Imperial College of Science, Technology and Medicine
Department of Computing

Incorporating Prior Knowledge Into Deep Neural Networks Without Handcrafted Features

Marta Garnelo Abellanas

11.01.2021

Submitted in partial fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London and
the Diploma of Imperial College London

Declaration of Originality

This statement is to certify that the work presented in this thesis is my own original research. Collaborators involved in the different projects have been acknowledged. Finally, this thesis is not being submitted for any other degree.

Copyright Declaration

Abstract

Deep learning (DL) is currently the largest area of research within artificial intelligence (AI). This success can largely be attributed to the data-driven nature of the DL algorithms themselves: unlike previous approaches in AI which required handcrafting and significant human intervention, DL models can be implemented and trained with little to no human involvement. The lack of handcrafting, however, can be a two-edged sword. DL algorithms are notorious for producing uninterpretable features, generalising badly to new tasks and relying on extraordinarily large datasets for training. In this thesis, on the assumption that these shortcomings are symptoms of the under-constrained training setup of deep networks, we address the question of how to incorporate knowledge into DL algorithms without reverting to complete handcrafting in order to train more data efficient algorithms. We start by motivating this line of work with an example of a DL architecture which, inspired by symbolic AI, aims at extracting symbols from a simple environment and using those for quickly learning downstream tasks. Our proof-of-concept model shows that it is possible to address some of the data efficiency issues as well as obtaining more interpretable representations by reasoning at this higher level of abstraction. Our second approach for data-efficiency is based on pre-training: the idea is to pre-train some parts of the DL network on a different, but related, task to first learn useful feature extractors. For our experiments we pre-train the encoder of a reinforcement learning agent on a 3D scene prediction task and then use the features produced by the encoder to train a simulated robot arm on a reaching task. Crucially, unlike previous approaches that could only learn from fixed view-points, we are able to train an agent using observations captured from randomly changing positions around the robot arm, without having to train a separate policy for each observation position. Lastly, we focus on how to build in prior knowledge through the choice of dataset. To this end, instead of training DL models on a single dataset, we train them on a *distribution* over datasets that captures the space of tasks we are interested in. This training regime produces models that can flexibly adapt to any dataset within the distribution at test time. Crucially they only need a small number of observations in order to adapt their predictions, thus addressing the data-efficiency challenge at test time. We call this family of meta-learning models for few-shot prediction *Neural Processes* (NPs). In addition to successfully learning how to carry out few-shot regression and classification, NPs produce uncertainty estimates and can generate coherent samples at arbitrary resolutions.

Contents

Acronyms, variables and mathematical symbols

Acronyms

Acronym	**Meaning**	**Section**
A3C	Asynchronous advantage actor-critic	2.5.4
CNN	Convolutional neural network	2.2.1
CNP	Conditional neural process	5
DDPG	Deep deterministic policy gradients	2.5.4
DP	Dynamic programming	2.5.2
DQN	Deep-Q-networks	2.5.4
DRAW	Deep recurrent attentive writer	4.2
ELBO	Evidence lower bound	2.3.1
GP	Gaussian Process	2.4.2
GQN	Generative query networks	4
KL (divergence)	Kullback Leibler divergence	2.3.1
kNN	k-nearest neighbours	2.4.1
LSTM	Long short-term memory network	2.2.2
MANN	Memory augmented neural networks	
MC	Monte Carlo	2.5.2
MCMC	Markov chain Monte Carlo	2.3.2
MLP	Multilayer perceptron	2.1
MN	Matching network	
MNIST	Modified National Institute of Standards and Technology database (dataset of handwritten digits)	
NN	Neural network	2.1

NP	Neural process	6
PPO	Proximal policy optimisation	2.5.4
RGB	Red Green Blue (colour model)	5.5
RL	Reinforcement learning	2.5
ReLU	Rectified linear unit	2.1
RN	Relation network	2.2.3
RNN	Recurrent Neural Network	2.2.2
SARSA	State-action-reward-state-action (a reinforcement learning algorithm)	2.5.2
SVG	Stochastic value gradients	2.5.4
TD	Temporal difference	2.5.2
TRPO	Trust region optimisation	2.5.4
VAE	Variational Autoencoder	2.3.2

Variables

Variable	**Meaning**	**Section**
a	Representation aggregator	4, 5, 6
a_τ	Agent action at time step τ (RL)	2
$\mathcal{A}(s)$	The action space of an agent in state s.	2
$\mathbf{b}$	Bias vector of a neural network	2
$\mathbf{c}$	LSTM cell state	2, 4
$C = (\mathbf{x}^c, \mathbf{y}^c)$	Context points	5, 6
$d_{final}, d_{palm}, d_{pinch}$	Distances from the Jaco hand to the target	4
$\mathbf{d}$	Binary variable for the cross entropy loss	2
$\mathcal{D}$	Dataset, data distribution	5, 6
f	Function $f : X \to Y$	5, 6
F	Stochastic process $F : \mathcal{X} \to \mathcal{Y}$	6
g	Neural network decoder	5, 6
h	Neural network encoder	5, 6
$\mathbf{h}$	Output of the LSTM	2, 4
$\mathbf{h}$	Hidden layer of a neural network	5, 6
I	Identity matrix	2, 5, 6
$J(\theta, \phi)$	expected reward for policy gradients	2
k	Bandit arm of the contextual bandit task.	6
K	Number of classes for classification tasks	5
l	Lengthscale of a GP kernel	2
L	Similarity measure for object tracking.	3
$\mathcal{L}$	Objective function of a neural network	2
m	Number of target points	4, 5, 6
n	Number of context points	4, 5, 6
ΔN	Difference in number of neighbouring objects	4, 5, 6
$p(x)$	Probability distribution over x	2
P	Period of the periodic GP kernel	2
q	Variational posterior	2
$q_\pi(s_\tau, a_\tau)$	State-value (RL)	2
$\mathcal{Q}$	Family of posterior functions	2
$\mathbf{r} = h(\mathbf{x},\, \mathbf{y})$	Latent representation generated by an encoder.	4, 5, 6
r_τ	Reward at time step τ (RL).	2
R_τ	Return (RL).	2

s_τ	Agent state at time step τ (RL).	2
$\mathcal{S}$	State space of an environment (RL).	2
τ	Iteration number (for RL or for recurrent algorithms)	2, 4
$T = (\mathbf{x}^t, \mathbf{y}^t)$	Target points	5, 6
$\mathcal{T}$	Total number of iterations of the DRAW algorithm	4
$\mathbf{u}$	Skip connections in the DRAW model	2
$v_\pi(s_\tau)$	State-value (RL)	2
$\mathbf{W}$	Weight matrix of a neural network	2
$x, \mathbf{x}$	Function input	2, 4, 5, 6
$\mathbf{x}^c = (\mathbf{x}^c_1, ..., \mathbf{x}^c_n)$	Inputs at context points	4, 5, 6
$\mathbf{x}^t = (\mathbf{x}^t_1, ..., \mathbf{x}^t_m)$	Inputs at target points	4, 5, 6
$y, \mathbf{y}$	Function output	2, 4, 5, 6
$\hat{y}, \hat{\mathbf{y}}$	Predicted function output	2, 4, 5, 6
$\mathbf{y}^c = (\mathbf{y}^c_1, ..., \mathbf{y}^c_n)$	Outputs at context points	4, 5, 6
$\mathbf{y}^t = (\mathbf{y}^t_1, ..., \mathbf{y}^t_m)$	Outputs at target points	4, 5, 6
$z, \mathbf{z}$	Latent variable sampled from a latent distribution.	4, 5, 6
α	Learning rate for gradient descent	2
γ	Discount factor for the expected reward (RL)	2
η	Neural network	4
ϑ	Neural network parameters	4
θ	Neural network parameters	2, 4, 5, 6
Θ_i	Objects extracted by neuro-symbolic model.	3
μ	Mean	2, 4, 5, 6
ξ	Neural network parameters	4
π	Policy (RL)	2
π^*	Optimal policy (RL)	2
ρ_x	Collection of joint distributions over x.	6
σ	Variance	2, 4, 5, 6
$\Sigma, \mathbf{\Sigma}$	Covariance	2, 4, 5, 6
τ	Time step in RL / iteration in RNNs	2
ϕ	Sufficient statistics that parametrise an output distribution	5
ψ	Permutation	6
ω	Neural network parameters	4

Mathematical symbols

Symbol	**Meaning**	**Section**
$\mathbb{E}$	Expectation	2
$f'(x) = \frac{\partial f}{\partial x}$	Derivative	2
$\frac{\partial}{\partial x}$	Partial derivative	2
∇	Gradient	2
$\mathcal{N}(\mu, \sigma)$	Gaussian distribution with mean μ and variance σ.	2
$\mathcal{R}^d$	Real space of dimension d.	2

List of Figures

List of Tables

Chapter 1

Introduction

Deep learning (DL) has attracted a lot of attention from the machine learning (ML) community over the past decade. The excitement around this particular field of research has, without a doubt, been fueled by its many exceptional successes [Levine et al., 2016, Mnih et al., 2015, Silver et al., 2016]. Crucially, DL has also been successfully applied to a wide range of real world problems beyond academic research [Karpathy and Fei-Fei, 2015, Xu et al., 2015, Vaswani et al., 2017, Senior et al., 2020]. At a higher level, the exceptional achievements of DL can largely be attributed to ongoing hardware improvements (such as GPUs and TPUs), as well as the increased availability of large training datasets. Equipped with these tools, practitioners have been able to train increasingly complex DL models, while establishing crucial theoretical and practical foundations along the way. These advances continue to drive the extraordinary growth of the DL field of research today.

Despite these successes some downsides of deep learning algorithms are becoming increasingly apparent. One of the most critical ones is their data inefficiency [Lake et al., 2017, Garnelo et al., 2016, Marcus, 2018]: training neural networks on supervised tasks, a problem which is largely considered to be solved today, usually requires datasets with thousands of samples. Training reinforcement learning (RL) agents often calls for several orders of magnitude more samples in the form of experience. Whether the data needs to be labelled, as in the case of supervised learning, or gathered from interactions with an environment, as in RL, these numbers are prohibitively large for most real-world applications. To make matters worse, a trained network will often not generalise well to test samples that lie outside of the distribution of the training set, limiting the application of deep models to data samples that are similar to the training samples.

The work presented in this thesis can be split into three separate research ideas. A common objective among all of them is to address the data intensity of deep learning algorithms. To this end, we take inspiration from three different research areas of artificial intelligence to explore whether various aspects of deep learning models can be rendered more efficient. We start by motivating the need for data efficiency with a simple experiment in chapter 3 where, inspired by symbolic methods, we test whether concept-level reasoning allows for faster adaptation to new tasks. Our second approach is based on model pre-training methods and addresses the question of whether pre-training parts of our deep neural model on different, but relevant tasks can result in increased data efficiency on other down-stream tasks. Finally, we take inspiration from few-shot prediction models (such as Gaussian processes) to develop neural algorithms that are flexible and data-efficient at test time.

We start with a simple motivating example in chapter 3 where we take inspiration from symbolic reasoning to design our neural network architecture. Symbolic methods are not subject to many of the drawbacks of deep learning algorithms. Given that the symbolic language is defined in terms of objects and relations, symbolic algorithms are able to reason at a higher level of abstraction than neural networks, whose input is usually higher-dimensional raw data samples (e.g. pixel values of an image). This abstraction allows symbolic methods to generalise at a higher conceptual level, whereas deep learning algorithms often fail to generalise to tasks that would seem trivial to a human (the prediction of a deep image classifier can be changed drastically by adding some human imperceptible noise, for example) [Akhtar and Mian, 2018]

Combining symbolic and deep methods, however, is not straightforward. Symbolic approaches owe the abstraction of their representations to careful human handcrafting, whereas deep learning models leverage a large part of their power from the data directly with little constraints on what the intermediate representations generated by the network should look like. As part of this work we build a deep architecture that extracts salient features from an image and returns a list of symbols and their relations. This list is then passed on to a reinforcement learning system that learns a policy based on the higher-level symbols rather than some unstructured latent representations. We show that by taking this approach the resulting agent is able to generalise better to unseen situations. Moreover, when trained on a single task, the agent doesn't overfit and is able to generalise to new tasks at test time.

In chapter 4 we tackle the data-efficiency problem from a pre-training point of view. Pre-training is a technique widely used in machine learning whereby a deep network that has been trained on a particular task is reused for a different task by either using it to initialise the parameters for fine-tuning or as a feature extractor. The idea behind pre-training is that there are common patterns between the two tasks that render the weights of the pre-trained network more useful than random initialisation. This method is particularly common in computer vision, where the image encoders often consist of large networks which have been pre-trained on natural images. Using these pre-trained networks has been shown to result in better performance on other down-stream tasks such as classification [Sharif Razavian et al., 2014, Erhan et al., 2010, Hendrycks et al., 2019].

In the case of image classification (or any similar task which requires image processing) it is clear why using a network that has been pre-trained on natural images can help speed up the learning process, as the first network will pick up common patterns of pixels that are present in most natural images. It is less straightforward, however, how this notion of pre-training can be generalised to tasks where the common pattern is not as obvious e.g. reinforcement learning (RL). In order to obtain a pre-trained network that is useful for other complex downstream tasks it is therefore crucial to design a suitable pre-training task. In chapter 4 our goal is to train an RL agent that successfully learns to carry out a simulated reaching task in a 3D environment. Crucially, we want the agent to take in observations which are taken from any point in 3D space (within a reasonable distance) as input, rather than images observed from a fixed camera position as is usually the case in RL. In order to train this agent in a data-efficient manner we pre-train the input encoder on a separate task. Given that we want the encoder to produce representations that are small and invariant to the position of the observer, we first train a generative query network (GQN) [Eslami et al., 2018a], a recently introduced model of scene understanding which is able to carry out scene predictions from varying camera angles, to reconstruct scenes from the reaching task. Using the trained GQN encoder to pre-process the agent's observations leads to significantly faster learning and more robust performance on the reaching task compared to architectures which are trained from scratch.

For the final part of this thesis we focus on data-efficiency at test time and to this end we draw inspiration from Gaussian processes (GPs). GPs are non-parametric, Bayesian models which are usually applied to few-shot regression tasks. Unlike neural networks, GPs are not trained. Depending on the kernel and its parameters a GP will capture a certain prior belief over the underlying ground truth function. This belief is expressed in terms of a mean estimate as well as a measure of uncertainty over the predicted values given any observations. As we add observations of the ground truth this belief is updated in a Bayesian fashion. Because GPs are regressing entire distributions over functions rather than a single function, they are able to adapt to any instantiation of a function at test time. In addition, since GPs carry a prior belief over what the function space looks like they can produce reasonable predictions from few observations already.

In order to combine ideas from GPs with neural networks we modify the training regime of standard deep models in chapters 5 and 6. Instead of training on a dataset that consists of several mini-batches drawn from a single function we use datasets consisting of distributions over functions. For example, rather than training a network to regress a function describing the dynamical model of a single cartpole pendulum we train our model on functions that model lots of different cartpoles with different properties. As a result our model learns a whole distribution over functions from the dataset, rather than a single one and can narrow down its predictions on what cartpole it is observing based on observations it is given at test time. This type of training setup is currently also often referred to as meta-learning and it has been successfully applied to classification [Vinyals et al., 2016, Rusu et al., 2018] as well as reinforcement learning tasks [Finn et al., 2017]. In chapters 5 and 6 we introduce Neural Processes (NPs), which learn to carry out these few-shot regression tasks and are able to generate accurate predictions from a limited number of observations at test time.

In addition to viewing the work in this thesis through the lens of data-efficiency it is also possible to interpret the different approaches as alternative ways of building in prior knowledge into a deep learning system. In the case of the symbolic approach the prior knowledge is built in by handcrafting parts of the architecture to produce individual symbols. In doing so we are explicitly incorporating a notion of objects and object persistence over time. In addition, by focusing on the relation between symbols rather than considering them individually we build in our belief that relational information is important for higher level reasoning. In the case of the pre-trained model in the following chapter we are building in prior knowledge about our goal task by designing pre-training tasks that will capture relevant information about the final task and speed up the training process. In this case the prior is not explicitly built into the architecture but is instead expressed through the choice of pre-training task. Finally, neural processes learn to capture some prior belief over what the underlying ground truth function looks like from the data directly. Unlike Gaussian processes, for NPs the prior knowledge is not incorporated through the choice of kernel. Instead, we build in prior knowledge by selecting a training dataset that we believe captures a reasonable prior over the test-time distribution of interest.

The rest of this thesis is structured as follows:

- We introduce the relevant technical background in Chapter 2. This includes neural networks (Section 2.1), generative models (Section 2.3), non-parametric methods (Section 2.4) and reinforcement learning (Section 2.5).
- In Chapter 3 we present our neuro-symbolic model and compare its performance against DQN on a simple pick-up game.
- We use the representations from a pre-trained generative query network to train a reinforcement learning agent in Chapter 4 and show that doing so improves robustness and accelerates learning.
- In Chapter 5 we introduce conditional neural processes, a family of models that extend the training regime of generative query networks to tasks beyond scene understanding such as few-shot regression and few-shot classification.
- We introduce neural processes, the latent variable version of conditional neural processes in Chapter 6 and show how this extension allows us to generate different coherent predictions on the few-shot regression tasks.

1.1 Publications

The following peer-reviewed publications have come out of the research presented in this thesis:

Garnelo, Marta, Kai Arulkumaran, and Murray Shanahan. Towards deep symbolic reinforcement learning. *NIPS - Deep Reinforcement Learning Workshop*, 2016

For this publication I developed the main idea and the experimental setup in collaboration with my PhD advisor Prof Murray Shanahan. I implemented the environment and the model and carried out all of the experiments except for the baseline on DQN which was done by the second author. The paper was written jointly with Prof Shanahan.

SM Ali Eslami, Danilo Jimenez Rezende, Frederic Besse, Fabio Viola, Ari S Morcos, **Garnelo, Marta**, Avraham Ruderman, Andrei A Rusu, Ivo Danihelka, Karol Gregor, et al. Neural scene representation and rendering. *Science*, 360(6394):1204–1210, 2018b

As part of this paper I carried out the experiments that used GQN for reinforcement learning. All other sections of the paper focus on the scene understanding task, while this section shows that the representations learned by GQN can be applied to other downstream tasks as well. I created the environments and the models and trained GQN and RL agents (the latter with the help of Andrei Rusu). I also wrote the RL part of the paper.

Garnelo, Marta, Dan Rosenbaum, Christopher Maddison, Tiago Ramalho, David Saxton, Murray Shanahan, Yee Whye Teh, Danilo Jimenez Rezende, and Ali Eslami. Conditional neural processes. In *International Conference on Machine Learning*, 2018a

The idea for CNPs was primarily developed by me (in conversations with Dan Rosenbaum, Ali Eslami and Danilo Rezende). I designed, implemented and ran the experiments for the paper and also wrote the majority of the paper.

Garnelo, Marta, Jonathan Schwarz, Dan Rosenbaum, Fabio Viola, Danilo J Rezende, SM Eslami, and Yee Whye Teh. Neural processes. In *ICML Workshop on Theoretical Foundations and Applications of Deep Generative Models*, 2018b As with CNPs this project was primarily driven and carried out by me. I also received help writing from Prof Yee Whye Teh and Jonathan Schwarz ran the experiments for the section on Bayesian optimisation.

Chapter 2

Technical Background

2.1 Neural networks

Neural networks (NNs) lie at the core of deep learning research. At a higher level, NNs can be used as function approximators that map an input x to an output $y = f(x)$ via a sequence of transformations and non-linearities. More concretely, a neural network consists of several layers of 'neurons' (thus the name neural network) that are connected in a particular fashion. The first layer is referred to as input layer and, similarly, the last layer is called output layer. Any layers in between are termed hidden layers as their values are, in principle, unknown to the user. We will refer to the j^{th} neuron of the i^{th} hidden layer as $h_{i,j}$. In the case of feedforward networks, $h_{i,j}$ is connected to all the neurons of the previous layer $\mathbf{h}_{i-1}$ and the following layer $\mathbf{h}_{i+1}$. The activations of all the neurons $\mathbf{h}_i$ in layer i is therefore a function of the activations of $\mathbf{h}_{i-1}$ weighted by the connection weights $\mathbf{W}_i$.

$$\mathbf{h}_i = \mathbf{W}_i \mathbf{h}_{i-1} + \mathbf{b}_i \tag{2.1}$$

where $\mathbf{h_i}$ and $\mathbf{h}_{i-1}$ are vectors of length n_i and n_{i-1} respectively and n_i and n_{i-1} correspond to the number of neurons in layers i and $i-1$. $\mathbf{W}_i$ is the weight matrix of shape $n_i \times n_{i-1}$ containing the weights of the connections between layers i and $i-1$ and finally $\mathbf{b}_i$ is the vector of biases of length n_i.

As a result of stacking several of these layers deep neural networks of sufficient capacity are able to approximate functions of arbitrary complexity. In order to achieve this, however, one needs to include non-linear activation functions between the linear layers described in equation 2.1. These activation functions are usually

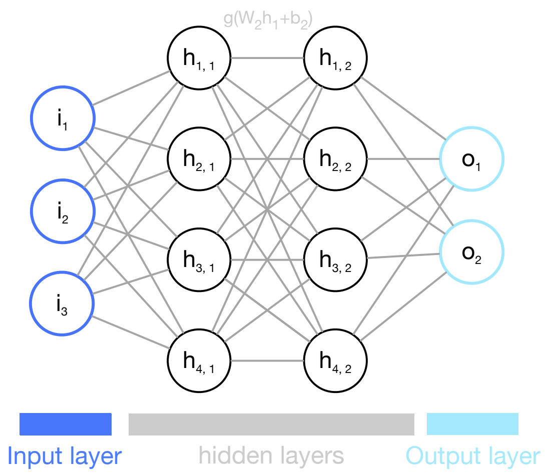


Figure 2.1: Schematic of a fully connected neural network with two hidden layers.

added on top of each layer in the network as:

$$\mathbf{h}_i = g(\mathbf{W}_i\mathbf{h}_{i-1} + \mathbf{b}_i) \tag{2.2}$$

Some of the most common choices include (see Figure 2.2 for plots of the functions):

- **Rectified Linear Units (ReLU)** - ReLU is the the most commonly used activation function. It is given by

  $$g(x) = \max(0, x) \tag{2.3}$$

  which is an easy function for optimisation as it corresponds to a linear function for non-negative values. Extensions of this activation function include leaky ReLU [Maas et al., 2013] where a fixed small value is chosen instead of 0 in equation 2.3 or parametric ReLU [He et al., 2015] where this small value is learned.

- **Logistic Sigmoid** - The sigmoid activation function is defined as:

  $$g(x) = \frac{1}{1 + \exp(-x)} \tag{2.4}$$

  Sigmoid units are bounded from above and below and quickly saturate for very positive and very negative values. Because they are only sensitive to a small range of values around 0 their use is now discouraged.

- **Hyperbolic Tangent** - The hyperbolic tangent function is closely related to the sigmoid function with:

  $$tanh(x) = 2\sigma(2x) - 1 \tag{2.5}$$

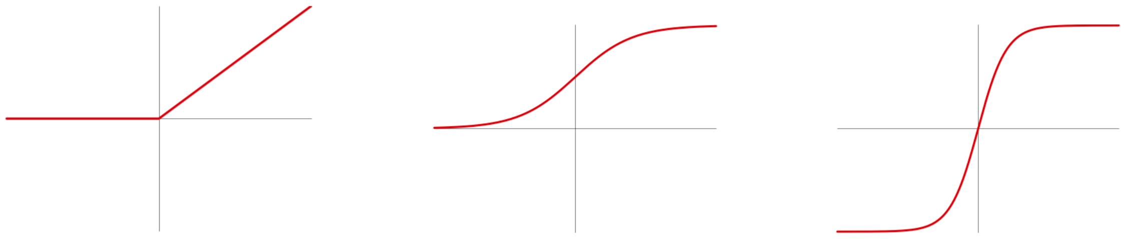

(a) ReLU activation. (b) Sigmoid activation. (c) Hyperbolic tangent activation.

Figure 2.2: Three different types of activation functions.

where $\sigma(x)$ corresponds to the sigmoid function described in equation 2.6. The hyperbolic tangent is generally preferred over the sigmoid as it passes through the origin (0, 0) and therefore its behaviour is somewhat more similar to that of a linear function.

The output of the final layer is often left as is (this could be considered as using a linear unit as the output activation). An alternative approach used for representing probability distributions over the output values is the softmax activation. This output unit is a generalisation of the sigmoid unit and is dfined as:

$$g(x) = \frac{\exp(x)}{\sum \exp(x_j)} \tag{2.6}$$

Softmax functions are usually applied to classification tasks where they produce a probablility distribution over the possible labels.

2.1.1 Optimisation

The idea behind training a neural network is to learn the parameters θ (consisting of the weights $\mathbf{W}$ and biases $\mathbf{b}$ in equation 2.1) such that the resulting network $f_\theta(\mathbf{x})$ is able to approximate the desired target function. More concretely, our goal is to update the parameters θ such that the network's performance, as assessed by an objective function $\mathcal{L}(\theta)$ (also called cost function or loss function), improves with every iteration. As such, at its core this training procedure is an optimisation problem, where the best possible performance will correspond to the global minimum of our objective function.

$$\theta^* = \arg\min \mathcal{L}(\theta) \tag{2.7}$$

The most common approaches to this optimisation problem are gradient descent-based methods. At a higher level these methods use the derivative of the objective function $\mathcal{L}'(\theta) = \frac{d\mathcal{L}}{d\theta}$ to update the parameters along the descending slope of $\mathcal{L}$. While gradient-based methods work well in practice, the fact that $\mathcal{L}$ is non-convex can lead the optimisation to converge on critical points that don't correspond to the global minimum but where $\mathcal{L}'(\theta) = 0$ still holds, such as local minima and saddle points.

In the context of neural networks the derivative is taken with respect to a vector rather than a single value and as such we speak of gradients. More concretely, the gradient is a vector containing the partial derivatives $\frac{\partial}{\partial x_i} f(\mathbf{x})$ for all dimensions i of $\mathbf{x}$. We denote the gradient of $f(\mathbf{x})$ as $\nabla_{\mathbf{x}} f(\mathbf{x})$. The update rule for the network's parameters θ is then given by:

$$\theta' = \theta - \alpha \nabla_\theta \mathcal{L}(\theta) \tag{2.8}$$

where α is the learning rate.

The gradient of the objective function $\nabla_\theta \mathcal{L}(\theta)$ is calculated via back-propagation. Given some measure $l(\mathbf{y}, \hat{\mathbf{y}})$ that evaluates the prediction $\hat{\mathbf{y}} = f_\theta(\mathbf{x})$ from our network for any input $\mathbf{x}$ we define the objective function as

$$\mathcal{L}(\mathbf{x}) = l(f_{\mathbf{y},\theta}(\mathbf{x})) \tag{2.9}$$

The relation between the gradients of this objective function $\mathcal{L}$ with respect to θ and f is given by the chain rule:

$$\nabla_\theta \mathcal{L} = \left(\frac{\partial f_\theta(\mathbf{x})}{\partial \mathbf{x}} \right)^\top \nabla_f \mathcal{L} \tag{2.10}$$

As a result in order to determine the gradients we need to calculate the Jacobians of the operations in our computation graph and use them to propagate the gradient backwards.

So far we have not specified the loss function $l(\mathbf{y}, \hat{\mathbf{y}})$ that we are using for our network. There are many options and some of the most common ones are:

- **Mean Squared Error (MSE)**: commonly used for linear regression.

$$l(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{N} \sum_{n=1}^{N} \left(\mathbf{y}_n - \hat{\mathbf{y}}_n \right)^2 \tag{2.11}$$

- **Squared L2 Norm**: effectively an unnormalised MSE loss.

$$l(\mathbf{y}, \hat{\mathbf{y}}) = \sum_{n=1}^{N} \left(\mathbf{y}_n - \hat{\mathbf{y}}_n \right)^2 \tag{2.12}$$

- **Cross entropy**: a common choice for classification tasks. In the binary case the loss is given by:

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\mathbf{d} \log(\hat{\mathbf{y}}) + (1 - \mathbf{d}) \log(1 - \hat{\mathbf{y}}) \tag{2.13}$$

where **d** is a binary variable that corresponds to 1 if the data point was assigned to the correct class and 0 if it was not.

Finally one needs to choose an optimiser to carry out the gradient descent updates. There are a large number of variations (see [Ruder, 2016] for an extensive overview) that usually differ in the way they adapt the learning rate α in Equation 2.10. For the majority of the experiments we will make use of an optimiser called Adam (Adaptive Moment Estimation). At a high level Adam adapts the learning rate of each individual parameter by taking into consideration the recent history of each parameter's gradient as well as each parameter's squared gradient. Using the parameter's previous gradients corresponds to adding 'momentum' to the learning rate, which encourages larger values of α along steeper directions and dampens oscillatory behaviour. In addition, Adam uses the square gradients of the previous time steps to encourage larger learing rates for parameters that correspond to infrequent features in the data.

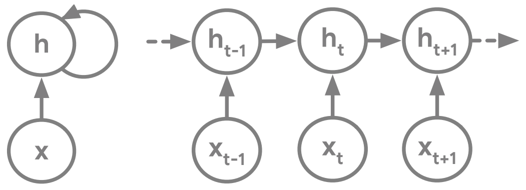
2.2 Common types of Neural Networks

In recent years a number of neural modules have established themselves as fundamental building blocks for current deep architectures. In the following we outline a few of the most important ones.

2.2.1 Convolutional neural networks

Convolutional neural networks (CNNs) are a common type of network that is primarily applied to image data. CNNs differ from fully connected networks in the wiring between layers: rather than being connected to every neuron of the previous layer h_{i-1}, neurons are only connected to units in its so-called receptive field. In the case of 2-dimensional data this is usually a square patch of small dimensions. Crucially, the weights that connect this patch to the neuron in the next layer are shared across all of the patches in that layer. As an illustration imagine a 3×3 matrix of weights that is shifted pixel-wise across an image and at every position we obtain the neuron of the next layer by summing the products of the weight matrix with the pixel values. This type of architecture has some desirable properties:

- **Sparse connectivity**: rather than connecting all the neurons between two layers CNNs capture local correlations in the data. These local correlations play a crucial role for image processing, for example.
- **Translation invariance**: given that the same weights are applied to the different patches, features of the data (e.g. egdes) will be captured regardless of their position within the image.
- **Parameter sharing** : since all receptive fields share the same weights, the number of parameters of CNNs is significantly smaller than feedforward networks.



(a) RNN (b) Unrolled RNN

Figure 2.3: The graphical model for recurrent neural networks.

Some CNNs include so-called pooling layers between the convolutional layers that reduce the dimensionality of the image by summarising the value of a patch using a single value. This is usually achieved by either selecting the maximum value within the receptive field (max-pooling) or taking the average over the patch (average-pooling).

Throughout this thesis we will make use of CNNs whenever we need to process image input (Chapters 3, 4 and 5).

2.2.2 Recurrent neural networks

When working with sequential data the most common family of models are Recurrent Neural Networks (RNNs). Unlike feedforward networks RNNs are defined recursively:

$$h^{(\tau)} = f_\theta(h^{(\tau-1)}, x^{(\tau)}) \tag{2.14}$$

where $h^{(\tau)}$ is the output at the τ^{th} iteration. In contrast to feedforward networks where the number of layers is fixed, the number of iterations in RNNs can be of variable length. The diagram in Figure 2.3a is a visualisation the update rule described in equation 2.14. However it is often easier to visualize the computational graph of RNNs by unfolding it as shown in figure 2.3b.

Long Short-Term Memory networks (LSTM)

While RNNs are well suited for time series, training them over large numbers of iterations can lead to vanishing gradients and long-term dependencies might not be captured. LSTMs [Hochreiter and Schmidhuber, 1997] overcome this problem by introducing an additional variable c that captures the long-term dependencies along with a differentiable gating mechanism that determines how c is updated at every iteration (see Figure 2.4 for a schematic of the LSTM and how it compares to an RNN).

More concretely, at every iteration τ the input x_τ and the previous state $h_{\tau-1}$ are combined and passed through three different gating mechanisms. The first one is termed the 'forget gate' as it calculates a mask

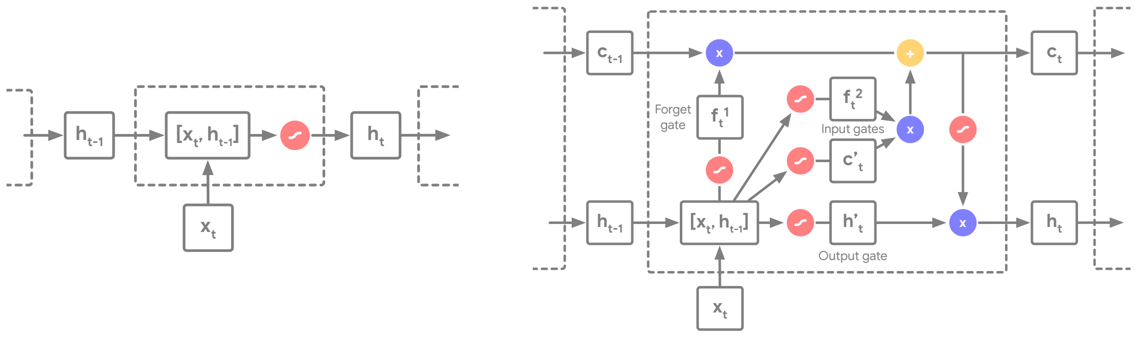


Figure 2.4: RNN (left) vs LSTM (right). See section 2.2.2 for a detailed explanation.

f^1_τ over the values in the long-term memory $c_{\tau-1}$. The second gate is the 'input gate' and it combines a second mask f^2_τ with the updates c'_τ for the long-term memory, to determine how the long-term memory is going to be modified after having been processed by the forget gate. Finally the 'output gate' takes in the previous state $h_{\tau-1}$ and returns an updated estimate h'_τ which is combined with the updated long-term memory c_τ to produce the next state h_τ.

LSTMs have been applied to a wide range of tasks such as speech recognition [Graves et al., 2013, Fernández et al., 2007, Graves and Schmidhuber, 2005], time series prediction [Schmidhuber et al., 2005] or handwriting recognition [Graves and Schmidhuber, 2009, Graves et al., 2008] and have established themselves as the most effective deep learning model for processing time series. In addition to being powerful and flexible models RNNs and LSTMs have the following benefits:

- **Variable length**: given that linear algebra underlies most of deep learning, the sizes and shapes of most elements (layers, weights, inputs) have to be specified in advance and remain fixed. RNNs and LSTMs, however, are capable of working with data of varying length.
- **Weight sharing**: Similarly to CNNs that apply the same set of weights to each receptive field, RNNs share the weights between the computations at every step. This significantly reduces the overall number of parameters that need to be trained.

Although none of the work in this thesis handles temporal sequences directly we make use of LSTMs as part of a recurrent generative model for images in the decoder of the generative query network (this is described in more detail in section 4.2).

2.2.3 Relational Modules

So far we have considered convolutional networks, which incorporate a spatial bias and recurrent networks which are suitable for processing sequential data. Another bias that is straight forward to build into a deep

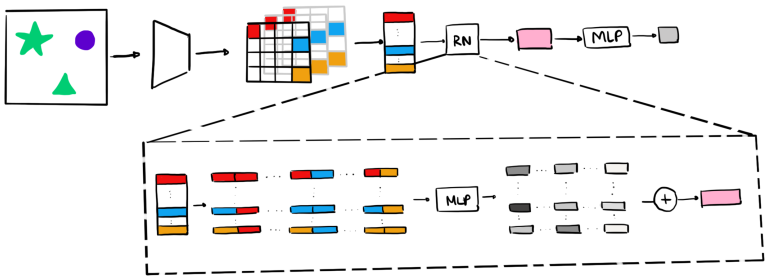


Figure 2.5: Schematic of a relation network.

learning architecture is a relational bias. This can be done by explicitly establishing connections between all the different sub-parts of the input data in our neural network architecture.

Relation Networks

Relation networks (RN) [Santoro et al., 2017] encode data (e.g. an image) into an intermediate representation r, which is divided into n smaller patches that capture different parts of the input. These patches are concatenated in pairs forming a $n \times n$ grid of patch combinations. The concatenated patches are passed individually through another network before being aggregated into a single representation by adding them up. More formally:

$$\mathrm{RN}(O) = g_{\vartheta}\Big(\sum_{i=1}^{n}\sum_{j=1}^{n} h_{\theta}(o_i, o_j)\Big) \tag{2.15}$$

where O is a set of n objects $\{o_1, o_2, ..., o_n\}$, $o_i \in \mathbb{R}^m$ (in practice these correspond to the patches in the images) and g and h are neural networks with parameters ϑ and θ respectively.

At a higher level, this type of architecture enforces the processing of the relations between the patches and has been shown to perform better than regular deep networks on tasks that require relational reasoning.

2.2.4 Autoencoders

Autoencoders are neural models for data compression that have become widely used for representation learning, particularly in the field of generative models. In their simplest form autoencoders are multi-layer feedforward networks with varying layer sizes, which are trained to produce an output that is as close as possible to the input.

In practice, the number of neurons in the first half of the network typically decreases with every layer, while the size of the layers in the second half increases layer-wise until it reaches the original size at the output-layer.

The smallest layer in the center can be seen as a compressed version of the input data and is often referred to as the intermediate representation of the autoencoder (although technically any intermediate representation would fit that description). Because this representation is usually the focal point of autoencoders people refer to the layers preceding it as the 'encoder' since it 'encodes' the input data (e.g. an image) into a lower-dimensional representation (e.g. a single vector). Similarly the layers that decompress the latent representation into the original input data are called the 'decoder'.

More formally an autoencoder can be described as:

$$\mathbf{r} = h_{\theta}(\mathbf{y}) \tag{2.16}$$

$$\hat{\mathbf{y}} = g_{\vartheta}(\mathbf{r}) \tag{2.17}$$

where $\mathbf{r}$ is the (generally) lower dimensional intermediate representation, h and g the encoder and decoder of the autoencoder with parameters ϑ and θ respectively and $\hat{\mathbf{y}}$ the model's prediction of the input $\mathbf{y}$. The parameters are updated using the difference between the prediction $\hat{\mathbf{y}}$ and the input $\mathbf{y}$ as the loss.

In Chapter 3 we make use of an autoencoder as a way of training an unsupervised neural model to pick up salient regions of the images. The encoder of the trained autoencoder is then used in a second stage to produce representations which are processed into symbols and fed to the reinforcement learning agent.

In the case of autoencoders the mapping from input data to latent representation and back to input space is completely deterministic. In the following section we will introduce variational autoencoders which take a probabilistic approach to autoencoders.

2.3 Generative models

Machine learning algorithms can be broadly divided into two categories: discriminative models and generative models. Given an observed variable x and a target variable y discriminative models model only $p(y|x)$ whereas generative models learn the joint probability $p(x, y)$. This can be particularly useful if the target variable y is an unobserved latent variable z and training $p(z|x)$ directly is not an option.

Consider a set of random variables with the joint distribution $p(x, z)$ where x are observed and z latent variables. We can generate samples of x by first sampling from $p(z)$ and then from $p(x|z)$. Alternatively we can compute the value of the unseen variable z given some observation x by sampling from $p(z|x)$. We refer to this second process as inference. Calculating $p(z|x)$, however, is intractable as it requires computing the following integral:

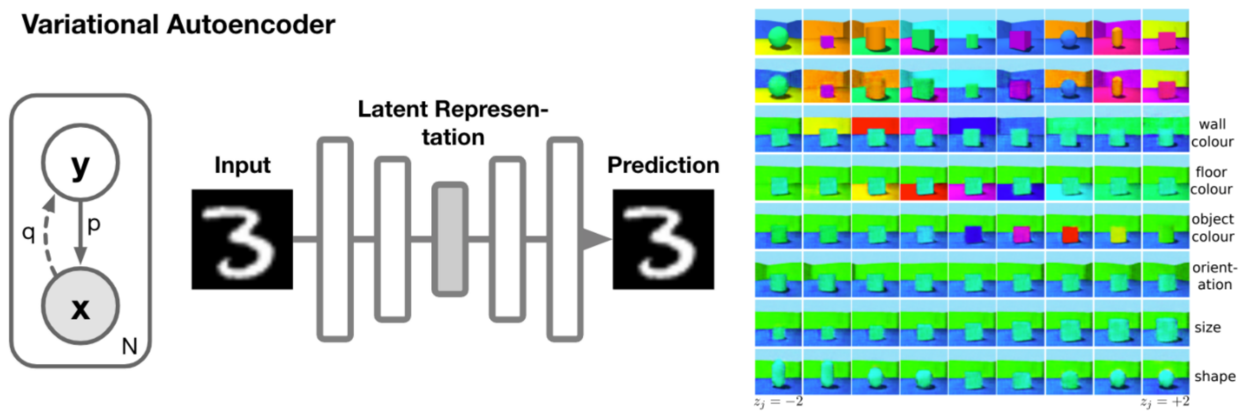


Figure 2.6: Left: Graphical model for a VAE. Middle: The prediction task carried out by a VAE on the MNIST dataset. Right: latent traversals with disentangled representations of models trained on the 3D Shapes dataset (figure from [Kim and Mnih, 2018])

$$p(z|x) = \frac{p(z,x)}{p(x)} = \frac{p(z,x)}{\int_z p(x,z)\mathrm{dz}} \tag{2.18}$$

One approach to approximating the density of $p(z|x)$ is to use sampling methods such as Markov chain Monte Carlo. A second approach is to apply variational inference, which we explain in the following section.

2.3.1 Variational Inference

Variational inference is a powerful statistical tool for approximating intractable integrals [Blei et al., 2017]. In the context of Bayesian inference it is often used as an alternative to Markov chain Monte Carlo methods to approximate posterior densities as in Equation 2.18.

The key idea is to choose a family of functions $\mathcal{Q}$ which are easy to optimise and try to find the $q(z) \in \mathcal{Q}$ that best matches $p(z|x)$. This turns the task of calculating an intractable integral into an optimisation problem over a family of functions of our choice. A common variational family is the mean-field variational family. The assumption for this type of funtions is that the latent variables are independent of each other, which simplifies the inference process:

$$q(z) = \prod_i q_i(z_i) \tag{2.19}$$

We measure the similarity between $q(z)$ and $p(z|x)$ using the Kullback-Leibler (KL) divergence, a measure for comparing probability distributions which is frequently used in variational inference. The KL divergence is defined as:

$$KL\big(q(x)||p(x)\big) = \mathbb{E}_q\left[\log\frac{q(x)}{p(x)}\right] \tag{2.20}$$

where the subscript in $\mathbb{E}_q$ indicates calculating the expected value with respect to $q(x)$:

$$\mathbb{E}_q\left[\log\frac{q(x)}{p(x)}\right] = \int_x \log\frac{q(x)}{p(x)}q(x)\mathrm{dx} \tag{2.21}$$

In the following section we describe in detail how variational inference is applied in the case of variational autoencoders.

2.3.2 Variational Autoencoders

The graphical model of variational autoencoders (VAEs) [Rezende et al., 2014, Kingma and Welling, 2013] is shown in Figure 2.6 and corresponds to the problem setup discussed at the beginning. As motivated earlier the goal is to learn a variational posterior $q(z)$ that best approximates the true posterior $p(z|x)$. We measure similarity using the KL divergence:

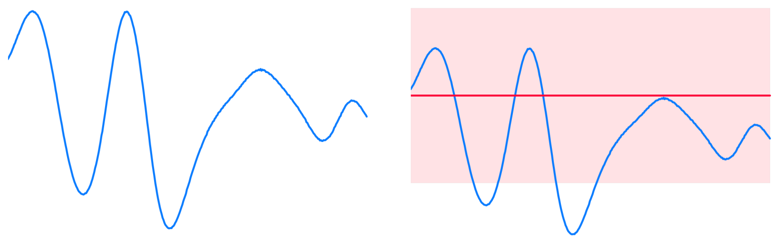
$$\begin{aligned} KL\big(q(z)||p(z|x)\big) &= \mathbb{E}_q\left[\log\frac{q(z)}{p(z|x)}\right] \\ &= \mathbb{E}_q\big[\log q(z)\big] - \mathbb{E}_q\big[\log p(z,x)\big] + \log p(x) \end{aligned} \tag{2.22}$$

Note that $\log p(x)$ is a constant which we don't have control over and therefore we can drop the last term when minimising equation 2.22. The remaining terms correspond to the negative evidence lower bound (ELBO):

$$\mathcal{L} = -\mathbb{E}_q\big[\log q(z)\big] + \mathbb{E}_q\big[\log p(z,x)\big] \tag{2.23}$$

Minimising the original KL divergence therefore corresponds to maximising the ELBO in equation 2.23. We can rearrange the terms of the ELBO to distinguish between different components:

$$\begin{aligned} \mathcal{L} &= -\mathbb{E}_q\big[\log q(z)\big] + \mathbb{E}_q\big[\log p(z,x)\big] \\ &= \mathbb{E}_q\big[-\log q(z) + \log p(x|z) + \log p(z)\big] \\ &= \mathbb{E}_q\big[\log p(x|z) + \log\frac{p(z)}{q(z)}\big] \end{aligned} \tag{2.24}$$



(a) Target function. (b) Prior mean and variance.

Figure 2.7: An example target function and the uninformed prior mean and variance of a GP.

In this case the first term $p(x|z)$ corresponds to the reconstruction error while the second term corresponds to the KL between the prior $p(z)$ and the approximate posterior $q(z)$.

From a practical point of view, $p(x|z)$ and $q(z)$ are implemented as neural networks whose outputs parametrise the mean and variance of a Gaussian distribution. The prior $p(z)$ is often set to a standard Gaussian, as this provides a closed form expression for the KL divergence.

Interestingly, it has been shown that weighting the KL-term induces factorised representations that are disentangled with respect to the attributes of the dataset. If the dataset consists of images of shapes in different colours and sizes, for example, different dimensions of a representation obtained with this type of VAE are going to capture different attributes of the data. In this particular example fixing all the values of the representation except for one and gradually modifying that one value would lead to changes of a single attribute (eg. colour or size) at output level (see Figure 2.6). This modified VAE is referred to as beta-VAE [Higgins et al., 2016].

At a higher level generative query networks (Chapter 4) and neural processes (Chapters 5 and 6) are conditional VAEs. As a result the objective functions and training setups will be variations of the original ELBO formulation described here.

2.4 Non-parametric methods

All of the models mentioned so far are implemented as neural networks and therefore fall under the category of parametric methods. This type of algorithms learn distributions with a pre-specified, fixed functional form which is governed by a set of learnable parameters. In the case of neural networks the form is defined by the neural architecture, and the network weights and biases constitute the parameters. In the following sections we will cover two of the most common non-parametric models: k-nearest neighbours and Gaussian processes. Unlike their parametric counterparts, non-parametric models don't assume a fixed functional form. Instead, their functional form grows with the number of data points and as a result they are generally more flexible than parametric methods. However, given that non-parametric models generate their predictions by storing and processing the data points observed so far they tend to be slower to query at test time.

2.4.1 Nearest Neighbours Methods

Despite its simplicity k-nearest neighbours (kNN) remains a common tool for data analysis. The idea behind kNN is the following: given some context observations $\mathbf{x}^c$ that have some associated values $\mathbf{y}^c$ (these can be labels in the case of classification or function outputs in the case of regression) the goal is to predict the value of $\mathbf{y}^t$ for a target point $\mathbf{x}^t$. This is achieved by measuring the distance from $\mathbf{x}^t$ to all of the context points $\mathbf{x}^c$ using any given metric (eg. Euclidean or Manhattan distance) and then selecting the k closest $\mathbf{x}^c$ points. The estimated value for $\mathbf{y}^t$ is the average of these k $\mathbf{y}^c$ values.

2.4.2 Gaussian Processes

Gaussian processes (GPs) are a powerful alternative to neural networks for regression tasks. Formally, GPs are defined as stochastic processes that consist of an infinite number of random variables. Crucially, the distribution over any finite number of these random variables is jointly Gaussian. In the following we will discuss how this property allows us to estimate function values $\mathbf{y}^t$ at unseen positions $\mathbf{x}^t$ based on some context observations $(\mathbf{x}^c, \mathbf{y}^c)$.

Consider the regression task mentioned in the previous section. As with kNNs, GPs make use of the context observations $(\mathbf{x}^c, \mathbf{y}^c)$ to make predictions at some target locations $\mathbf{x}^t$. However, instead of averaging over the k-nearest neighbours, GPs combine the values of $\mathbf{y}^c$ weighted by a metric which is defined by the so-called kernel $K(\mathbf{x}, \mathbf{x})$. The choice of kernel will have an impact on the form of the predictions generated by the GP and practitioners can build in any prior knowledge about the underlying data by choosing different kernels. Common options are:

- **Squared Exponential Kernel (Gaussian Kernel)**:

$$K_{SE}(\mathbf{x}^c, \mathbf{x}^t) = \sigma^2 \exp\left(- \frac{(\mathbf{x}^c - \mathbf{x}^t)^2}{2l} \right) \tag{2.25}$$

  where σ^2 is the output variance and l is the lengthscale that determines how far the GP can extrapolate.

- **Linear Kernel**:

$$K_{LIN}(\mathbf{x}^c, \mathbf{x}^t) = \sigma_b^2 + \sigma^2(\mathbf{x}^c - c)(\mathbf{x}^t - c) \tag{2.26}$$

  where σ_b^2 corresponds to the constant variance and c to the offset.

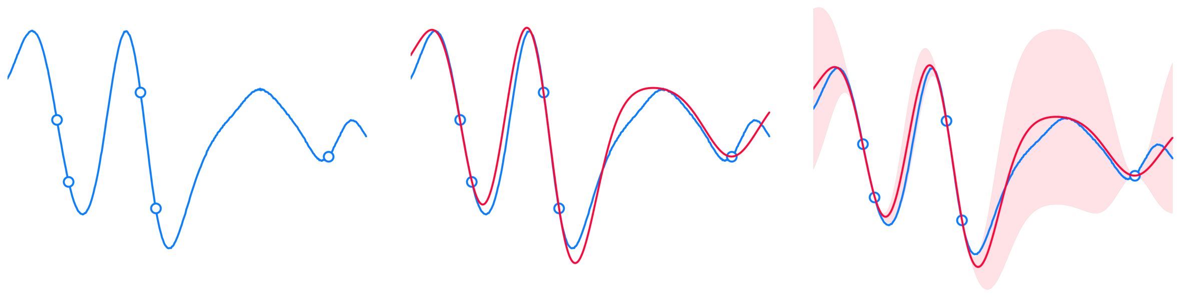

(a) Target with observations. (b) Posterior mean. (c) Posterior mean and variance.

Figure 2.8: The predicted mean and variance (red) given five observations (blue circles) from the target function (blue).

- **Periodic Kernel**:
$$K_{PER}(\mathbf{x}^c, \mathbf{x}^t) = \sigma^2 \exp\left(- \frac{sin^2(\pi|\mathbf{x}^c - \mathbf{x}^t|/P}{l^2}\right) \tag{2.27}$$
  where P is the period.

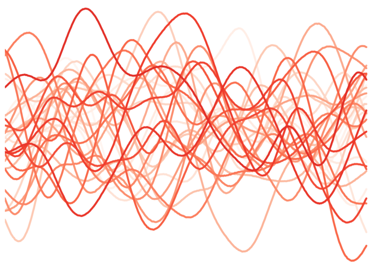
As mentioned above, the key underlying assumption of GPs is that the distribution of any subset of its random variables are jointly Gaussian. If we assume a zero-mean prior (i.e. we don't impose any prior knowledge about our distribution on the model) this can be expressed as:

$$y \sim \mathcal{N}(0, K(\mathbf{x}^t, \mathbf{x}^t)) \tag{2.28}$$

where $\mathcal{N}$ is a multivariate Gaussian parametrised by a zero-mean vector and a covariance defined by the kernel $K(\mathbf{x}, \mathbf{x})$. For the visualizations of this section we will make use of the squared exponential kernel, but all subsequent derivations hold under any choice of kernel.

Equation 2.28 captures the fact that for any set of target variables $\mathbf{x}^t$ (take, for example, a vector containing 100 values on the x-axis) each sample will produce a set of variables $\mathbf{y}^t$ (another vector of 100 values) that all come from the same sample function. Each of the functions in figure 2.9a corresponds to one sample of equation 2.28. Note that in the absence of any observations which narrow down our belief over the space of possible functions, repeatedly sampling from equation 2.28 will result in functions that cover the entire space. Importantly, the sampled functions will all behave similarly (i.e. have similar amplitudes and frequencies) as this is defined by the kernel, which is shared across samples.

Instead of sampling functions we can also visualize the model's predicted mean and variance on some regression task. Take the function depicted in figure 2.7a, for example, and let's assume that this function is our target function $y = f(x)$ that we are trying to approximate. f is unknown to us. Since we haven't observed any points on this function our prior belief over what it looks like corresponds to equation 2.28 and amounts


(a) Samples from prior.

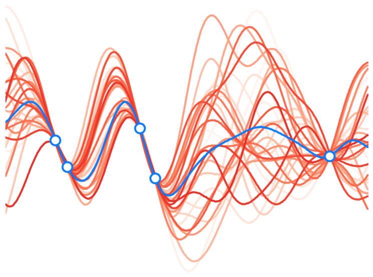

(b) Samples from posterior.

Figure 2.9: Generating samples from the prior and the posterior of a GP. Each curve corresponds to one sample.

to the zero-mean and constant variance visualized in figure 2.7b. This uninformed prior captures our belief over the set of possible functions before observing any context points $(\mathbf{x}^c, \mathbf{y}^c)$.

We now turn to the more interesting task of sampling functions that agree with a set of observation points $(\mathbf{x}^c, \mathbf{y}^c)$. We can split y in equation 2.28 into observed context variables $\mathbf{y}^c$ and un-observed target variables $\mathbf{y}^t$. Take the example in figure 2.8a: we have our unknown underlying ground truth function from which we have observed five context points $(\mathbf{x}^c, \mathbf{y}^c)$ (indicated by the small circles). The goal is to predict the y-values $\mathbf{y}^t$ of this function at other x-values $\mathbf{x}^t$. For visualization purposes we will choose $\mathbf{x}^t$ such that it covers the entire x-range of that function in small intervals. The crucial insight here is that if two random variables $\mathbf{y}^c$ and $\mathbf{y}^t$ are jointly Gaussian (which is true by eq. 2.28), their conditional probabilities are also Gaussian. We can thus rewrite equation 2.28 as:

$$\begin{bmatrix} \mathbf{y^c} \\ \mathbf{y^t} \end{bmatrix} = \mathcal{N}\left(0, \begin{bmatrix} K(\mathbf{x^c}, \mathbf{x^c}), K(\mathbf{x^c}, \mathbf{x^t}) \\ K(\mathbf{x^t}, \mathbf{x^c}), K(\mathbf{x^t}, \mathbf{x^t}) \end{bmatrix}\right) \tag{2.29}$$

Taking a Bayesian approach, the posterior probability $p(\mathbf{y}^t|\mathbf{x}^t, \mathbf{x}^c, \mathbf{y}^c)$ can then be calculated in closed-form as:

$$\begin{aligned} \mathbf{y^t}|\mathbf{x^t}, \mathbf{y^c}, \mathbf{x^c} \sim \mathcal{N}\big(&K(\mathbf{x^t}, \mathbf{x^c})K(\mathbf{x^c}, \mathbf{x^c})^{-1}\mathbf{y^c}, \\ &K(\mathbf{x^t}, \mathbf{x^t}) - K(\mathbf{x^t}, \mathbf{x^c})K(\mathbf{x^c}, \mathbf{x^c})^{-1}K(\mathbf{x^c}, \mathbf{x^t})\big) \end{aligned} \tag{2.30}$$

The derivation of equation 2.30 goes beyond the scope of this introduction. For a more detailed explanation see [Rasmussen and Williams, 2004].

From equation 2.30 we can see that computing posterior predictions for $\mathbf{y}^t$ only requires us to calculate the mean function

$$\mu^t = K(\mathbf{x^t}, \mathbf{x^c})K(\mathbf{x^c}, \mathbf{x^c})^{-1}\mathbf{y^c} \tag{2.31}$$

and the covariance function

$$\Sigma^t = K(\mathbf{x^t}, \mathbf{x^t}) - K(\mathbf{x^t}, \mathbf{x^c})K(\mathbf{x^c}, \mathbf{x^c})^{-1}K(\mathbf{x^c}, \mathbf{x^t}). \tag{2.32}$$

Calculating these two involves computing all the kernels in equation 2.29, which is straightforward once we have defined our kernel. Note that it is necessary to invert the kernel matrix $K(\mathbf{x}^c, \mathbf{x}^c)$ which is the reason behind the high computational cost of GPs that notoriously scale with $\mathcal{O}(n^3)$ where n is the number of context points.

The resulting mean and covariance functions for the example introduced in Figure 2.8a are shown in Figures 2.8b and 2.8c respectively. Alternatively, we can also sample entire functions conditioned on the observations, as shown in figure 2.9b.

While GPs are only used as baselines in this thesis, a large part of the motivation behind neural processes (Chapters 5 and 6) is to recreate some of the properties of GPs with neural networks, which is why we cover them in this technical background review as well.

2.5 Reinforcement learning

2.5.1 The Markov Decision Process Framework

Markov decision processes (MDPs) are the most commonly used framework to describe the sequential decision making problem encountered in Reinforcement Learning (RL) [Sutton et al., 1998]. In this setup, the learning problem is framed as follows:

Consider an agent that acts in an environment over several discrete time steps τ. At each time step the agent finds itself in a state $s_\tau \in \mathcal{S}$ of the environment and has to carry out one of the actions $a_\tau \in \mathcal{A}(s)$ associated with that state. As a result of its action a_τ the agent's state will be updated to state $s_{\tau+1}$ and it will receive a reward r_τ that is associated with carrying out action a_τ in state s_τ. This reward is captured by the reward function $p(r_\tau|a_\tau, s_\tau)$. In addition the environment defines the state-transition probabilities $p(s_{\tau+1}|a_\tau, s_\tau)$ that capture the probability of ending up in a certain state $s_{\tau+1}$ when taking action a_τ in state s_τ.

The goal of an agent is to learn some behaviour captured by a policy π that maximises the expected reward in the long run. This is captured by the expected return:

$$R_\tau = r_{\tau+1} + r_{\tau+2} + ... + r_T \tag{2.33}$$

where T corresponds to the final time step. For cases when the number of time steps is infinite it makes sense to think about the discounted return:

$$R_\tau = r_{\tau+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.34}$$

where γ is a value between 0 and 1 called the discount factor. The discount factor not only prevents the expected return from diverging, but also expresses the fact that the agent cares more about rewards in the near future than in the distant future.

The policy $\pi = p(a_\tau|s_\tau)$ itself describes a probability distribution over the possible actions $\mathcal{A}(s_\tau)$ conditional on the current state s_τ. The optimal policy π^* is the policy whose expected return is greater or equal to that of any other policy π_i for all states $s_\tau \in \mathcal{S}$. The goal of RL is to train agents that find this optimal policy π^* for a given environment.

State-Value Function (v-function)

We define $v_\pi(s_\tau)$ to be the expected return for being in state s_τ and acting according to policy π.

$$v_\pi(s_\tau) = \mathbb{E}_\pi[R_\tau|s_\tau] = \mathbb{E}\Big[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \Big| s_\tau\Big], \ \forall s_\tau \in \mathcal{S} \tag{2.35}$$

An important property of the value function is its recursive nature:

$$\begin{aligned} v_\pi(s_\tau) &= \mathbb{E}_\pi[R_\tau|s_\tau] \\ &= \mathbb{E}\Big[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \Big| s_\tau\Big] \\ &= \sum_{a_\tau} \pi(a_\tau|s_\tau) \sum_{s_{\tau+1}} \sum_{r_\tau} p(s_{\tau+1}, r_\tau|s_\tau, a_\tau)\Big[r_\tau + \gamma \mathbb{E}_\pi[R_{\tau+1}|s_{\tau+1}]\Big] \\ &= \sum_{a_\tau} \pi(a_\tau|s_\tau) \sum_{s_{\tau+1}, r_\tau} p(s_{\tau+1}, r_\tau|s_\tau, a_\tau)\Big[r_\tau + \gamma v_\pi(s_{\tau+1})\Big], \ \forall s_\tau \in \mathcal{S} \end{aligned} \tag{2.36}$$

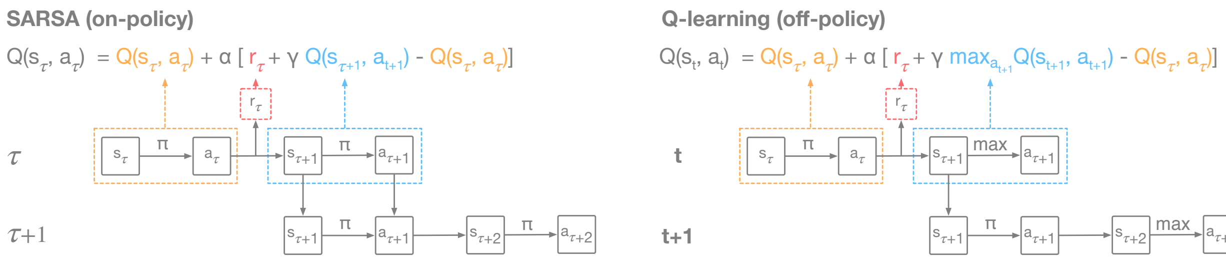

SARSA (on-policy)
$Q(s_\tau, a_\tau) = Q(s_\tau, a_\tau) + \alpha [r_\tau + \gamma Q(s_{\tau+1}, a_{t+1}) - Q(s_\tau, a_\tau)]$
τ
$\tau+1$
Q-learning (off-policy)
$Q(s_t, a_t) = Q(s_\tau, a_\tau) + \alpha [r_\tau + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_\tau, a_\tau)]$
t
t+1
max
π

Figure 2.10: Difference in the value function update between the on-policy and off-policy TD methods SARSA and Q-learning. The crucial difference is the term in blue, which corresponds to the update applied to the q-value and depends on the policy in the case of SARSA but is independent of it for Q-learning.

This equation, which captures the relation between the value of a state s_τ and the value of its following state $s_{\tau+1}$, is referred to as the Bellman equation for v_π [Sutton et al., 1998].

Action-Value Function (q-function)

The q-function $q_\pi(s_\tau, a_\tau)$ is defined as the expected return for being in state s_τ, carrying out action a_τ and acting according to policy π from there on.

$$q_\pi(s_\tau, a_\tau) = \mathbb{E}_\pi[R_\tau | s_\tau, a_\tau] = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_\tau, a_\tau\right], \ \forall s_\tau \in \mathcal{S}, a_\tau \in \mathcal{A}(s) \tag{2.37}$$

As with v_π the recursive definition of q_π can expressed by the Bellman equation for q_π.

2.5.2 Approximating the Value Functions

There are various different approaches to approximating value functions:

- **Dynamic programming**: Dynamic programming (DP) makes use of the Bellmann equation described in Equation 2.36 to iteratively compute an estimate of the value function. This method, however, requires the environment dynamics to be known, which limits its applicability.
- **Monte Carlo methods**: Monte Carlo (MC) methods don't require the model dynamics to be known in advance. Instead they rely on learning an approximation of the value function by averaging over trajectories from the agent's experience directly.
- **Temporal Difference Methods**: Temporal difference (TD) methods combine aspects of the two previous approaches. Like MC they don't require the environment model to be known and they use experience to update their estimate of the value funtion. In addition, like dynamic programming, TD

takes advantage of the recursive nature of the value function and uses estimations from previous updates to bootstrap the current value approximation.

On and off-policy learning

An important distinction that can be made for RL algorithms is whether they are trained 'on-policy' or 'off-policy'. The difference here is whether the policy that is used to explore is the same as the one that is being trained (on-policy) or not (off-policy). This difference is visualized in Figure 2.10 on the most common on-policy and off-policy TD algorithms: SARSA and Q-learning.

2.5.3 Policy gradient Methods

Instead of learning the value function and using it to find the best policy one can also learn a parametrised policy directly. This approach is called Policy Gradients (PG) and there are a few reasons why one might want to choose these methods over value function-based ones. For certain tasks, for example, the function describing the policy might be simpler to approximate than the value function. In addition, PG methods are able to approach both stochastic and deterministic policies, which is not the case for value-function methods.

In PG the performance of an agent is measured as the expected reward following policy π parametrised by θ:

$$J(\theta, \phi) = \mathbb{E}_{\pi_\theta(a_\tau|s_\tau)}[r_\phi(a_\tau)] \tag{2.38}$$

To be able to clarify an important point later on we will assume now that our reward function r_ϕ is parametrised by ϕ but this need not be the case in general. The update rule for the policy parameters θ is given by:

$$\theta_{\tau+1} = \theta_\tau + \alpha \nabla_\theta J(\theta) \tag{2.39}$$

where α is the learning rate. Unfortunately, the gradient $\nabla_{\theta,\phi} J(\theta)$ can usually not be computed in closed form and cannot be approximated using Monte Carlo methods. To demonstrate why this is the case we will follow the example in [Maddison et al., 2016] and compare the gradients of the computation graph for the expected reward. In this particular computation graph a_τ is sampled from the probability distribution $\pi_\theta(a_\tau|s_\tau)$ and fed to the reward function $r_\phi(a_\tau)$. Because both $\pi_\theta(a_\tau|s_\tau)$ and $r_\phi(a_\tau)$ may be stochastic we refer to this as a stochastic computation graph. We can now write out the gradients of this stochastic computation graph with respect to both parameters θ and ϕ. $\nabla_\phi J(\theta, \phi)$ can be rewritten as:

$$
\begin{aligned}
\nabla_\phi J(\theta,\phi) &= \nabla_\phi \mathbb{E}_{\pi_\theta(a_\tau|s_\tau)}[r_\phi(a_\tau)] \\
&= \nabla_\phi \int \pi_\theta(a_\tau|s_\tau) r_\phi(a_\tau) \mathrm{d}a_\tau \\
&= \int \pi_\theta(a_\tau|s_\tau) \nabla_\phi r_\phi(a_\tau) \mathrm{d}a_\tau \\
&= \mathbb{E}_{\pi_\theta(a_\tau|s_\tau)}[\nabla_\phi r_\phi(a_\tau)]
\end{aligned}
\tag{2.40}
$$

The gradient of the expectation with respect to ϕ is still an expectation and can thus be approximated with Monte Carlo methods. However this is not the case for $\nabla_\theta J(\theta,\phi)$:

$$
\begin{aligned}
\nabla_\theta J(\theta,\phi) &= \nabla_\theta \mathbb{E}_{\pi_\theta(a_\tau|s_\tau)}[r_\phi(a_\tau)] \\
&= \nabla_\phi \int r_\phi(a_\tau) \pi_\theta(a_\tau|s_\tau) \mathrm{d}a_\tau \\
&= \int r_\phi(a_\tau) \nabla_\theta \pi_\theta(a_\tau|s_\tau) \mathrm{d}a_\tau
\end{aligned}
\tag{2.41}
$$

Because the gradient is calculated for the term we are taking the expectation over, the overall expression is not an expectation itself and thus it is not possible to approximate it using Monte Carlo methods. There are two alternatives that overcome this issue: REINFORCE and the reparametrisation trick.

REINFORCE

REINFORCE [Williams, 1992] (also referred to as the score function estimator [Fu, 2006] or likelihood-ratio estimator [Glynn, 1990]) applies the identity $\nabla_\theta p_\theta(x) = p_\theta(x) \nabla_\theta \log p_\theta(x)$ to rewrite $\nabla_\theta J(\theta,\phi)$ as:

$$
\begin{aligned}
\nabla_\theta J(\theta,\phi) &= \nabla_\theta \mathbb{E}_{\pi\theta(a_\tau|s_\tau)}[r_\phi(a_\tau)] \\
&= \nabla_\theta \int \pi\theta(a_\tau|s_\tau) r_\phi(a_\tau) \mathrm{d}a_\tau \\
&= \int \nabla_\theta \pi\theta(a_\tau|s_\tau) r_\phi(a_\tau) \mathrm{d}a_\tau \\
&= \int \frac{\pi\theta(a_\tau|s_\tau)}{\pi\theta(a_\tau|s_\tau)} r_\phi(a_\tau) \nabla_\theta \pi\theta(a_\tau|s_\tau) \mathrm{d}a_\tau \\
&= \int \pi\theta(a_\tau|s_\tau) r_\phi(a_\tau) \nabla_\theta \log \pi\theta(a_\tau|s_\tau) \mathrm{d}a_\tau \\
&= \mathbb{E}_{\pi\theta(a_\tau|s_\tau)}[r_\phi(a_\tau) \nabla_\theta \log \pi\theta(a_\tau|s_\tau)]
\end{aligned}
\tag{2.42}
$$

This expectation can now be approximated using Monte Carlo. Note that only $\pi(a_\tau|s_\tau)$, and not r_ϕ, needs to be differentiable. Beyond that $r(a_\tau)$ can be a continuous or discrete random variable.

Reparametrization trick

A second option is to apply the reparametrization trick and express a stochastic variable x as a function of an independent noise variable z and calculate the gradient as:

$$\nabla_\theta \mathbb{E}_{\pi\theta(a_\tau|s_\tau)}[r(a_\tau)] = \mathbb{E}_{\pi\theta(a_\tau|s_\tau)}[\nabla_\theta r(a_\tau)] \tag{2.43}$$

Actor-critic algorithms

A common practice is to combine policy gradient with value function estimation in what is referred to as actor-critic methods. This type of model consists of a learned policy that corresponds to the 'actor' and a learned value function the 'critic'. During training the value function estimated by the critic is used to update the policy of the actor. Compared to regular PG methods actor-critic algorihtms have been shown to reduce the variance of the gradient estimators.

2.5.4 Common Deep Reinforcement Learning Agents

In recent years most RL agents have been implemented using deep networks and successes like [Mnih et al., 2013, Silver et al., 2016] established deep reinforcement learning as a core research field within machine learning. In the following we provide a high level description of the most common DRL agents:

- **Deep-Q-Networks - DQN** [Mnih et al., 2013]: This agent is a neural implementation of the q-learning algorithm described in 2.5.2. It also includes experience replay, which consists of an experience buffer that stores previous experiences and allows the agent to train on random steps taken from it.
- **Asynchronous advantage actor-critic - A3C** [Mnih et al., 2016]: As its name indicates this agent works by running on several parallel workers (asynchronous) and is implemented as an actor-critic agent where the critic approximates the advantage $(a_\pi(s,a) = q_\pi(s,a) - v_\pi(s))$ rather than the state-value function $q(s,a)$.
- **Deep Deterministic Policy Gradients - DDPG** [Lillicrap et al., 2015]: an off-policy, actor-critic agent that makes use of a stochastic policy for exploration but uses deterministic policy gradients to update the model. This agent works in continous action space.
- **Stochastic Value Gradients - SVG** [Heess et al., 2015]: SVG applies the pathwise derivative estimator to backpropagate through several time steps in the stochastic computation graph. In order to do this it requires a differentiable model of the environment.

- **Proximal policy optmisation - PPO** [Schulman et al., 2017]: PPO addresses the convergence issues of policy gradient methods by limiting the amount that a policy can be modified at each step. This is achieved with an additional regularization term consisting of the KL between the policy before and after the update. When the amount of regularization provided by the KL is adaptive the agent is referred to as Trust Region Policy Optimisation (TRPO).

For some of the chapters the ultimate goal is to apply the learned representations to RL. In particular we will train a tabular Q-learning agent on the symbolic representations obtained in Chapter 3 and use DQN as a baseline. In addition we will train an A3C agent on top of the representations obtained from generative query networks in Chapter 4.

Chapter 3

Symbols and Deep Reinforcement Learning

3.1 Introduction

Over the past decade deep learning (DL) has established itself as the most prolific area within machine learning and Artificial Intelligence (AI) research. The increase in computational power along with the availability of progressively larger datasets has enabled the development of powerful deep neural models that can be trained from data directly and with minimal handcrafting. Milestones such as reaching superhuman performance on image classification [Krizhevsky et al., 2012] and complex games [Mnih et al., 2013, Silver et al., 2017] or realistic image generation [Brock et al., 2018] have consolidated deep learning algorithms as fundamental tools of AI research.

However, despite their success current deep learning models still exhibit some critical flaws that result in a number of undesirable failure cases [Rocktäschel et al., 2015, Heaven, 2019, Evtimov et al., 2017]. Some of the most notable downsides include:

1. **Low data efficiency**: Deep networks can model complex functions thanks to the large number of parameters that are tuned during training. Unfortunately, training such a large number of parameters via gradient descent requires large amounts of data, with current algorithms often training on millions of examples.
2. **Weak generalisation**: Good performance on the training set does not guarantee good performance on the testing set. This lack of generalisation is particularly striking at a concept level as deep networks

tend to focus on low-level features (e.g. changing visual properties of the same image can decimate a network's performance). Humans, on the other hand, are very robust to this type of perturbations [Geirhos et al., 2018].

3. **Uninterpretable representations**: Even if deep learning algorithms learn to carry out a task successfully the intermediate representations produced by the networks are usually large tensors that are the result from non-obvious interactions between the input and a large set of learned weights. As such, these representations are not human readable and it is not straight forward to understand why a network produced a certain output. While understanding this is not always necessary, some applications like medical recommendations or legal decisions usually require some transparency that deep neural networks can not provide.

An alternative paradigm within Artificial Intelligence (AI) is symbolic AI, where algorithms reason about symbols and their relations with language-like propositions. Symbolic methods do not suffer from many of the drawbacks mentioned above: given the abstract nature of symbols and predicates, symbolic methods are good at higher-level generalisation, for example. In addition, because the symbols are handcrafted by humans, the outputs produced by these algorithms are easily interpretable. This handcrafting, however, is also the main reason symbolic methods are currently not as popular as deep learning approaches: rather than grounding the features in data directly, symbolic methods require a large amount of human intervention in order to define the symbols and predicates.

Inspired by symbolic methods we propose two desiderata for deep learning algorithms.

Abstraction We argue that reasoning at concept-level abstraction is a key component for data-efficient learning and good for generalisation and transfer learning. As such we would like deep learning algorithms to generate representations at different levels of abstraction from the data directly.

Common sense priors There are a number of fixed common sense priors that govern most of our everyday lives (e.g. we live in a world that contains objects which, for the most part, persist over time and generally move on continuous trajectories), which are helpful to carry out predictions and generalise to unseen situations. Ideally these priors should be learned by an intelligent system from data or interactions with the real world directly. So far it has proven difficult to train such an unconstrained system to learn these priors, so alternatively we argue for building in constraints (through the choice of architecture or training regime, for example) which embody these priors and which facilitate learning.

With these desiderata in mind we propose a proof-of-concept model which constitutes a first step towards integrating deep learning and symbolic approaches. By defining an architecture that extracts individual objects and focuses on their relations we can explicitly reason at a higher level of abstraction than regular

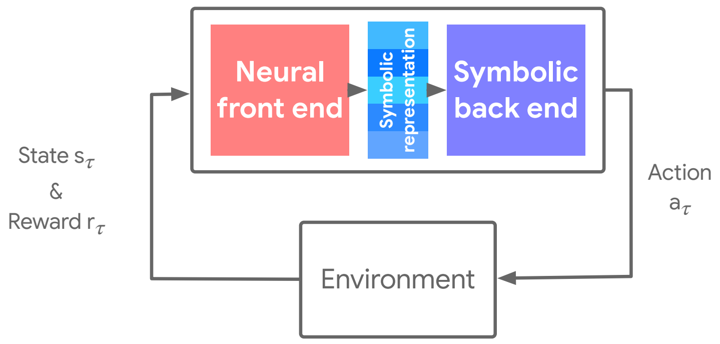


Figure 3.1: Our suggested neuro-symbolic architecture. The agent consists of a neural front end that extracts features from raw observations. These features are processed into a compositionally structured symbolic representation that serves as input to a reinforcement learning agent which maps these symbols to an action.

deep learning approaches. In addition, by focusing on salient regions of the image and their relations, as well as comparing symbols across time steps, we build in the concept of objects and relations as well as object persistence. We apply our model to a small 2D game and show that this type of algorithm outperforms current Q-learning based reinforcement learning algorithms when it comes to data efficiency and generalising to unseen situations and in addition it produces interpretable intermediate representations.

The rest of the chapter is structured as follows:

- In Section 3.2 we introduce the environment and the different conditions that we use for our experiments.
- We describe the different parts of our model as well as the training procedure in Section 3.3.
- After introducing our model we cover related research in Section 3.5.
- The results of the experiments are described in Section 3.4.
- Finally in Section 7 we conclude the chapter with a brief discussion.

3.2 The environment

As a benchmark for our prototype system, we implemented several variants of a simple game where the agent (shaped as a '+') has to learn either to avoid or to collect objects depending on their shape. Once the agent reaches an object using one of four possible move actions (up, down, left, or right), this object disappears and the agent obtains either a positive or a negative reward. Encountering a circle ('o') results in a negative reward while collecting a cross ('x') yields a positive reward.

We applied the system to four variants of this game (Figure 3.2) wherein the type of objects involved and their initial positions vary as follows:

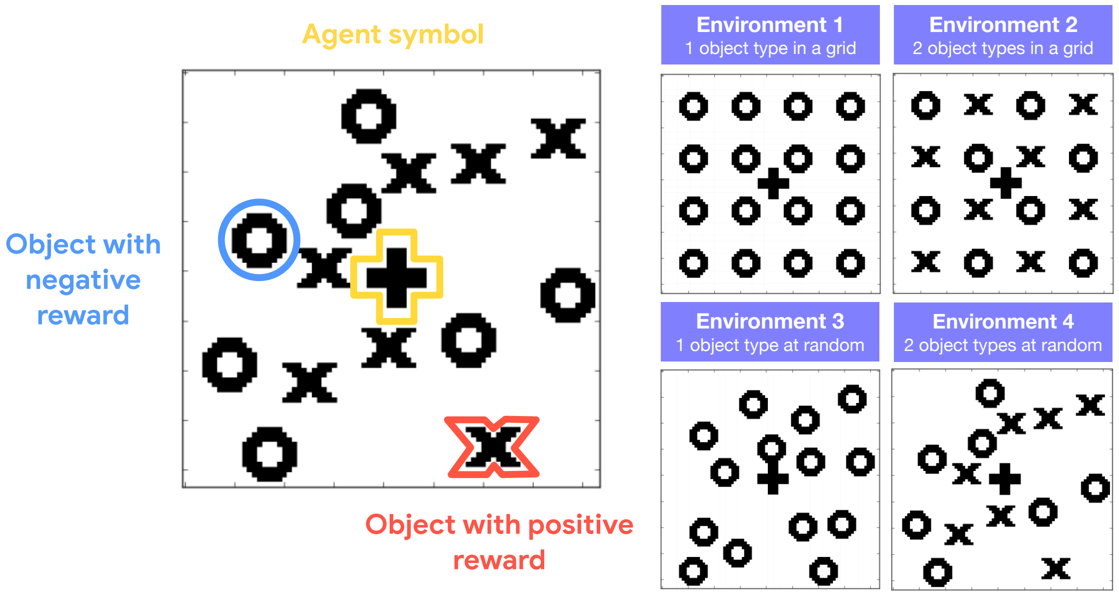


Figure 3.2: The four different game environments. The agent is represented by the '+' symbol. The static objects return positive or negative reward depending on their shape ('x' and 'o' respectively). The images produced by the environment are 84 x 84 pixels and the individual objects are up to 12 pixels in diameter.

Variant 1. In this environment there are 16 objects that all have the same shape ('o') and all return a negative reward of -1. The layout is the same for every new game: the 16 objects are positioned on a 4x4 grid and at the beginning of the game the player is located in the middle of the board.

Variant 2. The layout is the same as in version 1 but there are two types of objects. As before, collecting a circle results in a reward of -1 points and collecting a cross returns a reward of 1.

Variant 3. As in version, 1 this game only contains one type of object that returns a negative reward. In order to increase the difficulty of the learning problem however, the position of the objects is set at random at the beginning of each new episode. We lower the number to 14 objects to reduce the probability of objects overlapping.

Variant 4. This version combines the randomness in the position of the 14 objects from environment 3 with the two different object types from version 2.

3.3 Methods

Our algorithm can be thought of as a pipeline comprising three stages: low-level symbol generation, representation building, and reinforcement learning (see Figure 3.1).

3.3.1 Low-level symbol generation

The goal of this first stage is to generate, in an unsupervised manner, a set of symbols that can be used to represent the objects in a scene. We train a convolutional autoencoder on 5000 randomly generated images of varying numbers of game objects scattered across the screen. Given the simplicity of the images, which consist of objects with three different geometric shapes (cross, plus sign, and circle) on a uniform background, we don't need to train a very deep network to detect the different features for our current game benchmark. Our network consists of two 5x5 convolutional layers with 32 channels each followed by a 2x2 max pooling layer plus the corresponding decoding layers. The activations across features in the middle layer of the CNN are used directly for the detection of the objects in the scene.

Object detection and characterisation. The next step of the symbol generation stage uses the salient regions of the convoluted image (Figure 3.3). As shown by Li et al [Li et al., 2016], salient areas in an image will result in higher activations throughout the layers of a convolutional network. Given that our games are geometrically very simple, this property is enough to enable the extraction of the individual objects from any given frame. To do this, we first select, for each pixel of the intermediate representation, the feature with the highest activation. We then threshold these activation values, forming a list of those that are sufficiently salient. Ideally, each member of this list is a representative pixel for a single object. The objects identified this way are then assigned a symbolic type according to the geometric properties computed by the autoencoder. This is done by comparing the activation spectra of the salient pixels across features. In the current implementation, this comparison is carried out using the sum of the squared distances, which involves setting an ad hoc threshold for the maximal distance between two objects of the same type. More recent approaches to object extraction are able to successfully extract object level representation from more complex environments using models that are trained end-to-end and do not require such thresholding [Burgess et al., 2019].

The information extracted at this stage consists of a symbolic representation of the positions of salient objects in the frame along with their types. (See the left-hand table of Figure 3.4.)

3.3.2 Representation building

Once we have extracted the low-level symbols from a single snapshot of the game we need to be able to track them across frames in order to observe and learn from their dynamics. To do this, we need to build in the first common sense prior: object persistence across time. The concept of persistence we deploy is based on three measures combined into a single value. This value, which captures how likely an object Θ_i^τ identified in one frame is the same as object $\Theta_i^{\tau+1}$ identified in the next frame, is a function of three components: spatial proximity, type transitions and neighbourhood.

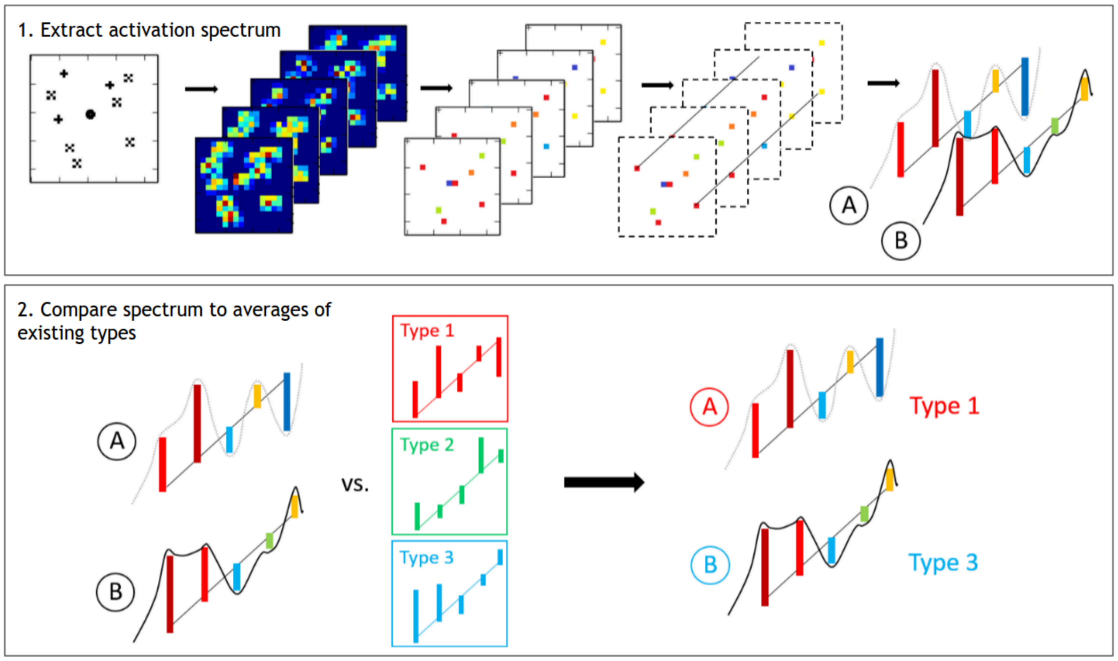

1. Extract activation spectrum
A
B
2. Compare spectrum to averages of existing types
A
B
vs.
Type 1
Type 2
Type 3
A
Type 1
B
Type 3

Figure 3.3: Unsupervised extraction of low-level symbols from the information provided by the convolutional autoencoder. The spectrum of activations across features is obtained by selecting the pixels with the highest activations across features. In a second step each symbol is assigned one of the existing types by comparing their spectra. If no match is found a new type is created. Figure taken from [Garnelo et al., 2016]

Spatial proximity. We build in the notion of continuity by defining the likelihood that an object in frame τ is the same as another object in frame $\tau + 1$ to be inversely proportional to the distance between the two objects in consecutive frames. We have

$$L_{dist}(d) = \frac{1}{1 + d}$$

where d is the Euclidean distance between two objects Θ_i^τ and $\Theta_j^{\tau+1}$ in consecutive frames τ and $\tau + 1$ respectively. Distance, however, is not the sole determiner of an object's identity, given that objects can replace each other.

Type transitions. Because we don't constrain the the types generated by the filters of the convolutional neural network to have any particular form, the type of the same object might change between frames (the same object will produce different features depending on its location within the convolution, for example). As long as these transitions are consistent throughout training and the same for all objects of a certain shape this behaviour is fine as long as we take the type changes into account when tracking the individual symbols. Given the types of two objects $type(\Theta_i^\tau)$ and $type(\Theta_j^{\tau+1})$ in consecutive frames, we can determine the probability that they are the same object that has changed from one type to the other by learning a transition matrix $\mathcal{T}$ from previously observed frames. Given that this is a transition matrix it is already normalised to be a probability.

$$L_{trans} = \mathcal{T}_{type(\Theta_i^\tau) \to type(\Theta_j^{\tau+1})}$$

In order to be able to describe all transitions, including the ones that correspond to objects appearing and disappearing, we introduce the object type 0. This type corresponds to 'non-existent'. An object of type 1 that appears in frame t has thus carried out the transition $0 \to 1$. If that object disappears later on it would transition $1 \to 0$. This addition can only be carried out after the tracking step and once every object has been assigned its corresponding type in the previous time steps.

Neighbourhood. The neighbourhood of an object will typically be similar from one frame to the next. This allows us to discriminate between objects that are spatially close but approaching from different directions. For now we just consider the difference in the number of neighbours, ΔN between two objects, where we define a neighbour to be any object, Θ_i, within a distance d_{max} of another object Θ_j. Future improvements could include considering the types of neighbours rather than just the number.

$$L_{neigh}(\Delta N) = \frac{1}{1 + \Delta N}$$

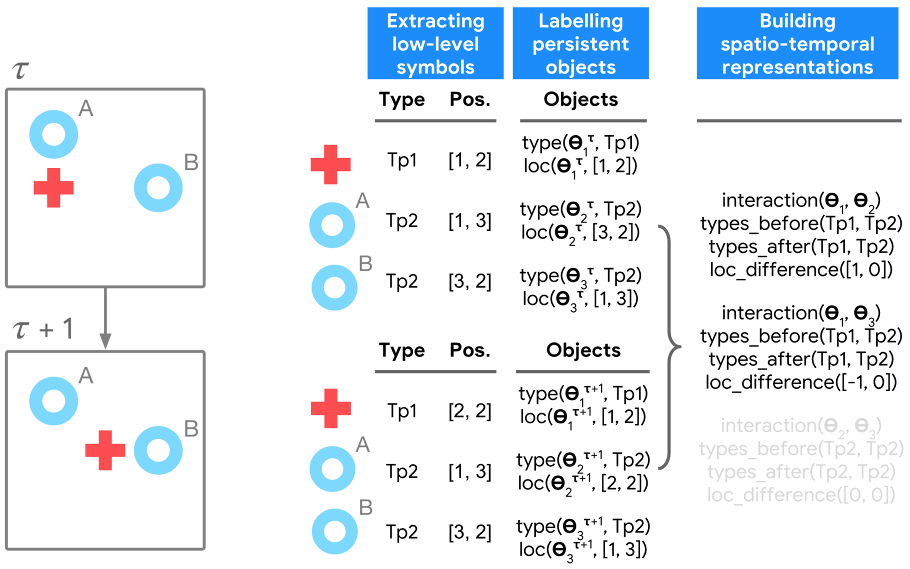
We track an object by combining the three measures just described. We have

$$L = w_1 L_{dist} + w_2 L_{trans} + w_3 L_{neigh} \tag{3.1}$$

where $w_1, ..., w_3$ are ad hoc weights. In future implementations these weights could be learned and set dynamically as the importance of each term varies during the learning process.

As a result of the tracking process, we acquire an additional attribute for each object in the scene: a unique identifier that labels it across time. This identifier is added to the symbolic representation of the ongoing game state. (See the middle table of Figure 3.4.)

Symbolic interactions and dynamics. So far, information about the game is expressed in terms of static frame-by-frame representations (albeit taking advantage of information in consecutive frames). But the final, reinforcement learning stage of the algorithm will require information about the dynamics of objects and their spatial interactions. This is obtained from the static representations constructed so far using two principles. First, we consider the difference between frames rather than working with single frames, thus moving to a temporally extended representation. Second, we represent the positions of objects relative to other objects rather than using absolute coordinates.



Extracting low-level symbols
Labelling persistent objects
Building spatio-temporal representations
τ
τ + 1
A
B
Type
Pos.
Objects
Tp1 [1, 2]
Tp2 [1, 3]
Tp2 [3, 2]
type(Θ1τ, Tp1) loc(Θ1τ, [1, 2])
type(Θ2τ, Tp2) loc(Θ2τ, [3, 2])
type(Θ3τ, Tp2) loc(Θ3τ, [1, 3])
Tp1 [2, 2]
Tp2 [1, 3]
Tp2 [3, 2]
type(Θ1τ+1, Tp1) loc(Θ1τ+1, [1, 2])
type(Θ2τ+1, Tp2) loc(Θ2τ+1, [2, 2])
type(Θ3τ+1, Tp2) loc(Θ3τ+1, [1, 3])
interaction(Θ1, Θ2) types_before(Tp1, Tp2) types_after(Tp1, Tp2) loc_difference([1, 0])
interaction(Θ1, Θ3) types_before(Tp1, Tp2) types_after(Tp1, Tp2) loc_difference([-1, 0])
interaction(Θ2, Θ3) types_before(Tp2, Tp2) types_after(Tp2, Tp2) loc_difference([0, 0])

Figure 3.4: Overview of the symbolic representations extracted after each of the three main steps. The images on the left show two consecutive frames of a toy example where the cross symbol moves one step to the right. The first column corresponds to the information obtained from these frames after low-level symbol extraction from the convolutional autoencoder. After tracking, a unique label is assigned to each persistent object (Θ_1, Θ_2 or Θ_3) and its associated type and position is recorded. The last column on the right lists the interactions between objects which are used for the reinforcement learning stage. The information is now expressed in terms of relative distance rather than absolute position and corresponds to the difference between two frames. Relations between objects that don't change (like the third element of the rightmost column) are not considered. Figure adapted from [Garnelo et al., 2016].

We now have a concise spatio-temporal representation of the game situation, one that captures not only what objects are in the scene along with their locations and types, but also what they are doing. In particular, it represents frame-to-frame interactions between objects, and the changes in type and relative position that result (see the right-hand table in Figure 3.4). This is the input to reinforcement learning, the third and final stage of the pipeline.

In practise we only record relative positions of objects that lie within a certain maximum distance of each other in order to reduce computational cost. In doing so we are building in the assumption that local relations between objects are more relevant than global ones. Of course, this assumption is not always valid, even in a simple game like ours, so the interactions captured by this system will only be an approximation of the real dynamics. We discuss further discuss the effect of this approximation in the following section.

3.3.3 Reinforcement learning

The spatio-temporal representation constructed in stages one and two of the system pipeline can now be used to learn an effective policy for game play. Since most interactions between objects are independent of each other in our environment we will focus on individual pairwise interactions in order to reduce the size of the state-space. As such, instead of defining a single global state space which contains all pairwise interaction we will define an independent state space for every interaction between two object types and train separate Q functions for each. This independence assumption allows for faster learning and enables generalisation across object types but, again, only yields an approximation of the real value function. It is worth stressing that the fact that we can decompose this Q function in this way is a result of the simplicity of our environments, where only the player moves and the relative changes in position are always only between player and objects. For more complex environments this decomposition would be too simplistic and not scale well. We discuss the effects of approximating the global value function with multiple local ones later on in this section.

More concretely, we train a separate Q function for each interaction between two object types. The main idea is to learn several Q functions for the different interactions and query those that are relevant for the current situation. Given the simplicity of the game and the reduced state space that results from the sparse symbolic representation we can approximate the optimal policy using tabular Q-learning. The update rule for the interaction between objects of types i and j is therefore

$$Q^{ij}(s^{ij}_\tau, a_\tau) \leftarrow Q^{ij}(s^{ij}_\tau, a_\tau) + \alpha \left(r_{t+1} + \gamma(\max_a Q^{ij}(s^{ij}_{t+1}, a) - Q^{ij}(s^{ij}_\tau, a_\tau)) \right) \tag{3.2}$$

where α is the learning rate, γ is the temporal discount factor, and each state s^{ij}_τ represents an interaction between object types i and j at time step t. In this case the interactions are changes in relative distance between the objects in question. The state space S^{ij} thus describes the different possible relations between two objects of types i and j. Given that we have limited the interactions to a certain radius of proximity this state space is bounded but learning is not guaranteed to converge on a global optimum. Finally, in order to choose the next action we add up all values obtained from the Q functions of interactions that are present at the current time step and pick the action that will return the highest reward overall.

$$a_{t+1} = \arg\max_a (\sum_Q (Q(s_{t+1}, a))$$

During training we use an ϵ-greedy exploration strategy with a 10% chance of choosing an action at random.

Note that, in the present benchmark games, the only moving object is the one the agent acts on directly (the plus sign), which ensures that every currently relevant Q function pertains to a possible agent action. In a more general implementation it would be necessary to automatically identify which objects in the world the agent controls directly (its own body, for example) and restrict the Q function accordingly.

Approximating the global Q-function with multiple interaction-specific Q-functions

Choosing separate Q-functions instead of a single global one deviates from the established Q-learning learning regime. In the following we show that in most situations the behaviour obtained with our approximation doesn't deviate from the behaviour of an agent with a single Q-function and we address the exception. There are two questions that we need to address:

1. Does the sum over the local Q-functions correspond to the global Q-function?
2. How can we ensure the reward is assigned to the correct pairwise interaction?

In the following we cover both questions. For simplicity we assume perfect identification of the symbols by the neural front end, although the arguments hold for any representations generated by the encoder, as long as they are consistent.

One way to think about how the interaction-specific Q-functions relate to the global Q-function is to compare them for a simple environment and then consider increasingly complex setups.

- **Environments with a single object of one type**: In this case learning a type-specific Q-function corresponds to learning a global Q-function exactly.
- **Environments with several objects of one type**: In this case there is still only a single Q-function, which again corresponds to a global Q-function. The main difference to regular Q-learning is that in order to choose the next action the policy sums over the Q-values of all symbols in the scene rather

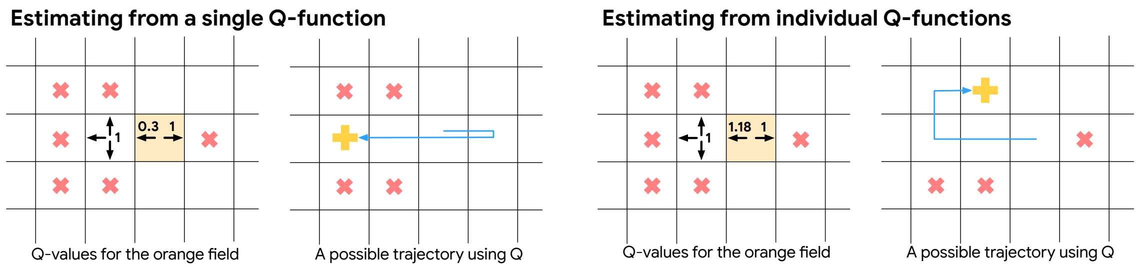

Estimating from a single Q-function
0.3
1
1
Q-values for the orange field
A possible trajectory using Q
Estimating from individual Q-functions
1.18
1
1
Q-values for the orange field
A possible trajectory using Q

Figure 3.5: Behaviour difference between agents that estimate a global Q-function versus the sum of symbol-specific Q-functions. While both Q-functions are likely to produce the same behaviour in most situations, there are cases like the one depicted above where the outcome of the policy will differ. This is a result of the latter up-weighing areas with more objects.

than just picking a single value for the global state. The sum of Q-functions and the global Q-function have the same values except where the symbol-specific Q-functions both have non-zero support. For those states the values of the different Q-functions will be added and will therefore be higher than in the regular global Q-function. While this makes a numerical difference, in practice the policy only takes the max, which is unaffected by this scaling in many cases.

There can be a few exceptions like the one depicted in Figure 3.5. As shown in the schematic for some particular symbol arrangements the effect of summing over several symbols can change the policy's behaviour. In particular, regular Q-learning tends to be more myopic and favour short-term rewards over multiple long-term ones. This is a result of only considering the action with maximum reward when bootstrapping the value. By taking the sum, our policy can consider several values and in cases where the agent can pick multiple rewards this might be a better suited approach since it weights the Q-value by the number of symbols. It's important to point out that while the Q-values are weighted by the number of symbols, in most of the cases this will not influence the policy's behaviour, as the discount factor will enforce a behaviour that is as short-sighted as the global Q-function.

- **Environments with a single object that can be of one of two types**: In this case different types means different rewards (+1 and -1 for our environment). Types with the same reward could be seen as a special case of the previous category. This environment learns separate Q-functions for separate symbols and since these symbols don't appear together in the same game, this is the same as learning two separate tasks with a global Q-function for each.

- **Environments with several objects that can be of one of two types**: A key insight here is the fact that the Q updates only use the max function. As a result the negative rewards will only affect the Q-functions for state and action pairs that directly lead to negative rewards. This is also the case for the symbol-specific Q-functions. As such, having an additional negative type of object leads to the same behaviour in both the global and the type-specific Q-functions.

Therefore the only difference between a global Q-function and our symbol-based Q-functions is the fact that in the case of multiple, neighbouring symbols summing over several Q-functions leads to a Q estimate that is weighted by the number of objects. However, it's worth noting, that while this weighted sum differs from the original Q-learning formulation, the cases where this actually affects behaviour are likely to be very rare, given that the discount quickly reduces the value of future rewards and that we are currently only comparing objects within a limited radius.

We now address the second question of how to ensure that the right value function is updated with the current reward. In our environment only one interaction at a time can lead to reward (whenever the agent picks up another object). In our formulation, however, we have a different Q-function update for every interaction in the game, even between objects that are not involved in the action that generated the reward. As such, the question arises of how to associate the reward with the Q-update that corresponds to the agent picking up the object that returned the reward. We overcome this issue by updating the Q-values of all the interactions at a given time point. The idea behind this is that in expectation the wrong updates will be cancelled out and only the true signal will be consistently reinforced.

Approximations and limitations

Beyond setting some hard-coded values (such as the threshold for the type clustering), which is common practice even for deep learning algorithms, we have simplified the task by making some assumptions and approximations. In the following we discuss the limitations that result from doing so.

An important assumption is the plainness of the environment. The ability of the encoder to extract objects relies on the visual simplicity of the game. In principle a simple encoder like ours could still work with complex environments but the level of abstraction would be greatly reduced. For visually rich environments with complex patterns and objects of different sizes it would be necessary to resort to more sophisticated object detection methods. Building deep unsupervised object detection algorithms, remains a challenge in machine learning, with some promising successes such as [Burgess et al., 2019, Engelcke et al., 2019, Greff et al., 2019]. Additionally, the fact that all objects are stationary greatly reduces the required amount of computation. While in theory our setup should work with moving objects as well, the number of relations that the model would be tracking would increase exponentially.

Approximating the global Q-function using several symbol-centric ones also deviates from the standard RL framework. While the behaviour obtained with multiple Q-functions is a good approximation there are cases where a policy that makes use of these approximations might choose different actions than the global policy. Note that this need not be a bad property of the model. By focusing only on the single object with the highest reward for each action, Q-learning is optimising for single immediate rewards, rather than several rewards from different sources over time. Since our system is able to model the different reward sources separately it has the ability to weigh these into its decision process.

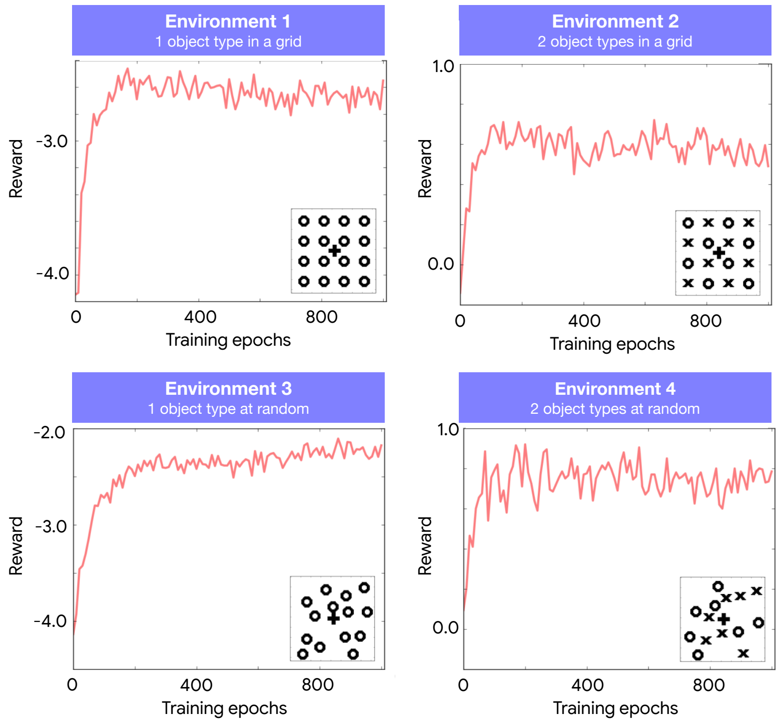


Figure 3.6: Average scores for the different game environments. Every training epoch consists of 100 steps and the reward is averaged over 20 repeats. Figure adapted from [Garnelo et al., 2016].

Finally, we have limited the interactions that we track to a certain distance between symbols. As mentioned above this was done for computational purposes and while it takes away the guarantee for a global optimum, in practice it works well enough for this proof of concept. Note that for an agent that is placed in the center the circumference of visibility will still cover over 34% of the game's surface.

3.4 Results

Agents were trained in epochs of 100 time steps for a maximum of 1000 epochs. We trained 20 agents separately and tested them for 200 time steps on 10 games at every tenth epoch. The resulting average score is plotted in Figure 3.6 for all four games. In all four cases the score increases within the first few hundred epochs and remains approximately constant for the remaining time.

While plotting the average score in this way is a common way of visualising successful learning for this type of experiment, this measure can produce an incomplete characterisation of the learning process in environments with both positive and negative objects. There are two reasons for this. First, given that the scores returned by the objects can cancel each other out, there is more than one way to achieve any given final score. For example, collecting ten positive and nine negative objects will result in the same final score as only collecting one positive object. Yet in the first case the number of positive objects the agent collects is about 53%

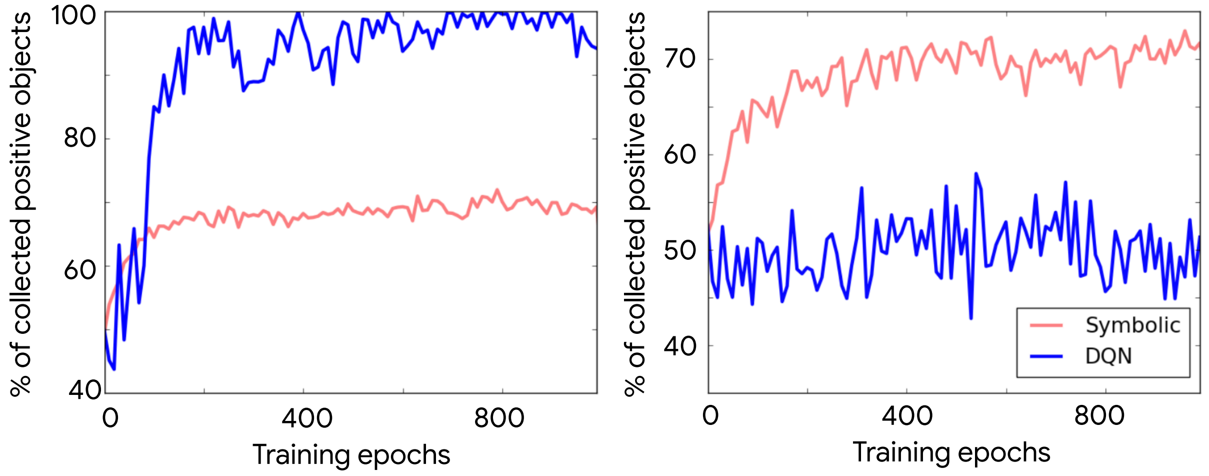


Figure 3.7: Comparison between DQN and our symbolic approach. We show the average percentage of objects collected over 200 games that return positive reward in the grid environment (left) and in the random environment (right). Figure adapted from [Garnelo et al., 2016].

while in the second scenario it is 100%. Therefore, although they both have the same score, this percentage measure reveals that the agent in the latter case has learned to react to the different objects correctly while the agent in the former case collects items without regard for their type. Second, our agent only has a limited radius of view and can therefore get stuck in a location surrounded by negative objects. In this case the agent is forced to spend most of the test time avoiding these negative objects and won't obtain a high score. For these reasons we introduce a second measure: the percentage of positive objects collected of the total number of objects encountered. Rather than measuring how well our agent performs on a global level, this measure shows whether or not the agent has learned to interact correctly with the individual objects at a local level.

The results for the two game variants that feature objects of two different types are shown in Figure 3.7. As expected, about 50% of the objects that the agent initially collects are positive. As training continues the percentage increases to approximately 70% in both cases.

Finally, we tested the transfer learning capabilities of our algorithm by training an agent only on games of the grid variant then testing it on games of the random variant. While this setup is similar to the experiments on the random game in the sense that the grid setup can be seen as one among the possible random initialisations, the difference lies in the fact that the agent is exposed to just this one type of environment during training, whereas in the random experiments the agent experiences numerous random variations. As shown in Figure 3.8, the learning curve is comparable to the random case, albeit showing a slightly inferior performance.

3.4.1 Comparison to DQN

Finally we compare our approach to DQN[1]. The environments that are suited the best are those with two types of objects as the initial score will be independent of the speed of the agent at the beginning given that they cancel each other out. Figure 3.7 shows the performance of DQN over time. It's important to note that our system's convolutional network was pre-trained on 5000 images, which corresponds to 50 epochs worth of frames. We don't include these in the plots because this pre-training is applicable to all the games and only has to be carried out once.

Thanks to the geometrical simplicity of the grid scenario, the DQN agent quickly learns to move down diagonally to collect only positive objects and avoid negative ones. As a result, the relative number of objects with positive reward for this game variant reaches 100% after only a few hundred epochs of training, while our agent can only achieve 70%. On the other hand, when the objects are positioned at random, the DQN agent is not able to learn an effective policy within 1000 epochs, with the number of positive items collected fluctuating around 50%. So on this game variant, our agent's performance is markedly better than DQN's.

This is also the case for our final experiment where the agent is trained on the grid variant and tested on the random variant (Figure 3.8). While our agent rapidly attains a percentage of approximately 70%, and then fluctuates around that value, DQN is again unable to do better than chance after 1000 epochs. Although we haven't run the experiment long enough to confirm this, we hypothesise that, while DQN might eventually learn to play the random game effectively when trained on the same game, it will never achieve competence at the random game when trained on the grid setup, as it will overfit on the latter.

3.5 Related work

The goal of this chapter is to implement an algorithm that combines ideas from symbolic learning with current deep learning methods. Despite both being prevalent areas of research the combination of the two is not as common. We will divide up the space of related research into work that is more closely related to the symbolic methods, which we will refer to as 'explicitly symbolic', and work from the deep learning literature that has been inspired by symbolic reasoning, which we call 'deep methods inspired by symbolic reasoning'.

3.5.1 Explicitly symbolic

Object oriented reinforcement learning Object oriented reinforcement learning [Liao and Poggio, 2017, Keramati et al., 2018] combines object-level representations of the environment with the classic re-

[1]For this comparison we used an open source implementation of DQN: https://github.com/Kaixhin/Atari

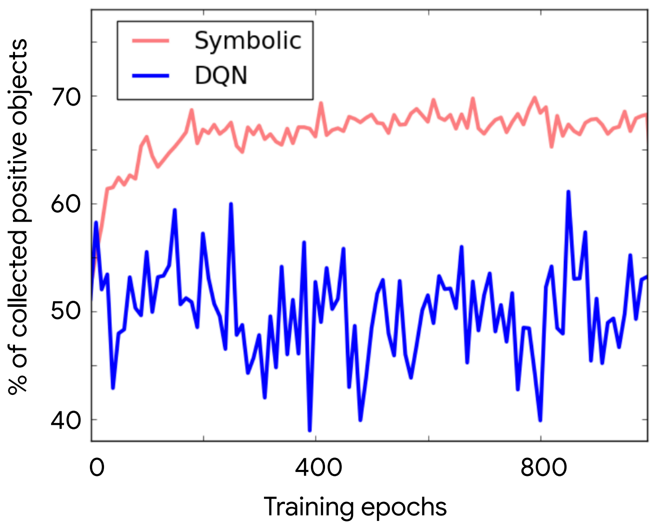


Figure 3.8: Average percentage of objects collected over 200 games that return positive reward by an agent that is trained on the grid environment and tested on random environments. Figure adapted from [Garnelo et al., 2016].

inforcement learning framework. Despite early successes, this type of methods still requires hand-crafted preprocessing of the input or even direct access to the ground truth object descriptions [Woof and Chen, 2018], which has limited their application. In contrast, out method picks up symbols automatically depending on their saliency in the image.

Follow-up work More directly related are follow-up papers which extend the work described in this chapter itself. Garcez et al improved the performance of our model by building in some 'common sense'-inspired constraints, such as weighting the Q-values by the distance of objects to the agent [Garcez et al., 2018]. In addition Dutra et al compare our model's performance on a simpler version of the task against DQN in order to further understand its generalisation abilities [Dutra and Garcez, 2017].

Theorem proving Mathematics Finally, a separate branch of research on deep networks which extract and handle symbolic representations from the data focuses on theorem proving [Cai et al., 2018, Irving et al., 2016, Rocktäschel and Riedel, 2017, Minervini et al., 2018] as well as logical reasoning [Serafini and Garcez, 2016, Garcez et al., 2008]. Recent work on combining deep networks with probabilistic logic programming is also related [Manhaeve et al., 2018]. In contrast to our work, this line of research does not focus directly on training RL agents on games and has more explicitly symbolic representations.

Common sense For this project we also mention the concept of injecting 'prior' or 'common sense' knowledge in the form of algorithmic constraints (such as focusing on object saliency and tracking objects over time to capture object persistence). This idea of common sense knowledge has also been considered in the context of machine comprehension [Chabierski et al., 2017]. In this case, however this common sense was learned from an explicit Question Answer dataset using Inductive Logic programming directly, rather than encoded as an inductive bias.

3.5.2 Deep methods inspired by symbolic reasoning

Interpretability A growing area of research in deep learning that shares some of its motivations with our line of work is research on interpretable neural network representations. The goal of work in this area is to train deep networks that successfully carry out a task (such as image generation) with the additional constraint that the network's representations or outputs need to be human-readable.

Probably the largest area of research within the field of interpretability is on learning models with disentangled latent representations. Although definitions vary, at a high level a representation generated by a deep neural network is said to be disentangled when different dimensions encode different higher-level attributes of the underlying data distribution. For a more formal definition of disentanglement see [Higgins et al., 2018a]. Disentangled representations can be induced by constraining the objective function directly [Chen et al., 2016, Higgins et al., 2016, Kim and Mnih, 2018, Siddharth et al., 2016] and representations obtained this way have been shown to be useful for downstream tasks [Higgins et al., 2017] as well as learning hierarchical concepts [Higgins et al., 2018b]. While our method learns individual representations for the different symbols our focus is not to learn features that have disentangled dimensions for the different properties of the objects in the environment.

A different approach to inducing interpretable representations is to build a renderer into the network [Wu et al., 2017, Ganin et al., 2018], thus forcing the network to produce human-readable representations that adhere to the language of the renderer. Finally, it is also possible to train a network to generate interpretable representations that capture the individual objects and their properties by rendering each object at a time or just encoding them separately [Eslami et al., 2016, Greff et al., 2016, Nash et al., 2017].

Beyond training models to generate interpretable representations one can also design a model that produces entire interpretable programs [Parisotto et al., 2016, Bhupatiraju et al., 2017, Reed and De Freitas, 2015, Tremblay et al., 2018], graph structured outputs [Li et al., 2018] and policies [Verma et al., 2018] for reinforcement learning.

Relations In addition to considering objects the model we introduce in this chapter also takes into consideration their relations. A similar approach is taken by relation networks [Santoro et al., 2017], which explicitly encode relational computations in a comparable way to our model. Relation networks have been shown to outperform regular feed forwad architectures on relational tasks such as the CLEVR dataset [Johnson et al., 2017] as well as reasoning tasks [Barrett et al., 2018]. Also related is a more recent line of work which focuses on relational learning on propositional representations extracted from pixel data directly [Shanahan et al., 2019].

Other areas In addition, larger research areas are related to some of the ideas discussed in this chapter. As we are learning separate Q functions for the same environment, our model could be described through the lens of multi agent reinforcement learning [Buşoniu et al., 2010] where each interaction between two types is an agent. Also related is work on value function factorisation [Rashid et al., 2018] that introduces a formal way of combining per-agent values from local observations into a joint action-value.

3.6 Discussion

In this chapter we have introduced a neuro-symbolic proof-of-concept architecture that addresses some of the drawbacks of current deep learning algorithms. Unlike most deep learning approaches our model extracts explicit symbols and focuses on their relations when generating representations for a downstream RL agent. As a result of working with these symbolic-like representations, the final agent is able to generalise to unseen environments at test time and to produce intermediate representations that are interpretable.

While the results obtained with our neuro-symbolic architecture are promising for future research on combining symbolic and deep methods, our implementation is still an early proof-of-concept architecture and thus has a number of challenges that could be addressed. First of all, the neural front end is comparably simple and, while it works on environments that are not too complex (i.e. flat background and simple shapes that are approximately the same size and which are, for the most part, stationary), it is likely not going to perform well on more intricate environments. In addition, we built in an explicit measure of how likely a symbol in one frame would correspond to a symbol in the next frame in order to track it across time. This approach reflects common-sense knowledge about the world such as object persistence and it would be interesting to see if this could be learned rather than handcrafted. Finally, we made some simplifying assumptions for our RL agent in order to make use of the symbolic structure of our input. While we show that such an agent is able to learn, this way of approximating the Q-function with several independent ones doesn't have the guarantees of regular RL algorithms. As an alternative one could consider different approaches such as relational reinforcement learning [Džeroski et al., 2001].

Designing neural architectures that produce representations with some particular, desirable properties, as we have done in this chapter, is one way of encouraging data-efficient learning. In the following chapter we take an alternative approach and consider pre-training parts of a network on tasks that will speed up the learning process on a second target task, thereby shifting the main challenge from handcrafting architectures to designing meaningful pre-training tasks.

Chapter 4

Generative Query Networks for Reinforcement Learning

4.1 Introduction

In the previous chapter we introduced a proof-of-concept neuro-symbolic model that addressed some of the drawbacks of DL algorithms. In particular, our approach tackled generalisation and abstraction by explicitly building in the notions of objects and relations into the architecture. The resulting neuro-symbolic architecture is able to abstract away from pixels and reason at the level of symbols which leads to better generalisation performance on unseen game environments.

In this chapter we concern ourselves again with generalisation and data efficiency. However, unlike in the previous chapter the goal here is to learn the representations that improve our model's generalisation ability, rather than hand-crafting them. Given that we would like these representations to be learned from data directly, the emphasis now lies on defining datasets and tasks that require the model to learn general representations in order to succeed.

An example of such an approach are Generative Query Networks (GQN) [Eslami et al., 2018a], a recent deep generative model for scene understanding. Given a few images rendered from a 3D scene and the position that these images were observed from, GQN learns to predict what the 3D scene would look like from any different viewpoint (see Figure 4.1A). While the model's ability to accurately reconstruct scenes from new viewpoints is interesting in itself, we will focus on the properties of the representations that are generated by a trained GQN.

As shown in the original publication) [Eslami et al., 2018a], GQN is able to generalise to held-out combinations

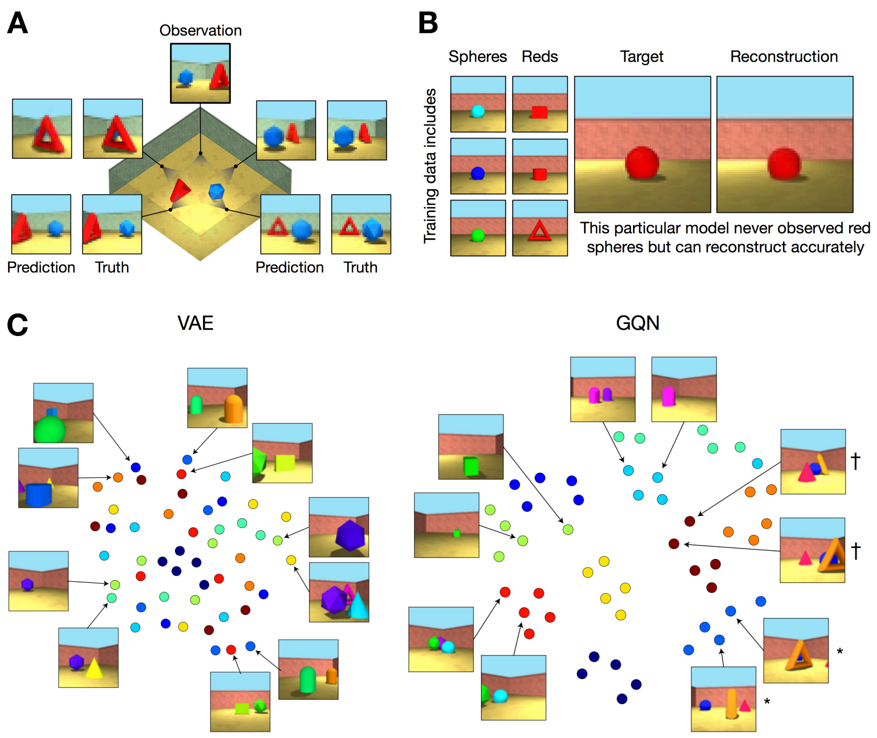


Figure 4.1: Figures from the original GQN publication [Eslami et al., 2018a]. A: The GQN scene reconstruction task on the MuJoCo room environment. The model's predictions given the single observation at the top are shown alongside the ground truth images for each of the query views. B: Testing the generalisation abilities of GQN on a held out object. GQN is able to reconstruct combinations of object attributes that it didn't encounter during training. C: t-SNE projection of representations obtained from images of different rooms with a VAE vs GQN. Each dot corresponds to an image. Points of the same colour correspond to images of the same room from different views. As shown in the plot the representations generated by GQN cluster by room rather than pixel similarity even for rooms that cotain identical objects in different possitions (marked with † and *). Figure adapted from [Eslami et al., 2018a].

of attributes in the dataset. More concretely, the authors trained GQN on a dataset of rooms that contain a single object which can vary in colour and shape. In the training set the object can be red or a sphere but is never both at the same time. Despite never having been trained on a red sphere, when asked to reconstruct one at test time GQN is able to carry this out successfully (see Figure 4.1B). The ability to generalise in such a way is likely to result from the choice of dataset. By training on a dataset containing objects with a large range of colours and shapes the resulting model is not able to get away with overfitting to a small set of attribute combinations. Instead, it is likely to learn a more distributed representation that allows for combinatorial combinations of the different object attributes.

Beyond performing well on generalisation tasks the representations obtained with GQN also seem to capture higher-level semantics of the data (Figure 4.1C). When clustering lower dimensional projections of the representations, these clusters are formed by room rather than just by pixel similarity as is the case with representations obtained with a regular VAE. This is likely a result of the task design. Because GQN has to be able to reconstruct the scene from an arbitrary point of view during training, it has to put more emphasis on encoding the objects in the scene rather than just focusing on reconstructing pixel-level patterns as is the case with VAEs.

With this in mind, the goal of this chapter is to investigate whether using the representations generated by a pre-trained model, which have been shown to exhibit such desirable properties, leads to improved performance on learning other downstream tasks. In particular, we are interested in applying the representations obtained with a pre-trained GQN model to training a reinforcement learning (RL) agent in a 3D environment. This setup is interesting for a number of reasons. First, it consolidates results obtained with GQN by establishing the usefulness of its learned representations for tasks beyond scene rendering. In addition, it addresses the question whether using pre-trained parts of a model can prove more successful and potentially less data intensive than training deep networks end-to-end. Finally, a working pipeline would be able to take advantage of the experience GQN gained from learning to reason about a 3D environment and result in more robust RL agents. In particular, the agent would be robust to the position of the source for its visual input, which is, to the best of our knowledge, something that has not been achieved with deep reinforcement learning systems and would contribute towards training more data-efficient agents.

The chapter is structured as follows:

- We discuss related work on transfer learning and model pre-training in section 4.5.
- In section 4.2 we give an overview of the GQN model introduced in [Eslami et al., 2018a] and outline its training procedure.
- In section 4.3 we present the Jaco arm reaching task and describe the experimental setup as well as the agent baselines for our experiments.
- Section 4.4 covers the results obtained with GQN on the reconstruction of the Jaco arm environment as well as the performance of our agent on different conditions of the reaching task.
- In section 4.6 we summarize the findings of this chapter.

4.2 Generative Query Networks

4.2.1 Model

The Scene Rendering Task

Generative Query Networks (GQN) learn to generate representations that capture the information about an underlying 3D scene provided by 2D renderings from that scene. More concretely, given a set of 2D

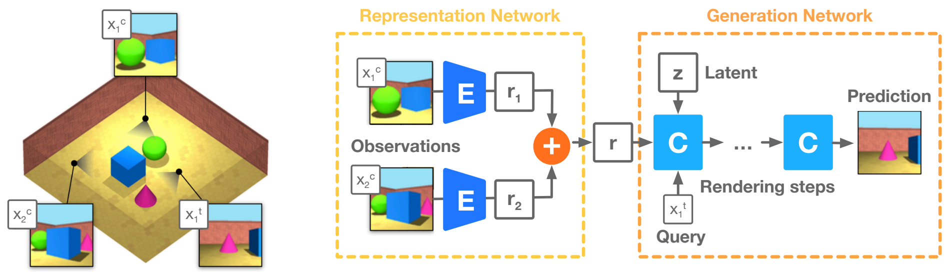


Figure 4.2: Left: Training environment for GQN. A 3D room environment contains objects of different shapes, colours, sizes and positions. The observations $(\mathbf{x}^c, \mathbf{y}^c)$ are shown alongside the target image $\mathbf{y}^t$ at $\mathbf{x}^t$. Right: the GQN model. E corresponds to the encoder that computes the representations $\mathbf{r}_i$ and C is a cell of the LSTM in the generation network. Figure adapted from [Eslami et al., 2018a].

observations $\mathbf{y}^c$ of a 3D environment and the positions $\mathbf{x}^c$ within that environment from which they were rendered the task is to predict what the 2D rendering of that scene would look like from an unobserved viewpoint $\mathbf{x}^t$. There are several reasons why this task is interesting from a machine learning research point of view:

- **Flexibility at test time**: Because GQN is presented with a different scene at every training iteration it has to learn to use the context $C = (\mathbf{x}^c, \mathbf{y}^c)$ to make accurate predictions about the current one. As a result even once the training phase is over GQN is still able to form predictions about any scene within the data distribution.
- **Representation learning**: In order to succeed at reconstructing the 2D view of a 3D scene, an algorithm needs to learn how to capture the main characteristics of that environment (e.g. objects, shapes, colours etc). In addition it needs to learn about more abstract concepts like occlusion and lighting.
- **Uncertainty**: When uncertain about certain aspects of the underlying scene, the model should learn to generate predictions that are coherent and agree with the information provided by the observations $(\mathbf{x}^c, \mathbf{y}^c)$, while suggesting a variety of possible alternatives.

GQN Model

The architecture of GQN (see Figure 4.2 for a schematic) reflects the scene rendering task described above. We will refer to the observations as the context points $\{(\mathbf{x^c}, \mathbf{y^c})_i\}_{i=1}^n$, which consist of n pairs of images $\mathbf{y}_i^c$ and the corresponding position from which they were rendered $\mathbf{x}_i^c$. Each of these $(\mathbf{x}^c, \mathbf{y}^c)_i$ pairs is processed by a shared encoder module which generates an individual representation $\mathbf{r}_i$ for each. Given that the information about the scene obtained from the different observations is cumulative we aggregate the individual representations $\mathbf{r}_i$ into a single representation $\mathbf{r}$. The simplest form of aggregation that reflects the order-invariance of the

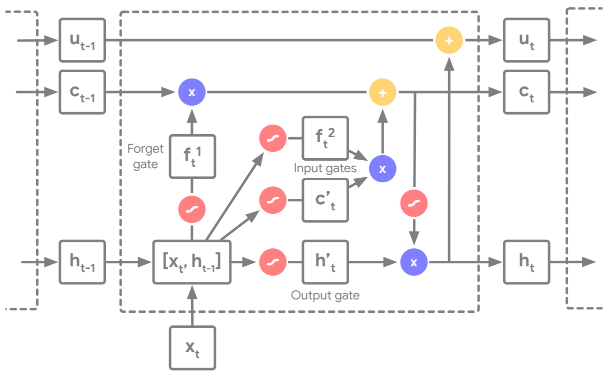


Figure 4.3: The computations carried out by a cell of the LSTM in the generation network of GQN. The cell corresponds almost exactly to the original LSTM cell (Figure 2.4) except for the additional skip-connections through u_t.

observation is to take the sum over all $\mathbf{r}_i$. While this is a very simple operation we found that in practice it works well and is straightforward to implement.

The representation $\mathbf{r}$ captures all the information about the underlying scene that the model can gather from the context points. In the next step this representation is fed alongside the new target position $\mathbf{x}^t$ and a source of noise $\mathbf{z}$ into a generation network that produces a rendering $\mathbf{y}^t$ of the scene from the target position $\mathbf{x}^t$. The generation network is implemented as a conditional Deep Recurrent Attentive Writer (DRAW), a deep network for image generation introduced by Gregor et al [Gregor et al., 2015]. At a higher level this model can be seen as a VAE (see Section 2.3.2) that generates images over several iterations using an LSTM (Section 2.2.2). As a result of its recursive nature DRAW models an auto-regressive density:

$$p(\mathbf{z}|\mathbf{x}^t, \mathbf{r}) = \prod_{\tau=1}^{\mathcal{T}} p(\mathbf{z}_\tau|\mathbf{x}^t, \mathbf{r}, \mathbf{z}_{<\tau}) \tag{4.1}$$

where τ indexes the current iteration and $\mathcal{T}$ is the total number of iterations. The auto-regressive definition of $p(\mathbf{z}|\mathbf{x}^t, \mathbf{r})$ makes it possible to approximate complex densities, including multi-modal ones.

We can summarize the computations carried out by GQN using the following equations:

$$\begin{aligned}
&\text{Encoded representation} && \mathbf{r} = \sum_{i=1}^{n} \eta_\vartheta(\mathbf{x}_i^c, \mathbf{y}_i^c) && (4.2)\\
&\text{Initial state (prior)} && (\mathbf{c}_0^p, \mathbf{h}_0^p, \mathbf{u}_0^p) = (\mathbf{0}, \mathbf{0}, \mathbf{0}) && (4.3)\\
&\text{Initial state (posterior)} && (\mathbf{c}_0^q, \mathbf{h}_0^q) = (\mathbf{0}, \mathbf{0}) && (4.4)\\
&\text{State update (posterior)} && (\mathbf{c}_{\tau+1}^q, \mathbf{h}_{\tau+1}^q) = C_\xi^q(\mathbf{y}^t, \mathbf{x}^t, \mathbf{r}, \mathbf{z}, \mathbf{c}_\tau^p, \mathbf{h}_\tau^p, \mathbf{h}_\tau^q, \mathbf{u}_\tau) && (4.5)\\
&\text{Prior distribution} && p(\mathbf{z}^p|\mathbf{x}^t, \mathbf{r}, \mathbf{z}_{<\tau}^p) = \mathcal{N}\big(\eta_\theta(\mathbf{h}_\tau^p)\big) && (4.6)\\
&\text{Posterior distribution} && p(\mathbf{z}^q|\mathbf{x}^t, \mathbf{r}, \mathbf{z}_{<\tau}^q) = \mathcal{N}\big(\eta_\xi(\mathbf{h}_\tau^q)\big) && (4.7)\\
&\text{Prior sample} && \mathbf{z}^p \sim p(\mathbf{z}^p|\mathbf{x}^t, \mathbf{r}, \mathbf{z}_{<\tau}^p) && (4.8)\\
&\text{Posterior sample} && \mathbf{z}^q \sim p(\mathbf{z}^q|\mathbf{x}^t, \mathbf{r}, \mathbf{z}_{<\tau}^q) && (4.9)\\
&\text{State update (prior)} && (\mathbf{c}_{\tau+1}^p, \mathbf{h}_{\tau+1}^p, \mathbf{u}_{\tau+1}^p) = C_\theta^p(\mathbf{x}^t, \mathbf{r}, \mathbf{z}^p, \mathbf{c}_\tau^p, \mathbf{h}_\tau^p, \mathbf{u}_\tau^p) && (4.10)\\
&\text{Observation sample} && \mathbf{x} \sim \mathcal{N}\big(\mathbf{y}|\eta_\omega(\mathbf{u}_T)\big) && (4.11)
\end{aligned}$$

Where $\mathbf{c}$, $\mathbf{h}$ and $\mathbf{u}$ correspond to the cell state, output and skip connections of the LSTM respectively and η denotes a neural network. More concretely: η_ϑ is the convolutional encoder network that generates the representations $\mathbf{r}_i$ from the context observations $(\mathbf{x}^c, \mathbf{y}^c)_i$. η_θ and η_ξ are networks that take in the LSTM state $\mathbf{h}$ and produce the sufficient statistics for the Gaussian latent distributions of the prior and the posterior distribution respectively. Finally, η_ω corresponds to a network that outputs the sufficient statistics for the output distribution which generates the final observations from the final $\mathbf{u}_T$.

The variables in Equations 4.2 to 4.11 are indexed by iteration τ as well as p or q depending on whether the variable is associated with the computation of the prior or posterior distribution. The detailed computations carried out within one of the LSTM cells C are depicted in Figure 4.3.

Generation

Equations 4.2 to 4.11 capture both the generation and the inference process. However, during generation only a subset of these equations needs to be evaluated. In the interest of clarity we highlight those computations that are part of the generation path:

$$\begin{aligned}
\mathbf{r} &= \sum_{i=1}^{n} h_\vartheta(\mathbf{x}_i^c, \mathbf{y}_i^c) \\
(\mathbf{c}_0^p, \mathbf{h}_0^p, \mathbf{u}_0^p) &= (\mathbf{0}, \mathbf{0}, \mathbf{0}) \\
(\mathbf{c}_0^q, \mathbf{h}_0^q) &= (\mathbf{0}, \mathbf{0}) \\
(\mathbf{c}_{\tau+1}^q, \mathbf{h}_{\tau+1}^q) &= C_\xi^q(\mathbf{y}^t, \mathbf{x}^t, \mathbf{r}, \mathbf{z}, \mathbf{c}_\tau^p, \mathbf{h}_\tau^p, \mathbf{h}_\tau^q, \mathbf{u}_\tau) \\
p(\mathbf{z}^p|\mathbf{x}^t, \mathbf{r}, \mathbf{z}_{<\tau}^p) &= \mathcal{N}\big(\eta_\theta(\mathbf{h}_\tau^p)\big) \\
p(\mathbf{z}^q|\mathbf{x}^t, \mathbf{r}, \mathbf{z}_{<\tau}^q) &= \mathcal{N}\big(\eta_\xi(\mathbf{h}_\tau^q)\big) \\
\mathbf{z}^p &\sim p(\mathbf{z}|\mathbf{x}^t, \mathbf{r}, \mathbf{z}_{<\tau}^p) \\
\mathbf{z}^q &\sim p(\mathbf{z}|\mathbf{x}^t, \mathbf{r}, \mathbf{z^q}_{<\tau}) \\
(\mathbf{c}_{\tau+1}^p, \mathbf{h}_{\tau+1}^p, \mathbf{u}_{\tau+1}^p) &= C_\theta^p(\mathbf{x}^t, \mathbf{r}, \mathbf{z}^p, \mathbf{c}_\tau^p, \mathbf{h}_\tau^p, \mathbf{u}_\tau^p) \\
\mathbf{x} &\sim \mathcal{N}\big(\mathbf{y}|\eta_\omega(\mathbf{u}_T)\big)
\end{aligned}$$

4.2.2 Training GQN

In order to train GQN we sample a number B of different scenes from the environment and render M observations from each. The number of observations M is sampled at random for each scene to ensure the model does not overfit to a specific number of observations.

Objective function

Given that, in essence, GQN is a conditional VAE the loss function resembles that of the VAE:

$$\mathcal{L}(\theta, \xi, \omega, \vartheta) = \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim D, \mathbf{z}\sim q}\Big[-\ln \mathcal{N}\big(\mathbf{y}|\eta_\omega(\mathbf{u}_T)\big) + \sum_{\tau=1}^{\mathcal{T}} \mathrm{KL}\big[\mathcal{N}\big(\mathbf{z}|\eta_\theta(\mathbf{h}_\tau^p)\big)||\mathcal{N}\big(\mathbf{z}|\eta_\xi(\mathbf{h}_\tau^q)\big)\big]\Big] \tag{4.12}$$

As in Equation 6.9 the first term of the right hand side corresponds to the reconstruction error, while the second term encourages the posterior containing the target observations to match the prior.

4.3 The Jaco Arm Reaching Task

The Jaco arm reaching task is embedded into the MuJoCo room environment [Todorov et al., 2012]. A MuJoCo reproduction of the robotic Jaco arm is placed in the middle of the room along with one target object. The arm has nine joints. The appearance of the room is modified for each episode by randomly

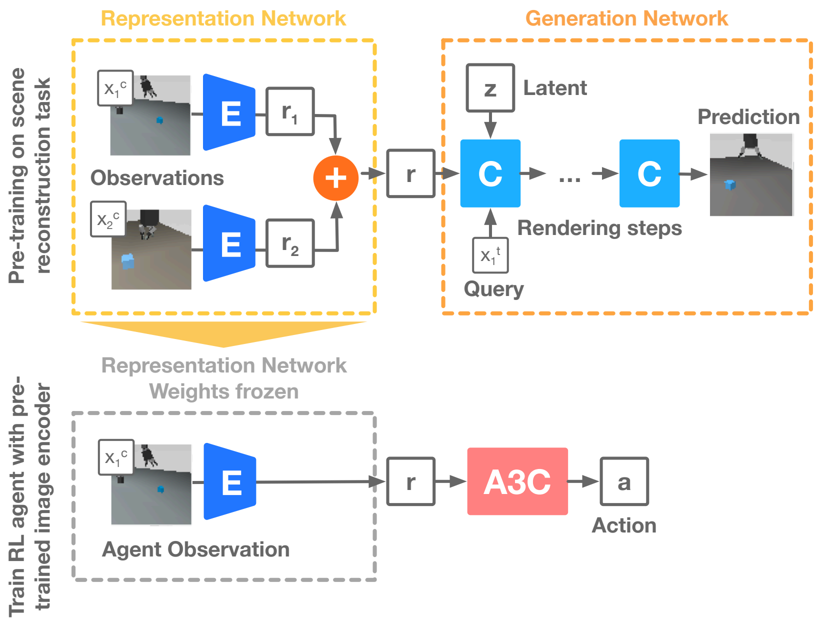


Figure 4.4: Experimental setup of the pre-training pipeline. First GQN is trained on the scene reconstruction task with images from the Jaco arm environment. The weights of the trained encoder are then frozen and used for the input encoder for an A3C agent that is trained on the reaching task from pixels.

modifying both colour and position of the target. In the second version of the environment described later on we also change the texture for the walls and floor from a fixed pool of options. Finally, the joint angles of the arm are also initialised at random within a range of physically sensible positions. The goal of the RL task is for the hand to reach the target and remain close to it for the remaining duration of the episode. The reward obtained at every step is a decreasing function of the distance from the hand to the target:

$$d_{final} = 1 - \tanh^2\left(\max\left(0, \left(\frac{d_{palm} + d_{pinch}}{2} - 0.15\right) \times 10\right)\right) \tag{4.13}$$

where d_{palm} and d_{pinch} are the distance from the target to either the palm of the hand or the pinch site weighted by [1.41, 1.41, 1] along the x, y and z axes, respectively.

In order to carry out the reaching experiments we train two models: first we pre-train a GQN model on the scenes of the room containing the Jaco arm. We then replace the encoder of an A3C agent (see Section 2.5.4) by the trained GQN encoder and train the agent following the same setup as [Rusu et al., 2017]. To ensure that GQN learns a complete representation of the Jaco-arm space we generate a dataset with a variety of arm positions. We achieve this by selecting random points in 3D space as targets for a proprioception-driven agent and recording one random intermediate state from the resulting trajectory. We do this with 50 independently trained agents to ensure diversity. We sample 4 million scenes and 20 images per scene to construct the dataset.

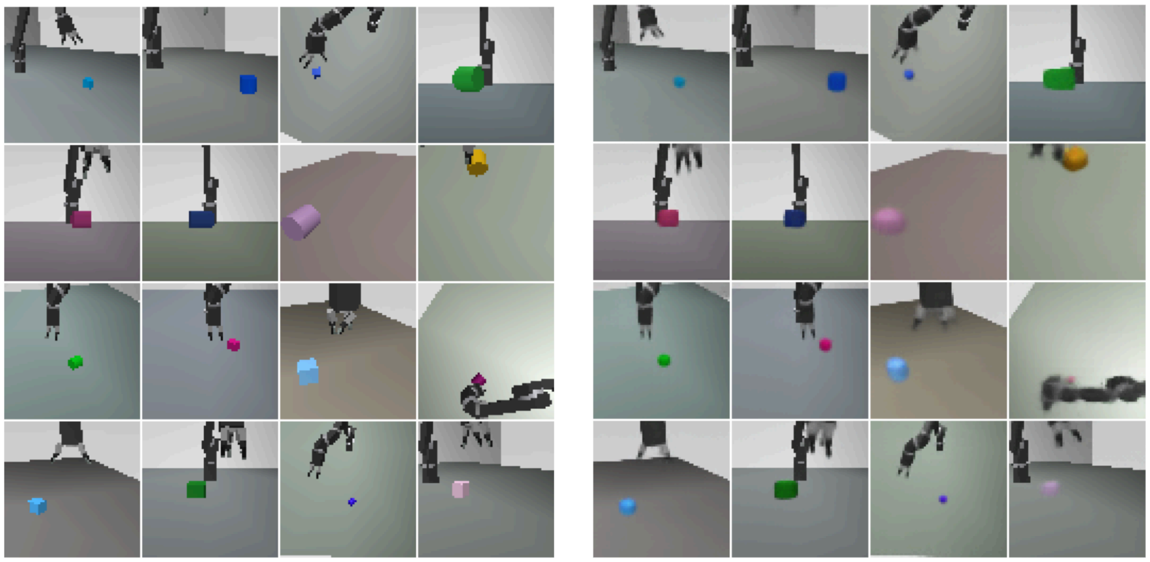

(a) Ground truth images. (b) Predictions.

Figure 4.5: Predictions of GQN on the Jaco arm dataset.

The model is trained by conditioning on n observations, where n is randomly chosen between 1 and 7 for each mini-batch. Finally, to avoid having a very large state space for reinforcement learning we modify the representation network by adding two fully connected layers after the convolutional layers. These layers reduce the representation size from 8x8x128 to 64x1.

Crucially, while GQN is trained using several input images at each step, we only feed one image at every step during RL training in order to remain close to current experimental protocols. When training the agent, the pre-trained weights of the representation network are not updated.

We compare our agent to a few baselines (see Figure 4.4 for a schematic):

- **A3C without pre-trained weights**: We train a feed-forward A3C [Rusu et al., 2017] agent from pixels using nine independent policies for each of the nine arm joints. The only difference to the previously published setup, apart from the change in environment, is that we modify the architecture of the A3C baseline input network to be identical to the representation network architecture of GQN for comparison.
- **GQN with fixed random weights**: This agent makes use of the GQN encoder as well, but in this particular case the weights are set at random and not updated during training. This setup provides lower bound on the performance of the agent.
- **Beta-VAE**: The encoder of this agent corresponds to a beta-VAE as in [Higgins et al., 2017]. We only run this baseline where applicable (i.e. where the input for the agent comes from a single fixed camera).

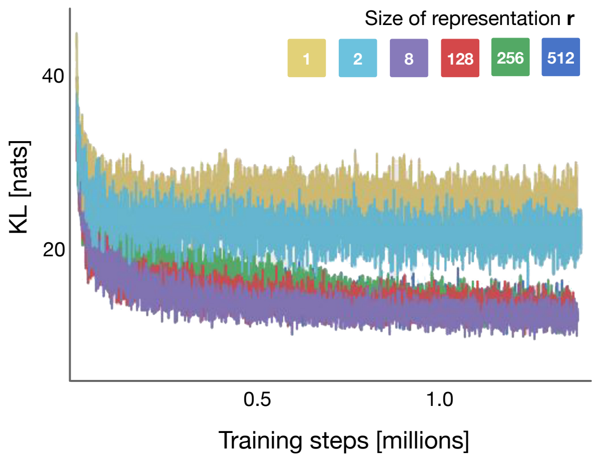


Figure 4.6: Evolution of the KL divergence between the prior and the posterior distribution for different sizes of the representation **r** over training.

4.4 Results and analysis

4.4.1 Training GQN on the Jaco Arm dataset

The results of training GQN on the first Jaco arm environment are shown in Figure 4.5. As evidenced by the generated predictions, the model learns to carry out the task successfully. While the predictions sometimes appear blurry in some parts of the image, they capture the overall structure (position of the arm, shape and colour of the target etc.) correctly. The model also successfully reproduces lighting and shadows.

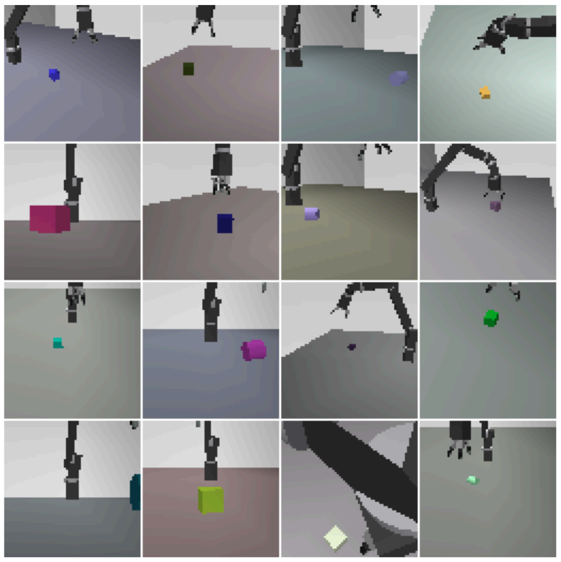
Varying the representation size

As mentioned in Section 4.3 the representations generated by the GQN encoder are one-dimensional vectors of length λ. In order to analyse the model's response to different representation sizes we train GQN setting λ to the following sizes: 1, 2, 8, 128, 256, 512.

Note that the degrees of freedom in this dataset are limited. The model only needs to encode:

- The 9 joint angles of the robot arm.
- The hue of the table.
- The shape and colour of the target.
- The position of the target.

In addition, the joint angles are likely to be correlated (e.g. the finger joints generally behave similarly) and a small model could get away with only encoding the position of the hand and interpolating the arm position.


(a) Ground truth images.

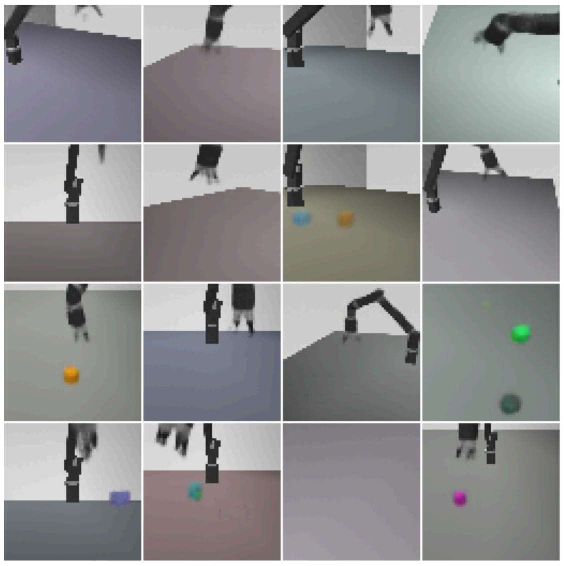

(b) Predictions.

Figure 4.7: Predictions of the Jaco arm environment generated with GQN. The size of the representation $\mathbf{r}$ for this set of experiments was set to 2.

As a result of this observation we looked into the effect of reducing the size of $\mathbf{r}$ (effectively the information bottleneck) on the model's performance.

Given that the posterior distribution observes the target image $\mathbf{y}^t$, we can measure the KL-divergence between the prior and the posterior distribution to track how much information is missing from the representation. The results of this experiment are shown in Figure 4.6 and they confirm our initial hypothesis that this task does not require a large bandwidth at the bottleneck: using vectors of length 1 or 2 for the model's representation is not sufficient, but beyond that there is not much difference between a representation of length 8 or 512. This is particularly convenient since we want to use this representation as the input to our RL agent and the size of the state space will depend on the size of the representation.

Finally, we provide some examples of the images generated by a GQN with representation size $\lambda = 2$. Interestingly, while the prediction of the arm and the target object are wrong in the majority of the cases, the colour of the table is correct for all. This behaviour is reasonable, since the model gets a significantly larger loss-signal from the large number of pixels that belong to the table than from the smaller number of pixels depicting the arm or the target.

4.4.2 GQN representations for RL

GQN trained from all possible view points

In the following experiments we use the trained GQN encoder as pre-processing for the input of an RL agent. The motivation behind this is to test whether training an agent using representations that exhibit desirable properties such as good generalisation and, to a certain extent, camera invariance leads to more robust performance and faster learning on RL tasks as well.

For our first set of experiments we define three different setups to test the flexibility and robustness of the agent when combined with GQN. Each setup requires the agent to carry out the same reach task described above. The key difference between the three is the number of different camera-positions used to train the same agent. Note, however, that the data used to pre-train the GQN is the same across all environments and corresponds to images taken at random from a dome around the arm. The different down-stream RL environments are:

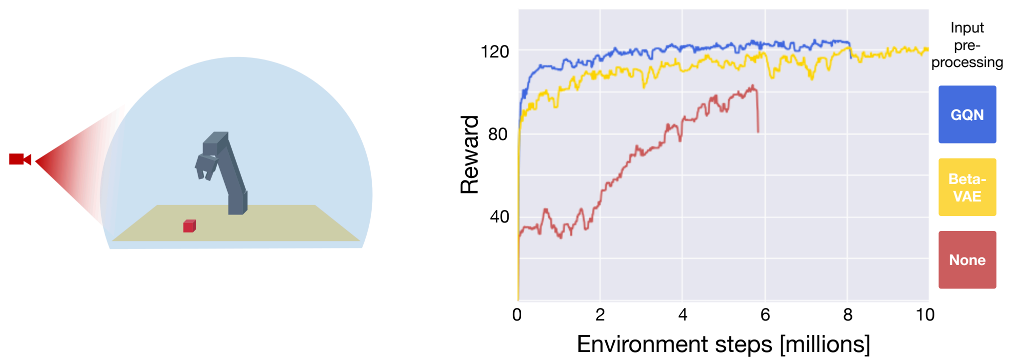
- **Environment 1**: Produces observations from a single fixed camera only.
- **Environment 2**: Generates observations from one of five possible fixed camera positions. The camera position is chosen at random at the beginning of each episode.
- **Environment 3**: Produces observations from a camera position chosen at random from a dome around the Jaco arm. As before, a new camera position is chosen at the beginning of each episode.

The results for these experiments including the three baseline agents are summarised in figure 4.8. We only compare performance against beta-VAE on the first experimental setup, as this is the only one provided by the authors in [Higgins et al., 2017].

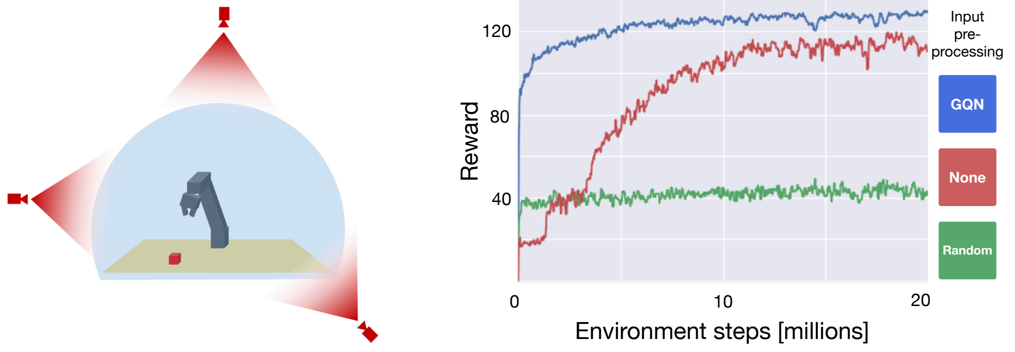
For all three task the combination of A3C + GQN learns the fastest and achieves the best performance. It is especially interesting that learning from a representation that was optimised to capture 3D scenes renders the agent more robust against changing camera views.

Comparing different types of pre-training

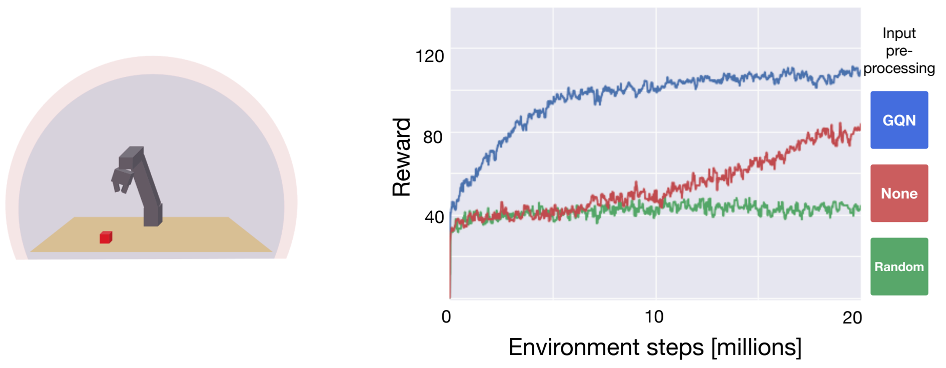
The main idea behind the research in this chapter is to take a pre-trained model and apply it to a down-stream task that is different from the one it was trained on. In the era of end-to-end training this is not necessarily a popular approach. In this section we compare the performance between four different experimental setups to investigate if splitting the training procedure can be beneficial for some tasks. The four variations of the task differ in their encoder (see Figure 4.9 for a schematic of each setup).



(a) Left: Training setup. GQN (blue) was trained on images sampled from any view point on the dome around the Jaco arm. The RL agent (red) was trained on images from a single camera. Right: Reward over training iterations for three agents with different image pre-processing: none (standard A3C, with an untrained GQN encoder), beta-VAE and GQN.



(b) Left: Training setup. GQN (blue) was trained on images sampled from any view point on the dome around the Jaco arm. The RL agent (red) was trained on images from five different fixed cameras. Right: Reward over training iterations for three agents with different image pre-processing: none (standard A3C, with an untrained GQN encoder), random weights and GQN



(c) Left: Training setup. GQN (blue) and the RL agent (red) were both trained on images sampled from any view point on the dome. Right: Reward over training iterations for three agents with different image pre-processing: none (standard A3C, with an untrained GQN encoder), random weights and GQN.

Figure 4.8: Agent's performance on the reaching task for different experimental setups. The different baselines differ in the pre-processing procedures applied to the input frames.

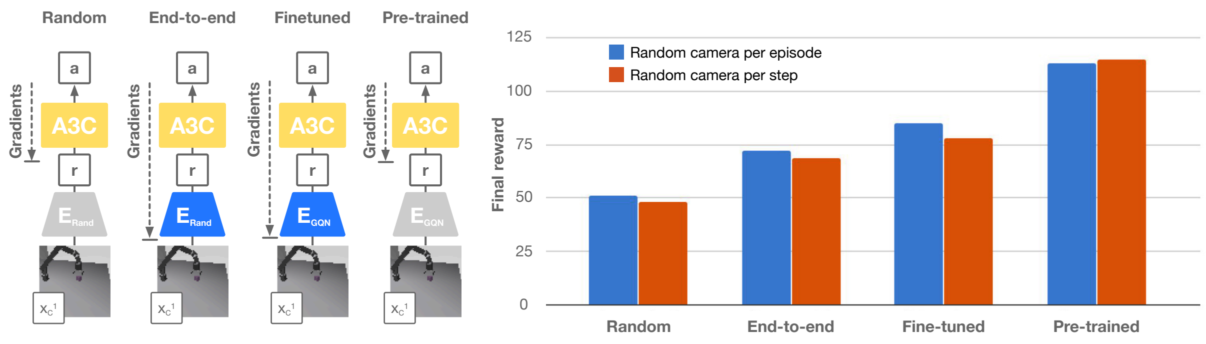


Figure 4.9: Maximum score achieved on the reaching task for four different training setups. Left: Schematic of the four setups used to generate the bar plots on the right. **Random** initialises the weights of GQN at random and does not update them when training the RL agent. **End-to-end** starts with randomly initialised weights, but allows them to be trained with the agent. The weights of the GQN in **Finetuned** are trained separately first but then updated when the agent is trained. **Pre-trained** also makes use of pre-trained weights for the GQN network, but doesn't update them during RL training. All of these conditions are compared on the reaching task where the camera is randomly switched once every episode and where the camera is randomly switched at every step. Right: The performance of the different setups.

- **Random**: The weights of the encoder are not pre-trained. Instead they are initialised randomly and not updated during training. This is the agent we are using as a lower bound baseline throughout this chapter.
- **End-to-end**: The weights of the encoder are not pre-trained. They are initialised randomly and trained together with the agent. This corresponds to the A3C baseline that we present throughout the chapter. Current deep learning agents are usually trained in this fashion.
- **Fine-tuned**: The weights of the encoder correspond to the pre-trained weights of the GQN encoder. During RL training these weights are also updated with the agent.
- **Pre-trained**: The weights of the encoder correspond to the pre-trained weights of the GQN encoder. During RL training, however, these weights remain fixed and are not updated. This corresponds to our main GQN+A3C agent.

We apply these four setups to the reaching task where the camera position for the observations is picked at random from around the Jaco arm either once per episode or at every time step. The results from this experiment are shown in Figure 4.9. As we observed in the previous experiments the random agent performs the worst and our GQN+A3C agent performs the best. Interestingly, initialising the agent with the trained GQN weights and updating them during RL training leads to worse performance than leaving them fixed. We hypothesize that since the gradients from the agent are noisy at the beginning of RL training the initial updates to the encoder weights are actually counter-productive and lead to catastrophic forgetting. Nevertheless, both methods with pre-trained weights still perform better than the agent trained end-to-end from scratch. Overall these results seem to suggest not only that pre-training can be beneficial for learning down-stream tasks efficiently but also that updating pre-trained weights on a new task might not actually improve performance.

There is an additional benefit to pre-training fixed models that we don't test in this chapter. Given that the GQN encoder improves performance on a task it was not trained for, it is likely that it will improve performance on a number of other tasks that involve processing 3D information. So, although pre-training a separate model does require more computation than training a single agent when tested on one task, if we were to use it for a whole set of RL tasks we could overall end up being more data efficient.

GQN trained from a small number of possible view points

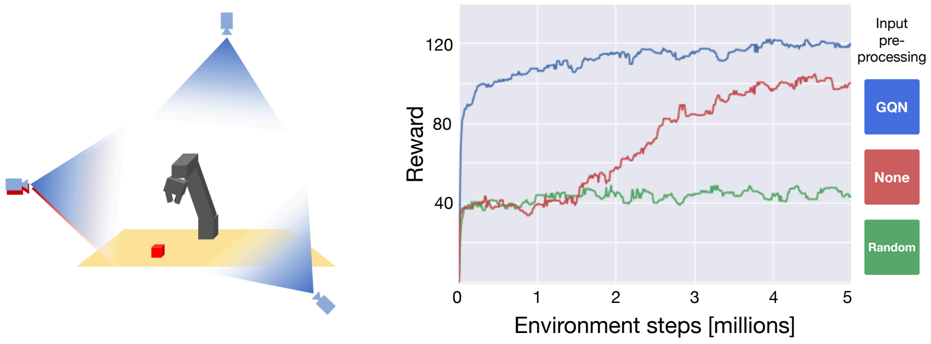
The previous set of experiments showed that by training agents on representations generated by the encoder of GQN we achieve better and more robust performance. However, the experimental setup would not be realistic for a real life robot arm since it requires taking pictures of the scene from almost every possible view point around the arm in order to train GQN.

In this section we try out a more realistic approach and train GQN on a small number of fixed cameras only, which will later also be used to train the RL agent. We choose the frontal and side view cameras (to make sure there are some views that do capture the Jaco arm well) and three additional random cameras. We use all five as the view points to train GQN and the representation obtained with the trained encoder are used to train an A3C model. As before, we have different setups for the RL training:

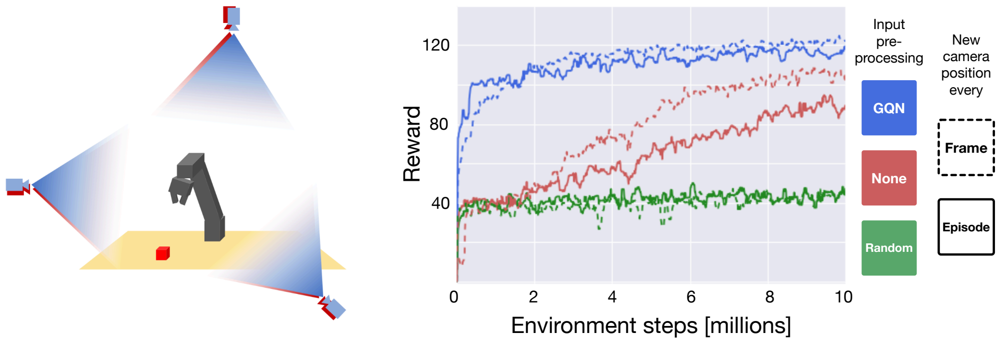
- **Environment 1**: Produces observations from a single fixed camera only.
- **Environment 2**: Generates observations from one of five possible fixed camera positions. There are two versions of this environment: one where the camera is chosen at random at the beginning of each episode and one where the camera is changed at every time step.

We compare our agent's performance against an A3C agent without the pre-trained GQN encoder and an A3C agent where the encoder weights are set at random and not updated during training. The latter serves as a lower bound on performance as it is essentially acting randomly.

As before, an agent trained on GQN representation learns faster and reaches higher performance for both environments. In addition this performance remains consistent across both versions of environment 2, which means the agent is indifferent as to whether it is given the same camera view throughout an episode or a different one at each step. Interestingly, the performance of the standard A3C agent improves when trained on the switching cameras task. We hypothesize that it might have learned to aggregate the information provided to it from the different camera views over time in a similar fashion to GQN.



(a) Left: Training setup. GQN (blue) was trained on images from 5 fixed cameras whereas the RL agent (red) was only trained on images from one fixed camera. Right: Reward over training iterations for three agents with different image pre-processing: none, random weights and GQN



(b) Left: Training setup. GQN (blue) and the RL agent (red) were both trained on images from five fixed cameras. Right: Reward over training iterations for three agents with different image pre-processing: none, random weights and GQN. For each setting we also compare switching camera every frame versus every episode.

Figure 4.10: Agent's performance on the reaching task with different experimental setups that all involve a small number of fixed cameras. The different baselines differ in the pre-processing procedures applied to the input frames.

4.4.3 Realistic-looking room environment

Finally, our goal is to reproduce the results within an environment that matches the other experiments in [Eslami et al., 2018a] in terms of appearance and complexity of textures.

The updated Jaco arm environment

We update the previous environment by adding walls around the Jaco arm table and changing the textures to match those of the other experiments in [Eslami et al., 2018a]. These textures contain more detail than the original gray table and add to the complexity of the learning task. There are three different wall textures and seven different floor textures that are sampled at random for each episode. In addition we change the target object to a sphere, for aesthetic purposes. The friction of the sphere was set to a very large value, to prevent it from rolling away when touched by the arm, as the goal of these experiments is to reach for a

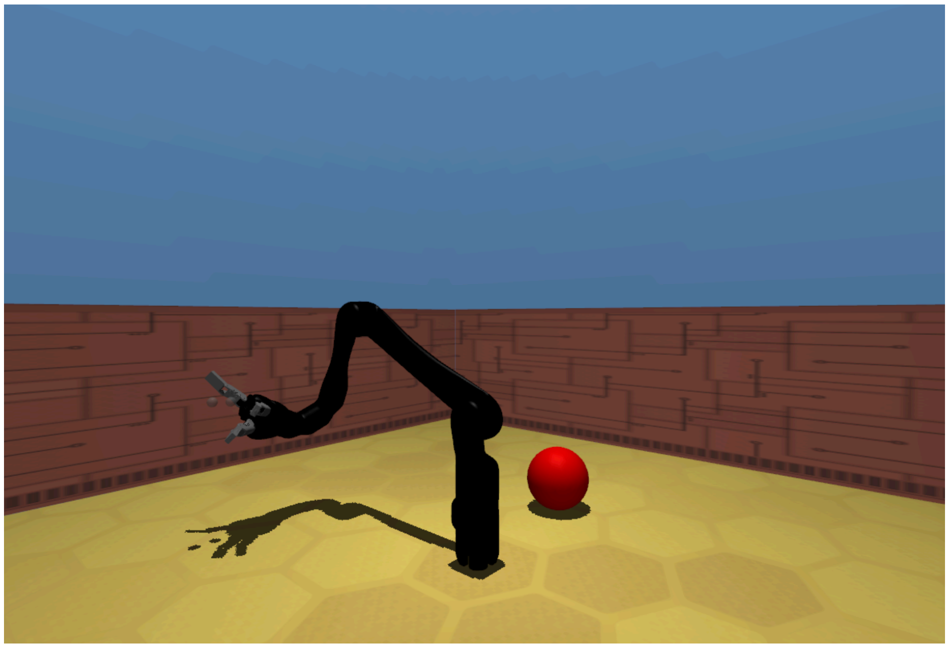


Figure 4.11: A high resolution rendering of the updated Jaco arm environment.

stationary target.

The resulting environment is shown in Figure 4.11. Given that the input resolution for all the models was only 32×32 we use a simplified model of the Jaco arm, that approximates the shape with rectangular cuboids rather than the more realistic looking mesh. Furthermore we remove the shadow of the robot arm to match the experiments in [Rusu et al., 2017] which we use as a baseline.

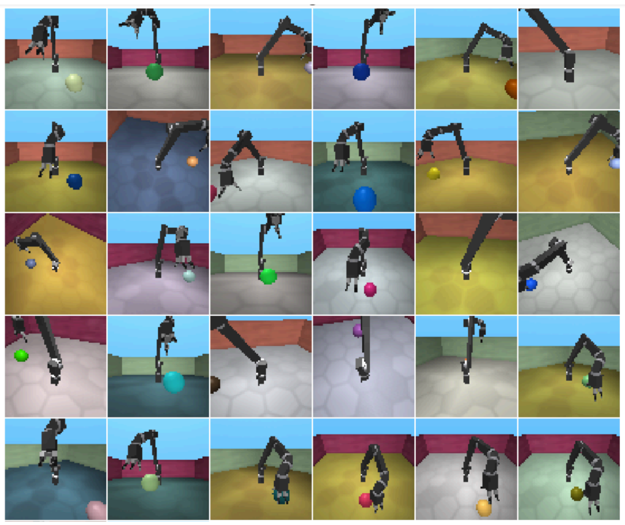
GQN Performance on the New Environment

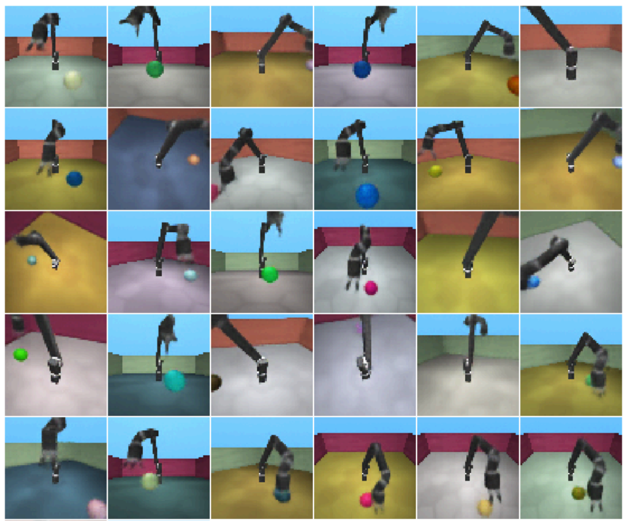
We train GQN on this new environment using the same procedure as before. As shown in Figure 4.12, GQN is able to successfully carry out the prediction task. Although this is not necessarily relevant for the RL tasks it is important to note that GQN is able to reproduce the new textures of the walls and floors.

Reaching Task in the New Environment

Finally, we use the trained GQN model to train an agent in this updated environment. For this final experiment we will focus on the setup where GQN is trained on images from all possible view points around the arm. We train two versions of the reaching task: one with a fixed camera and one where the camera is changing at every step.

As before, our baseline is an A3C agent whose encoder matches the architecture of the GQN encoder, but has not been pre-trained. In addition to this baseline we trained and agent that was given propioception


(a) Ground truth images.


(b) Reconstructions.

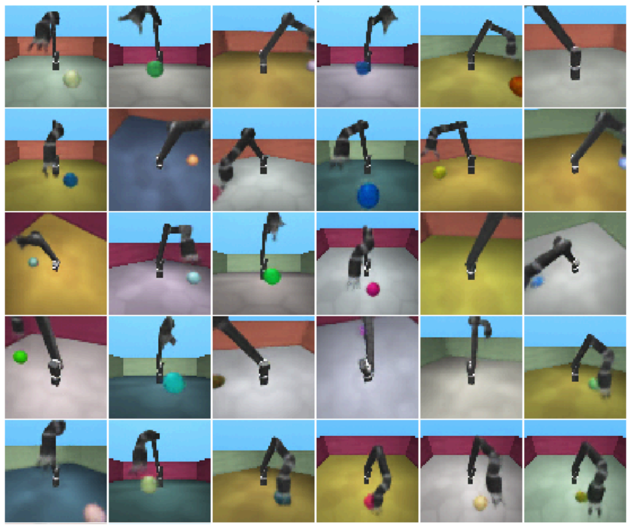

(c) Predictions.

Figure 4.12: Reconstructions and predictions produced by GQN on the updated Jaco arm environment.

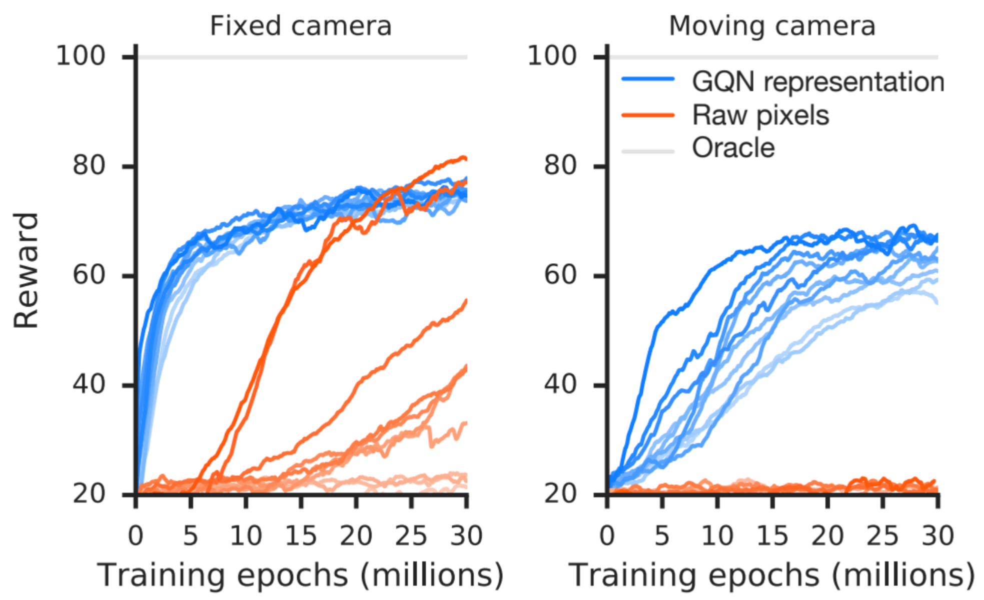


Figure 4.13: Agent performance on both versions of the reaching task in the updated Jaco arm environment. Figure adapted from [Eslami et al., 2018a].

information of the jaco arm as well as the position of the target as input. This agent served as an upper bound on performance. We also trained an agent with random pre-processing in the encoder that served as a lower bound. For this set of experiments we have normalised the performance of our agents by the upper and lower bound and the results of the training process are shown in Figure 4.13.

As before, an agent learning from the representations generated by the GQN encoder learns the task faster for both variations. It is interesting that in the experimental setup involving a single camera, the A3C agent, while slower, is sometimes able to reach a higher performance than our GQN agent. The reason for this might be that by training an encoder whose loss is independent of the follow-up RL task it might miss some details that are necessary for the reaching task but not as important for the reconstruction. Given that the encoder is not updated, if GQN is not capturing something that is relevant for RL the agent will not have any way of accessing that information during training. Despite this, the learning process of the GQN agent is much more robust as evidenced by the variance in the performance across learning curves.

In the case of the moving camera the performance of the GQN agent varies more, but it still achieves high scores overall, while the regular A3C agent is not able to learn at all in that period of training time.

4.5 Related Work

The idea of re-using some pre-trained network to facilitate the training of an RL agent on some new task lies at the heart of transfer learning research in machine learning [Pan et al., 2010]. Some of the models in this area focus on how to transfer knowledge between agents [Gupta et al., 2016] or from a labelled subset of the training data to the rest of the unlabelled data [Finn et al., 2016b]. A different branch of transfer learning research considers the task of continual learning, where the same agent has to learn a number of different tasks with different reward functions. This has been achieved by progressively growing the network with each new task that is being trained [Rusu et al., 2016] or by decoupling the environment dynamics from the reward in the value function [Barreto et al., 2017], for example.

More related to our line of research are approaches that aim at learning a representation that will lend itself to transfer learning tasks. These representations can be learned in the loop with a controller [Finn et al., 2016a] to enable fast adaptation in robots. Another possibility is to learn several state representations and a task-specific gate that filters them as in [Raffin et al., 2016].

Both of these approaches learned task-specific representations that were trained alongside the agent. Alternatively one can train a model to produce task-independent representations that facilitate transfer learning. This is the case in Darla [Higgins et al., 2017], where beta-VAE (Section 2.2.2) is trained to encode the environment and the disentangled representations are used for robust RL. This approach is the closest to the research described in this chapter. As in Darla, we train our generative model on a task that is unrelated to the subsequent RL task and use the trained model as pre-processing for the agent's input during RL training. Unlike Darla, we don't explicitly aim to have disentangled representations, but rather make use of representations that were useful for 3-D reasoning tasks.

Finally it might seem ill-fitted to compare our work to transfer learning algorithms given that we only apply the learned representations to the Jaco arm reaching task. However, since the scene prediction task that GQN is trained on is unrelated to the reaching task we can consider this to fall into the transfer learning category. Furthermore, while we have not applied it to any other RL task there is nothing inherently 'reaching-task specific' about the scene reconstruction experiments used to train GQN that would suggest that the benefits of using its representations would not also apply to other RL tasks.

4.6 Discussion

In this chapter we have used a pre-trained generative model of 3D scenes as the encoder for the visual input of an RL agent. The purpose of this was threefold. First, we wanted to test whether the representations generated by GQN are useful for other tasks beyond scene understanding. Second, we wanted to investigate whether using pre-trained parts for a model can be more beneficial than training end-to-end. Finally, another goal was to train a robust RL agent that can receive as input images from any point of view in a 3D environment.

With the exception of these experiments, the results presented in the original GQN publication all revolved around scene understanding. As such we wanted to test whether the representations generated by GQN are helpful for other tasks as well. For all of the experiments presented in this chapter the agent trained with the GQN encoder has proven to be the most robust and the most data efficient. In the majority of the cases it also reaches the best performance. From this we can conclude that the GQN representations capture 3D scene information in a way that is useful for other 3D tasks beyond scene reconstruction.

Our results comparing pre-trained encoders with agents that are trained end-to-end seem to imply that pre-training models can lead to higher performance. Interestingly fine-tuning the pre-trained parts of the model to specific tasks is not necessarily beneficial for training and can, indeed, decrease final performance.

Finally, the proposed agent architecture is able to carry out the task it is trained on from visual information taken at any position within the dome. To achieve such a level of performance with an agent trained from scratch one would have to train multiple agents for the different camera views. As such using pre-trained model parts can result in better data efficiency.

A large part of these results goes back to the power of the pre-trained model itself: the fact that the representations generated by GQN have a range of interesting properties such as generalisation and abstraction is what drives our agent's performance. These properties of GQN are a result of its training regime. In the following chapters we extend this training regime beyond the specific task of scene understanding to more general machine learning tasks such as regression and classification as we introduce Neural Processes.

Chapter 5

Conditional Neural Processes

5.1 Introduction

In the previous chapter we looked at Generative Query Networks (GQN) and how their training regime produces latent representations with interesting properties that can be applied to different downstream tasks, such as reinforcement learning. We focused on two properties in particular: flexibility at test time and generalisation to unseen attribute combinations.

Beyond that, the representations generated by GQN have been shown to display other desirable traits. In the original GQN publication the authors show, for example, that GQN is able to capture the uncertainty over its predictions given the observed context. This is demonstrated on the top-down view prediction task in Figure 5.1. In these experiments GQN is provided with first-person views from a randomly generated maze environment and has to predict the top-down view of the maze. As demonstrated by the predicted images in the bottom two rows, the accuracy of GQN's predictions increase as it is provided with more context observations. In addition to looking at the predictions we can also measure the model's predicted uncertainty at different positions in the room. We do this by calculating the expected information gain obtained from including a context point at a given location. As illustrated by the heat-maps the uncertainty estimates of GQN agree with the observations of the room it has seen so far.

Altogether, GQN produces representations with three desirable traits that are not common for deep learning algorithms:

1. The ability to generalise to held-out attribute combinations.
2. The flexibility to adapt the predictions to a set of observations at test time.

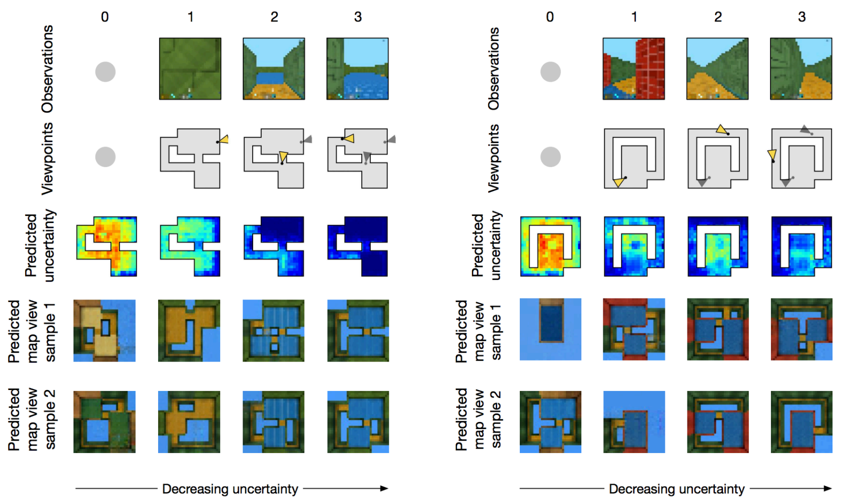


Figure 5.1: Uncertainty estimation in generative query networks. Given some observations within a small virtual maze (top row) and the positions from which they were taken (second row) GQN is trained to predict the top down view of the maze. The two bottom rows show two predictions of the model. GQN starts off with no context observations and is provided with an increasing number of observations. Each column shows the new observation and the model's predictions given the observations so far. As the number of observations grows the accuracy of the predictions increases and the samples look more and more alike. Crucially, we can measure the model's predicted uncertainty by calculating the information gain of each location given the context, as shown in the third row. Figure adapted from [Eslami et al., 2018a].

3. The capacity to estimate the uncertainty over its predictions.

The goal of the research presented in this chapter is to take advantage of this meta-learning training setup and apply it to more general machine learning tasks beyond scene understanding, such as regression and classification. To that aim we introduce Conditional Neural Processes (CNPs), a meta-learning algorithm that expands the training regime of GQN to few-shot regression and classification algorithms. To understand why CNPs are different from current regression models, it is useful to compare their training set-up and properties to those of other algorithms. Take conventional neural networks (NNs), for example (see Section 2.1). NNs are generally trained to approximate a single function (e.g. a value function in reinforcement learning) by updating the randomly initialised parameters of the network using a large number of training examples. Once the training phase is over, these parameters remain fixed and the hope is that the learned function is a good approximation for the unseen test points. Some of the benefits of neural network models include:

Data-driven learning. NNs are able to learn the model parameters from data directly using gradient descent updates.

Complex functions. The large number of parameters of neural networks makes it possible to model complex functions with few functional restrictions.

Fast evaluation. Evaluating neural networks at test time is very fast, as it only involves a forward pass which boils down to matrix multiplication.

Gaussian processes provide a different approach to regression (see Section 2.4.2). If we assume the kernel and its parameters are known in advance, GPs don't need to be trained as is the case with NNs. At inference time, the GP updates its belief over the underlying distribution using the observations from the target function. GPs display some desirable properties:

Distribution over functions. Rather than just approximating a single function that agrees with the observations GPs learn a whole distribution over functions. This allows them to flexibly adapt their predictions at inference time.

Measure of uncertainty. Given a choice of kernel, GPs have a statistically grounded measure of uncertainty over their predictions.

CNPs aim at combining the benefits of both. They are implemented as neural networks, but rather than approximating a single function CNPs learn to approximate a whole distribution over them. This distribution is learned from the data directly by updating the CNP's parameters using gradient descent. This is a result of the data: unlike traditional machine learning algorithms, CNPs train on whole sets of functions rather than just a single function (see Figure 5.3). As with NNs the parameters of CNPs remain fixed at test time, but the model can still adapt its predictions to the context information it is given. CNPs thus combine benefits of NNs and GPs:

Data-driven learning. CNPs learn the model parameters via gradient descent updates from datasets that consist of distributions over functions.

Complex functions. CNPs are implemented as neural networks and thus the large number of parameters allows it to model complex functions.

Fast evaluation. Evaluating CNPs at test time only involves a forward pass and has a runtime of $\mathcal{O}(n+m)$ where n are the number of context observations and m the number of targets.

Distribution over functions. Rather than just approximating a single function CNPs learn to capture the distribution over functions from the dataset.

Measure of uncertainty. Given some context points CNPs are able to not only predict the expected function value, but also the uncertainty of that prediction, as captured by the variance in the training set.

Finally, CNPs and their extension Neural Processes (introduced in Chapter 6) can be interpreted as a neural approximation of a stochastic process (thus the name Neural Processes). We will elaborate on this relation in the next chapter.

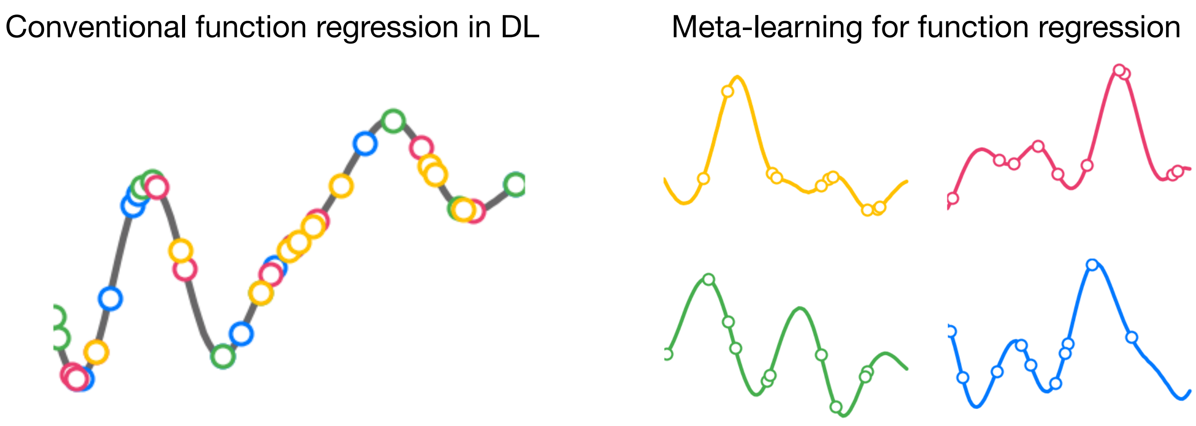


Figure 5.2: Conventional deep learning datasets versus meta-learning datasets. Traditional DL datasets consist of points from a single target function. At every iteration we select a subset of these data points (called a batch) and use them to update the network's parameters. datasets for meta learning algorithms on the other hand conform to a hierarchical structure and consist of a dataset over datasets (e.g. a dataset of functions). At every episode we first select a subset (i.e. a function) and then choose our training batch from this subset.

The rest of this chapter is structured as follows:

- In Section 5.2 we introduce the regression task and the CNP model.
- In Section 5.4 we discuss related research.
- In Section 5.3 we present the results on few-shot regression and few-shot classification.
- In section 5.5 we summarise the findings and motivate an extension of CNPs introduced in the next chapter.

5.2 Model

5.2.1 The Regression Task

As mentioned before, CNPs are flexible enough to be applied to a number of tasks that can range from regression to classification. In the following we lay the foundations for CNPs by focusing on the regression task only. We will build on this when applying CNPs to classification in the results section later on.

The formulation of the regression task is similar to that of the GQN scene rendering task. Where in the previous chapter we started by picking a room, in the case of CNPs we first sample a function $f \sim \mathcal{D}$ from the distribution over functions $\mathcal{D}$. This could mean sampling a function $f \sim \mathcal{GP}$ from a Gaussian process or choosing an image from a dataset of images (in this last case each of the pixels would correspond to a point on the function, see Figure 5.5). Given n context points $C = \{(\mathbf{x}^c, \mathbf{y}^c)_i\}_{i=1}^n$ on f, the goal is to predict the value of $\mathbf{y}^t = f(\mathbf{x}^t)$ at the target position $T = \{\mathbf{x}_i^t\}_{i=1}^m$.

5.2.2 Conditional Neural Process Model

The CNP model (shown in Figure 5.11) can be summarised as follows:

$$\mathbf{r}_i = h_\theta(\mathbf{x}_i^c, \mathbf{y}_i^c) \qquad \forall(\mathbf{x}_i^c, \mathbf{y}_i^c) \in C \tag{5.1}$$

$$\mathbf{r} = \mathbf{r}_1 \oplus \mathbf{r}_2 \oplus \dots \mathbf{r}_{n-1} \oplus \mathbf{r}_n \tag{5.2}$$

$$\phi_i = g_\vartheta(\mathbf{x}_i^t, \mathbf{r}) \qquad \forall(\mathbf{x}_i^t) \in T \tag{5.3}$$

$$\mathbf{y}_i^t \sim p(\mathbf{y}_i^t | \phi_i) \tag{5.4}$$

where $h_\theta : X \times Y \to \mathbb{R}^d$ and $g_\vartheta : X \times \mathbb{R}^d \to \mathbb{R}^e$ are neural networks with parameters θ and ϑ, $\oplus$ is a commutative operation that takes a row of elements in $\mathbb{R}^d$ and maps them into a single element of $\mathbb{R}^d$, and p is a probability distribution with sufficient statistics ϕ. Depending on the task, the model learns to parametrise a different output distribution. For regression tasks we use ϕ_i to parametrise the mean and variance $\phi_i = (\mu_i, \sigma_i^2)$ of a Gaussian distribution $\mathcal{N}(\mu_i, \sigma_i^2)$ for every $\mathbf{x}_i^t \in T$. For classification tasks ϕ_i parametrises the logits of the class probabilities p_c over the c classes of a categorical distribution. In most of our experiments we take $a_1 \oplus \dots \oplus a_n$ to be the mean operation $(a_1 + \dots + a_n)/n$.

This architecture ensures permutation invariance and $\mathcal{O}(n + m)$ scaling for conditional prediction. We note that, since $r_1 \oplus \dots \oplus r_n$ can be computed in $\mathcal{O}(1)$ from $r_1 \oplus \dots \oplus r_{n-1}$, this architecture supports streaming observations with minimal overhead.

Within this specification of the model there are still some aspects that can be modified to suit specific requirements. The exact implementation of h, for example, can be adapted to the data type. For low dimensional data the encoder can be implemented as an MLP, whereas for inputs with larger dimensions and spatial correlations it can also include convolutions.

5.2.3 Training Conditional Neural Processes

To train CNPs we first sample $f \sim \mathcal{D}$ and then select a subset n of points as the context $C = (\mathbf{x}^c, \mathbf{y}^c) = \{(\mathbf{x}_i^c, \mathbf{y}_i^c)\}_{i=1}^n$. We pick a second subset n' as targets and combine them with C to form our full target set $T = (\mathbf{x}^t, \mathbf{y}^t) = \{(\mathbf{x}_i^t, \mathbf{y}_i^t)\}_{i=1}^m$, where $m = n + n'$. Asking the model to predict the values of f both at the new target positions as well as at the known context points ensures that the network makes use of the context. Both n and n' are sampled randomly with $n \sim$ uniform$[1, ..., N]$ where N is the maximum number of context

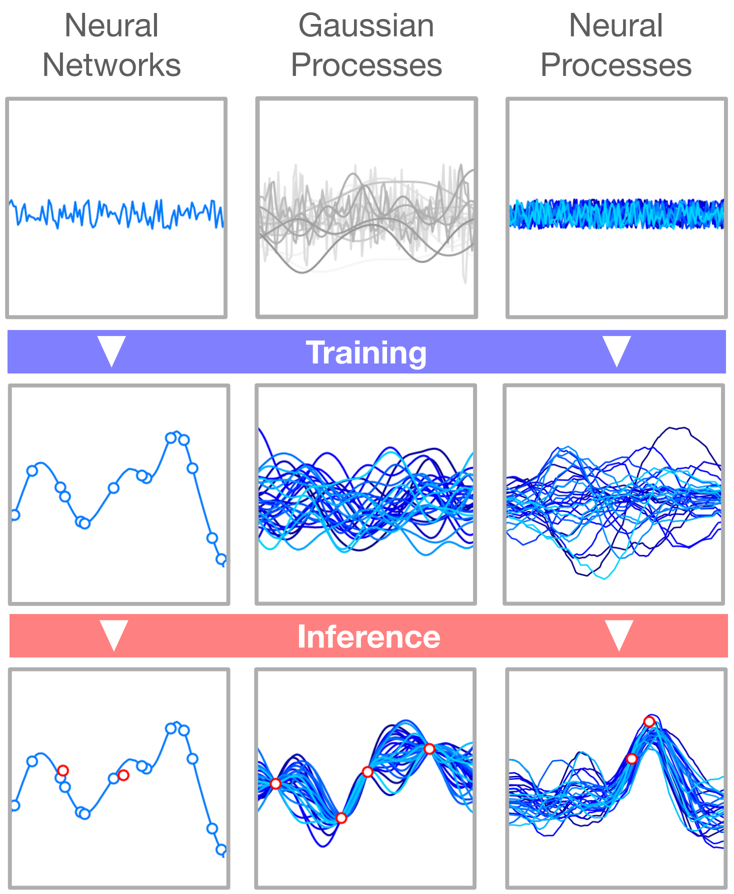


Figure 5.3: Training and inference stages of neural networks (NNs), Gaussian processes (GPs) and neural processes (NPs). The first row illustrates the predictions of the different regression algorithms before training. As we are not addressing the choice of kernel for the GPs in this work, we assume there is no training phase which we have illustrated by plotting some examples of possible kernels in grey. The middle row shows the models' predictions after training and the bottom row shows the updated predictions upon observing some new points at test time.

points. We use C and T to minimise the negative conditional log probability

$$\mathcal{L}(\theta) = -\mathbb{E}_{f\sim\mathcal{D}}\left[\mathbb{E}_N\left[\log p_\theta(\mathbf{y}^t|\mathbf{x}^t, (\mathbf{x}^c, \mathbf{y}^c), \phi)\right]\right] \tag{5.5}$$

In practice, we take Monte Carlo estimates of the gradient of this loss by sampling f and N.

5.3 Experimental Results

5.3.1 Function Regression

As a first experiment we test CNP on the classical 1D regression task that is used as a common baseline for GPs. We generate two different datasets that consist of functions generated from a GP with an exponential

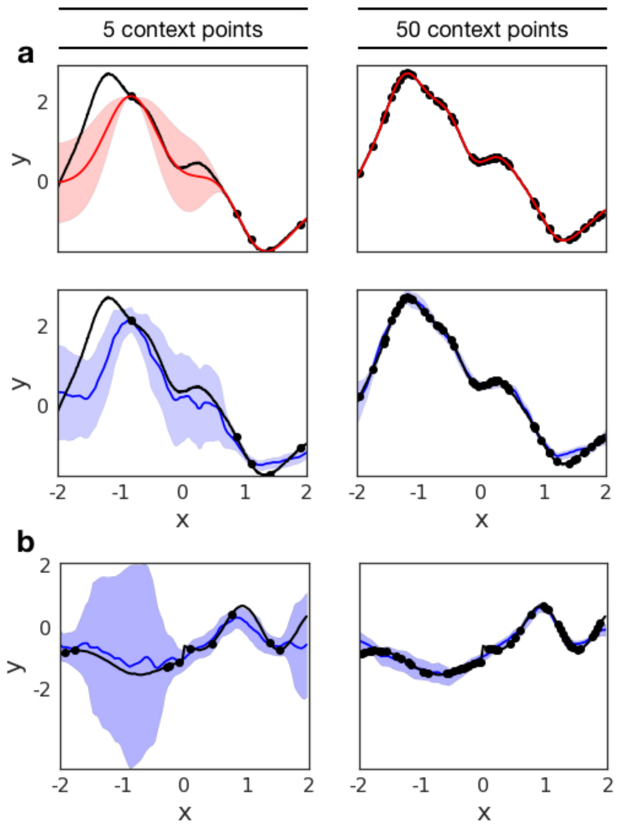


Figure 5.4: **1-D Regression**. Regression results on a 1-D curve (black line) using 5 (left column) and 50 (right column) context points (black dots). The first two rows show the predicted mean and variance for the regression of a single underlying kernel for GPs (red) and CNPs (blue). The bottom row shows the predictions of CNPs for a curve with switching kernel parameters. Figure adapted from [Garnelo et al., 2018a].

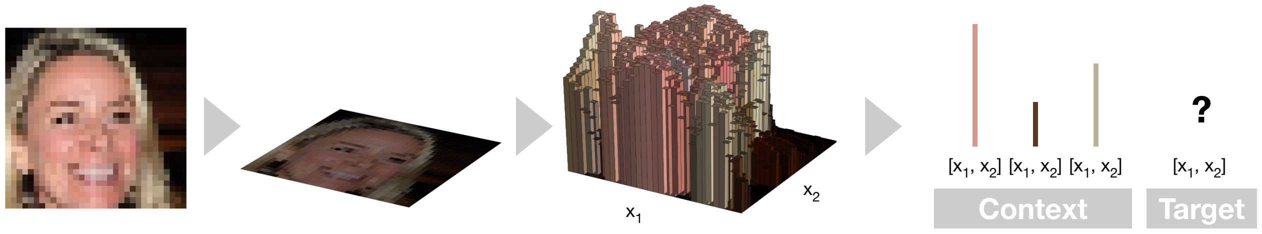


Figure 5.5: Image completion as a regression task. This figure illustrates how image completion can be phrased as a regression task where the position of a pixel is the function input x and the colour of the pixel the function output $f(x)$. When training neural processes on this task we provide a varying number of pixel positions and their colour and query the colour of pixels at other target locations. Figure adapted from [Garnelo et al., 2018a].

kernel. In the first dataset we use a kernel with fixed parameters, and in the second dataset the function switches at some random point on the real line between two functions, each sampled with different kernel parameters.

At every training step we sample a curve from the GP, select a subset of n points $(\mathbf{x}^c, \mathbf{y}^c)$ as observations, and a subset of points $(\mathbf{x}^t, \mathbf{y}^t)$ as target points. Using the model described in Figure 5.11, the observed points are encoded using a three layer MLP encoder h with a 128 dimensional output representation $\mathbf{r}_i$. The representations are aggregated into a single representation $\mathbf{r} = \frac{1}{n}\sum \mathbf{r}_i$ which is concatenated to the target inputs $\mathbf{x}_t$ and passed to a decoder g consisting of a five layer MLP. The decoder outputs a Gaussian mean and variance for the target outputs $\hat{\mathbf{y}}_t$. We train the model to maximise the log-likelihood of the target points using the Adam optimiser of Kingma and Ba [2014].

Two examples of regression results obtained for each of the datasets are shown in Figure 5.4. We compare the model to the predictions generated by a GP with the correct hyperparameters, which constitutes an upper bound on our performance. Although the prediction generated by the GP is smoother than the CNP's prediction both for the mean and variance, the model is able to learn to regress from a few context points for both the fixed kernels and switching kernels. As the number of context points grows, the accuracy of the model improves and the approximated uncertainty of the model decreases. Crucially, we see the model learns to estimate its own uncertainty given the observations very accurately. Moreover it provides a good approximation that increases in accuracy as the number of context points increases.

Furthermore the model achieves similarly good performance on the switching kernel task. This type of regression task is not trivial for GPs whereas in our case we only have to change the dataset used for training.

5.3.2 Image Completion

We consider image completion as a regression task over functions in either $f : [0,1]^2 \rightarrow [0,1]$ for greyscale images, or $f : [0,1]^2 \rightarrow [0,1]^3$ for RGB images. The input $\mathbf{x}$ is the 2D pixel coordinates normalised to $[0,1]^2$, and the output $\mathbf{y}$ is either the greyscale intensity or a vector of the RGB intensities of the corresponding pixel. For this completion task we use exactly the same model architecture as for 1D function regression (with the exception of making the last layer 3-dimensional for RGB).

We test CNP on two different datasets: the MNIST handwritten digit database LeCun et al. [1998] and large-scale CelebFaces Attributes (CelebA) dataset Liu et al. [2015]. The model and training procedure are the same for both: at each step we select an image from the dataset and pick a subset of the pixels as observations. Conditioned on these, the model is trained to predict the values of all the pixels in the image (including the ones it has been conditioned on). Like in 1D regression, the model outputs a Gaussian mean

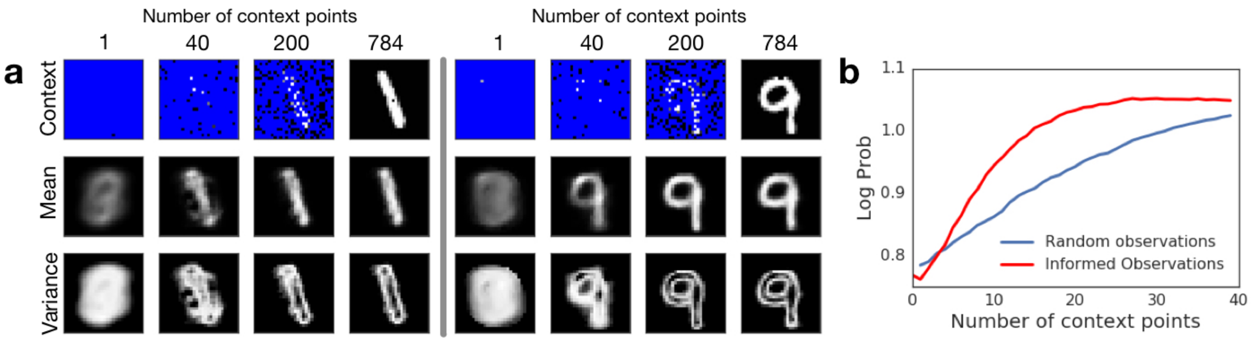


Figure 5.6: **Pixel-wise image regression on MNIST.** Left: Two examples of image regression with varying numbers of observations. We provide the model with 1, 40, 200 and 728 context points (top row) and query the entire image. The resulting mean (middle row) and variance (bottom row) at each pixel position is shown for each of the context images. Right: average model accuracy over 100 runs with increasing number of observations that are either chosen at random (blue) or by selecting the pixel with the highest variance (red). Figure adapted from [Garnelo et al., 2018a].

and variance for each pixel and is optimised with respect to the log-likelihood of the ground-truth image. It is important to point out that we choose images as our dataset because they constitute a complex 2-D function that is easy to evaluate visually, not to compare to generative models benchmarks.

MNIST

We first test CNP on the MNIST dataset and use the test set to evaluate its performance. As shown in Figure 5.6a the model learns to make good predictions of the underlying digit even for a small number of context points. Crucially, when conditioned only on one non-informative context point (e.g. a black pixel on the edge) the model's prediction corresponds to the average over all MNIST digits. As the number of context points increases, the predictions become more similar to the underlying ground truth. This demonstrates the model's capacity to extract dataset specific prior knowledge. It is worth mentioning that even with a complete set of observations the model does not achieve pixel-perfect reconstruction, as we have a bottleneck at the representation level.

An important aspect of the model is its ability to estimate the uncertainty of the prediction. As shown in the bottom row of Figure 5.6a, as we add more observations, the variance shifts from being almost uniformly spread over the digit positions to being localised around areas that are specific to the underlying digit, specifically its edges. Being able to model the uncertainty given some context can be helpful for many tasks. One example is active exploration, where the model has a choice over where to observe. We test this by comparing the predictions of CNP when the observations are chosen according to uncertainty (i.e. the pixel with the highest variance at each step), versus random pixels (Figure 5.6b). This method is a very simple way of doing active exploration, but it already produces better prediction results than selecting the conditioning points at random.

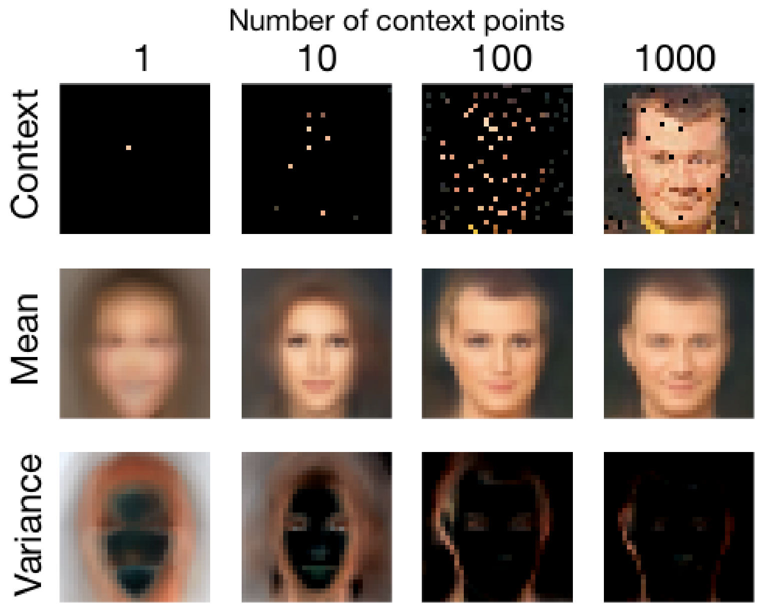


Figure 5.7: **Pixel-wise image completion on CelebA.** Two examples of CelebA image regression with varying numbers of observations. We provide the model with 1, 10, 100 and 1000 context points (top row) and query the entire image. The resulting mean (middle row) and variance (bottom row) at each pixel position is shown for each of the context images. Figure adapted from [Garnelo et al., 2018a].

CelebA

We also apply CNP to CelebA [Liu et al., 2015], a dataset of images of celebrity faces, and report performance obtained on the test set. As shown in Figure 5.7 our model is able to capture the complex shapes and colours of this dataset with predictions conditioned on less than 10% of the pixels being already close to ground truth. As before, given few context points the model averages over all possible faces, but as the number of context pairs increases the predictions capture image-specific details like face orientation and facial expression. Furthermore, as the number of context points increases the variance is shifted towards the edges in the image.

An important aspect of CNPs demonstrated in Figure 5.8 is its flexibility not only in the number of observations and targets it receives, but also with regards to their input values. It is interesting to compare this property to GPs on one hand, and to trained generative models van den Oord et al. [2016], Gregor et al. [2015] on the other hand.

The first type of flexibility can be seen when conditioning on subsets that the model has not encountered during training. Consider conditioning the model on one half of the image, for example. This forces the model to not only predict pixel values according to some stationary smoothness property of the images, but

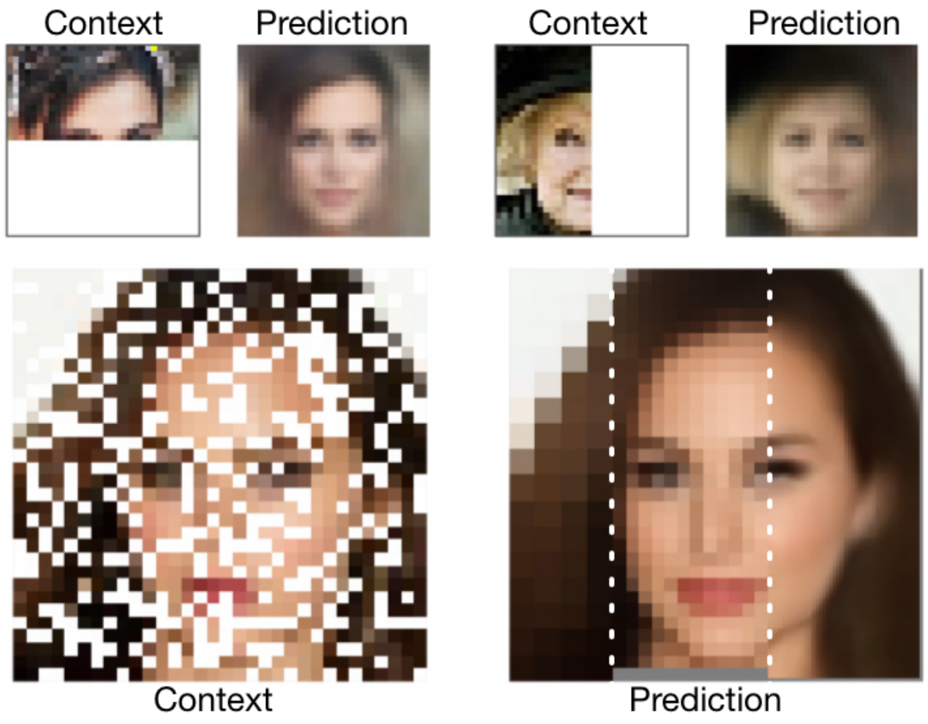


Figure 5.8: **Flexible image completion**. In contrast to standard conditional models, CNPs can be directly conditioned on observed pixels in arbitrary patterns, even ones which were never seen in the training set. Similarly, the model can predict values for pixel coordinates that were never included in the training set, like subpixel values in different resolutions. The dotted white lines were added for clarity after generation. Figure adapted from [Garnelo et al., 2018a].

also according to global spatial properties, e.g. symmetry and the relative location of different parts of faces. As seen in the first row of the figure, CNPs are able to capture those properties. A GP with a stationary kernel cannot capture this, and in the absence of observations would revert to its mean (the mean itself can be non-stationary but usually this would not be enough to capture the interesting properties).

In addition, the model is flexible with regards to the target input values. This means, e.g., we can query the model at resolutions it has not seen during training. We take a model that has only been trained using pixel coordinates of a specific resolution, and predict at test time subpixel values for targets between the original coordinates. As shown in Figure 5.8, with one forward pass we can query the model at different resolutions. While GPs also exhibit this type of flexibility, it is not the case for trained generative models, which can only predict values for the pixel coordinates on which they were trained. In this sense, CNPs capture the best of both worlds – it is flexible in regards to the conditioning and prediction task, and has the capacity to extract domain knowledge from a training set.

We compare CNPs quantitatively to two related models: kNNs and GPs (see Section 2.4.1 and 2.4.2 for an introduction to kNNs and GPs, respectively). As shown in Table 5.1, CNPs outperform the latter when number of context points is small (empirically when half of the image or less is provided as context). When the majority of the image is given as context exact methods like GPs and kNN will perform better. From the table we can also see that the order in which the context points are provided is less important for CNPs, since providing the context points in order from top to bottom still results in good performance. Both insights point to the fact that CNPs learn a data-specific ‘prior’ that will generate good samples even when the number of context points is very small.

	Random Context			Ordered Context		
#	10	100	1000	10	100	1000
kNN	0.215	0.052	0.007	0.370	0.273	0.007
GP	0.247	0.137	**0.001**	0.257	0.220	**0.002**
CNP	**0.039**	**0.016**	0.009	**0.057**	**0.047**	0.021

Table 5.1: Pixel-wise mean squared error for all of the pixels in the image completion task on the CelebA dataset with increasing number of context points (10, 100, 1000). The context points are chosen either at random or ordered from the top-left corner to the bottom-right. With fewer context points CNPs outperform kNNs and GPs. In addition CNPs perform well regardless of the order of the context points, whereas GPs and kNNs perform worse when the context is ordered.

5.3.3 Classification

The Few-shot Classification Task

So far we have been focusing on few-shot regression tasks. Conditional neural processes, however, can be easily extended to few-shot classification as well. As with regression the few-shot classification framework differs from the traditional deep learning classification regime.

Take image classification (Figure 5.9): traditionally every image is associated to a fixed label (e.g. images of dogs, are always labelled ‘dog’). In the case of few-shot classification this is different: at every training iteration a subset K of the available classes are selected and assigned the labels 1 to K at random. For the next iteration we sample a different set of K classes and re-label them. As a result the same class can have different labels at different iterations. The task is therefore not to learn to associate a label with an image, but to be able to compare the context images with the target images and determine which class the targets are most likely to be part of. If we look at the image vs class distribution (depicted on the side in blue in Figure 5.9) in traditional DL classification this distribution remains fixed, whereas in few-shot learning the distribution is different at every iteration. As before we can therefore claim that CNPs learn to approximate a distribution over functions rather than a single function.

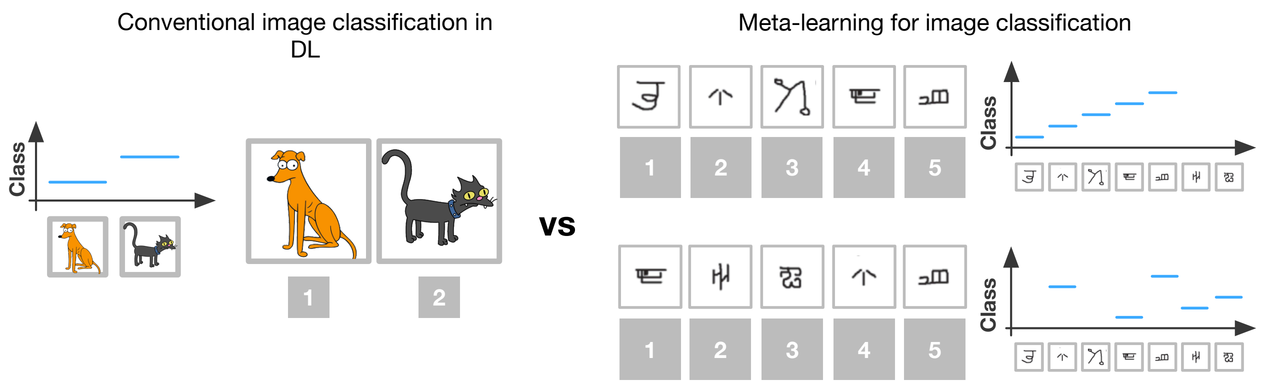

Conventional image classification in DL
Meta-learning for image classification
Class
1
2
vs
3
4
5

Figure 5.9: Conventional image classification versus few-shot classification. In the conventional training regime the labels for each image remain fixed. In the few-shot classification setup the labels of the images are changed at every iteration.

Few-Shot Classification of Omniglot Images

We apply the model to one-shot classification using the Omniglot dataset of Lake et al. [2015] (see Figure 5.10 for an overview of the task). This dataset consists of 1,623 classes of characters from 50 different alphabets. Each class has only 20 examples and as such this dataset is particularly suitable for few-shot learning algorithms. As in [Vinyals et al., 2016] we use 1,200 randomly selected classes as our training set and the remainder as our testing dataset. In addition we augment the dataset following the protocol described in Santoro et al. [2016]. This includes cropping the image from 32×32 to 28×28, applying small random translations and rotations to the inputs, and also increasing the number of classes by rotating every character by 90 degrees and defining that to be a new class. We generate the labels for an N-way classification task by choosing N random classes at each training step and arbitrarily assigning the labels $0, ..., N-1$ to each.

Given that the input points are images, we modify the architecture of the encoder h to include convolution layers as mentioned in section 5.2. In addition we only aggregate over inputs of the same class by using the information provided by the input label. The aggregated class-specific representations are then concatenated to form the final representation. Given that both the size of the class-specific representations and the number of classes are constant, the size of the final representation is still constant and thus the $\mathcal{O}(n+m)$ runtime still holds.

The results of the classification are summarised in Table 5.2. CNPs achieve higher accuracy than models that are significantly more complex (like MANN [Santoro et al., 2016]). While CNPs do not beat state-of-the-art for one-shot classification, our accuracy values are comparable. Crucially, we reach those values using a significantly simpler architecture (three convolutional layers for the encoder and a three-layer MLP for the decoder) and with a lower runtime of $\mathcal{O}(n+m)$ at test time as opposed to $\mathcal{O}(nm)$.

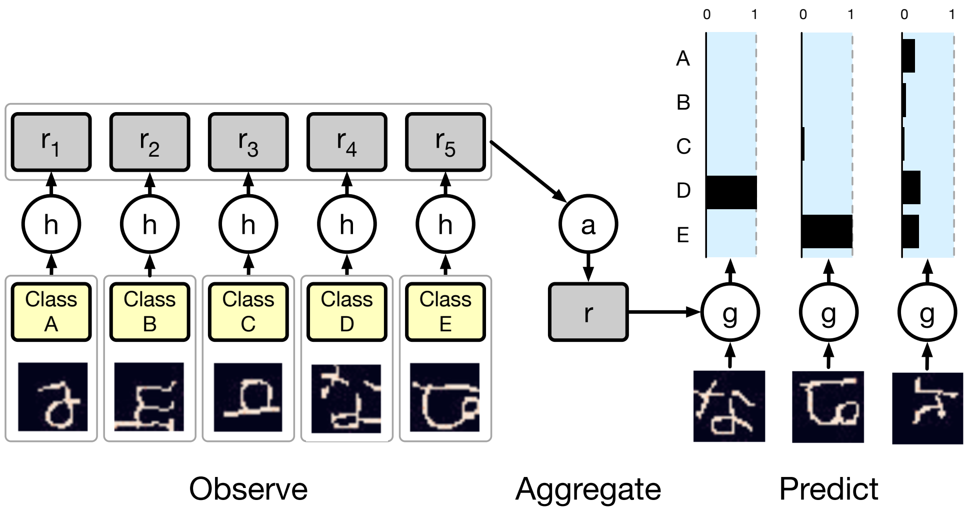


Figure 5.10: **One-shot Omniglot classification**. At test time the model is presented with a labelled example for each class, and outputs the classification probabilities of a new unlabelled example. The model uncertainty grows when the new example comes from an un-observed class. Figure adapted from [Garnelo et al., 2018a].

	5-way Acc		20-way Acc		Runtime
	1-shot	5-shot	1-shot	5-shot	
MANN	82.8%	94.9%	-	-	$\mathcal{O}(nm)$
MN	**98.1%**	**98.9%**	**93.8%**	**98.5%**	$\mathcal{O}(nm)$
CNP	95.3%	98.5%	89.9%	96.8%	$\mathcal{O}(n+m)$

Table 5.2: **Classification results on Omniglot**. Results on the same task for MANN [Santoro et al., 2016], and matching networks (MN) [Vinyals et al., 2016] and CNP.

5.4 Related research

5.4.1 Gaussian Processes

The goal of our research is to incorporate ideas from GP inference into a NN training regime to overcome certain drawbacks of both. There are a number of papers that address some of the same issues within the GP framework. Scaling issues with GPs have been addressed by sparse GPs [Snelson and Ghahramani, 2006], while overcoming the limited expressivity resulting from functional restrictions is the motivation for Deep GPs [Damianou and Lawrence, 2013, Salimbeni and Deisenroth, 2017]. The authors of Deep Kernel learning [Wilson et al., 2016], also combine ideas from DL and GPs. Their model, however, remains closer to GPs than Neural Processes as the neural network is used to learn more expressive kernels that are then used within a GP.

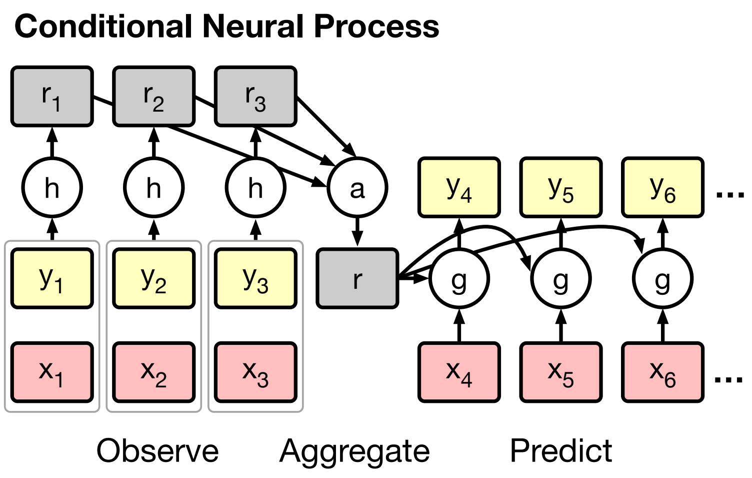


Figure 5.11: The CNP model. The context pairs $(\mathbf{x}^c, \mathbf{y}^c)$ are processed individually by an encoder h and aggregated by an aggregator a to obtain a single representation $\mathbf{r}$. This representation is fed into the decoder g along with the target query $\mathbf{x}^t$ to generate prediction $\mathbf{y}^t$. Figure adapted from [Garnelo et al., 2018a].

5.4.2 Meta-Learning

Deep learning models are generally more scalable than GPs and are very successful at learning features and prior knowledge from the data directly. However, they tend to be less flexible with regards to input size and order. Additionally, in general they only approximate one function as opposed to distributions over functions. Meta-learning approaches address the latter and share our core motivations. Recently meta-learning has been applied to a wide range of tasks such as RL [Wang et al., 2016, Finn et al., 2017] and program induction [Devlin et al., 2017].

More related to Neural Processes are meta-learning algorithms which are implemented as deep generative models that learn to do few-shot estimations of the underlying density of the data. One way to achieve this is by updating existing models like PixelCNN [van den Oord et al., 2016] and augmenting them with attention mechanisms [Reed et al., 2018] or including a memory unit in a VAE model [Bornschein et al., 2017]. Another successful latent variable approach is to explicitly condition on some context during inference [J. Rezende et al., 2016]. Given the generative nature of these models, they are usually applied to image generation tasks, but models that include a conditioning class-variable can be used for classification as well. In general these approaches have not been used in the context of regression.

Classification itself is another common task in meta-learning. Few-shot classification algorithms usually rely on some distance metric in feature space to compare target images to the observations provided [Koch et al., 2015], [Santoro et al., 2016]. Matching networks (MNs) [Vinyals et al., 2016, Bartunov and Vetrov, 2017] are closely related to CNPs. In their case, features of samples are compared with target features using an attention kernel. At a higher level, one can interpret this model as a CNP where the aggregator is just the concatenation over all input samples and the decoder g contains an explicitly defined distance kernel. In this sense matching networks are closer to GPs than to CNPs, since they require the specification of a distance kernel that CNPs learn from the data instead. In addition, as MNs carry out all-to-all comparisons they scale with $\mathcal{O}(n \times m)$, although they can be modified to have the same complexity of $\mathcal{O}(n+m)$ as CNPs [Snell et al., 2017].

A model that is conceptually very similar to CNPs (and in particular the latent variable version) is the "neural statistician" [Edwards and Storkey, 2017] and the related variational homoencoder [Hewitt et al., 2018]. As with the other generative models, the neural statistician learns to estimate the density of the observed data but does not allow for targeted sampling at what we have been referring to as input positions x_i. Instead, it can only generate i.i.d. samples from the estimated density.

5.5 Discussion

In this chapter we have introduced Conditional Neural Processes (CNPs), a model for few-shot regression and classification that is both flexible at test time and has the capacity to extract prior knowledge from training data. We have demonstrated its ability to perform a variety of tasks including regression, classification and image completion. We compared CNPs to Gaussian Processes on one hand, and deep learning methods on the other, and also discussed the relation to meta-learning and few-shot learning. As such, this work can be seen as a step towards learning high-level abstractions, one of the grand challenges of contemporary machine learning. Functions learned by most conventional deep learning models are tied to a specific, constrained statistical context at any stage of training. A trained CNP is more general, in that it encapsulates the high-level statistics of a family of functions, that can be reused for multiple tasks.

Despite these advantages CNPs remain a deterministic model, which means it can only generate one prediction for each set of context points. In the following chapter we overcome this shortcoming by introducing a latent variable extension of CNPs we call Neural Processes (NPs). In addition to making it possible to generate multiple samples the latent variable in NPs allows us to draw parallels to stochastic processes and a range of conditional latent variable models.

Chapter 6

Neural Processes

6.1 Introduction

In the previous chapter we introduced Conditional Neural Processes (CNPs) as flexible meta-learning models for few-shot regression and classification. We showed that CNPs learn to model a whole distribution over functions rather than a single function, which allows them to adapt their prediction to the observed context points even once the training phase is over. In addition, CNPs can capture the uncertainty of their own estimates and are computationally cheap to evaluate.

Despite these benefits, CNPs fundamentally can only learn a deterministic mapping from observations to targets. Take the example of image completion introduced in the previous chapter (Figure 5.5). For a fixed set of context points CNPs can only generate a single prediction for each pixel and therefore we can only generate one image. If we wanted to sample different images that agreed with the observed context points, the only distribution we could sample from are the output distributions parametrised by the means and variances produced by the decoder. These distributions, however, are generated independently for each target point and thus the values of different targets are not correlated (the covariance of the joint distribution over all target points is therefore a diagonal rather than a full covariance matrix). Sampling from these independent output distributions results in noisy variations of the mean image (see Figure 6.1) instead of different coherent images.

There are a number of possible alternatives to obtaining varied, coherent samples. One could build a model that learns a covariance matrix rather than just the independent variances of the target points. This approach would be more similar to a Gaussian process than the original CNP. However, the dependency of each target point on every other target point would result in a higher computational cost and would reduce the flexibility of the decoder. Another option would be to generate the target points auto-regressively, i.e. generate

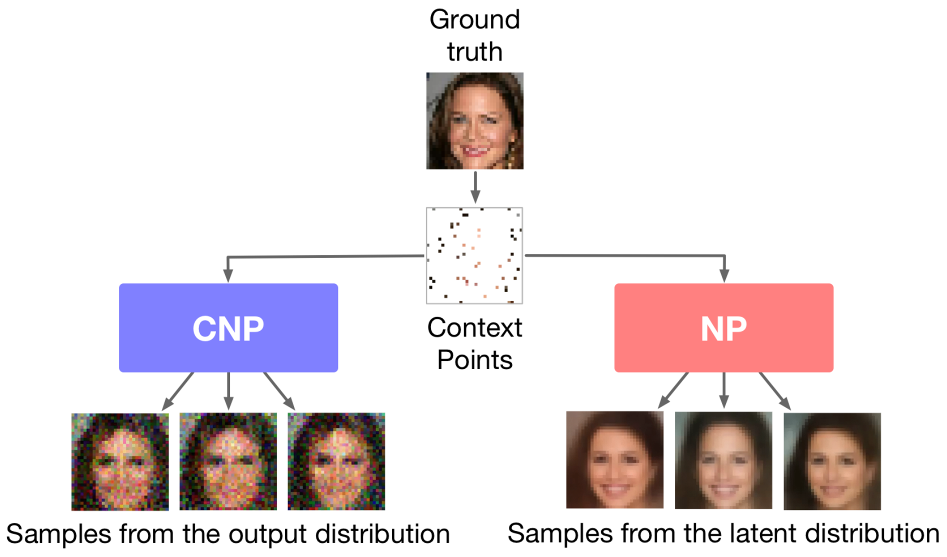

Ground truth
Context Points
CNP
NP
Samples from the output distribution
Samples from the latent distribution

Figure 6.1: Generating samples using a CNP versus an NP. The only way to sample a distribution with a CNP is to sample the independent output distributions. Because these are not correlated the resulting samples are just noisy variations of the mean image. By sampling the latent variable of the NP the mean values for the different samples will generate different coherent images that agree with the context observations.

one target point at a time conditioned on all the target points predicted so far. While this approach has proven very successful [van den Oord et al., 2016], auto-regressive models don't lend themselves as well to parallelisation and tend to be slower.

In this chapter we opt for a different approach: we introduce a latent variable $\mathbf{z}$ parametrised by the representation generated by the encoder. Each time we want to sample a different consistent set of targets (e.g. a new, consistent image) we sample one $\mathbf{z}$ from the latent distribution and condition all the target points on it. The images generated using this approach are shown in Figure 6.1. We refer to this model as Neural Processes (NPs).

In addition to enabling coherent sampling the latent variable also makes it easier to draw parallels to stochastic processes and conditional latent variable models. This allows us to compare models across different research areas, which we carry out in the main model section and the related work section.

The chapter is structured as follows:

- In Section 6.2 we introduce the model and discuss how NPs are related to stochastic processes.
- We compare our model to related work from different areas in Section 6.5.
- In Section 6.3 we introduce a number of different regression experiments carried out using NPs and report the results obtained on a range of different tasks.
- Finally, we summarise and discuss the contributions of this chapter in Section 6.6.

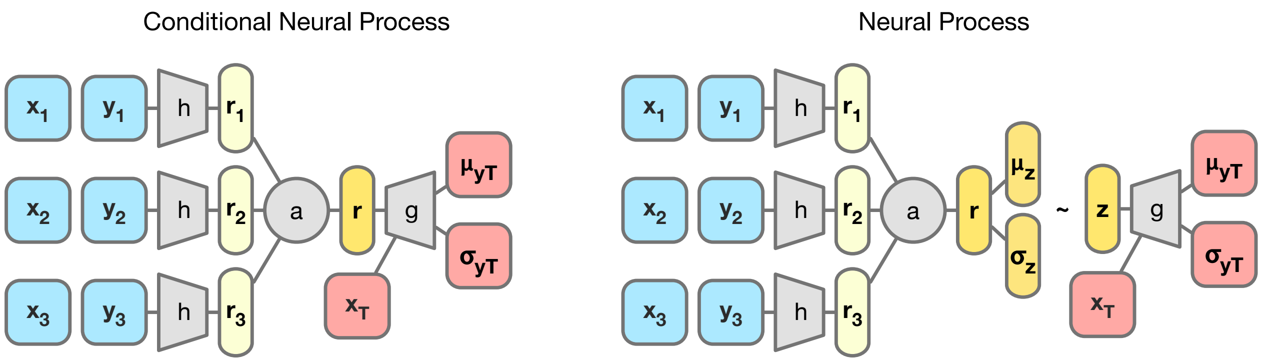


Figure 6.2: The CNP model (left) and the NP model (right). The main difference is the latent variable parametrised by the representation.

6.2 Model

6.2.1 The neural process model

Neural Processes (NP) and Conditional Neural Processes (CNP) share a large part of their architecture. The main difference between the two is the handling of the intermediate representation (see Figure 6.2). CNPs just pass this representation $\mathbf{r}$ generated by the encoder to the decoder along with the target input $\mathbf{x}^t$. NPs on the other hand use $\mathbf{r}$ to parametrise the mean and variance of a multivariate Gaussian $\mathcal{N}(\mathbf{z}|\mu(\mathbf{r}), I\sigma(\mathbf{r}))$ which is used to sample $\mathbf{z}$. The same $\mathbf{z}$ is used to predict all the target outputs $\mathbf{y}^t$ from the target inputs $\mathbf{x}^t$. As with CNPs the model of NPs can be boiled down to three core components:

- An **encoder** h from input space into representation space that takes in *pairs* of $(\mathbf{x}^c, \mathbf{y}^c)_i$ context values and produces a representation $\mathbf{r}_i = h((\mathbf{x}^c, \mathbf{y}^c)_i)$ for each of the pairs. We parameterise h as a neural network.
- An **aggregator** a that summarises the encoded inputs. We are interested in obtaining a single order-invariant global representation $\mathbf{r}$ that parameterises the latent distribution $z \sim \mathcal{N}(\mu(\mathbf{r}), I\sigma(\mathbf{r}))$. The simplest operation that ensures order-invariance and works well in practice is the mean function $\mathbf{r} = a(\mathbf{r}_i) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{r}_i$. Crucially, the aggregator reduces the runtime to $\mathcal{O}(n+m)$ where n and m are the number of context and target points respectively.
- A **conditional decoder** g that takes as input the sampled global latent variable $\mathbf{z}$ as well as the new target locations $\mathbf{x}^t$ and outputs the predictions $\hat{\mathbf{y}}^t$ for the corresponding values of $f(\mathbf{x}^t) = \mathbf{y}^t$.

6.2.2 Neural processes as stochastic processes

The additional latent variable allows us to formulate NPs as approximations of stochastic processes. In the following we define stochastic processes and derive the NP model from this definition.

The standard approach to defining an infinite-dimensional stochastic process is via its finite-dimensional marginal distributions. Specifically, we consider the process as a random function $F : \mathcal{X} \to \mathcal{Y}$ and, for each finite sequence $\mathbf{x}_{1:n} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$ with $\mathbf{x}_i \in \mathcal{X}$, we define the marginal joint distribution over the function values $\mathbf{y}_{1:n} := (F(\mathbf{x}_1), \ldots, F(\mathbf{x}_n))$. For example, in the case of GPs, these joint distributions are multivariate Gaussians parameterised by a mean and a covariance function.

Given a collection of joint distributions $\rho_{\mathbf{x}_{1:n}}$ we can derive two necessary conditions to be able to define a stochastic process F such that $\rho_{\mathbf{x}_{1:n}}$ is the marginal distribution of $(F(\mathbf{x}_1), \ldots, F(\mathbf{x}_n))$, for each finite sequence $\mathbf{x}_{1:n}$. These conditions are: (finite) exchangeability and consistency. As stated by the Kolmogorov Extension Theorem [Øksendal, 2003], these conditions are sufficient to define a stochastic process.

Exchangeability This condition requires the joint distributions to be invariant to permutations of the elements in $\mathbf{x}_{1:n}$. More precisely, for each finite n, if ψ is a permutation of $\{1, \ldots, n\}$, then:

$$\begin{aligned} \rho_{\mathbf{x}_{1:n}}(\mathbf{y}_{1:n}) &:= \rho_{\mathbf{x}_1,\ldots,\mathbf{x}_n}(\mathbf{y}_1, \ldots, \mathbf{y}_n) \\ &= \rho_{\mathbf{x}_{\psi(1)},\ldots,\mathbf{x}_{\psi(n)}}(\mathbf{y}_{\psi(1)}, \ldots, \mathbf{y}_{\psi(n)}) =: \rho_{\psi(\mathbf{x}_{1:n})}(\psi(\mathbf{y}_{1:n})) \end{aligned} \tag{6.1}$$

where $\psi(\mathbf{x}_{1:n}) := (\mathbf{x}_{\psi(1)}, \ldots, \mathbf{x}_{\psi(n)})$ and $\psi(\mathbf{y}_{1:n}) := (\mathbf{y}_{\psi(1)}, \ldots, \mathbf{y}_{\psi(n)})$.

Consistency If we marginalise out a part of the sequence the resulting marginal distribution is the same as that defined on the original sequence. More precisely, if $1 \leq m \leq n$, then:

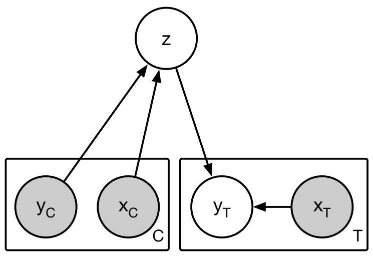
$$\rho_{\mathbf{x}_{1:m}}(\mathbf{y}_{1:m}) = \int \rho_{\mathbf{x}_{1:n}}(\mathbf{y}_{1:n}) d\mathbf{y}_{m+1:n}. \tag{6.2}$$

Take, for example, three different sequences $\mathbf{x}_{1:n}, \psi(\mathbf{x}_{1:n})$ and $\mathbf{x}_{1:m}$ as well as their corresponding joint distributions $\rho_{\mathbf{x}_{1:n}}, \rho_{\psi(\mathbf{x}_{1:n})}$ and $\rho_{\mathbf{x}_{1:m}}$. In order for these joint distributions to all be marginals of some higher-dimensional distribution given by the stochastic process F, they have to satisfy equations 6.1 and 6.2 above.

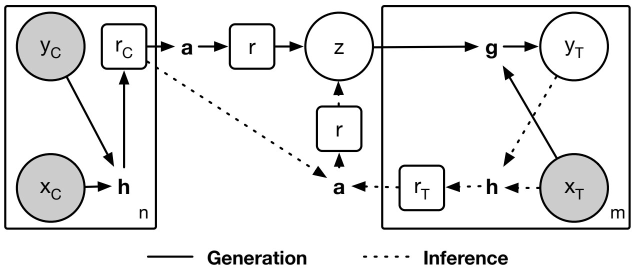
Given a particular instantiation of the stochastic process f the joint distribution is defined as:

$$\rho_{\mathbf{x}_{1:n}}(\mathbf{y}_{1:n}) = \int p(f) p(\mathbf{y}_{1:n} | f, \mathbf{x}_{1:n}) df. \tag{6.3}$$

Here p denotes the abstract probability distribution over all random quantities. To satisfy both conditions



(a) Graphical model



(b) Computational diagram

Figure 6.3: **Neural process model. (a)** Graphical model of a neural process. x and y correspond to the data where $y = f(x)$. C and T are the number of context points and target points respectively and z is the global latent variable. A grey background indicates that the variable is observed. **(b)** Diagram of our neural process implementation. Variables in circles correspond to the variables of the graphical model in (a), variables in square boxes to the intermediate representations of NPs and unbound, bold letters to the following computation modules: h - encoder, a - aggregator and g - decoder. In our implementation h and g correspond to neural networks and a to the mean function. The continuous lines depict the generative process, the dotted lines the inference. Figure adapted from [Garnelo et al., 2018b].

above we define the joint probability $p(\mathbf{y}_{1:n}|f, \mathbf{x}_{1:n})$ to be i.i.d. Gaussians $\mathbf{y}_i \sim \mathcal{N}(F(\mathbf{x}_i), \sigma^2)$ with some observation noise:

$$p(\mathbf{y}_{1:n}|f, \mathbf{x}_{1:n}) = \prod_{i=1}^{n} \mathcal{N}(\mathbf{y}_i|f(\mathbf{x}_i), \sigma^2). \tag{6.4}$$

Inserting this into equation 6.3 the stochastic process is specified by:

$$\rho_{\mathbf{x}_{1:n}}(\mathbf{y}_{1:n}) = \int p(f) \prod_{i=1}^{n} \mathcal{N}(\mathbf{y}_i|f(\mathbf{x}_i), \sigma^2) df. \tag{6.5}$$

According to de Finetti's Theorem De Finetti [1937] given that $\mathbf{y}_{1:n}$ are conditionally independent given f the exchangability condition is met. In addition by defining the joint as the product of i.i.d variables, we ensure that the consistency condition also holds. In other words, exchangeability and consistency of the collection of joint distributions $\{\rho_{\mathbf{x}_{1:n}}\}$ imply the existence of a stochastic process F such that the observations $\mathbf{y}_{1:n}$ become i.i.d. conditional upon F.

In order to represent a stochastic process using a NP, we will approximate it with a neural network, and assume that F can be parameterised by a high-dimensional random vector $\mathbf{z}$, and write $F(\mathbf{x}) = g(\mathbf{x}, \mathbf{z})$ for some fixed and learnable function g (i.e. the randomness in F is due to that of $\mathbf{z}$). The generative model (Figure 6.3a) then follows from (6.5):

$$p(\mathbf{z}, \mathbf{y}_{1:n}|\mathbf{x}_{1:n}) = p(\mathbf{z}) \prod_{i=1}^{n} \mathcal{N}(\mathbf{y}_i|g(\mathbf{x}_i, \mathbf{z}), \sigma^2) \tag{6.6}$$

where, following ideas of variational auto-encoders, we assume $p(\mathbf{z})$ is a multivariate standard normal, and $g(\mathbf{x}_i, \mathbf{z})$ is a neural network which captures the complexities of the model.

6.2.3 Global latent variable

As mentioned above, neural processes include a latent variable $\mathbf{z}$ that captures F. This latent variable is of particular interest because it captures the global uncertainty, which allows us to sample at a global level – one function f at a time, rather than at a local output level – one $\mathbf{y}_i$ value for each $\mathbf{x}_i$ at a time (independently of the remaining $\mathbf{y}^t$).

In addition, since we are passing all of the context's information through this single variable, we can formulate the model in a Bayesian framework. In the absence of context points C the latent distribution $p(\mathbf{z})$ would correspond to a data specific prior the model has learned during training. As we add observations the latent distribution encoded by the model amounts to the posterior $p(\mathbf{z}|C)$ over the function given the context. On top of this, as shown in equation 6.7, instead of using a zero-information prior $p(\mathbf{z})$, we condition the prior on the context. As such this prior is equivalent to a less informed posterior of the underlying function. This formulation makes it clear that the posterior given a subset of the context points will serve as the prior when additional context points are included. By using this setup, and training with different sizes of context, we encourage the learned model to be flexible with regards to the number and position of the context points.

6.2.4 Training Neural Processes

As before, we need a training procedure that reflects the task of representing a distribution over functions rather than a single function. More formally, to train a NP we form a dataset that consists of functions $f : X \to Y$ that are sampled from some underlying data distribution $\mathcal{D}$. As an illustration consider a dataset consisting of functions $f_{\mathcal{GP}}(x) \sim \mathcal{GP}$ that have been generated using a Gaussian process with a fixed kernel. For each of the functions $f_{\mathcal{GP}}(x)$ our dataset contains a number of $(\mathbf{x}, \mathbf{y})_i$ tuples where $\mathbf{y}_i = f_{\mathcal{GP}}(\mathbf{x}_i)$. For training purposes we divide these points into a set of n context points $C = \{(\mathbf{x}^c, \mathbf{y}^c)_i\}_{i=1}^n$ and a set of $m = n + n'$ target points which consists of all points in C as well as n' additional unobserved points $T = \{(\mathbf{x}^t, \mathbf{y}^t)_i\}_{i=1}^m$. During testing the model is presented with some context C and has to predict the target values $\mathbf{y}^t = f(\mathbf{x}^t)$ at target positions $\mathbf{x}^t$.

In order to be able to predict accurately across the entire dataset a model needs to learn a distribution that covers all of the functions observed in training and be able to take into account the context data at test time.

Since the decoder g is non-linear, we can use amortised variational inference to learn it. Let $q(\mathbf{z}|\mathbf{x}^t, \mathbf{y}^t)$ be a variational posterior of the latent variables $\mathbf{z}$, parameterised by another neural network that is invariant to permutations of the sequences $\mathbf{x}^t$ and $\mathbf{y}^t$. Then the evidence lower-bound (ELBO) is given by:

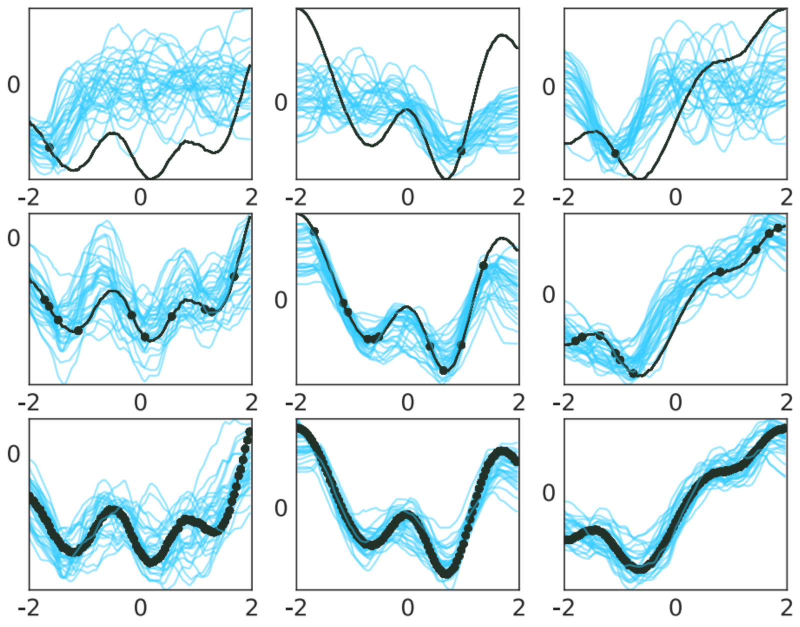


Figure 6.4: **1-D function regression.** The plots show samples of curves conditioned on an increasing number of context points (1, 10 and 100 in each row respectively). The true underlying curve is shown in black and the context points as black circles. Each column corresponds to a different example. With only one observation the variance away from that context point is high between the sampled curves. As the number of context points increases the sampled curves increasingly resemble the ground truth curve and the overall variance is reduced. Figure adapted from [Garnelo et al., 2018b].

$$\log p(\mathbf{y}^t|\mathbf{x}^c, \mathbf{y}^c, \mathbf{x}^t) \geq \mathbb{E}_{q(\mathbf{z}|\mathbf{x}^c,\mathbf{y}^c)}\left[\sum_{i=1}^{m} \log p(\mathbf{y}_i^t|\mathbf{z}, \mathbf{x}_i^t) + \log \frac{p(\mathbf{z}|\mathbf{x}^c, \mathbf{y}^c)}{q(\mathbf{z}|\mathbf{x}^t, \mathbf{y}^t)}\right] \tag{6.7}$$

Since there are no restrictions on how we can define the prior, we use the same encoder h that is used to generate the approximate posterior $q(\mathbf{z}|\mathbf{x}^t, \mathbf{y}^t)$ to encode $p(\mathbf{z}|\mathbf{x}^c, \mathbf{y}^c)$. To make it clear that the prior and the posterior are obtained using the same network we can express the ELBO as:

$$\log p(\mathbf{y}^t|\mathbf{x}^c, \mathbf{y}^c, \mathbf{x}^t) \geq \mathbb{E}_{q(\mathbf{z}|\mathbf{x}^c,\mathbf{y}^c)}\left[\sum_{i=1}^{m} \log p(\mathbf{y}_i^t|\mathbf{z}, \mathbf{x}_i^t) + \log \frac{q(\mathbf{z}|\mathbf{x}^c, \mathbf{y}^c)}{q(\mathbf{z}|\mathbf{x}^t, \mathbf{y}^t)}\right] \tag{6.8}$$

6.3 Results

6.3.1 1-D function regression

In order to test whether neural processes indeed learn to model distributions over functions we first apply them to a 1-D function regression task. The functions for this experiment are generated using a GP with varying kernel parameters for each function. At every training step we sample a set of values for the Gaussian

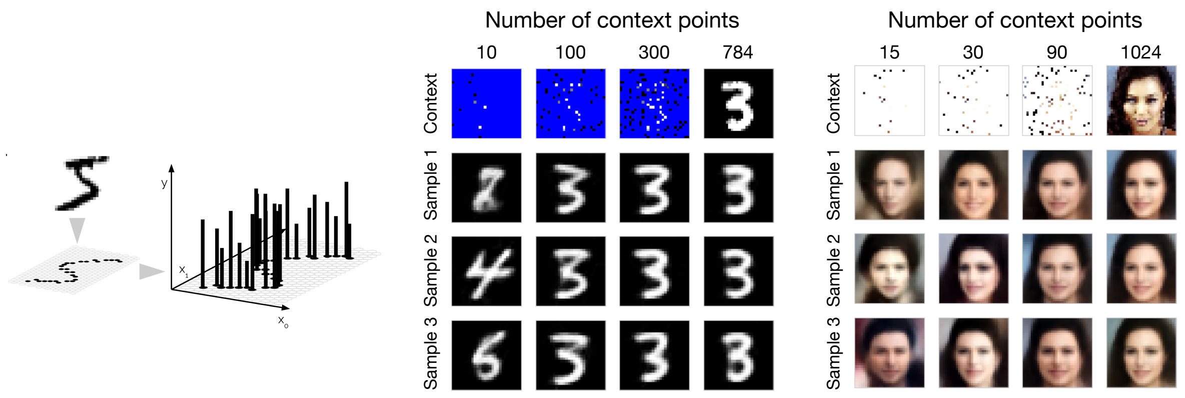
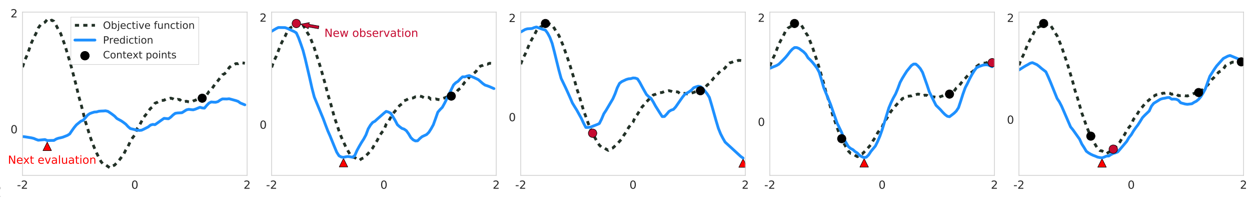


Figure 6.5: **Pixel-wise regression on MNIST and CelebA** The diagram on the left visualises how pixel-wise image completion can be framed as a 2-D regression task where f(pixel coordinates) = pixel brightness. The figures to the right of the diagram show the results on image completion for MNIST and CelebA. The images on the top correspond to the context points provided to the model. For better clarity the unobserved pixels have been coloured blue for the MNIST images and white for CelebA. Each of the rows corresponds to a different sample given the context points. As the number of context points increases the predicted pixels get closer to the underlying ones and the variance across samples decreases. Figure adapted from [Garnelo et al., 2018b].

kernel of a GP and use those to sample a function $f_D(\mathbf{x})$. A random number of the $C = (\mathbf{x}, \mathbf{y})$ pairs are passed into the decoder of the NP as context points. We pick additional target pairs $T = (\mathbf{x}, \mathbf{y})$ which we combine with the observed context points C as targets and feed $\mathbf{x}^t$ to the decoder that returns its estimate $\hat{\mathbf{y}}^t$ of the underlying value of $\mathbf{y}^t$.

Some sample curves are shown in Figure 6.4. For the same underlying ground truth curve (black line) we run the neural process using varying numbers of context points and generate several samples for each run (light-blue lines). As evidenced by the results the model has learned some key properties of the 1-D curves from the data such as continuity and the general shape of functions sampled from a GP with a Gaussian kernel. When provided with only one context point the model generates curves that fluctuate around 0, the prior of the data-generating GP. Crucially, these curves go through or near the observed context point and display a higher variance in regions where no observations are present. As the number of context points increases this uncertainty is reduced and the model's predictions better match the underlying ground truth. Given that this is a neural approximation the curves will sometimes only approach the observations points as opposed to go through them as it is the case for GPs. On the other hand once the model is trained it can regress more than just one dataset i.e. it will produce sensible results for curves generated using any kernel parameters observed during training.



*2*2

Figure 6.6: **Thompson sampling with neural processes on a 1-D objective function.** The plots show the optimisation process over five iterations. Each prediction function (blue) is drawn by sampling a latent variable conditioned on an incresing number of context points (black circles). The underlying ground truth function is depicted as a black dotted line. The red triangle indicates the next evaluation point which corresponds to the minimum value of the sampled NP curve. The red circle in the following iteration corresponds to this evaluation point with its underlying ground truth value that serves as a new context point to the NP. Figure adapted from [Garnelo et al., 2018b].

6.3.2 2-D function regression

One of the benefits of neural processes is their functional flexibility as they can learn non-trivial 'kernels' from the data directly. In order to test this we apply NPs to a more complex regression problem. We carry out image completion as a regression task, where we provide some of the pixels as context and do pixel-wise prediction over the entire image. In this formulation the $\mathbf{x}_i$ values would correspond to the Cartesian coordinates of each pixel and the $\mathbf{y}_i$ values to the pixel intensity (see Figure 6.5 for an explanation of this). It is important to point out that we choose images as our dataset because they constitute a complex 2-D function and they are easy to evaluate visually. It is important to point out that NPs, as such, have not been designed for image generation like other specialised generative models.

We train separate models on the MNIST [LeCun et al., 1998] and the CelebA [Liu et al., 2015] datasets. As shown in Figure 6.5 the model performs well on both tasks. In the case of the MNIST digits the uncertainty is reflected in the variability of the generated digit. Given only a few context points more than just one digit can fit the observations and as a result the model produces different digits when sampled several times. As the number of context points increases the set of possible digits is reduced and the model produces the same digit, albeit with structural modifications that become smaller as the number of context points increases.

The same holds for the CelebA dataset. In this case, when provided limited context the model samples from a wider range of possible faces and as it observes more context points it converges towards very similar looking faces. We do not expect the model to reconstruct the target image perfectly even when all the pixels are provided as context, since the latent variable $\mathbf{z}$ constitutes a strong bottleneck. This can be seen in the final column of the figure where the predicted images are not only not identical to the ground truth but also vary between themselves. The latter is likely a cause of the latent variance which has been clipped to a small value to avoid collapsing, so even when no uncertainty is present we can generate different samples from $p(\mathbf{z}|C)$.

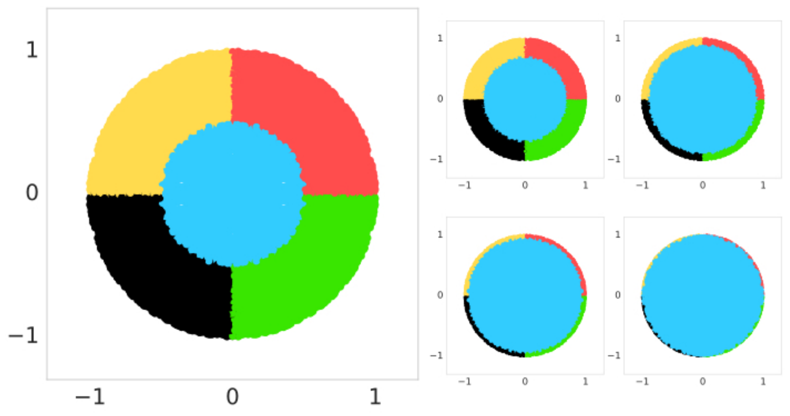


Figure 6.7: The wheel bandit problem with varying values of δ. Figure adapted from [Garnelo et al., 2018b].

6.3.3 Black-box optimisation with Thompson sampling

To showcase the utility of sampling entire consistent trajectories we apply neural processes to Bayesian optimisation on 1-D function using Thompson sampling Thompson [1933]. Thompson sampling (also known as randomised probability matching) is an approach to tackle the exploration-exploitation dilemma by maintaining a posterior distribution over model parameters. A decision is taken by drawing a sample of model parameters and acting greedily under the resulting policy. The posterior distribution is then updated and the process is repeated. Despite its simplicity, Thompson sampling has been shown to be highly effective both empirically and in theory. It is commonly applied to black box optimisation and multi-armed bandit problems [e.g. Agrawal and Goyal, 2012, Shahriari et al., 2016].

Neural processes lend themselves naturally to Thompson sampling by instead drawing a function over the space of interest, finding its minimum and adding the observed outcome to the context set for the next iteration. As shown in in Figure 6.4, function draws show high variance when only few observations are available, modelling uncertainty in a similar way to draws from a posterior over parameters given a small dataset. An example of this procedure for neural processes on a 1-D objective function is shown in Figure 6.6.

We report the average number of steps required by an NP to reach the global minimum of a function generated from a GP prior in Table 6.1. For an easier comparison the values are normalised by the amount of steps required when doing optimisation using random search. On average, NPs take four times fewer iterations than random search on this task. An upper bound on performance is given by a Gaussian process with the same kernel than the GP that generated the function to be optimised. NPs do not reach this optimal performance, as their samples are more noisy than those of a GP, but are faster to evaluate since merely a forward pass through the network is needed. This difference in computational speed is bound to get more

Neural process	**Gaussian process**	**Random Search**
0.26	0.14	1.00

Table 6.1: **Bayesian Optimisation using Thompson sampling.** Average number of optimisation steps needed to reach the global minimum of a 1-D function generated by a Gaussian process. The values are normalised by the number of steps taken using random search. The performance of the Gaussian process with the correct kernel constitutes an upper bound on performance.

notable as the dimensionality of the problem and the number of necessary function evaluations increases.

δ	**0.5**	**0.7**	**0.9**	**0.95**	**0.99**
Cumulative regret					
Uniform	100.00 ± 0.08	100.00 ± 0.09	100.00 ± 0.25	100.00 ± 0.37	100.00 ± 0.78
LinGreedy ($\epsilon = 0.0$)	65.89 ± 4.90	71.71 ± 4.31	108.86 ± 3.10	102.80 ± 3.06	104.80 ± 0.91
Dropout	7.89 ± 1.51	9.03 ± 2.58	36.58 ± 3.62	63.12 ± 4.26	98.68 ± 1.59
LinGreedy ($\epsilon = 0.05$)	7.86 ± 0.27	9.58 ± 0.35	19.42 ± 0.78	33.06 ± 2.06	74.17 ± 1.63
Bayes by Backprob Blundell et al. [2015]	1.37 ± 0.07	3.32 ± 0.80	34.42 ± 5.50	59.04 ± 5.59	97.38 ± 2.66
NeuralLinear	**0.95** ± 0.02	**1.60** ± 0.03	4.65 ± 0.18	9.56 ± 0.36	49.63 ± 2.41
MAML Finn et al. [2017]	2.95 ± 0.12	3.11 ± 0.16	4.84 ± 0.22	7.01 ± 0.33	22.93 ± 1.57
Neural Processes	1.60 ± 0.06	1.75 ± 0.05	**3.31** ± 0.10	**5.71** ± 0.24	**22.13** ± 1.23
Simple regret					
Uniform	100.00 ± 0.45	100.00 ± 0.78	100.00 ± 1.18	100.00 ± 2.21	100.00 ± 4.21
LinGreedy ($\epsilon = 0.0$)	66.59 ± 5.02	73.06 ± 4.55	108.56 ± 3.65	105.01 ± 3.59	105.19 ± 4.14
Dropout	6.57 ± 1.48	6.37 ± 2.53	35.02 ± 3.94	59.45 ± 4.74	102.12 ± 4.76
LinGreedy ($\epsilon = 0.05$)	5.53 ± 0.19	6.07 ± 0.24	8.49 ± 0.47	12.65 ± 1.12	57.62 ± 3.57
Bayes by Backprob Blundell et al. [2015]	0.60 ± 0.09	1.45 ± 0.61	27.03 ± 6.19	56.64 ± 6.36	102.96 ± 5.93
NeuralLinear	**0.33** ± 0.04	**0.79** ± 0.07	**2.17** ± 0.14	**4.08** ± 0.20	35.89 ± 2.98
MAML Finn et al. [2017]	2.49 ± 0.12	3.00 ± 0.35	4.75 ± 0.48	7.10 ± 0.77	22.89 ± 1.41
Neural Processes	1.04 ± 0.06	1.26 ± 0.21	2.90 ± 0.35	5.45 ± 0.47	**21.45** ± 1.3

Table 6.2: Results on the wheel bandit problem for increasing values of δ. Shown are mean and standard errors for both cumulative and simple regret (a measure of the quality of the final policy) over 100 trials. Results normalised wrt. the performance of a uniform agent. Table adapted from [Garnelo et al., 2018b].

6.4 Contextual bandits[1]

Finally, we apply neural processes to the wheel bandit problem introduced in Riquelme et al. [2018], which constitutes a contextual bandit task on the unit circle with varying needs for exploration that can be smoothly parametrised.

The problem can be summarised as follows: the unit circle is divided into five areas (see Figure 6.7). The blue area in the center is a low-reward region and the other four areas around it are the high-reward areas. As shown in Figure 6.7 the size of the areas can vary depending on the radius δ of the blue center area. This value δ is sampled anew at the start of every episode. Once the radius has been established the agent is given

[1]These experiments were run by Jonathan Schwartz.

the coordinates $X = (x_1, x_2)$ of some point within the circle and has to choose which of the $k = 5$ arms it wants to pull. The reward obtained for pulling a certain arm will depend on which area the coordinates fall into. If the point lies anywhere on the blue center area (so $||X|| \leq \delta$) then:

- **Arm 1** returns a reward drawn from $R \sim \mathcal{N}(1.2, 0.01^2)$
- **Arms 2-5** return a reward drawn from $R \sim \mathcal{N}(1.0, 0.01^2)$

The optimal action in this case is therefore to pull arm 1. If the point falls within any of the remaining areas the reward depends on the area. For the coordinates in area k the arms will return:

- **Arm 1** returns a reward drawn from $R \sim \mathcal{N}(1.2, 0.01^2)$
- **Arm k** returns a reward drawn from $R \sim \mathcal{N}(50.0, 0.01^2)$
- **Arms 2-5 except k** return a reward drawn from $R \sim \mathcal{N}(1.0, 0.01^2)$

However, the radius δ is unknown to the agent, so it has to figure out the size of the areas by trial-and-error in order to be able to choose the optimal arm.

We compare our model to a large range of methods that can be used for Thompson sampling, taking results from Riquelme et al. [2018], who kindly agreed to share the experiment and evaluation code with us. Neural Processes can be applied to this problem by training on a distribution of tasks before applying the method. Since the methods described in Riquelme et al. [2018] do not require such a pre-training phase, we also include Model-agnostic meta-learning (MAML, Finn et al. [2017]), a method relying on a similar pre-training phase, using code made available by the authors. For both NPs and MAML methods, we create a batch for pre-training by first sampling M different wheel problems $\{\delta_i\}_{i=1}^{M}, \delta_i \sim \mathcal{U}(0, 1)$, followed by sampling tuples $\{(X, a, r)_j\}_{j=1}^{N}$ for context X, arm a and associated reward r for each δ_i. We set $M = 64, N = 562$, using 512 context and 50 target points for Neural Processes, and an equal amount of data points for the meta- and inner-updates in MAML. Note that since gradient steps are necessary for MAML to adapt to data from each test problem, we reset the parameters after each evaluation run. This additional step is not necessary for neural processes.

Table 6.2 shows the quantitative evaluation on this task. We observe that Neural Processes are a highly competitive method, performing similar to MAML and the *NeuralLinear* baseline in Riquelme et al. [2018], which is consistently among the best out of 20 algorithms compared.

6.5 Related work

6.5.1 Gaussian processes

We start by considering models that, like NPs, lie on the spectrum between neural networks (NNs) and Gaussian processes (GPs). Algorithms on the NN end of the spectrum fit a single function that they learn from a very large amount of data directly. GPs on the other hand can represent a distribution over a family of functions, which is constrained by an assumption on the functional form of the covariance between two points.

Scattered across this spectrum, we can place recent research that has combined ideas from Bayesian non-parametrics with neural networks. Methods like [Calandra et al., 2016, Huang et al., 2015] remain fairly close to the GPs, but incorporate NNs to pre-process the input data. Deep GPs have some conceptual similarity to NNs as they stack GPs to obtain *deep* models [Damianou and Lawrence, 2013]. Approaches that are more similar to NNs include for example neural networks whose weights are sampled using a GPs [Wilson et al., 2011] or networks where each unit represents a different kernel [Sun et al., 2018].

There are two models on this spectrum that are closely related to NPs: matching networks (MN, Vinyals et al. [2016]) and deep kernel learning (DKL, Wilson et al. [2016]). As with NPs both use NNs to extract representations from the data, but while NPs learn the 'kernel' to compare data points implicitly these other two models pass the representation to an explicit distance kernel. MNs use this kernel to measure the similarity between contexts and targets for few shot classification while the kernel in DKL is used to parametrise a GP. Because of this explicit kernel the computational complexity of MNs and DKL would be quadratic and cubic instead of $\mathcal{O}(n+m)$ like it is for NPs. To overcome this computational complexity DKL replace a standard GP with a kernel approximation given by a KISS GP [Wilson and Nickisch, 2015], while prototypical networks [Snell et al., 2017] are introduced as a more light-weight version of MNs that also scale with $\mathcal{O}(n+m)$.

Finally concurrent work by Ma et al introduces variational implicit processes, which share large part of the motivation of NPs but are implemented as GPs [Ma et al., 2018]. In this context NPs can be interpreted as a neural implementation of an implicit stochastic process.

On this spectrum from NNs to GPs, neural processes remain closer to the neural end than most of the models mentioned above. By giving up on the explicit definition of a kernel NPs lose some of the mathematical guarantees of GPs, but trade this off for data-driven 'priors' and computational efficiency.

6.5.2 Meta-learning

In contemporary meta-learning vocabulary, NPs and GPs can be seen to be methods for 'few-shot function estimation'. In this section we compare with related models that can be used to the same end. A prominent example is matching networks Vinyals et al. [2016], but there is a large literature of similar models for classification [Koch et al., 2015, Santoro et al., 2016], reinforcement learning [Wang et al., 2016], parameter update [Finn et al., 2017, 2018], natural language processing [Bowman et al., 2016] and program induction [Devlin et al., 2017]. Related are generative meta-learning approaches that carry out few-shot estimation of the data densities [van den Oord et al., 2016, Reed et al., 2018, Bornschein et al., 2017, J. Rezende et al., 2016].

Meta-learning models share the fundamental motivations of NPs as they shift workload from training time to test time. NPs can therefore be described as meta-learning algorithms for few-shot function regression, although as shown in the previous chapter they can also be applied to few-shot learning tasks beyond regression.

6.5.3 Bayesian methods

The link between meta-learning methods and other research areas, like GPs and Bayesian methods, is not always evident. Interestingly, recent work by Grant et al. [2018] out the relation between model agnostic meta learning (MAML, Finn et al. [2017]) and hierarchical Bayesian inference. In this work the meta-learning properties of MAML are formulated as a result of task-specific variables that are conditionally independent given a higher level variable. This hierarchical description can be rewritten as a probabilistic inference problem and the resulting marginal likelihood $p(\mathbf{y}|C)$ matches the original MAML objective. The parallels between NPs and hierarchical Bayesian methods are similarly straightforward. Given the graphical model in Figure 6.8d we can write out the conditional marginal likelihood as a hierarchical inference problem:

$$\log p(\mathbf{y}^t|C, \mathbf{x}^t) = \log \int p(\mathbf{y}^t|\mathbf{z}, \mathbf{x}^t)p(\mathbf{z}|C)d\mathbf{z} \tag{6.9}$$

Another interesting area that connects Bayesian methods and NNs are Bayesian neural networks [Gal and Ghahramani, 2016, Blundell et al., 2015, Louizos et al., 2017, Louizos and Welling, 2017]. These models learn distributions over the network weights and use the posterior of these weights to estimate the values of $\mathbf{y}^t$ given $\mathbf{y}^c$. In this context NPs can be thought of as amortised version of Bayesian DL.

6.5.4 Conditional latent variable models

We have covered algorithms that are conceptually similar to NPs and algorithms that carry out similar tasks to NPs. In this section we look at models of the same family as NPs: conditional latent variable models. Such models (Figure 6.8a) learn the conditional distribution $p(\mathbf{y}^t|\mathbf{y}^c, \mathbf{z})$ where $\mathbf{z}$ is a latent variable that can be sampled to generate different predictions. Training this type of directed graphical model is intractable and as with variational autoencoders (VAEs, Rezende et al. [2014], Kingma and Welling [2013]), conditional variational autoencoders (CVAEs, Sohn et al. [2015]) approximate the objective function using the variational lower bound on the log likelihood:

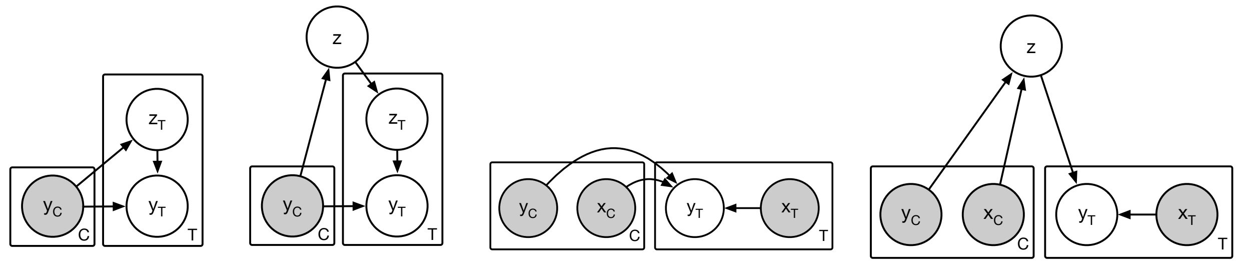
$$\log p(\mathbf{y}^t|\mathbf{y}^c) \geq \mathbb{E}_{q(\mathbf{z}^t|\mathbf{y}^c,\mathbf{y}^t)}\Bigg[\log p(\mathbf{y}^t|\mathbf{z}^t, \mathbf{y}^c) + \log \frac{p(\mathbf{z}^t|\mathbf{y}^c)}{q(\mathbf{z}^t|\mathbf{y}^c, \mathbf{y}^t)}\Bigg] \tag{6.10}$$

We refer to the latent variable of CVAEs $\mathbf{z}^t$ as a local latent variable in order to distinguish from global latent variables that are present in the models later on. We call this latent variable local as it is sampled anew for each of the output predictions $\mathbf{y}^t_i$. This is in contrast to a global latent variable that is only sampled once and used to predict multiple values of $\mathbf{y}^t$. In the CVAE, conditioning on the context is done by adding the dependence both in the prior $p(\mathbf{z}^t|\mathbf{y}^c)$ and decoder $p(\mathbf{y}^t|\mathbf{z}^t, \mathbf{y}^c)$ so they can be considered as deterministic functions of the context.

CVAEs have been extended in a number of ways for example by adding attention [J. Rezende et al., 2016]. Another related extension is generative matching networks (GMNs, Bartunov and Vetrov [2017]), where the conditioning input is pre-processed in a way that is similar to the matching networks model.

A more complex version of the CVAE that is very relevant in this context is the neural statistician (NS, Edwards and Storkey [2017]). Similar to the neural process, the neural statistician contains a global latent variable $\mathbf{z}$ that captures global uncertainty (see Figure 6.8b). A crucial difference is that while NPs represent the distribution over functions, NS represents the distribution over sets. Since NS does not contain a corresponding $\mathbf{x}$ value for each $\mathbf{y}$ value, it does not capture a pair-wise relation like GPs and NPs, but rather a general distribution of the $\mathbf{y}$ values. Rather than generating different $\mathbf{y}$ values by querying the model with different $\mathbf{x}$ values, NS generates different $\mathbf{y}$ values by sampling an additional local hidden variable $\mathbf{z}^t$.

The ELBO of the NS reflects the hierarchical nature of the model with a double expectation over the local and the global variable. If we leave out the local latent variable for a more direct comparison to NPs the



(a) CVAE (b) NS (c) Conditional neural process (d) Neural process

Figure 6.8: Graphical models of related models (a: conditional VAE, b: neural statistician, c: conditional neural processes) and of the neural process (d). Gray shading indicates the variable is observed. C stands for context variables and T for target variables i.e. the variables to predict given C. Figure adapted from [Garnelo et al., 2018b].

ELBO becomes:

$$\begin{aligned}\log p(\mathbf{y}^t, \mathbf{y}^c) \geq \mathbb{E}_{q(\mathbf{z}|\mathbf{y}^c,\mathbf{y}^t)} \Bigg[&\log p(\mathbf{y}^t|\mathbf{z}) \\ &+ \log \frac{p(\mathbf{z})}{q(\mathbf{z}|\mathbf{y}^c, \mathbf{y}^t)} \Bigg]\end{aligned} \tag{6.11}$$

Notably the prior $p(\mathbf{z})$ of NS is not conditioned on the context. The prior of NPs on the other hand is conditional (equation 6.7), which brings the training objective closer to the way the model is used at test time. Another variant of the ELBO is presented in the variational homoencoder [Hewitt et al., 2018], a model that is very similar to the neural statistician but uses a separate subset of data points for the context and predictions.

As reflected in Figure 6.8 the main difference between NPs and the conditional latent variable models is the lack of an $\mathbf{x}$ variable that allows for targeted sampling of the latent distribution. This change, despite seeming small, drastically changes the range of applications. Targeted sampling, for example allows for generation and completion tasks (e.g. the image completion tasks) or the addition of some downstream task (like using an NP for reinforcement learning). It is worth mentioning that all of the conditional latent variable models have also been applied to few-shot classification problems, where the data space generally consists of input tuples $(\mathbf{x}, \mathbf{y})$, rather than just single outputs $\mathbf{y}$. The models are able to carry this out by framing classification either as a comparison between the log likelihoods of the different classes or by looking at the KL between the different posteriors, thereby overcoming the need of working with data tuples.

6.6 Discussion

In this chapter we have introduced an latent variable extension of CNPs called Neural Processes. Neural processes, form a family of models which combine the benefits of stochastic processes and neural networks.

NPs learn to represent distributions over functions and make flexible predictions at test time conditioned on some context input. Instead of requiring a handcrafted kernel, NPs learn an implicit measure from the data directly. As a result of including a latent variable, NPs are able to generate different coherent predictions for a given set of context points, which was not possible with CNPs.

We have successfully applied NPs to a range of benchmarks ranging from regression to optimisation in order to showcase their flexibility. Note that the main motivation for the research described in this chapter was to introduce NPs as an alternative family of algorithms for few-shot regression and to assess their abilities on a variety of tasks. As a result, the tasks presented here are diverse but relatively low-dimensional. We leave it to future work to scale NPs up to higher dimensional problems that are likely to highlight the benefit of lower computational complexity and data driven representations.

Chapter 7

Conclusion and Future Work

In this thesis we have introduced three different approaches to tackling the data intensive nature of deep learning models. In chapter 3 we first considered symbolic methods. We argued in favour of concept-level reasoning, one of the fundamental characteristics of these methods, for obtaining algorithms capable of data-efficient generalisation. With this in mind we posed the question of how to incorporate ideas from symbolic reasoning into deep learning architectures. We addressed this question by identifying some key concepts (such as objects and relations) that we claimed are crucial for concept-level reasoning and incorporated these concepts as architectural biases. As an example, we picked out objects from images by identifying salient features and we emphasized the importance of relations by considering pairwise relations between objects rather than the individual objects themselves.

The goal of chapter 3 was not to find universal architectural constraints that can be applied to any network to allow for symbolic reasoning. The assumptions that motivated our architectural choices worked well for our particular small environment, but are unlikely to translate to more complex environments. Instead, the motivation of this chapter was to highlight the benefits of considering symbolic reasoning and to give an example of how to guide the design of deep networks to incorporate this type of prior knowledge. Other successful deep learning methods, such as relation networks [Santoro et al., 2017] are also examples of how to explicitly incorporate concept-level priors into neural networks.

Despite the promising initial results, the model itself is very simple and to a certain degree specifically tailored to the simple game environment. Some of the architectural assumptions would not translate to more complex environments. For example, we assume that salient areas of the image correspond to individual objects, which does not necessarily hold for more complex images. The fact that the player is the only object in the scene that moves also simplifies the task and is not a realistic assumption for more complex environments.

Future improvements to the model can be divided into four areas: 1. Improving the object/symbol detection pipeline. Developing models that learn how to extract meaningful objects and concepts from images in an unsupervised fashion is currently an important area of research in artificial intelligence [Burgess et al., 2019] and is crucial for building more complex and robust versions of our model. 2. Improving object tracking. This involves having models that capture object persistence [Creswell et al., 2020] but also models with memory modules that can keep track of the objects in the scene and their evolution. 3. Improving object-based reinforcement learning. There are some approaches of combining object-level representations with the current reinforcement learning (RL) frameworks [Diuk et al., 2008], as well as relational RL [Džeroski et al., 2001] but to date there is no good framework to match the results using the regular RL setup with object-level approaches. 4. Finding relevant training tasks and environments. Many of our current RL environments are overly simplistic (like Atari [Bellemare et al., 2013]) and might not benefit from object-level reasoning, or might not be conducive to models learning abstract concepts in the first place. We therefore need to design environments and tasks that reflect the properties of real world tasks. While these four directions of improvements span a large area of current machine learning research, research can be carried out on the individual branches separately to gradually improve a system that combines them all.

In chapter 4 we considered an alternative approach to reducing the amount of data or experience required to train deep neural models: pre-training parts of the network. The idea behind this technique is to train a subset of a network on a task that is distinct from, but related to, the original target task. The learned weights can then be used either for initialisation or as fixed weights when training the rest of the network on the original task. The fact that the weights are reused for different tasks by itself contributes to improving the data-efficiency of our deep models. For example, people often use pre-trained image encoders that are publicly available online when training a classifier instead of training network weights from scratch. Beyond this, however, training networks in separate stages can enable faster learning.

The challenge we face when pre-training parts of deep learning models, however, is to choose the right pre-training task. Historically, the most common pre-trained networks were used for computer vision, where practitioners rely on the common patterns present in natural images that can be useful for a range of downstream tasks. In chapter 4 our goal was to pre-train the encoder of an RL-agent that moves in a 3D environment. While training on natural images certainly would have provided some useful image encoders, we further tailored the pre-training to the RL-task by training the encoder on a 3D scene reconstruction task. In doing so the agent's encoder was provided with the ability to integrate information from 3D scenes from different viewpoints and the agent was able to learn to flexibly act using observations from any viewpoint in 3D space. As we move towards training increasingly larger and more complex neural models, it is likely that we will rely more and more on pre-trained components. It is therefore important to think about designing the right auxiliary pre-training tasks that will improve down-stream performance.

Finally, we shifted our attention to algorithms that are data-efficient at test time, i.e. that can adapt their predictions to different tasks without having to be re-trained. One way of achieving this is to train models that approximate a distribution over functions rather than a single function. By doing this, the trained models are able to adapt their predictions to a range of different functions depending on some small number of observations at test time. This area of research is generally referred to as meta-learning or 'learning to learn'.

In chapters 5 and 6 we introduced neural processes (NPs), a family of deep few-shot prediction models. We introduced two versions of the model: a deterministic one and a latent one and applied them to a number of regression and classification tasks. A crucial part of the training setup is the choice of dataset. In order for NPs to learn to model the uncertainty correctly and be able to do accurate predictions we need to train on a dataset that consists of a distribution over functions. Crucially, this distribution should cover the space of all functions that we are interested in at test time. As such, the choice of dataset can be seen as a way of incorporating prior knowledge into our algorithms instead of handcrafting features or specifying non-parametric kernels.

The papers presented in these chapters introduce the basic deterministic and latent variable Neural Processes. While they produce good results on simple regression and classification tasks there are many possible extensions that can be built on top of this original model. As a matter of fact, since the publication of the original NP papers there has been a large number of follow-up papers that cover different additions to the models: one important extension is that of adding attention to NPs [Kim et al., 2019] which greatly improves the predictions at the observed context points. There has been work into adding structure to the aggregation of the context points using either recurrent architectures [Qin et al., 2019, Willi et al., 2019] or graph architectures [Carr and Wingate, 2019, Nassar et al., 2018]. Different versions of the variational objective have been analysed in [Le et al., 2018] and NPs have been applied to more complex tasks, for example in the context of sequential decision making [Galashov et al., 2019].

In sum, in this thesis we investigate a number of different approaches that address data efficiency of deep neural networks: by constraining the architecture to capture some high level concepts, by finding pre-training tasks to train different parts of the network or by considering datasets that capture our belief over the distribution of possible targets we are interested in. In addition to the methods that we focus on there are other approaches, such as transfer learning [Tan et al., 2018, Weiss et al., 2016], data augmentation [Shorten and Khoshgoftaar, 2019] or generating synthetic data [Nikolenko, 2019] that share our motivation. In general, however, addressing the data efficiency of deep learning models, remains an open question. As our deep models grow and the complexity of goal tasks increases, it is likely that we will have to turn to a combination of methods.

Acknowledgements

First and foremost, I want to thank my PhD advisor Prof Murray Shanahan. I can't stress enough how much he has supported me over the past years. Without his advice and efforts I would certainly not be where I am today. I am also very grateful that we went through the DM journey together from the beginning and I am proud to have managed to get him to try some chocolate cake (probably the biggest accomplishment of my PhD).

It takes a village to write a PhD and I am lucky that I have had about 3 medium-sized towns to support me. The only downside of this is the fact that this 'Acknowledgements' section will probably be the longest part of my thesis. In the interest of brevity I would like to just provide a bullet point list with no particular order other than maybe some attempt at chronologically sorting people in my head. So with that, I would like to thank:

- My family. My sister (chip de la hermandad-broes, before hoes), Mum and Dad. I am who I am because of them, period.
- Dan Burgess, for being one of the biggest supports in my life and also making me appreciate really tiny mangy dogs that look really worried.
- Sorin, for making my life look put together by comparison even in my lowest of lows. Also for introducing me to Maldon salt.
- Chris, for his hair colour and being a good TensorFlow student (it's all about Jax now, though).
- Irene (my sister from another mister) for getting Sorin out of the house every evening and boiling the kettle.
- Tiago for teaching me BigTable and becoming my best mate.
- My amazing lab at Imperial:
  - Kitsy for being the perfect co-parent to the Basil and just having eternal patience with all of us.

  - Pedro for literally making me laugh with any of his Spanish expressions and having unrealistic faith in the intelligence of the rest of of the lab.
  - Nat for always being a kind face (that will TRASH you at office basketball).
  - Zaf for having the most insane ideas and carrying them out with a straight face.
  - Christos for motivating me to finish not TOO late after him.

- The rest of the Imperial gang (and work colleagues):
  - Matt for being an expert on all good things in life and forcing me to sign up to all airline reward schemes (I can't emphasize enough how this is one of the best things to happen during my PhD).
  - Hugh for putting up with all of my attempts to marry him and just forcing us all to love GPs.
  - Ben for taking my fear of asking questions and making the whole group much more kind and humane.
  - Kai for managing my Twitter presence through his account.
  - Marc for having faith in me and always including me in his group.
  - Petr, my first 'Computing friend' for hanging out with me in Master lectures, Chinese class and bachelor parties.
  - Claudia for being the perfect mix between a replica of me and a replica of Taylor Swift.
  - Silvia for taking us out to dance.
  - Hadeel for introducing me to the saudi sweet pockets and for providing me with a thousand times more dates than any dating app.
  - Doniyor for not giving up on me despite my bad texting habits.
  - Carlos for letting us bully him into visiting us all the way from Spain.
  - Stefan's dad for always being nice to me and also creating Stefan.

- Ali for unexpectedly becoming an honorary additional PhD advisor to me, while also remaining a good friend. Thank you for referring me to DeepMind and making my whole experience so far incredibly fun and interesting. Thanks also for putting up with my continuous stream of weird (but really good) jokes, life drama and 3D printed presents.

- Danro for being my lunch buddy and always incredibly chilled in an otherwise often stressful environment. Also the whole Roseforest for adopting me into their family and teaching my subconscious Hebrew.

- Danilo for being one of the smartest and kindest people at DM. As I always say, I want to be Danilo when I grow up.

- Fabio for making me laugh pretty much all the time and forcing me to use reverse search on the terminal.
- Fred for introducing me to the Oculus quest as well as BUILD files.
- The coolest gang at DM, my lovely members of the DeepSix (in the order that I met them):
  - Courtney for being a ray of sunshine in literally everyone's lives and at this point like a sister to me.
  - Ellen (pronounced "eeeeeeaaahl'n") for being one of my favourite flatmates and agreeing to dance to RuPaul all day and all night with me. Also for being my PgM for life.
  - Hannah for always being really sweet while really embracing all of my debauchery.
  - Claire, my gym and fitness bro, as well as olympic acro-yoga partner (who I am slowly realising is somehow the same person as me) for helping me put Will back in his place.
  - Will Hawkinggs for the interesting research conversations, the sudoku and age of empires games, my yellow hoodie and the overall classic banter.
- Makro for not judging me when I randomly show up with the weirdest technical questions once a year and for listening to my crazy life with interest (and amusement) but always without judgement.
- Chloe, my on-off PgM for being always incredibly supportive and understanding and always in a great mood.
- Shakir for introducing me to the machine learning community and always being a bright ball of excitement and energy.
- Heiner for being my official coding buddy when I joined DM and putting up with a long, long, long list of bizarre questions.
- Toni for being my perfect climbing partner and the biggest supporter of my cuisine.
- Anna Bortsova for taking me to crypts to look at (and draw) semi-nude people.
- Adam for introducing me to one-legged squats and Polish sweets.
- Yarosha for always being incredibly kind and introducing me to the illuminated DeepMinder starter pack life.
- Karen for breaking my heart, like all interns when they become our best friends and leave us after a few months and for showing me great music.
- Mihaela for always being such an inspiration as a strong and intelligent researcher as well as being a great friend and advisor on pasta-alternatives.
- Dada for our great toilet chats and for co-creating the post-it Pusheen with me.

- Olivier for teaching me how to straight chill and being a great DJ at parties.
- Bobak for showing up at family dinner even when it was in the far far West and for agreeing to take the *beautiful* mirror with him.
- Claire for being the coolest flatmate and for putting up with the mattress in the hallway for almost a year. Also for introducing me to Magnesium to stop my eye from twitching.
- Brian for always lending us chairs when we didn't have enough… wait, no I mean for watering my plants (which is a full time job) and bringing me Spruengli chocolate spread from Zurich.
- Team America:
  - Kevin for being more European than any of us by now.
  - Ian for introducing me to Keenan Thompson's French lessons.
  - Jackie for being the best Thanksgiving help-line we could have asked for.
- Dhruva, my same random seed from another continent, for being one of the kindest people I know, for his patience and support during a time that was truly difficult for me and for all the good jokes.
- David Saxton for joining the Neural Processes project before it was cool and joining me on some other journeys as well.
- Hels for being a kind support from the NIPS party just before I joined DeepMind until today and for teaching me how to make Macarons (a skill that I had long given up on).
- Lila for being an incredible inspiration, for taking the time to be my mentor and to teach me that you can be a strong leader, while being incredibly kind and loving what to do. Also for not freaking out when Hels and I torched about 20 creme brulees next to her desk.
- Gauthier for introducing me to great new music and for my bike.
- Anna Harutyunyan for being my soul sister and wife for life. For helping me grow and supporting me throughout the weirdest and most interesting period of my life and above all for all the TikToks she has shared with me.
- Hamza for being like a tree without the shade and the older brother I never had.
- Sasha for being my hipsternism senpai.
- Wojtek for questioning all of my statements, but always with a ';)'.
- Friends from the past that have kept me sane:
  - Ioan for having faith in my ML skills before there were any.

  - Tomke for being a constant ray of sunshine and forever my queen.
  - Ana for teaching me to not wear skin-tone tights.
  - Gabi for introducing me to Google and Imperial. I would not be where I am today if we had not been roomates for a few days at Caltech.
  - Katia for staying my best friend throughout life, regardless of how much time passes.
  - Tere for being my unofficial twin.
  - Ewa, my older sister for inspiring me on how to be a kick-ass career woman as well as a mother.
  - Jess for introducing me to most of the ML community.

- Anyone that has been on a paper with me, thanks for your help and advice.

Bibliography

Shipra Agrawal and Navin Goyal. Analysis of thompson sampling for the multi-armed bandit problem. In *Conference on Learning Theory*, pages 39–1, 2012.

Naveed Akhtar and Ajmal Mian. Threat of adversarial attacks on deep learning in computer vision: A survey. *IEEE Access*, 6:14410–14430, 2018.

André Barreto, Rémi Munos, Tom Schaul, and David Silver. Successor Features for Transfer in Reinforcement Learning. *Advances in Neural Information Processing Systems*, 2017.

David Barrett, Felix Hill, Adam Santoro, Ari Morcos, and Timothy Lillicrap. Measuring abstract reasoning in neural networks. In *International Conference on Machine Learning*, pages 511–520, 2018.

Sergey Bartunov and Dmitry P Vetrov. Fast adaptation in generative models with generative matching networks. *International Conference on Leaarning Representations*, 2017.

Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Deep api programmer: Learning to program with apis. *arXiv preprint arXiv:1704.04327*, 2017.

David M Blei, Alp Kucukelbir, and Jon D McAuliffe. Variational inference: A review for statisticians. *Journal of the American Statistical Association*, 112(518):859–877, 2017.

Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. Weight uncertainty in neural networks. *International Conference on Machine Learning*, 2015.

Jörg Bornschein, Andriy Mnih, Daniel Zoran, and Danilo J. Rezende. Variational memory addressing in generative models. In *Advances in Neural Information Processing Systems*, pages 3923–3932, 2017.

Samuel R Bowman, Luke Vilnis, Oriol Vinyals, Andrew M Dai, Rafal Jozefowicz, and Samy Bengio. Generating sentences from a continuous space. *Conference on Computational Natural Language Learning*, 2016.

Andrew Brock, Jeff Donahue, and Karen Simonyan. Large scale gan training for high fidelity natural image synthesis. *arXiv preprint arXiv:1809.11096*, 2018.

Christopher P Burgess, Loic Matthey, Nicholas Watters, Rishabh Kabra, Irina Higgins, Matt Botvinick, and Alexander Lerchner. Monet: Unsupervised scene decomposition and representation. *arXiv preprint arXiv:1901.11390*, 2019.

Lucian Buşoniu, Robert Babuška, and Bart De Schutter. Multi-agent reinforcement learning: An overview. In *Innovations in multi-agent systems and applications-1*, pages 183–221. Springer, 2010.

Cheng-Hao Cai, Yanyan Xu, Dengfeng Ke, and Kaile Su. Learning of human-like algebraic reasoning using deep feedforward neural networks. *Biologically inspired cognitive architectures*, 25:43–50, 2018.

Roberto Calandra, Jan Peters, Carl Edward Rasmussen, and Marc Peter Deisenroth. Manifold gaussian processes for regression. In *International Joint Conference on Neural Networks*, pages 3338–3345. IEEE, 2016.

Andrew Carr and David Wingate. Graph neural processes: Towards bayesian graph neural networks. *arXiv preprint arXiv:1902.10042*, 2019.

Piotr Chabierski, Alessandra Russo, Mark Law, and Krysia Broda. Machine comprehension of text using combinatory categorial grammar and answer set programs. CEUR Workshop Proceedings, 2017.

Xi Chen, Yan Duan, Rein Houthooft, John Schulman, Ilya Sutskever, and Pieter Abbeel. Infogan: Interpretable representation learning by information maximizing generative adversarial nets. In *Advances in neural information processing systems*, pages 2172–2180, 2016.

Antonia Creswell, Kyriacos Nikiforou, Oriol Vinyals, Andre Saraiva, Rishabh Kabra, Loic Matthey, Chris Burgess, Malcolm Reynolds, Richard Tanburn, Marta Garnelo, et al. Alignnet: Unsupervised entity alignment. *arXiv preprint arXiv:2007.08973*, 2020.

Andreas Damianou and Neil Lawrence. Deep gaussian processes. In *Artificial Intelligence and Statistics*, pages 207–215, 2013.

Bruno De Finetti. La prévision: ses lois logiques, ses sources subjectives. In *Annales de l'institut Henri Poincaré*, volume 7, pages 1–68, 1937.

Jacob Devlin, Rudy R Bunel, Rishabh Singh, Matthew Hausknecht, and Pushmeet Kohli. Neural program meta-induction. In *Advances in Neural Information Processing Systems*, pages 2077–2085, 2017.

Carlos Diuk, Andre Cohen, and Michael L Littman. An object-oriented representation for efficient reinforcement learning. In *Proceedings of the 25th international conference on Machine learning*, pages 240–247, 2008.

Aimore RR Dutra and Artur S d'Avila Garcez. A comparison between deep q-networks and deep symbolic reinforcement learning. In *NeSy*, 2017.

Sašo Džeroski, Luc De Raedt, and Kurt Driessens. Relational reinforcement learning. *Machine learning*, 43 (1-2):7–52, 2001.

Harrison Edwards and Amos Storkey. Towards a neural statistician. 2017.

Martin Engelcke, Adam R Kosiorek, Oiwi Parker Jones, and Ingmar Posner. Genesis: Generative scene inference and sampling with object-centric latent representations. *arXiv preprint arXiv:1907.13052*, 2019.

Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent, and Samy Bengio. Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research*, 11 (Feb):625–660, 2010.

SM Ali Eslami, Nicolas Heess, Theophane Weber, Yuval Tassa, David Szepesvari, Geoffrey E Hinton, et al. Attend, infer, repeat: Fast scene understanding with generative models. In *Advances in Neural Information Processing Systems*, pages 3225–3233, 2016.

SM Ali Eslami, Danilo Jimenez Rezende, Frederic Besse, Fabio Viola, Ari S Morcos, Marta Garnelo, Avraham Ruderman, Andrei A Rusu, Ivo Danihelka, Karol Gregor, et al. Neural scene representation and rendering. *Science*, 360(6394):1204–1210, 2018a.

SM Ali Eslami, Danilo Jimenez Rezende, Frederic Besse, Fabio Viola, Ari S Morcos, **Garnelo, Marta**, Avraham Ruderman, Andrei A Rusu, Ivo Danihelka, Karol Gregor, et al. Neural scene representation and rendering. *Science*, 360(6394):1204–1210, 2018b.

Ivan Evtimov, Kevin Eykholt, Earlence Fernandes, Tadayoshi Kohno, Bo Li, Atul Prakash, Amir Rahmati, and Dawn Song. Robust physical-world attacks on deep learning models. *arXiv preprint arXiv:1707.08945*, 2017.

Santiago Fernández, Alex Graves, and Jürgen Schmidhuber. An application of recurrent neural networks to discriminative keyword spotting. In *International Conference on Artificial Neural Networks*, pages 220–229. Springer, 2007.

Chelsea Finn, Xin Yu Tan, Yan Duan, Trevor Darrell, Sergey Levine, and Pieter Abbeel. Deep spatial autoencoders for visuomotor learning. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 512–519. IEEE, 2016a.

Chelsea Finn, Tianhe Yu, Justin Fu, Pieter Abbeel, and Sergey Levine. Generalizing skills with semi-supervised reinforcement learning. *International Conference on Learning Representations*, 2016b.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. *International Conference on Machine Learning*, 2017.

Chelsea Finn, Kelvin Xu, and Sergey Levine. Probabilistic model-agnostic meta-learning. *arXiv preprint arXiv:1806.02817*, 2018.

Michael C Fu. Gradient estimation. *Handbooks in operations research and management science*, 13:575–616, 2006.

Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *International Conference on Machine Learning*, pages 1050–1059, 2016.

Alexandre Galashov, Jonathan Schwarz, Hyunjik Kim, Marta Garnelo, David Saxton, Pushmeet Kohli, SM Eslami, and Yee Whye Teh. Meta-learning surrogate models for sequential decision making. *arXiv preprint arXiv:1903.11907*, 2019.

Yaroslav Ganin, Tejas Kulkarni, Igor Babuschkin, SM Eslami, and Oriol Vinyals. Synthesizing programs for images using reinforced adversarial learning. *arXiv preprint arXiv:1804.01118*, 2018.

Artur d'Avila Garcez, Aimore Resende Riquetti Dutra, and Eduardo Alonso. Towards symbolic reinforcement learning with common sense. *arXiv preprint arXiv:1804.08597*, 2018.

Artur SD'Avila Garcez, Luis C Lamb, and Dov M Gabbay. *Neural-symbolic cognitive reasoning*. Springer Science & Business Media, 2008.

Marta Garnelo, Kai Arulkumaran, and Murray Shanahan. Towards deep symbolic reinforcement learning. *NIPS - Deep Reinforcement Learning Workshop*, 2016.

Marta Garnelo, Dan Rosenbaum, Christopher Maddison, Tiago Ramalho, David Saxton, Murray Shanahan, Yee Whye Teh, Danilo Jimenez Rezende, and Ali Eslami. Conditional neural processes. In *International Conference on Machine Learning*, 2018a.

Marta Garnelo, Jonathan Schwarz, Dan Rosenbaum, Fabio Viola, Danilo J Rezende, SM Eslami, and Yee Whye Teh. Neural processes. In *ICML Workshop on Theoretical Foundations and Applications of Deep Generative Models*, 2018b.

Robert Geirhos, Carlos RM Temme, Jonas Rauber, Heiko H Schütt, Matthias Bethge, and Felix A Wichmann. Generalisation in humans and deep neural networks. In *Advances in Neural Information Processing Systems*, pages 7538–7550, 2018.

Peter W Glynn. Likelihood ratio gradient estimation for stochastic systems. *Communications of the ACM*, 33(10):75–84, 1990.

Erin Grant, Chelsea Finn, Sergey Levine, Trevor Darrell, and Thomas Griffiths. Recasting gradient-based meta-learning as hierarchical bayes. *International Conference on Learning Representations*, 2018.

Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, 18(5-6):602–610, 2005.

Alex Graves and Jürgen Schmidhuber. Offline handwriting recognition with multidimensional recurrent neural networks. In *Advances in neural information processing systems*, pages 545–552, 2009.

Alex Graves, Marcus Liwicki, Horst Bunke, Jürgen Schmidhuber, and Santiago Fernández. Unconstrained on-line handwriting recognition with recurrent neural networks. In *Advances in neural information processing systems*, pages 577–584, 2008.

Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE, 2013.

Klaus Greff, Antti Rasmus, Mathias Berglund, Tele Hao, Harri Valpola, and Juergen Schmidhuber. Tagger: Deep unsupervised perceptual grouping. In *Advances in Neural Information Processing Systems*, pages 4484–4492, 2016.

Klaus Greff, Raphaël Lopez Kaufman, Rishabh Kabra, Nick Watters, Chris Burgess, Daniel Zoran, Loic Matthey, Matthew Botvinick, and Alexander Lerchner. Multi-object representation learning with iterative variational inference. *arXiv preprint arXiv:1903.00450*, 2019.

Karol Gregor, Ivo Danihelka, Alex Graves, Daan Wierstra, et al. Draw: A recurrent neural network for image generation. In *CoRR*. Citeseer, 2015.

Abhishek Gupta, Coline Devin, YuXuan Liu, Pieter Abbeel, and Sergey Levine. Learning invariant feature spaces to transfer skills with reinforcement learning. *International Conference on Learning Representations*, 2016.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

Douglas Heaven. Why deep-learning ais are so easy to fool. *Nature*, 574(7777):163, 2019.

Nicolas Heess, Gregory Wayne, David Silver, Timothy Lillicrap, Tom Erez, and Yuval Tassa. Learning continuous control policies by stochastic value gradients. In *Advances in Neural Information Processing Systems*, pages 2944–2952, 2015.

Dan Hendrycks, Kimin Lee, and Mantas Mazeika. Using pre-training can improve model robustness and uncertainty. *arXiv preprint arXiv:1901.09960*, 2019.

Luke Hewitt, Andrea Gane, Tommi Jaakkola, and Joshua B Tenenbaum. The variational homoencoder: Learning to infer high-capacity generative models from few examples. 2018.

Irina Higgins, Loic Matthey, Arka Pal, Christopher Burgess, Xavier Glorot, Matthew Botvinick, Shakir Mohamed, and Alexander Lerchner. beta-vae: Learning basic visual concepts with a constrained variational framework. *International Conference on Learning Representations*, 2016.

Irina Higgins, Arka Pal, Andrei Rusu, Loic Matthey, Christopher Burgess, Alexander Pritzel, Matthew Botvinick, Charles Blundell, and Alexander Lerchner. Darla: Improving zero-shot transfer in reinforcement learning. In *International Conference on Machine Learning*, pages 1480–1490, 2017.

Irina Higgins, David Amos, David Pfau, Sebastien Racaniere, Loic Matthey, Danilo Rezende, and Alexander Lerchner. Towards a definition of disentangled representations. *arXiv preprint arXiv:1812.02230*, 2018a.

Irina Higgins, Nicolas Sonnerat, Loic Matthey, Arka Pal, Christopher P Burgess, Matko Bošnjak, Murray Shanahan, Matthew Botvinick, Demis Hassabis, and Alexander Lerchner. SCAN: Learning hierarchical compositional visual concepts. In *Proc. International Conference on Learning Representations*, 2018b.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Wen-bing Huang, Deli Zhao, Fuchun Sun, Huaping Liu, and Edward Y Chang. Scalable gaussian process regression using deep neural networks. In *International Joint Conferences on Artificial Intelligence*, pages 3576–3582, 2015.

Geoffrey Irving, Christian Szegedy, Alexander A Alemi, Niklas Eén, François Chollet, and Josef Urban. Deepmath-deep sequence models for premise selection. In *Advances in Neural Information Processing Systems*, pages 2235–2243, 2016.

Danilo J. Rezende, Ivo Danihelka, Karol Gregor, Daan Wierstra, et al. One-shot generalization in deep generative models. In *International Conference on Machine Learning*, pages 1521–1529, 2016.

Justin Johnson, Bharath Hariharan, Laurens van der Maaten, Li Fei-Fei, C. Lawrence Zitnick, and Ross Girshick. CLEVR: A diagnostic dataset for compositional language and elementary visual reasoning. In *Proc. IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

Andrej Karpathy and Li Fei-Fei. Deep visual-semantic alignments for generating image descriptions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3128–3137, 2015.

Ramtin Keramati, Jay Whang, Patrick Cho, and Emma Brunskill. Fast exploration with simplified models and approximately optimistic planning in model based reinforcement learning. 2018.

Hyunjik Kim and Andriy Mnih. Disentangling by factorising. *arXiv preprint arXiv:1802.05983*, 2018.

Hyunjik Kim, Andriy Mnih, Jonathan Schwarz, Marta Garnelo, Ali Eslami, Dan Rosenbaum, Oriol Vinyals, and Yee Whye Teh. Attentive neural processes. *arXiv preprint arXiv:1901.05761*, 2019.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

Gregory Koch, Richard Zemel, and Ruslan Salakhutdinov. Siamese neural networks for one-shot image recognition. In *ICML - Deep Learning Workshop*, volume 2, 2015.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

Brenden M Lake, Tomer D Ullman, Joshua B Tenenbaum, and Samuel J Gershman. Building machines that learn and think like people. *Behavioral and Brain Sciences*, 40, 2017.

Tuan Anh Le, Hyunjik Kim, Marta Garnelo, Dan Rosenbaum, Jonathan Schwarz, and Yee Whye Teh. Empirical evaluation of neural process objectives. In *NeurIPS workshop on Bayesian Deep Learning*, 2018.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 17(39):1–40, 2016.

Guiying Li, Junlong Liu, Chunhui Jiang, and Ke Tang. Relief Impression Image Detection: unsupervised Extracting Objects Directly From Feature Arrangements of Deep CNN. *arXiv:1601.06719*, 2016.

Yujia Li, Oriol Vinyals, Chris Dyer, Razvan Pascanu, and Peter Battaglia. Learning deep generative models of graphs. *arXiv preprint arXiv:1803.03324*, 2018.

Qianli Liao and Tomaso Poggio. Object-oriented deep learning. Technical report, Center for Brains, Minds and Machines (CBMM), 2017.

Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, December 2015.

Christos Louizos and Max Welling. Multiplicative normalizing flows for variational bayesian neural networks. *International Conference on Machine Learning*, 2017.

Christos Louizos, Karen Ullrich, and Max Welling. Bayesian compression for deep learning. In *Advances in Neural Information Processing Systems*, pages 3290–3300, 2017.

Chao Ma, Yingzhen Li, and José Miguel Hernández-Lobato. Variational implicit processes. *arXiv preprint arXiv:1806.02390*, 2018.

Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *in ICML Workshop on Deep Learning for Audio, Speech and Language Processing*. Citeseer, 2013.

Chris J Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. *International Conference on Learning Representations*, 2016.

Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Deepproblog: Neural probabilistic logic programming. *Advances in Neural Information Processing Systems*, 31: 3749–3759, 2018.

Gary Marcus. Deep learning: A critical appraisal. *arXiv preprint arXiv:1801.00631*, 2018.

Pasquale Minervini, Matko Bosnjak, Tim Rocktäschel, and Sebastian Riedel. Towards neural theorem proving at scale. *arXiv preprint arXiv:1807.08204*, 2018.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

Charlie Nash, SM Ali Eslami, Chris Burgess, Irina Higgins, Daniel Zoran, Theophane Weber, and Peter Battaglia. The multi-entity variational autoencoder. *NIPS Workshops*, 2017.

Marcel Nassar, Xin Wang, and Evren Tumer. Conditional graph neural processes: A functional autoencoder approach. *arXiv preprint arXiv:1812.05212*, 2018.

Sergey I Nikolenko. Synthetic data for deep learning. *arXiv preprint arXiv:1909.11512*, 2019.

Bernt Øksendal. Stochastic differential equations. In *Stochastic differential equations*, page 11. Springer, 2003.

Sinno Jialin Pan, Qiang Yang, et al. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.

Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. 2016.

Shenghao Qin, Jiacheng Zhu, Jimmy Qin, Wenshuo Wang, and Ding Zhao. Recurrent attentive neural process for sequential data. *arXiv preprint arXiv:1910.09323*, 2019.

Antonin Raffin, Sebastian Höfer, Rico Jonschkowski, Oliver Brock, and Freek Stulp. Unsupervised learning of state representations for multiple tasks. 2016.

Tabish Rashid, Mikayel Samvelyan, Christian Schroeder De Witt, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. Qmix: Monotonic value function factorisation for deep multi-agent reinforcement learning. *arXiv preprint arXiv:1803.11485*, 2018.

Carl Edward Rasmussen and Christopher KI Williams. Gaussian processes in machine learning. In *Advanced lectures on machine learning*, pages 63–71. Springer, 2004.

Scott Reed and Nando De Freitas. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015.

Scott Reed, Yutian Chen, Thomas Paine, Aäron van den Oord, SM Eslami, Danilo J. Rezende, Oriol Vinyals, and Nando de Freitas. Few-shot autoregressive density estimation: Towards learning to learn distributions. 2018.

Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models. *International Conference on Machine Learning*, 2014.

Carlos Riquelme, George Tucker, and Jasper Snoek. Deep bayesian bandits showdown: An empirical comparison of bayesian deep networks for thompson sampling. *International Conference on Learning Representations*, 2018.

Tim Rocktäschel and Sebastian Riedel. End-to-end differentiable proving. In *Advances in Neural Information Processing Systems*, pages 3788–3800, 2017.

Tim Rocktäschel, Sameer Singh, and Sebastian Riedel. Injecting logical background knowledge into embeddings for relation extraction. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1119–1129, 2015.

Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

Andrei A. Rusu, Neil C. Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive Neural Networks. *International Conference on Learning Representations*, 2016.

Andrei A Rusu, Matej Večerík, Thomas Rothörl, Nicolas Heess, Razvan Pascanu, and Raia Hadsell. Sim-to-real robot learning from pixels with progressive nets. In *Conference on Robot Learning*, pages 262–270, 2017.

Andrei A Rusu, Dushyant Rao, Jakub Sygnowski, Oriol Vinyals, Razvan Pascanu, Simon Osindero, and Raia Hadsell. Meta-learning with latent embedding optimization. *arXiv preprint arXiv:1807.05960*, 2018.

Hugh Salimbeni and Marc Deisenroth. Doubly stochastic variational inference for deep gaussian processes. In *Advances in Neural Information Processing Systems*, pages 4591–4602, 2017.

Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy Lillicrap. One-shot learning with memory-augmented neural networks. *NIPS - Deep Learning Symposium*, 2016.

Adam Santoro, David Raposo, David G Barrett, Mateusz Malinowski, Razvan Pascanu, Peter Battaglia, and Tim Lillicrap. A simple neural network module for relational reasoning. In *Advances in Neural Information Processing Systems*, pages 4967–4976, 2017.

Jürgen Schmidhuber, Daan Wierstra, and Faustino J Gomez. Evolino: Hybrid neuroevolution/optimal linear search for sequence prediction. In *Proceedings of the 19th International Joint Conferenceon Artificial Intelligence (IJCAI)*, 2005.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Andrew W Senior, Richard Evans, John Jumper, James Kirkpatrick, Laurent Sifre, Tim Green, Chongli Qin, Augustin Žídek, Alexander WR Nelson, Alex Bridgland, et al. Improved protein structure prediction using potentials from deep learning. *Nature*, pages 1–5, 2020.

Luciano Serafini and Artur d'Avila Garcez. Logic tensor networks: Deep learning and logical reasoning from data and knowledge. *arXiv preprint arXiv:1606.04422*, 2016.

Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016.

Murray Shanahan, Kyriacos Nikiforou, Antonia Creswell, Christos Kaplanis, David Barrett, and Marta Garnelo. An explicitly relational neural network architecture. *arXiv preprint arXiv:1905.10307*, 2019.

Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. Cnn features off-the-shelf: an astounding baseline for recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 806–813, 2014.

Connor Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(1):60, 2019.

N Siddharth, Brooks Paige, Alban Desmaison, Van de Meent, Frank Wood, Noah D Goodman, Pushmeet Kohli, Philip HS Torr, et al. Inducing interpretable representations with variational autoencoders. *arXiv preprint arXiv:1611.07492*, 2016.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

Jake Snell, Kevin Swersky, and Richard Zemel. Prototypical networks for few-shot learning. In *Advances in Neural Information Processing Systems*, pages 4080–4090, 2017.

Edward Snelson and Zoubin Ghahramani. Sparse gaussian processes using pseudo-inputs. In *Advances in Neural Information Processing Systems*, pages 1257–1264, 2006.

Kihyuk Sohn, Honglak Lee, and Xinchen Yan. Learning structured output representation using deep conditional generative models. In *Advances in Neural Information Processing Systems*, pages 3483–3491, 2015.

Shengyang Sun, Guodong Zhang, Chaoqi Wang, Wenyuan Zeng, Jiaman Li, and Roger Grosse. Differentiable compositional kernel learning for gaussian processes. *arXiv preprint arXiv:1806.04326*, 2018.

Richard S Sutton, Andrew G Barto, Francis Bach, et al. *Reinforcement learning: An introduction.* MIT press, 1998.

Chuanqi Tan, Fuchun Sun, Tao Kong, Wenchang Zhang, Chao Yang, and Chunfang Liu. A survey on deep transfer learning. In *International conference on artificial neural networks*, pages 270–279. Springer, 2018.

Garnelo, Marta, Kai Arulkumaran, and Murray Shanahan. Towards deep symbolic reinforcement learning. *NIPS - Deep Reinforcement Learning Workshop*, 2016.

Garnelo, Marta, Dan Rosenbaum, Christopher Maddison, Tiago Ramalho, David Saxton, Murray Shanahan, Yee Whye Teh, Danilo Jimenez Rezende, and Ali Eslami. Conditional neural processes. In *International Conference on Machine Learning*, 2018a.

Garnelo, Marta, Jonathan Schwarz, Dan Rosenbaum, Fabio Viola, Danilo J Rezende, SM Eslami, and Yee Whye Teh. Neural processes. In *ICML Workshop on Theoretical Foundations and Applications of Deep Generative Models*, 2018b.

William R Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294, 1933.

Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 5026–5033. IEEE, 2012.

Jonathan Tremblay, Thang To, Artem Molchanov, Stephen Tyree, Jan Kautz, and Stan Birchfield. Synthetically trained neural networks for learning human-readable plans from real-world demonstrations. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–5. IEEE, 2018.

Aaron van den Oord, Nal Kalchbrenner, Lasse Espeholt, Oriol Vinyals, Alex Graves, et al. Conditional image generation with pixelcnn decoders. In *Advances in Neural Information Processing Systems*, pages 4790–4798, 2016.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. *arXiv preprint arXiv:1804.02477*, 2018.

Oriol Vinyals, Charles Blundell, Tim Lillicrap, Daan Wierstra, et al. Matching networks for one shot learning. In *Advances in Neural Information Processing Systems*, pages 3630–3638, 2016.

Jane X Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z Leibo, Remi Munos, Charles Blundell, Dharshan Kumaran, and Matt Botvinick. Learning to reinforcement learn. *arXiv preprint arXiv:1611.05763*, 2016.

Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang. A survey of transfer learning. *Journal of Big data*, 3(1):9, 2016.

Timon Willi, Jonathan Masci, Jürgen Schmidhuber, and Christian Osendorfer. Recurrent neural processes. *arXiv preprint arXiv:1906.05915*, 2019.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

Andrew Wilson and Hannes Nickisch. Kernel interpolation for scalable structured gaussian processes (kiss-gp). In *International Conference on Machine Learning*, pages 1775–1784, 2015.

Andrew Gordon Wilson, David A Knowles, and Zoubin Ghahramani. Gaussian process regression networks. *arXiv preprint arXiv:1110.4411*, 2011.

Andrew Gordon Wilson, Zhiting Hu, Ruslan Salakhutdinov, and Eric P Xing. Deep kernel learning. In *Artificial Intelligence and Statistics*, pages 370–378, 2016.

William Woof and Ke Chen. Learning to play general video-games via an object embedding network. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE, 2018.

Jiajun Wu, Joshua B Tenenbaum, and Pushmeet Kohli. Neural scene de-rendering. In *Proc. Proc. IEEE Conference on Computer Vision and Pattern Recognition*, volume 2, 2017.

Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhutdinov, Richard Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. *arXiv preprint arXiv:1502.03044*, 2015.

Copyright Clearance

License Number	4915910599984
License date	Sep 25, 2020
Licensed Content Publisher	The American Association for the Advancement of Science
Licensed Content Publication	Science
Licensed Content Title	Neural scene representation and rendering
Licensed Content Author	S. M. Ali Eslami,Danilo Jimenez Rezende,Frederic Besse,Fabio Viola,Ari S. Morcos,Marta Garnelo,Avraham Ruderman,Andrei A. Rusu,Ivo Danihelka,Karol Gregor,David P. Reichert,Lars Buesing,Theophane Weber,Oriol Vinyals,Dan Rosenbaum,Neil Rabinowitz,Helen King,Chloe Hillier,Matt Botvinick,Daan Wierstra,Koray Kavukcuoglu,Demis Hassabis
Licensed Content Date	Jun 15, 2018
Licensed Content Volume	360
Licensed Content Issue	6394

Volume number	360
Issue number	6394
Type of Use	Thesis / Dissertation
Requestor type	Author of the AAAS published paper
Format	Print and electronic
Portion	Figure
Number of figures/tables	4
Title	Incorporating Prior Knowledge Into Deep Neural Networks Without Handcrafted Features
Institution name	Imperial College London
Expected presentation date	Oct 2020
Portions	Parts of figures 1, 2 and 3 and Figure 5
Requestor Location	Imperial College Student Flat 50, 98 Camley Street London, N1C 4PF United Kingdom Attn: Imperial College Student
Total	0.00 USD

Terms and Conditions

American Association for the Advancement of Science TERMS AND CONDITIONS