UNIVERSITY OF OULU

FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Roni Latva
Santtu Orava
Casimir Saastamoinen

AUTOENCODER IMPLEMENTATIONS IN THE PREDICTIVE CODING FRAMEWORK

Bachelor's Thesis
Degree Programme in Computer Science and Engineering
June 2022

Latva R., Orava S., Saastamoinen C. (2022) Autoencoder Implementations in the Predictive Coding Framework. University of Oulu, Degree Programme in Computer Science and Engineering, 42 p.

ABSTRACT

We study the implementation and functionality of autoencoders based on the predictive coding model and the free energy framework, which have seen relatively little experimentation. This framework offers an alternative approach to constructing artificial neural networks in place of traditional backpropagation networks. The limited number of studies published on the subject indicate that the framework could provide better solutions to applications employing artificial intelligence.

This work is meant to accessible to any university student wishing to gain a preliminary understanding for the concepts involved. To this end we provide a detailed walkthrough of the core mathematical ideas behind the implementation using Bogacz's great tutorial as a guide.

We document the implementation process of two autoencoders that learn to recreate handwritten digits from the MNIST dataset in an unsupervised learning scenario. Both of these implementations utilize fully connected layers and are tasked with encoding and decoding of handwritten digits from the MNIST dataset. We analyze graphs of the different variable values and compare the final images produced by the autoencoder to the original ones.

The first implementation is an attempt at constructing an original network and serves as an example of how error sensitive the construction of these networks from the ground up can be. We study the applicability of the theory of predictive coding in practice and diagnose the issues that we encounter. In particular, we showcase problems relating to the update of variances within the network and general difficulties in achieving convergence for all nodes in the network.

The second implementation is built on top of a predictive coding library built by B. Millidge and A. Tschantz and showcases the potential of predictive coding model as a basis for a functional autoencoder. We partially replicate the results obtained by Millidge to establish a baseline for the network's performance. Furthermore, we study the effects of tuning different aspects of these networks to better understand the function of these types of networks. These aspects include the network depth, number of nodes per layer and activation functions. Subjective evaluation on the effects of these modifications is conducted.

Our findings regarding the second implementation indicate that the most important factor in determining final image quality and classification capability is the width of the code layer of the autoencoder. Our experiments using different activation functions do not reveal significant performance gains for any of the functions used. Lastly, we look at the effects of deepening the network but find equal or worse performance when compared to shallow networks.

Keywords: Predictive coding, free energy principle, neural network, autoencoder, MNIST

TABLE OF CONTENTS

1. INTRODUCTION

Understanding how the brain extracts information from the stimuli it receives is an important long-standing problem in neuroscience. The predictive coding model coupled together with Fristons later work on the free-energy principle offer a compelling framework for examining this problem. In addition to locality, Rao & Ballard demonstrated that the nodes in this type of network exhibit behavior similar to biological neurons in that the features a node recognizes resemble receptive fields of neurons [1]. These types of connections between predictive coding and biological brains make the theory relevant to neuroscience. In short, the predictive coding model suggests that the brain generates predictions about its own states. It calculates prediction errors defined as the difference between the predicted and the actual activities of neurons. Then improves its predictions over time by minimizing these prediction errors. According to Friston, this process can be understood more generally as the minimization of the system's statistical free-energy [2]. This explanation is quite interesting because it has been proposed that networks structured in this manner could potentially be implemented in a biologically plausible way. Computations carried out by a predictive coding network are local and only depend on the activity of presynaptic and postsynaptic neurons or nodes excluding couple nuanced cases. This emphasis on locality is a key difference between a predictive coding network and a classical artificial neural network that utilizes a global loss function in learning.

After Rao & Ballards paper demonstrated the initial connections to biological circuits through emphasis on locality, plasticity and the exhibited receptive field effects, the field of predictive coding has been expanded upon significantly. These extensions include the role of action in error minimization, learning the variance of features (as opposed to just the mean values of features), formal mathematical work and further connections to other related theories in the study of perception, namely the free-energy principle. However, relatively little work has been done on studying implementations of predictive coding networks.

The work that has been done on this suggests predictive coding can outperform traditional artificial neural networks in certain tasks [3]. Because of this, we feel it is prudent to probe the efficacy of predictive coding networks in different tasks as the model could provide superior solutions to existing and future problems where artificial intelligence is employed.

We will implement an autoencoder following the mathematics and principles provided in Bogacz's tutorial [4] by coding everything ourselves using Python as the programming language. We will also utilize the implementation of the predictive coding in Python by Millidge[5] as our second implementation's framework. Our final goal is to compare these two implementations by testing what results they give and how many iterations they need. Our hope is that by evaluating the results of each implementation we can help future work on practical implementations of predictive coding networks.

1.1. Background

The idea of perception as an unconscious inference was first presented by Hermann von Helmholtz [6] and can be seen as the precursor for later work which led to the conception of predictive coding. The predictive coding model as such properly originated in the late 1900s. While there were other studies published on predictive coding in the 1980s and 1990s it was the work of Rao Ballard [1] that acted as the catalyst for newfound interest in the field. The paper showcased predictive coding in the context of artificial neural networks for the first time which yielded new practical results and the previously mentioned possible biological connections. They demonstrated through simulations that a network consisting of both feedforward and feedback connections was capable of exhibiting some extra-classical receptive-field effects such as back-stopping.

The biological plausibility of the predictive coding model has made it an exciting research topic in both neuroscience and computational fields as it suggests possible explanations for empirical observations from studies on the different mechanisms of the brain. It should be noted however, that there are many unanswered questions regarding both the function of biological neurons and the predictive coding model's accuracy in representing those neurons. The biological plausibility has two constraints. Firstly, a neuron must do computations only based on its input neurons and synaptic weights associated to those inputs. Secondly, the change of synaptic plasticity has to be based only on the pre-synaptic and post-synaptic neurons activity. This means that the synaptic weights must only be altered by the two neurons which the synapse connects.

The model described in Rao and Ballard's paper [1] is a network where higher level nodes predict the activity of lower level nodes through a feedback connection and the lower level nodes send prediction errors to higher levels through a feedforward connection. These prediction errors describe the difference between the prediction and the actual activity and they are used to enhance the predictions made at the higher level about the lower level. The network learns by maximizing the likelihood of model parameters given the stimulus the network encounters. This maximization is implemented by minimizing a derived optimization function through gradient descent. This structure forms the basis for any predictive coding network.

A later model on predictive coding developed by Karl Friston [7] did not completely satisfy the biological constraints. The alterations and extensions needed to mend the model are introduced in a tutorial written by Rafal Bogacz [4]. In this tutorial Bogacz illustrates how learning can be framed as the minimization of free-energy. This ties together predictive coding and an idea called free-energy principle presented in a paper written by Friston, Kilner and Harrison [8]. The tutorial builds up the mathematical elements of the model and applies them to simple scenarios that illustrate the functionality of the model. Our first implementation will refer to the mathematical equations used in Bogacz's tutorial [4]. In particular, we will utilize the internal dynamics and the parameter update rules. We feel Bogatz's tutorial [4] is an excellent introduction to the predictive coding model, and we provide a detailed walkthrough of this paper in Section 2.

1.2. The Free Energy Principle

The free energy principle aims to describe how the brain works by stating that every change in the brain is trying to minimize free energy. It was first introduced by K. Friston, J. Kilner and L. Harrison [8]. Minimizing free energy is described with Bayesian equations. The free energy principle states that the brain has internal states and the world has external or hidden states. These states do not interact directly, but the external states provide information which is seen as sensations by the internal states in predictive coding. The sensations affect internal states which in turn can respond to the sensations by taking actions to affect the external states. These actions try to steer the world toward something that is less bewildering and the brain is more familiar with. The brain tries to minimize surprise. Brain continuously corrects its internal model in order to better match its own model of reality.

Mathematically the minimization of free energy of a system can be achieved by minimizing the Kullback-Leibler divergence of two probability distributions. The first distribution is the system's model for the environment and the second is the true probability distribution of the environment. Kullback-Leibler divergence can be hard to understand but it can be thought of as the ratio of the two distributions with the property that this ratio gets larger as the distributions get farther apart and goes to zero if the distributions are equal.

1.3. Predictive Coding Network Models

Predictive coding networks can be employed to accurately perform a variety of different artificial intelligence tasks. In their paper on deep predictive coding for video prediction[9] Lotter and Kreima use neural network named "PredNet" which is based on the predictive coding paradigm to predict future frames of a video sequence. Their network was able to predict future movements of synthetic objects well and scale up to complex real world image streams from which it was able to take into account parallax and movement of other objects. Another example of a predictive coding network model is the deep predictive coding network used for recognising objects in a 2018 study [3]. In this study, a deep predictive coding network was evaluated against classical and state-of-the-art models in multiple image recognition tasks. The predictive coding network outperformed its feedforward-only counterpart and was also competitive with the much wider and deeper state-of-the-art models. Additionally, the study makes note of the fact that the predictive coding network was able to reconstruct the input image with good accuracy from its top-down representations even when it is trained to classify images (as opposed to generate them).

A deep predictive coding network used for object recognition is introduced in Beren Millidge's study about different applications of the free energy principle[5]. In this study Millige considers a traditional convolutional neural network (CNN) and implements feedforward, feedback and recurrent connections forming a predictive coding network (PCN). Then he compares this PCN against the CNN. He found that the PCN outperforms the CNN in various tasks. The PCN achieved higher accuracy given more cycles of computation. Compared with state of the art models it achieved competitive performance with fewer layers and less optimization, leaving room to improve and iterate on this design.

A predictive coding algorithm used to encode elements which can be predicted across time is presented in a study about planning in latent space[10]. In this study the writers showcase how using a predictive coding network for encoding and then applying reinforcement learning to the generated latent space allowed them to achieve better performance on a challenging modification of a DeepMindControl task. These are tasks where the agent must perform certain actions based on its environment.

It has been discovered that applying a reinforcement learning algorithm to these situations directly is inefficient due to the complexity of the scenes. Because of this, some type of encoder is usually employed to reduce the dimensionality of the problem to a point where reinforcement algorithms can solve them effectively. When the backgrounds of these environments (which should be irrelevant to the control tasks) were replaced with natural video, the predictive coding network outperformed other encoder solutions[10]. This indicates that the predictive coding model may be more efficient at extracting the most important representations from the input space when learning the latent states. This property of a system being robust against excessive noise in the input could be crucial in certain real world applications of automated systems. The predictive coding encoder also stayed competitive in the standard non-natural video background task where noise in the input space was reduced.

1.4. Biological Constraints

The possible biological plausibility of the predictive coding model has made it an exciting research topic in neuroscience and computational fields. The constraints mentioned in Section 1.1 were that a neuron has to make computations based only on its input neurons and that the change of synaptic weights must be based only on the pre-synaptic and post-synaptic neurons activity. This means that the synaptic weights have to only be altered by the neuron pair which is connected by the synapse. There has been limited research focused on tightening these constraints.

In [11] the writers make two modifications to the predictive coding framework to better match neurological reality. The first modification is related to the issue of bidirectional synapses and transferring the transpose of forward weights. In [11] the writers propose two solutions for achieving this. The first of these is by using random matrices instead of the transpose, derived from research done on back propagating networks. The second proposed solution uses a method developed by Kolen and Pollack [12] and extended by Akrout et al. [13], in which they train backward matrices so that they converge to match the transposes of the forward matrices.

The second modification relates to the continuous values of neurons interpreted as the firing rate of the neuron. This value can go negative, but biologically neurons never have negative firing rates. In the paper the neuronal activity is fixed to zero if it gets negative. This however will lead to a gradient loss, so to prevent the loss a constant positive bias term is added to the model. This way the values will always stay positive. The writers present three error neuron encoding schemes: subtractive separated encoding, subtractive threshold encoding, and division mismatch encoding. These are all aimed to keep the neuron firing rates positive and biologically plausible.

Comparing the resulting network against a traditional backpropagating network and an unmodified predictive coding network using the MNIST dataset demonstrates that the modifications have little impact on performance.

1.5. Autoencoder Applications

Autoencoders are neural networks that take in data, compress and encode the input data and then reconstruct the data from the encoded representation to produce an output of the same size as the input [14]. An autoencoder is trained until loss or error is minimal and the image or data that is fed in is recreated as accurately as possible. Because the data is compressed, the autoencoder has to learn the important features found in the data. These features are sometimes called components. Autoencoders can be trained to do various things such as removing noise from images and detecting anomalies from a given data set.

In [5], Beren Millidge presents an autoencoder based on the predictive coding framework. The autoencoder model demonstrates capability by recreating digits from the MNIST dataset successfully, also recreating unseen MNIST digits with a slightly lower fidelity. Additionally it can generate (or, as they put it, "dream") new never before seen digits. Millidge also tested the network with CIFAR images, achieving very high fidelity. Having learned the latent space, the network was able to interpolate between images. We use this autoencoder as reference when evaluating the results of our second implementation, which is built on top of the same predictive coding library done by Millidge and Tschantz [15].

1.6. Scope

The predictive coding model can be described mathematically through variational methods that formalize the ideas contained within the theory and links it to other related subjects [16]. However, in this thesis we focus primarily on the software side of the implementation since in order to scrutinize the mathematical aspects of the framework a substantially deeper mathematical background than the authors possess is required. Even so, there are many important mathematical concepts and steps that need to be understood in order to make sense of how the network functions. To this end, we provide a walkthrough of the required concepts in Section 2.

Once a functional network has been obtained, we will devote our attention to studying its properties and comparing it to other similar implementations, namely [15]. As for the scope of our implementation, we will initially limit ourselves to studying the autoencoder in an unsupervised learning task utilizing the MNIST dataset of handwritten digits. The most important aspect will be the proper function of the network, meaning that it performs reasonably well in its task and that the implementation follows the predictive coding model properly. Where we are forced to deviate from the theory, this will be noted and its implications acknowledged accordingly.

The dataset we use in the training and testing is the MNIST dataset which contains a large collection of images of handwritten digits [17]. The images are 28x28 pixels in size and are in grayscale. The dataset contains 60000 training images and 10000 testing images. Usually it is used to train classifier type neural networks where the output layer is connected to the correct labels for each image. In this thesis we use the MNIST dataset to train an autoencoder in an unsupervised setting where correct labels will not be presented to the network.

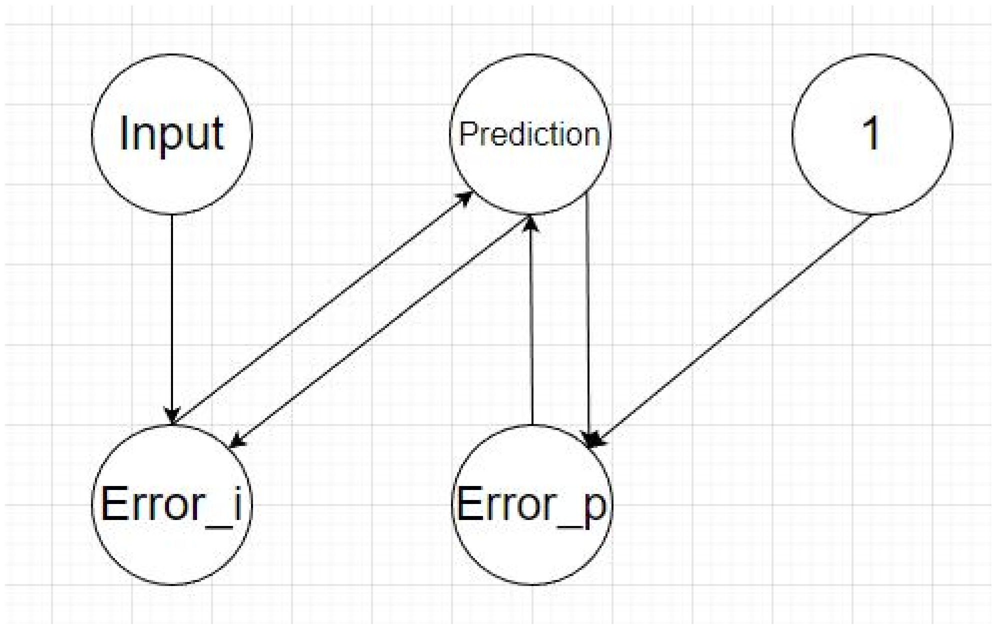

Figure 1. A minimalistic representation of a predictive coding network. In Section 2 we showcase the dynamics which lead to convergence of the prediction and error values.

2. PREDICTIVE CODING WALKTHROUGH

In this section we provide a walkthrough of the core principles guiding the predictive coding paradigm. We aim to provide the reader with an intuitive feel for the type of network we are building. This section is heavily inspired by Bogacz's tutorial [4].

2.1. Task Introduction and Framework Setup

In order to help the reader gain an understanding of the predictive coding framework we first assume the simplest conceivable scenario of perception. In this scenario, we task the network with inferring a single feature from a single stimulus. We borrow the example used in [4] where this single feature is the diameter of a food object s and the stimulus is the light intensity i that is observed from the object. We assume that these two values are connected via some function f. This function could in principle be of an arbitrary form but for simplicity we will assume $f(s) = s^2$. Intuitively, this equation could be interpreted to simply mean that the light intensity reflected off an object is proportional to its surface area (as s is a measure of length, s^2 would be a measure of area). Also, we will assume that i (our observed stimulus) is noisy. This means that instead of i always equaling $f(s)$ it is distributed according to some probability distribution. For simplicity, let us assume that i is normally distributed with mean $f(s)$ and some arbitrary variance $\Sigma_i > 0$. In addition, we will assume that over time the network accumulates some sense for what size the food objects tend to be. Like the observed light intensity, we will assume that this "prior" expectation for the size is normally distributed with mean s_p and variance Σ_p. We can now write the probability density of any expectation $p(s)$ or observation $p(i|s)$ as

$$N(x; \mu, \Sigma) = \frac{1}{\sqrt{2\pi\Sigma}} \exp\left(-\frac{(x-\mu)^2}{2\Sigma}\right). \tag{1}$$

In the case of the observation we substitute i, $f(s)$ and Σ_i for x, μ and Σ respectively. For the expectation we similarly substitute the values s, s_p and Σ_p. This completes the setup for the assumed scenario.

We will now tackle the problem of predicting the size of a food object based on the observed stimuli and prior knowledge about food object sizes. Our objective is to find the most likely size i.e. $p(s|i)$ given the stimuli and our prior knowledge. This probability density can be calculated exactly using Bayes' Theorem

$$p(s|i) = \frac{p(s)p(i|s)}{p(i)}. \tag{2}$$

The left hand side of equation (2) denotes the posterior probability of the diameter s given observed light intensity i. The term "posterior" signifies the fact that the probability distribution of s is evaluated after the evidence i has been observed. On the right hand side, the term $p(s)$ is called *prior probability* and it measures the probability of encountering an object of size s in general, without any knowledge about the stimulus. The term $p(i|s)$ is called *likelihood* and it is a measure of how well i the observed evidence fits our guess of the size s. Both terms in the numerator are easily obtained by substituting the corresponding values into equation (1). The term $p(i)$ of the right hand side denominator is called *marginal likelihood* or *model evidence* and acts as a static normalization term for the whole expression. That is to say, it ensures that the integral of the probability distribution $p(s|i)$ equals 1. Its value is defined as the integral

$$p(i) = \int p(s)p(i|s)ds. \tag{3}$$

In principle, knowing what all the terms are, it is possible to compute the entire probability distribution $p(s|i)$ for some i. From this distribution we can identify the value of s which maximizes the value of the probability density function as our best guess. As the evidence i changes so too does our best guess. This process of updating the probability of a hypothesis (like our guess for s) based on new evidence is called Bayesian inference and it is the central element of the predictive coding model as it is the method by which the network updates its predictions.

2.2. Solution Approximation with Model Variables

There are, however, two problems with this straightforward application of Bayes' Theorem to figure out the most likely value of s in terms of the model's biological feasibility. The first and more easily recognizable problem is that the evaluation of equation (2) requires the computation of the normalization term $p(i)$ which is defined through the integral in equation (3). This computation is generally thought to be infeasible in a biological circuit as it implies the summation of infinitely many values. The second problem is that the representation of the probability distribution $p(s|i)$ may require infinitely many calculations if it cannot be represented neatly with a finite amount of numbers (unlike for example a normal distribution which can be represented with just its mean and variance). According to Bogacz, this type of standard shape cannot be guaranteed as soon as the function f is non-linear (like it is in this example).

To avoid these problems we will focus solely on finding the value of s that maximizes the probability density $p(s|i)$ instead of mapping out the entire distribution. We call this the most likely size of the food object ϕ. When we calculate the posterior probability $p(\phi|i)$ we employ a similar equation as we did for s in equation (2). The posterior probability $p(\phi|i)$ is what we wish to maximize. A key difference in contrast to the exact solution in Section 2.1 is that the denominator $p(i)$ is independent of ϕ. This means that in looking for the ϕ that maximizes $p(\phi|i)$ we can ignore the denominator in equation (2) and focus only on maximizing the numerator $p(\phi)p(i|\phi)$. In order to make the following computations simpler (and to make the connection of this procedure to free energy more apparent later) we now take the logarithm of the numerator and use the property $\ln{(ab)} = \ln a + \ln b$ to write the quantity we are trying to maximize as

$$F(\phi) = \ln p(\phi) + \ln p(i|\phi). \tag{4}$$

Since "the logarithm" is a monotonic function it is maximized by the same value of ϕ as the numerator $p(\phi)(i|\phi)$. When we expand the two terms in F using the normal distributions defined earlier in equation (1) we obtain the following equation:

$$\begin{aligned} F &= \ln N(\phi; s_p, \Sigma_p) + \ln N(i; f(\phi), \Sigma_i) \\ &= \ln\left[\frac{1}{\sqrt{2\pi\Sigma_p}} \exp\left(-\frac{(\phi - s_p)^2}{2\Sigma_p}\right)\right] + \ln\left[\frac{1}{\sqrt{2\pi\Sigma_i}} \exp\left(-\frac{(i - f(\phi))^2}{2\Sigma_i}\right)\right] \\ &= \frac{1}{2}\left(-\ln\Sigma_p - \frac{(\phi - s_p)^2}{\Sigma_p} - \ln\Sigma_i - \frac{(i - f(\phi))^2}{\Sigma_i}\right) + C. \end{aligned} \tag{5}$$

The term C contains constant values that arise from expanding the expression in the second row. This term can safely be ignored since it will vanish during differentiation. By computing the partial derivative of F with respect to ϕ we obtain the direction in which we should move ϕ in order to increase the value of F the fastest. When F settles to its maximum (the calculated derivative becomes sufficiently small) we have found the approximate value of ϕ which maximizes $p(\phi)p(i|\phi)$ and subsequently $p(\phi|i)$ and thus the inference problem is solved. The partial derivative we are looking for works out to be:

$$\frac{\partial F}{\partial \phi} = \frac{s_p - \phi}{\Sigma_p} + \frac{i - f(\phi)}{\Sigma_i} f'(\phi). \tag{6}$$

In order to find the ϕ that maximizes F we simply have to calculate this derivative with some initialized values for the model parameters s_p, Σ_p and Σ_i. We will also have to specify the function f, the observed light intensity i and an initial guess for ϕ. Once the equation has been initialized, gradient ascent is performed by evaluating $\frac{\partial F}{\partial \phi}$, updating ϕ by adding to it this derivative times some small constant Δt and then repeating these two steps for some sufficient number of rounds. Bogacz notes that this derivative can be understood intuitively as a composition of two simultaneous effects acting on ϕ. The first term in equation (6) moves the current guess for ϕ towards the mean of the prior distributions while the second term moves ϕ depending on how well ϕ's true light intensity matches the observed light intensity. Additionally both terms are weighted by the variances of their associated distributions which can be understood as the model assigning reliability to these two measures. To keep all of the notation as simple as possible we will assign two new variables to these measures:

$$\epsilon_p = \frac{\phi - d_p}{\Sigma_p}, \qquad \epsilon_u = \frac{i - f(\phi)}{\Sigma_i}. \tag{7}$$

These two terms are called prediction errors and they encode how much the predicted size ϕ and the observed light intensity i differ from their respective expectations. Rephrasing equation (6) using the two measures in equation (7) gives us a simpler representation of the update rule for ϕ:

$$\frac{\partial F}{\partial \phi} = \epsilon_i f'(\phi) - \epsilon_p. \tag{8}$$

Similarly, we can derive dynamics for the prediction errors

$$\frac{\partial F}{\partial \epsilon_p} = \phi - s_p - \Sigma_p \epsilon_p, \tag{9}$$

$$\frac{\partial F}{\partial \epsilon_i} = i - f(\phi) - \Sigma_i \epsilon_i. \tag{10}$$

To understand where these dynamics come from we assume a state in which the network has converged to an equilibrium. In this state equation (9) and equation (10) equal 0. If we then solve these two equations for the prediction errors with the left hand side set to 0 we obtain the two forms in equation (7). Thus these dynamics provide us with a structure that has fixed points at desired values. Through these three differential equations we can also understand the implied architecture of the network. For instance, the node ϕ is connected to node ϵ_i with strength $f'(\phi)$ and node ϵ_p with strength 1. Nodes ϵ_i and ϵ_p implement their dynamics similarly . Positive signs indicate excitatory connections and negative signs inhibitory connections. With these dynamics implemented it is possible to infer the most likely size ϕ, given a static observed light intensity i, by following the flow defined by equations (8)-(10) for variable ϕ for a sufficient amount of time. It should also be noted that the above rules satisfy an important condition considering the biological feasibility of this model. This condition is referred to as local computation which demands that every neuron in the network only carries out calculations involving values of its input neurons (meaning physically neighbouring neurons) and their associated weights.

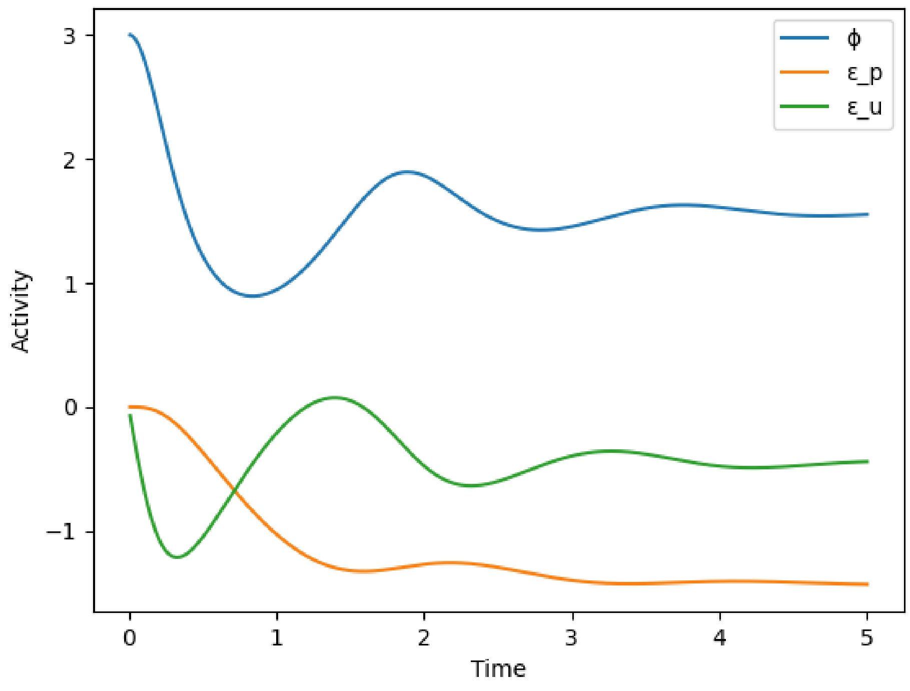

Figure 2. Convergence of prediction ϕ and prediction errors ϵ_i and ϵ_p.

2.3. Learning with Model Parameters

The logical next step in making the network more robust is to implement a mechanism by which it can improve the accuracy of the model parameters s_p, Σ_p and Σ_i. These parameters essentially encode the network's "world view" and their refinement can be understood as the network learning something new about the world it inhabits. According to the free energy principle the internal dynamics of a perceptual system can be interpreted, in terns of Bayesian inference, to maintain a probabilistic description of the outside world that minimises the surprisal of sensory input overtime. With this principle in mind it is sensible to choose the model parameters in such a way that the light intensities the network encounters are as expected as can be. This means we have to maximize $p(i)$ in relation to the model parameters. However, as mentioned earlier computing $p(i)$ is difficult and considered infeasible by a biological circuit. Because of this, a different approach is required. Luckily such an approach exists and it will simultaneously solve the problem at hand and illustrate a clear connection between the predictive coding model and the free energy principle.

To tackle this issue we will reframe a step we made earlier. Instead of finding the value of ϕ that maximizes $p(\phi)p(i|\phi)$ we will look for a simple probability distribution that best approximates the actual posterior distribution $p(s|i)$. Importantly, we have complete control over the shape of the new approximate distribution $q(s)$ so we can be sure that it can be characterized by a finite number of parameters. Following Bogacz's example we will define $q = \delta_\phi$ where δ_ϕ is a delta distribution centered at ϕ. The delta distribution is a probability distribution that has all of its mass concentrated at a single point resembling an impulse. Additionally its integral is defined to equal 1. As before, ϕ denotes the most likely value of s, given some observed i. Our objective now is to minimize the dissimilarity between the probability distributions $q(s)$ and $p(s|i)$ which is formally measured as their Kullback-Leibler divergence

$$KL(q(s), p(s|i)) = \int q(s) \ln \frac{q(s)}{p(s|i)} ds. \tag{11}$$

We are still unable to minimize this quantity directly since $p(s|i)$ includes the term $p(i)$ that we are trying to avoid. To circumvent this we must apply the definition of conditional probability to the term $p(s|i)$ i.e. $p(s|i) = \frac{p(i,s)}{p(i)}$. With this substitution we get

$$\begin{aligned} KL(q(s), p(s|i)) &= \int q(s) \ln \frac{q(s)p(i)}{p(i,s)} ds \\ &= \int q(s) \ln \frac{q(s)}{p(i,s)} ds + \int q(s) ds \ln p(i) \\ &= \int q(s) \ln \frac{q(s)}{p(i,s)} ds + \ln p(i). \end{aligned} \tag{12}$$

Here we have divided the integral into two parts using the property $\ln{(ab)} = \ln a + \ln b$. Additionally, we have used the fact that the integral of the delta distribution is 1. From this we can see that the term $\ln p(i)$ does not depend on ϕ and so the value of ϕ that minimizes the integral in the third line is the same value that minimizes the whole expression. Bogacz calls this integral free energy in [4]. Let us denote this integral by E and see what form it takes when $q = \delta_\phi$, as assumed earlier:

$$\begin{aligned} E &= \int q(s) \ln \frac{q(s)}{p(i,s)} ds \\ &= \int q(s) \ln q(s) ds - \int q(s) \ln p(i,s) ds \\ &= C_1 - \ln p(i,\phi). \end{aligned} \tag{13}$$

Again we divide the integral in two using the fact that $\ln \frac{a}{b} = \ln a - \ln b$ and apply the knowledge that $\int h(x)\delta(x) = h(\phi)$ when h is any function and δ is a delta distribution with the center of ϕ. Also, since the first integral does not depend on ϕ we simply assume it to be some constant C_1 (since it will disappear when we later perform the gradient ascent). If we now ignore the constant and apply the definition of conditional probability we obtain that $E = -\ln p(\phi)p(i|\phi)$ which is just the negative of equation (4) i.e. $-F$. This means that the task of minimizing the Kullback-Leibler divergence for the defined distributions, which is the same as finding the the most likely value of ϕ given a stimulus i and solved by minimizing E, is equivalently solved by maximizing F.

To achieve our original goal for this reframing let us see what this setup tells us about the computationally problematic $p(i)$. Denoting the integral of the last line in equation (12) as E (or $-F$) and rearranging to solve for $p(i)$ gives

$$\ln p(i) = KL(s,i) - E = KL(s,i) + F. \tag{14}$$

Here we can see that due to Kullback-Leibler divegence being non-negative, $\ln p(i)$ is maximized by either minimizing E or maximizing F. Thus we conclude that the model parameters can be optimized by maximizing F since maximizing F maximizes $\ln p(i)$. The difference in the optimization of the model parameters in contrast to the variables ϕ, ϵ_p and ϵ_i is that we want the parameters to maximize $p(i)$ over many different observations. Therefore, we will only slightly change the parameter values per observation instead of letting the system run to convergence (i.e. till F is maximized (or E is minimized)). Finally, we can obtain the dynamics for updating the model parameters by simply taking the partial derivative of F (equation (5)) with respect to each parameter

$$\frac{\partial F}{\partial s_p} = \frac{\phi - s_p}{\Sigma_p} = \epsilon_p, \tag{15}$$

$$\frac{\partial F}{\partial \Sigma_p} = \frac{1}{2}\left(\frac{(\phi - s_p)^2}{\Sigma_p^2} - \frac{1}{\Sigma_p}\right) = \frac{1}{2}\left(\epsilon_p^2 - \frac{1}{\Sigma_p}\right), \tag{16}$$

$$\frac{\partial F}{\partial \Sigma_i} = \frac{1}{2}\left(\frac{(i - f(s))^2}{\Sigma_i^2} - \frac{1}{\Sigma_i}\right) = \frac{1}{2}\left(\epsilon_i^2 - \frac{1}{\Sigma_i}\right). \tag{17}$$

We note here that all the above rules satisfy the condition of local plasticity. This means that when the network learns (when it tunes its parameters) the updates on the values of these parameters only depend on the activity of pre-synaptic and post-synaptic neurons. In other words, the change in the connection weight between neurons only depend on the activity of those particular neurons. Sometimes this principle is referred to as Hebbian learning [18]. The intuition behind it is that neurons that "fire together, wire together."

With dynamics (8)-(10) and (15)-(17) the network could now be initialized randomly and it could start learning to model its world through the model parameters. This is executed by inferring the most likely size of a food object given the light intensity the network observes (and given its current sense of the world) by maximizing F with respect to all model variables and then by updating its model model parameters (i.e. updating its sense of the world) by taking a small step in the direction of equations (15)-(17). By repeating this process over many different stimuli the network would learn to model its environment correctly meaning that it would predict food object sizes accurately.

The separation of time scales in updating the variables (ϕ, ϵ_p and ϵ_i) and the parameters (s_p, Σ_p and Σ_i) has a biological interpretation: the variables change often so their values indicate activity of neurons and the parameters change less often so their values are coded in the weights of synaptic connections between neurons. At the same time, this temporal separation can be seen as a potential flaw if biological circuits do not function with similar separation taking place.

2.4. Tuning the Activation Function

The last aspect of the network we have yet to cover in this simple case of a single inferred variable from a single observed stimulus is the function f relating these two. At the start we defined f as $f(s) = s^2$ but this information may not be axiomatically known by the network. If f is not known, it should be learned. For our purposes, this examination of f can be divided into two distinct scenarios: one, in which this function is linear i.e. of the form θs, and another where the function is nonlinear i.e. of the form $\theta h(s)$ where h is nonlinear function of s. Here θ is a parameter of $f(s, \theta)$ and it can be learned like all the other model parameters. The linear case turns out to be simple while the nonlinear case presents challenges for the neural implementation side of the model. In both scenarios, obtaining the dynamics for the model variables is easy as we only need to substitute the aforementioned forms in place of f in equations (8)-(10). It is similarly straightforward to obtain the update rule for θ as the partial derivative of F over θ as F now depends on θ. Dynamics for the linear case are given by the differential equations

$$\frac{\partial F}{\partial \phi} = \epsilon_u \theta - \epsilon_p, \tag{18}$$

$$\frac{\partial F}{\partial \epsilon_p} = \phi - s_p - \Sigma_p \epsilon_p, \tag{19}$$

$$\frac{\partial F}{\partial \epsilon_i} = i - \theta\phi - \Sigma_i \epsilon_i, \tag{20}$$

$$\frac{\partial F}{\partial \theta} = \epsilon_u \phi. \tag{21}$$

Analogously, the dynamics for the nonlinear case become

$$\frac{\partial F}{\partial \phi} = \epsilon_u \theta h'(\phi) - \epsilon_p, \tag{22}$$

$$\frac{\partial F}{\partial \epsilon_p} = \phi - s_p - \Sigma_p \epsilon_p, \tag{23}$$

$$\frac{\partial F}{\partial \epsilon_i} = i - \theta h(\phi) - \Sigma_i \epsilon_i, \tag{24}$$

$$\frac{\partial F}{\partial \theta} = \epsilon_u h'(\phi). \tag{25}$$

Here we note that the update rules for the other model parameters do not need to be reformulated depending on the form of f since the effect of the change in f will be dealt with in the calculation of the prediction errors (see the rightmost form of equations (15)-(17)).

In the linear case, all update rules are Hebbian and satisfy the rule of local computation. In the nonlinear case however, it is not obvious as to how one should organize the network in order for every node and connection to satisfy the constraints that make the model biologically feasible. It should also be noted that in the nonlinear case the network must be initialized with some $h(s)$ since the form of this function (polynomial, rational, exponential etc.) cannot be learned by the model presently. Only the multiplier θ can be tuned. Indeed, the nonlinear case requires further research in order for the model to explain perception and learning in biological circuits completely.

2.5. Multiple Features, Stimuli and Layers

We have now described all the aspects of the predictive coding model needed for operation in the simplest case of perception where we infer a single variable from a single noisy observation. Next we will present how these methods can be scaled up to include multiple stimuli, inferrable features and layers of neurons.

The extension to multiple stimuli and inferable features is relatively straightforward as we only have to reformulate F using the higher dimensional definition of the normal distribution

$$N(\vec{x};\vec{\mu},\boldsymbol{\Sigma}) = \frac{1}{\sqrt{(2\pi)^n \det \boldsymbol{\Sigma}}} \exp\left(-\frac{1}{2}(\vec{x}-\vec{\mu})^T\boldsymbol{\Sigma}^{-1}(\vec{x}-\vec{\mu})\right). \tag{26}$$

Here n denotes the length of the vector $\vec{x}$ and $\boldsymbol{\Sigma}$ is a matrix. Writing the terms $\ln p(\vec{\phi})$ and $\ln p(\vec{i}|\vec{\phi})$ using this new definition and applying the correct means and variance matrices we can obtain a more generalized F. Analogously to the simple one variable version, we can then take the partial derivatives with respect to each model variable and parameter to derive all the update rules necessary for the model to function. To further extend the model to include multiple layers we assume that the activity of a layer is dependent on the activity of the layer above through the function f. Thus,

$$E(\vec{s_i}) = f_i(\vec{s_{i+1}}, \boldsymbol{\Theta_i}) = \boldsymbol{\Theta_i} h_i(\vec{s_{i+1}}), \tag{27}$$

where the matrix $\boldsymbol{\Theta_i}$ is the generalized version of θ and $i = 1$ is the input layer. The likelihood of activity (the current estimation of s or the prediction), given the above layer's activity becomes

$$p(\vec{s_i}|\vec{s_{i+1}}) = N(\vec{s_i}; f_i(\vec{s_{i+1}}, \boldsymbol{\Theta_i}), \boldsymbol{\Sigma_i}). \tag{28}$$

Here the model no longer contains the individually named layers i and p but is instead described using the hierarchy of the layers. We can obtain update rules for the new model variables $\vec{\phi_i}$ and $\vec{\epsilon_i}$ and the model parameters $\boldsymbol{\Sigma_i}$ and $\boldsymbol{\Theta_i}$ by once again forming a new F based on equation (28) and computing the required partial derivatives. In this generalized model $\vec{\phi_i}$ is the activity in layer i, $\vec{\epsilon_i}$ is the prediction error in layer i, $\boldsymbol{\Sigma_i}$ is the covariance matrix for features in layer i and $\boldsymbol{\Theta_i}$ describes how the mean values of features in one layer depend on the next layer. The form of these rules will be displayed later in the implementation Section 3.3.1. It should be noted that there arises a problem with these update rules regarding the biological feasibility aspect of the model. It turns out that the update rule for $\boldsymbol{\Sigma_i}$ does not satisfy the locality constraints defined earlier. Bogacz [4] presents a solution to this in his tutorial and we showcase this in Section 3.3.4.

3. IMPLEMENTATION

In this section we describe our implementation process from the plans and objectives to the results of the implementations. We explain the network architecture and go through the experimentation and testing we did with these implementations.

3.1. Plans and Objectives

The first step in our design was to try and understand the mathematical concepts behind predictive coding and implementing them in Python. We went through the tutorial [4] and did the exercises provided in the paper in Python. This familiarized us with the mathematical concepts of the theory and gave us crude visualizations of the math in action to help grasp the inner workings of the process. Some tinkering of parameters in the code provided insight into their effects on the performance of the system. Due to our prior experience and the availability of great premade libraries like Pytorch we chose to implement these networks using Python.

Following these introductory tasks we set out to implement two different predictive coding networks which we planned to compare once they were complete. The first network was built from the ground up following the structure and mathematics presented in Bogacz's tutorial. The second was constructed on top of an existing implementation by Millidge [15] which contained all the necessary building blocks for our needs. The Millidge implementation contains scripts for both generative and classifier networks so we had to modify them to obtain the autoencoder we wished to obtain. Pursuing two different networks also had the advantage of giving our project some robustness against possible complications that we thought we might encounter. We recognized our limited experience with coding in general and neural network in particular and so we believed it would be wise to give ourselves the opportunity to lean on an already working implementation in the case that our original network did not function as we hoped.

Use cases for the autoencoders we had envisioned include image denoising and image compression. Noisy images could be restored by running them through an autoencoder which has been trained on clear images. For the compression application we can imagine a scenario where a large image file is sent through a low bandwidth connection. Using an autoencoder, the sender could compress the image on their end and the receiver could then recover the original image by decompressing the data they received with the same autoencoder. In practise, however, autoencoders are not used for these types of compression tasks due to the loss that occurs in the encoding and decoding of the image.

For this project, the main goal for both networks was to recreate MNIST digits in a way that they remain classifiable by human observers. The mathematics guiding all the learning that takes place should follow the theory presented by Bogacz's tutorial [4] as closely as possible. There are some key differences between the tutorial and the Millidge API when it comes to implementing these dynamics and they will be highlighted later when we discuss the technical details of that API. Furthermore, we studied the performance impacts of altering different aspects of the network such as the number of layers and nodes and the use of different activation functions and learning rates. We also pursued expanding the networks with more features as the timetable of the project permits it. These features include expanding the networks functionality to datasets other than MNIST, implementation of alternative learning rules and implementation of different connectivity schemes between layers.

3.2. Network Architecture

The initial macrostructure for both implementations was a simple symmetric network. Input layer is where the image is entered into the system. The MNIST dataset contains images of size 28x28 pixels meaning that 784 input nodes are required in total. The network needs to then compress the image into some small number of nodes. Then the network widens again and decodes and reconstructs the image back to 784 output nodes which can then be arranged back to a 28x28 pixel picture. This compression from the initial 784 data points down to a few dozen values, in a way where the original data can be reconstructed from these few values, signifies that the network has learned the most important components of the input space. While input and output layers are constrained by the size of the image, the inner layers can be tuned in terms of their amount and the number of nodes in each layer. To obtain a frame of reference we will initially implement a three layer network with 20 nodes in the middle layer similar to the one implemented by Millidge. Comparing our results to Millidge's will help us confirm whether the network is functioning properly or not.

Both networks will at least initially utilise a fully connected layer structure where each node in a given layer is connected to every other node in the adjacent layers. Although this approach is not optimal from the perspective of learning, it is the most straightforward to implement. Implementing convolutional variants at a later stage could be an interesting experiment since convolutional neural networks (CNN) perform very well on these types of tasks. In a CNN only certain nodes are connected to certain other nodes in the adjacent layers. The motivation behind this design is to help the network figure out the most important components of the input space by not including certain inputs in the same components. For example, it probably does not make sense to include the top-left and bottom-right pixels in the same component since there is a low likelihood that there exists a component sharing these two far away pixels. In essence, we are leveraging our intuition about the plausible components the network might look for and so we make the networks search for components easier.

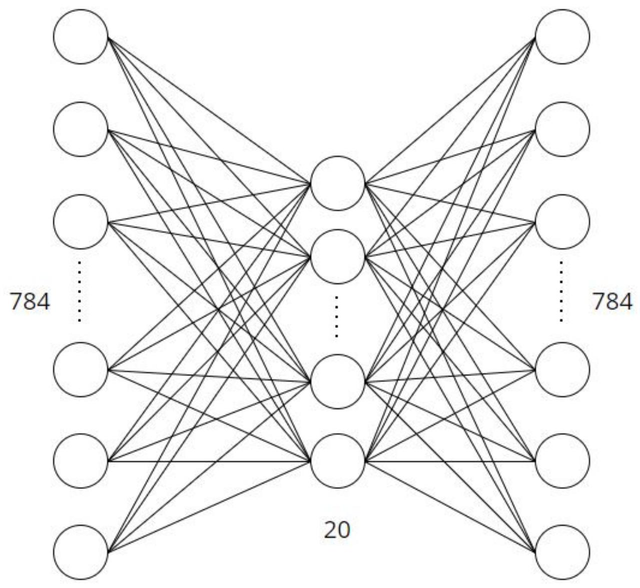

Figure 3. Fully connected design for the network architecture. 784 nodes for input and output layers with 20 nodes in the code layer.

3.3. Bogacz Inspired Original Implementation

We started the implementation process by trying to build a network from the ground up following the principles of the tutorial by Bogacz. The implementation of the first network demonstrated several difficulties associated with the construction of networks employing the predictive coding architecture. After extensive debugging and experimenting with different setups, we conclude that a straightforward translation of the equations in Bogacz's tutorial into code is not sufficient to obtain a functional neural network. The implementation of the model into a neural network revealed issues relating to the convergence of different values in the network. In particular, the update rules for the variances and covariance matrices in the model exhibit unwanted behavior, which we cover in more detail in Section 3.3.4. Additionally, the convergence of model variables in the high dimensional case, where we deal with multiple inputs and outputs, is not self-evident. In general, the model seems to be very sensitive to the initial values of the parameters when multiple inputs and outputs are involved. We have conducted extensive testing with different types of simpler networks in order to gain insight into to inner workings of the predictive coding model. We present all relevant findings in more detail in Sections 3.3.1 - 3.3.4.

3.3.1. Initial Implementation

Having familiarized ourselves with the predictive coding model in the previous phase we set out to implement the network we had envisioned. Our plan was to simply adapt the equations provided in the Bogacz tutorial into an appropriate architecture and then feed the MNIST images into the network as the input. The architecture we settled on would first compress the 28x28 pixel image into the values of 16 code neurons and then decode those values back to the original dimensionality of 784. In addition to the inputs and the prediction nodes, each layer requires the same amount of error nodes. That is to say, the first layer had 784 input nodes and 784 error nodes, the second layer had 16 prediction nodes and 16 error nodes and so on. For the parameters of the network we had to initialize two 784x784 covariance matrices for the input and output layers and one more 16x16 covariance matrix for the code layer. Lastly, for the connection weights between layers we needed weight matrices of dimensions 16x784 and 784x16 and a tonic weight vector with length 784. All of the values were initialized randomly with a uniform distribution between zero and one. Symmetry in the covariance matrices was achieved by multiplying each matrix by its transpose. We then divided the matrices with their maximum values to normalize all values between zero and one. In order for the input image to be fed into the network we flattened the 28x28 array into a vector of length 784 and normalized all pixel values by dividing by 255, which is the maximum intensity of a pixel in a grayscale image. This concludes the initialization of the values in the autoencoder network. We then proceeded to implement the inner loop of the predictive coding model. This is the loop that iterates the values of model variables until they converge. For each loop we would pull a new image from the MNIST dataset and feed it to the network. Here we calculate the partial derivatives of F, which is based on equation (28). These partial derivatives turn out to be

$$\frac{\partial F}{\partial \vec{\phi_i}} = -\vec{\epsilon_i} + h'(\vec{\phi_i}) * \boldsymbol{\Theta_{i-1}}^T \vec{\epsilon_{i-1}}, \tag{29}$$

$$\frac{\partial F}{\partial \vec{\epsilon_i}} = \vec{\phi_i} - \boldsymbol{\Theta_i} h(\vec{\phi_{i+1}}) - \boldsymbol{\Sigma_i} \vec{\epsilon_i}, \tag{30}$$

for predictions and errors respectively. These partial derivatives provide us with the direction we should step in to increase F the fastest and so we can perform gradient ascent by iterating these equations for a sufficient time. Initially we iterated these dynamics for 1000 steps with a learning rate of 0.01. Here, however, we hit our first major problem.

We found that the values of the variables do not always converge. At this point it was very difficult to guess as to why the provided equations would produce this result, since analysing the dynamics is not easy to visualise and monitor in this high a dimension. We proceeded to debug the network by varying the initialization values, but our control was limited here as configuring large matrices is not very efficient. We found that with certain constant multipliers we could get the values to not explode during the iteration, but there was never any guarantee that all the values would indeed converge and not just diverge slowly. These inconsistencies in the behaviour of the model prompted us to simplify the scenario significantly. This would allow us to easily monitor the progression of each value in the system and could potentially aid us in understanding why the more complex case is so sensitive to its initial conditions.

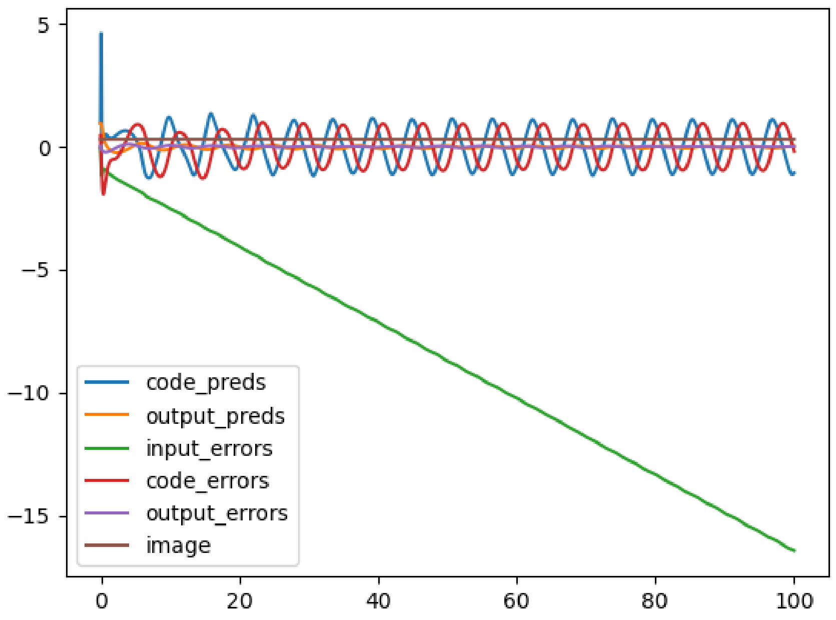

Figure 4. Divergence of different prediction and prediction error values in the complex case. Here we only showcase one arbitrarily chosen prediction and error for each layer (input, code and output) as plotting thousands of variables would make the image unintelligible.

3.3.2. A Simplified Architecture

In order to make sense of the inner workings of the dynamics at play, we simplified the network architecture as much as possible. We constructed the simplest conceivable network, which consisted of a single input, a single output and one error node for each of these nodes. This network was also used in one of the exercises in Bogacz's tutorial [4]. In this model we could easily configure all of the initial values of the system as there were only four nodes and three parameters (two variance values, one for each error node and a single tonic weight). We set things up analogously with the more complex architecture by constructing the inner loop with the appropriate dynamics following equations (8)-(10). For the function f we chose $f(\phi) = \phi^2$ in order to illustrate the effect of non-linearity. For the input, we sampled a random value from a normal distribution with some reasonable mean and variance.

Testing revealed the behaviour of this very simple network to be very robust against different initializations. The system would eventually diverge if the initialized values became unreasonably large (tens, hundreds or thousands). It was, however, possible to bring these divergent scenarios back in line by decreasing the learning rate in the gradient ascent step, which brought the step size low enough. Doing this does naturally result in the system requiring more iteration steps to reach the convergence point. Another nice property of this network is that if the solutions of the system of equations (8)-(10) do converge, the asymptotic values do not depend on the initial values of the variables. That is to say, for a given set of model parameters, the model variables have a unique limit. This is true even for initializations which diverge with one learning rate, but converge with another. The only case where divergence seems to be eminent is when we set one or both of the variances to negative values, which is reassuring, since a negative variance is nonsensical and we would not even wish for the model to function in such a case.

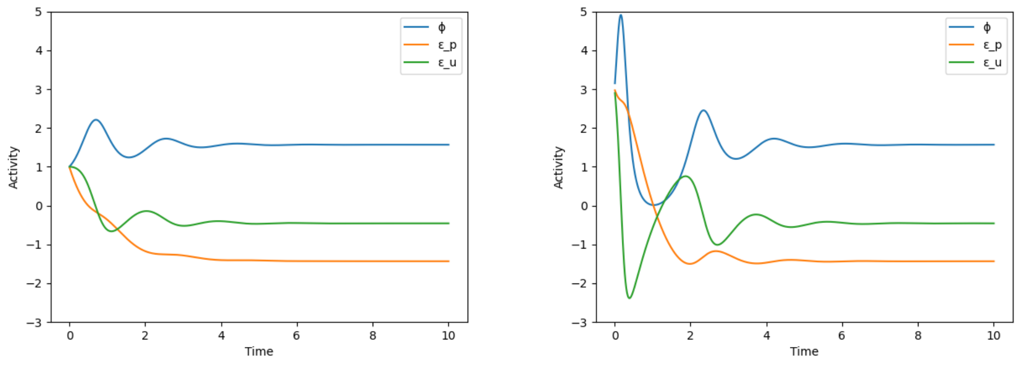

Figure 5. Two different initial values for the variables (prediction ϕ and prediction errors ϵ_p, ϵ_i) lead to convergence towards same asymptotic values (in our code $\epsilon_i = \epsilon_u$). On the left all variables start at 1 whereas on the right they start at 3.

This improved stability encouraged us to try and implement the model parameter update rules on top of this simple network. This was achieved by nesting the inner loop inside an outer loop that would contain the dynamics for the update of the parameters. These dynamics are represented by equations (15)-(17). Due to our choice of the function f we did not implement the update rule for the parameter Θ described by equation (21) or equation (25). A key difference in the implementation of the inner and the outer loops is that while we let the inner loop run for a long time (thousands of steps) in order for the variable values to reach convergence, we only evaluate the parameter updates once per input signal. This is done with the motivation that the parameter values should change with respect to the whole set of possible inputs and not be biased towards any particular input. Thus, we hope to obtain better performance over the whole input space. For testing we ran the outer loop for thousands or sometimes tens of thousands of iterations, choosing the input signal randomly from a predefined normal distribution for each loop. After each run through the outer loop we logged the three parameter values into lists and after the experiment concluded we constructed plots from these lists in order to visualise the behaviour of model parameters.

From these experiments we found that the update rule for the tonic weight behaves appropriately and converges to the real mean value of the chosen normal distribution with good accuracy. Unfortunately, however, the updates for the two variance parameters exhibit unwanted behaviour. These two values seem to be diverging towards negative infinity regardless of initialization of the systems values. This problem with the variance update has been a major area of research for us in our implementation phase and we will go over all the experiments we have conducted in Section 3.3.4. Here we just note that we have not found a perfect solution to this problem.

In summary, this simplification of the network architecture showcases a couple of important aspects of the framework. Firstly, it confirms that the foundation for the equations used for variable updates in the complex case is solid, since the analogous one-dimensional equivalents consistently produce correct results in the simple case. Secondly, it is quite clear that there are additional obstacles left to solve even after sorting out the partial divergence of the inner loop in the higher-dimensional case due to the surprising behaviour of the variance update rules.

3.3.3. Expanding the Simplest Case

To further investigate the behaviour of the model we proceeded to build some slightly more complex networks in comparison to the one in the previous Section 3.3.2. The first of these was a three layer network with a single activity and a single error node in each layer. The setup is completely analogous to the simple case with the obvious differences being the addition of an extra variance parameter and a pair of equations for the third layer's prediction and error. We found that the convergence of the inner loop works out just as in the simple network. The nice properties of a single point of convergence for each variable and imperviousness to initial conditions seemed to transition over from the simple case seamlessly. Implementing the outer loop into this network provided less familiar results. While the variance update resulted in similar behaviour as previously, the tonic weight update behaved unexpectedly. Although the tonic weight seemed to converge consistently, the value it converged to was never the mean of the distribution we sampled our inputs from. We have not investigated this effect fully but our initial hypothesis is that this value related to the real mean of the distribution logarithmically. That is to say, when we plot the values the tonic weight converges to with respect to the true mean, for some constant set of variances, the values follow a logarithmic function $y = a + b \ln \mu$, where μ is the true mean. The reasons and consequences of this behaviour remain unknown and could possibly be investigated further.

For the second expansion of the simple network architecture we increased the dimensionality of the network while keeping the layer count at two. This expansion has undergone a few iterations as we have increased the dimension from two to four during the course of the implementation phase. For these networks we have always matched the number inputs to the number of outputs to keep everything as simple as possible for testing purposes. Here we were able to manually set all the values in the required vectors and matrices, unlike in the original attempt. For the input signal we sampled four values from four distinct normal distributions into a vector. Again, we observed that the inner loop converges consistently but the values of different variables seem more densely grouped than in the one dimensional scenarios. Specifically, all error nodes in a layer seem to converge to the same value and predictions behave similarly. As for the outer loop and associated parameter updates, we found that the values do converge with the exception of the variances as has been the case for all the experiments we have presented so far. The convergence of the tonic weights here depend on the the connectivity of the input and output layer. If we connect the input layer error nodes one-to-one with the corresponding output layer prediction nodes we can learn all the true means of input distributions through the tonic weights. This setup is in some sense just four of our simple 1-1 networks stitched together. When we implement full connectivity between the two layers we no longer obtain the true means but the network does still converge like it did in the three layer experiment. This time, however, the tonic weights' relation to the mean values of all the input distributions is less clear.

The results from these experiments showcase that in principle there should not be any obstacles preventing us from scaling up the network towards the originally envisioned autoencoder structure. Unfortunately, getting the variables to converge in the inner loop remains an issue in practice. A logical next step in our experimentation would be to combine the previous two setups into a three layer network with more than one input and output. Even this might not provide an answer to the initial problem though, since it does seem like the sheer size of the matrices and other elements in the original setup play a role in its behaviour.

3.3.4. Problems with the Variance Update

In this section we take a closer look at the problems regarding the update rules for the variances in the network. These tests were conducted on the simple network with only a single input and output. Firstly, let us briefly analyse the equations

$$\frac{\partial F}{\partial \Sigma_p} = \frac{1}{2}\left(\epsilon_p^2 - \frac{1}{\Sigma_p}\right), \tag{31}$$

$$\frac{\partial F}{\partial \Sigma_i} = \frac{1}{2}\left(\epsilon_i^2 - \frac{1}{\Sigma_i}\right). \tag{32}$$

When we evaluate these expressions the most important aspect of that value is actually its sign. This is because regardless of the amplitude of the value, the sign alone will tell us to which direction to shift the parameter in question to increase F. Of course, the amplitude does encode how far off we still are from the correct value, but the speed at which we converge towards that value is not critical for the functionality of the network. Looking at equations (31) and (32) we can see that the first term is always positive and the second term is always negative because the variances $\boldsymbol{\Sigma_i}$, $\boldsymbol{\Sigma_p}$ are by definition non-negative. Therefore, we can infer that when the value of the variance parameters dives towards negative infinity, the error terms must be small. By studying the limits of the error variables we found that if the tonic weights are close to their correct values then the errors tend to be small. When this happens, the gradients tend to be negative since squaring the error makes it smaller still when we are in the range from zero to one. Negative gradient results in the variance becoming smaller and thus it makes the second term larger. This repeats causing the variance to drop ever lower. In our testing we would often see graphs of these variances dive downwards and then jumping very rapidly upwards once they went negative. This is explained by the fact that when the variance goes negative the gradient is guaranteed to be positive. Additionally, the magnitude of the jump is due to the variance happening to hit a value very close to zero, in which case the magnitude of the second term will be large. A possible solution to this problem could be to initialize the tonic weight far away from true mean of the input distribution, but in general this might not be the correct approach. In the case of the original autoencoder network, the means are not known at the start of the experiment so setting the weights correctly is probably not feasible.

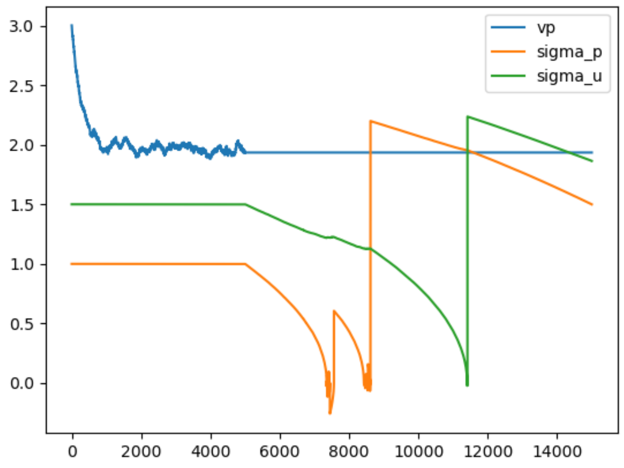

Figure 6. Variances exhibiting unwanted bouncing behaviour. The plotted values sigma_p, sigma_u and vp correspond to Σ_p, Σ_i and s_p respectively.

Another aspect worth considering turns out to be the order of the parameter updates. Since errors tend to be very small when the tonic weights are correct, it may be helpful for the stability of the updating scheme to run the variance updates first for some time and only then move on to updating the tonic weights. It should be noted that updating these parameters simultaneously has not yielded any positive results and the system becomes generally quite chaotic when this strategy is adopted. These ideas concerning the order in the parameter updates were only experimented with at a late stage in the implementation phase and so their full merits are yet to be determined with confidence.

To avoid the observed divergence in our implementation of the variance update rules, we experimented with alternative ways of updating these parameters. One of these was provided in [4] Chapter 5 and another was developed by us. The first alternative approach was to use a new set of equations that utilized an additional node in the network architecture called inter node. This idea was presented as a solution to a locality violation that arose in the multidimensional case of the update rule for the covariance matrices. Bogacz presents the following dynamics for the inter node e_i and the error node ϵ_i associated with layer i:

$$\dot{e}_i = \boldsymbol{\Sigma_i} \epsilon_i - e_i, \tag{33}$$

$$\dot{\epsilon}_i = \phi_i - f_i(\phi_{i+1}) - e_i. \tag{34}$$

We implemented these equations in a secondary inner loop which would run after the other model variables had converged in the primary loop. After the values of the error and inter node had converged we updated the corresponding variance according to the equation

$$\Delta\Sigma_i = \alpha(\epsilon_i e_i - 1), \tag{35}$$

also provided in [4] in (35), parameter α is some small constant that acts as the learning rate. As before, this update is only performed once per input signal. Similarly to equations (16) and (17), this rule also tended to drag variances downward until they hit a very small negative number after which the value would make a large jump upwards. The explanation here could be the same as before: when the tonic weight is correct or close to correct the values of the error and inter nodes are close to zero and so the gradient is dominated by the -1 in equation (35). Interestingly, this tending towards negative infinity seems to only affect the variance of the output layer and not both. The system is quite unpredictable when this strategy was used and it is hard to make any strong claims about the general behaviour of the network.

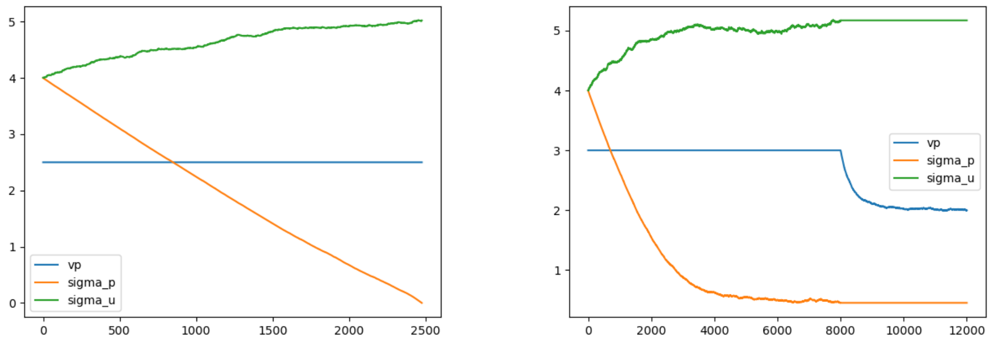

Figure 7. Divergence and convergence of variances when using inter nodes.

After experimenting with the above mentioned methods we decided to try and come up with our own rule for the parameter update in the case of variance. For these new rules we simply aim at moving the variance in the direction of our best guess. For example, the variance of ϕ can be estimated as $(\phi - s_p)^2$ and therefore we define the gradient to simply be $(\phi - s_p)^2 - \Sigma_p$. This value is then multiplied by a constant learning rate and added to the prior value of Σ_p. Similar reasoning can be used in the case of Σ_i to obtain an analogous update rule. These new rules seem to behave more predictably than the previous ones. This is likely due to the fact that the second term, which is responsible for decreasing the value of Σ, decreases as Σ itself decreases. In practice, this enables us to avoid the unwanted rapid changes to the value of these parameters when Σ gets close to zero. In our testing we observed that the value of Σ_p tends towards zero in a stable manner while the value of Σ_i approaches (the vicinity of) the variance of the input distribution. Additionally, the network seems to predict values near the tonic weight with more confidence as we tune the variances using the above method. However, this method for updating the variances does not conform to the locality constraints that are a core motivation for the predictive coding paradigm.

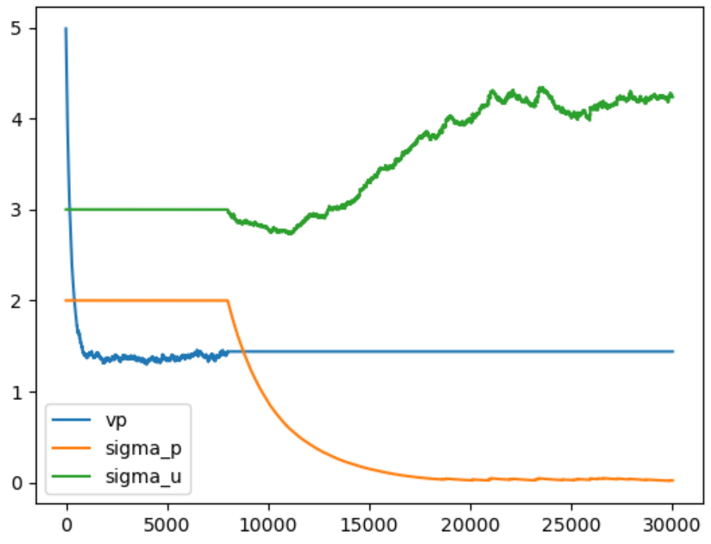

Figure 8. Variance update using the improvised learning rule. The guess for the mean is learned first and variances second. The variance of the input layer settles to the neigbourhood of the real value while the variance of the output layer approaches zero gently.

3.4. Implementation Based on an Existing Framework

After our own implementation of the predictive coding network did not produce desired results, we set our focus on producing results by modifying existing implementations. The aim was to create an autoencoder that functions correctly and produces concrete results in the form of recreated MNIST digit images. As the basis of our second implementation we used a Python source provided in [15, 19]. It is an implementation of predictive coding in Python utilizing PyTorch. It supports two modes of operation in the form of scripts. The supervised script is a classifier which takes an image as an input and produces an output that corresponds to the label it thinks is the most likely. The generative script is the reverse. It receives a label as an input and then it constructs an image that corresponds to that label.

The implementation [15, 19] is constructed using Python classes. It has appropriately named variables and is well organized but it was initially hard to grasp the structure as a whole with our limited experience with Python classes. After a few videos on Python's classes we began our analysis of the code. We started from the script files and followed through the code, jumping from file to file and trying to understand what every line did and where every variable came from. At the same time we are able to learn about mathematics and coding with this thorough approach.

There are some major differences between our first and second implementations. These include activation function performing its function in a different place, no gradient use for prediction errors, utilization of an adaptive gradient descent algorithm Adam and the simplification of the variance's role in the calculations. One of the biggest differences between the implementations is variance. While we tried to implement variance with gradient updates, Millidge leaves the updates out completely. All covariance matrices are effectively replaced with identity matrices, meaning all feature variances are set to 1 and all covariances are set to 0. These values remain static for the whole duration of a run. Updating the covariance matrices has been known to cause issues before in these types of networks.

Regarding the use of the activation function, in our implementation it is applied to the calculated value as the last step in forming the gradient. In Millidge's implementation it is used to calculate the activation strengths of all the connections which are then used to calculate the final gradient with other relevant values. Additionally, the activation function's form is significantly different from what was showcased earlier. In Millidge's implementation the connection weights are not parameters of the activation function and instead of an arbitrary polynomial, the function takes the form of a spesifically chosen function. These are the Tanh, ReLU, Sigmoid and Linear functions.

As for the calculation of prediction errors, we do not employ a gradient descent on the error values directly as Bogacz instructs. Instead, we simply define the prediction error at every step as the difference between a given nodes activity and the corresponding prediction. The predictions and activities are iterated using gradient decent and so the desired effect of closing in on a value is achieved indirectly, in the case of the errors. This approach is probably responsible for the elimination of the oscillation of model variables' values that we saw earlier in Figures 2 and 4.

For the purposes of updating the connection weights, [19] includes two different algorithms. The first is stochastic gradient descent which naively shifts a given value according to the associated gradient. The second is Adam, an adaptive optimizer, which allows for more intelligent shifting of the value in question. Adam takes into account not only the current gradient but also adjusts the learning rate dynamically based on how fast the value we are shifting has shifted in previous iterations [20]. Adam gives the network a considerable performance increase compared to SGD. That is to say, the network converges to a solution at a significantly greater pace [15, 19].

Our implementation was constructed in a way that makes the code very modular. This allows for an easy way to take an existing script and make small modifications to match our requirements. First we needed to change the network structure to support an autoencoder. This was easily accomplished by changing a few numbers inside a list to correspond to our desired form of 784 input nodes, 20 code layer nodes and 784 output nodes. We also needed to set both the input and output targets as the image we are having the network learn how to encode and decode. We found that these two changes were effectively all that was needed in order to obtain a functional autoencoder. The included accuracy measurement was not deemed applicable to our needs and so it was discarded. Millidge provides 3 options for an activation function: Linear, Rectified Linear Unit, or ReLU, and Tanh. We added the Sigmoid function as an additional option since it has been a popular option in these types of tasks. It should also be noted that the original step size of 0.01 for the inner loop was not large enough for the values to properly reach their convergent values in 50 steps. We found that a step size of 0.05 was sufficient.

3.5. Results of the Second Implementation

Here we showcase the results obtained by way of experimenting with different configurations of our implementation. We will go through the experiments one at a time with the motivation that the effects of any singular modification can be isolated from the others. The results will be presented and analyzed visually as no numeric evaluation method was devised or deemed necessary for understanding the impact of the modifications. As we present these results we will simultaneously comment on the experimentation process and share our interpretations regarding these results. All of the data used in these experiments comes from the MNIST dataset of hand written digits which is composed of a training set of 60000 images and a testing set of 10000 images. A summary of all the results along with concluding remarks on the project are provided in Section 4.2.

3.5.1. Convergence of the Model Variables

Before presenting the results of our experiments we wish to highlight a key difference in the behaviour of this network in comparison to the earlier attempt. This has to do with the way in which the model variables (predictions and prediction errors) settle onto their convergent values. As was mentioned in Section 3.4, the calculation for the updates of these variables does not include a separate gradient for the prediction errors. Instead, the prediction error is defined at every step to be the difference between the node's value and the corresponding prediction. The iteration of prediction values still utilizes a gradient that is almost identical to the one presented in [4] with a small change in the way the activation function is defined. Whereas in [4] the connection weight matrix was defined as a parameter of the activation function, Millidge [5] assumes it to be independent and so the form of the dynamics is slightly altered.

This discrepancy in the model definition probably does not adversely impact the network's performance. The fact that the update cycle for the variables contains a single gradient instead of the previous two is likely much more significant and we hypothesise that it is responsible for eliminating the oscillation of variable values which we saw earlier in Figures 2 and 4. Also, in contrast to the situation described in Figure 4, this network converges to a set of values for the variables consistently without problems. For the convergence of variable values see Figure 9.

3.5.2. Replicating Millidge's Results

In order to achieve a baseline for the performance of the network and to confirm that it was functioning as intended we first set out to recreate the results Millidge had showcased in his report [19]. In this experiment we used stochastic gradient decent, the learning rate was set to 0.01 and the experiment was run for 100 epochs. The activation function was set as the linear function, the code layer width was set to 20 nodes. The training set was limited to 10000 images instead of the full 60000 images, which sped up the training process significantly. In this configuration the network produces reconstructed images with decent fidelity, meaning that most of the digits

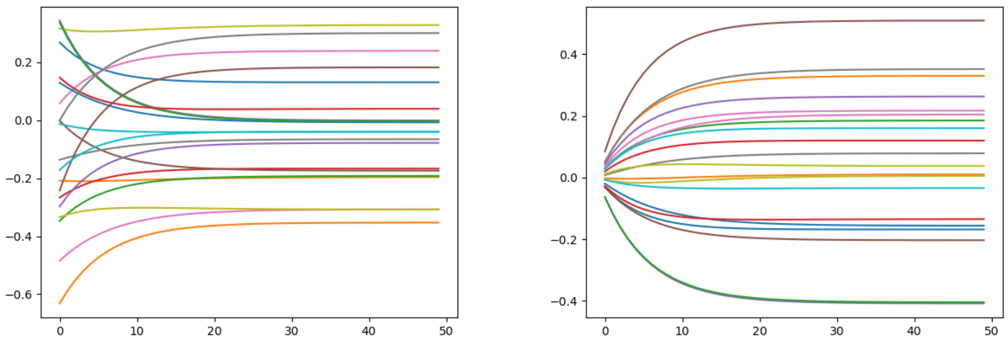

Figure 9. Convergence of code layer predictions and prediction errors respectively in the network implementation based on [15, 19].

are easily classifiable by human. While we did not see it to be necessary to construct a numerical method for the purposes of accuracy evaluation, we note here that our subjective quality of the reconstructions is clearly not perfect. This manifests most notably in the lack of contrast between the actual digit and the background and the smoothness of certain lines. In general the backgrounds remain light gray for the whole duration of the experiment as opposed to tending towards the complete black background present in the input image. Classifying certain digits such as three and five, may prove challenging due to the smoothness of the distinguishing edges. The results we obtained are largely consistent with the ones presented in [21]. This indicates that the network is performing correctly, providing justification for proceeding with further experiments.

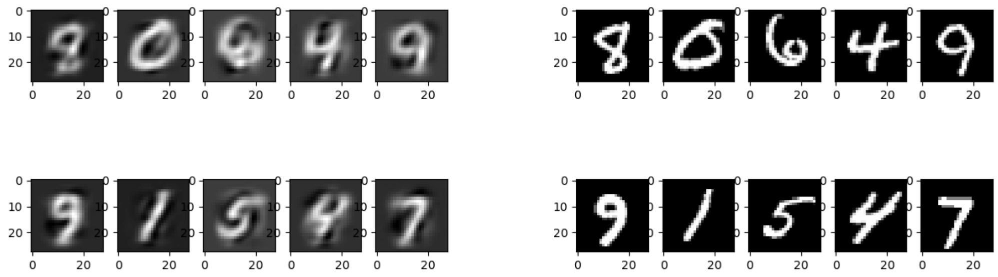

Figure 10. Left: A snapshot of the recreated digits using the network configuration provided in [15, 19]. Right: True images from the MNIST dataset.

3.5.3. *Activation Function Experiments*

Our implementation used a linear activation function. As previously mentioned the code provided us with two additional activation functions, Rectified Linear Unit, or ReLU, and Tanh, with sigmoid being our addition. Our goal is to test the different approaches to the activation function and see if they produce noticeably different, or desirably better images when compared to one another.

The different activation functions display distinct looking effects on the images they produce, though they are quite subtle. The results are compared in Figure 11. Effects vary from different hues of gray on the background to more smooth or ragged edges on the digits. None of the activation function demonstrated superior performance when compared with the others.

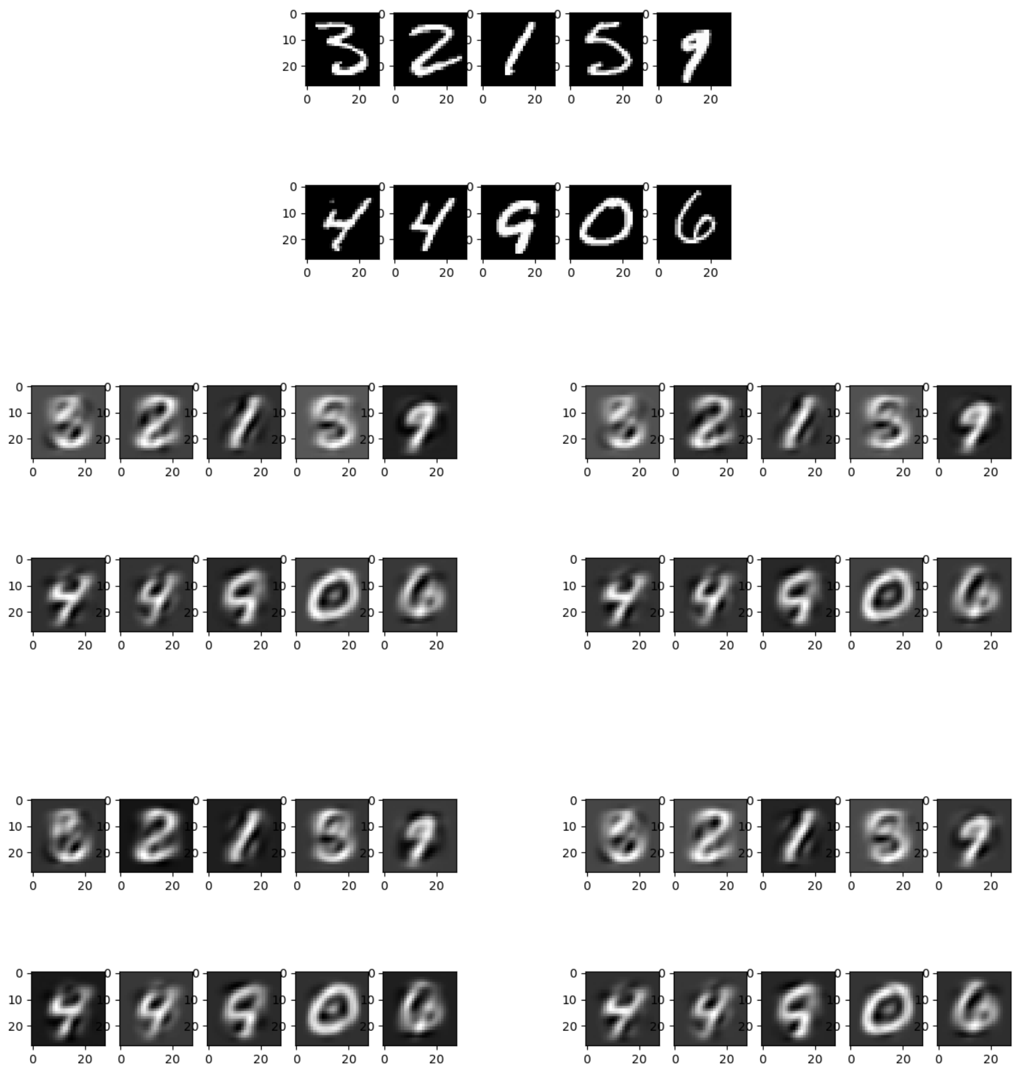

Figure 11. Top left: Linear; Top right: ReLU; Bottom left: Sigmoid; Bottom right: Tanh

The digits that appear in these results differ from the ones seen in the previous experiment because this time we did not limit the training set but in stead allowed for all images to be used in training. This causes the set of printed images to shift somewhat. Since each epoch had a more comprehensive set of data to learn from, we simultaneously decreased the number of epochs to 20 in order to keep the experiment duration manageable.

3.5.4. Impact of Code Layer Width

Next we moved on to study the effects of architectural changes to the network. The first of these experiments had to do with the width of the layers between the input and output layers. To isolate this parameter's effect we conducted tests in the three layer network where the code layer consisted of a single layer. The width of this layer is simply defined as the number of nodes in that layer which in the earlier experiments was kept at 20.

The results of this experiment were unsurprising and the explanation is similarly intuitive. As the width of the code layer increases so does the image fidelity over all the images. This is a straightforward consequence of the code layer acting as the most important bottleneck of the network as it defines the amount of compression that an input image is subjected to. The code layer's width can in some sense be thought of as the amount of features the network is capable of extracting from the entire collection of training data. Each of the code layer nodes can encode the strength of a certain configuration of input layer nodes. This can be understood more clearly if we consider a code layer whose width coincides with the input layer's width. In this scenario each pixel of the input image can in principle be represented by a single node in the code layer and thus no compression is required. This scenario is referred to as an identity mapping by Millidge [5].

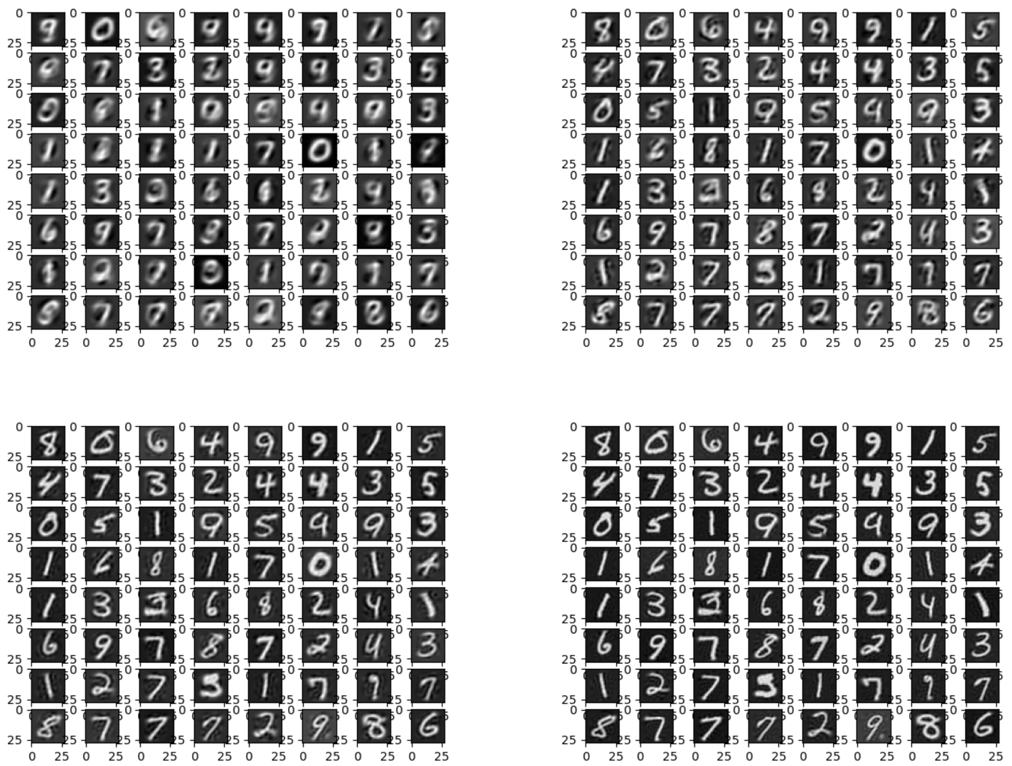
Figure 12. From top left 8, 32, 64, 128 nodes wide respectively. As the width of the code layer increases, the edges become sharper and the images representing the same digit become less uniform.

3.5.5. Deep Network Experiments

Continuing with the architectural investigation, we proceeded to make the network deeper by inserting additional layers into both the encoder and decoder parts. We kept the network symmetrical and added eight layers, making the code layer nine layers wide as a whole. This change increased the total number of weights in the network from about 30000 to just over a million. The bottleneck layer was increased from 20 nodes to 32. Resulting images can be observed in Figure 13. Results do not appear to improve over the original three-layer network.

It is not entirely clear why the performance of this deeper network is not visibly better when compared to more shallow ones especially with the extra width in the bottleneck. One possible explanation is that the extra complexity in the network simply does not have a corresponding element in the input space. That is to say, there is nothing meaningful to map onto the extra nodes of the network. This effect might be amplified by the fact that we are using fully connected layers which possibly prevents us from picking up on the potential features present in the compressed intermediate representations of the additional layers.

Another reason for the performance, or lack thereof, could be that the sizes of the intermediate layers have to be chosen more systematically, paying attention to the data that is presented to the network. When conducting this experiment we chose a relatively steadily narrowing architecture without any solid motivation for the structure of the network. These results were somewhat disappointing, but given that the duration of each experiment is quite long due to the large number of calculations required, we decided it was not prudent to proceed with further experiments on deep architectures in the context of this project.

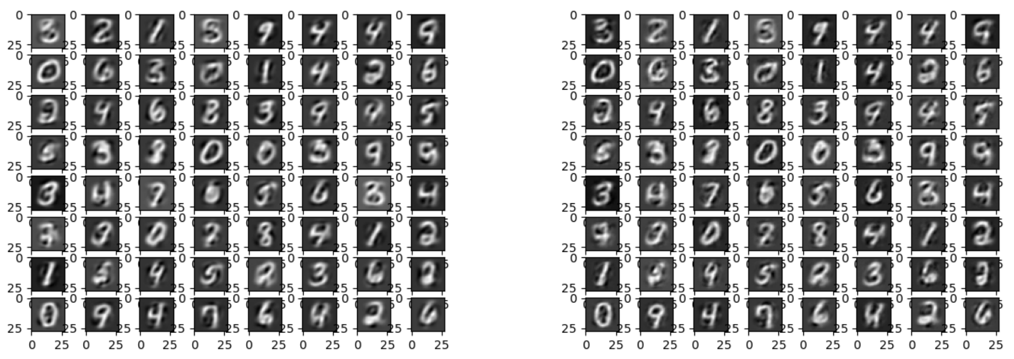

Figure 13. Left: Deep neural network with nine hidden layers. Right: The original three-layer network.

4. CONCLUSIONS

In this section summarize the key elements of this thesis. We also evaluate the two implementations produced in this project and offer our thoughts on the implementation process. We conclude with future work possibilities regarding what we have and have not achieved during the thesis work.

4.1. Summary

In this thesis we have introduced the topic of predictive coding alongside the most important related concepts. We briefly presented the history and background of the theory and motivated the model with both its apparent compatibility with biological circuits and its efficacy in different tasks involving artificial intelligence solutions. The model's relevance as a potential alternative to traditional back-propagation networks was showcased through a survey of related literature in which we highlighted some of the more interesting applications of networks of this kind. We concluded the introduction with a description of our plan to implement an autoencoder, using the predictive coding model as the basis for the calculations required for the network to achieve learning. The task assigned to this autoencoder is the reconstruction of handwritten MNIST digits from their compressed representations.

Before presenting our implementations, we provided a walkthrough of the predictive coding model and the free-energy framework using Bogacz's tutorial as inspiration. In this chapter we go through in detail all the key principles and ideas guiding this paradigm. We also showcase how the mathematical side of the predictive coding model links to statistical free energy. In addition to verbally explaining the core of the theory we present the relevant mathematics needed to implement the network. Most notably this includes the source and form of the dynamics which govern how the different variables and parameters are to be tuned in order for the network to achieve the goal set for it.

After introducing the predictive coding model, we moved on to present the implementation phase of the project. Here we described the implementation process and results of two different autoencoders, one of which was our attempt at creating an original network and the other was built on top of an existing predictive coding library written in python by Millidge. The implementation of the original network began by us familiarizing ourselves with the framework presented by Bogacz [4]. The goal then was to extrapolate the mathematics from the provided exercises into a proper autoencoder network capable of reconstructing MNIST digits. However it soon turned out to be a much more complicated procedure than we anticiåated. While we were ultimately unsuccessful in achieving expected functionality with the original implementation it served as a great lesson into the intricacies of building a neural network from the ground up. We documented the problems we encountered while implementing this network and gained some insight into what could be the source of the difficulties. In particular we encountered problems regarding the update rule of feature variance which did not function as intended even in an simplified network setup. As a result of these problems, no research results could be obtained regarding the original implementation excluding the problem analysis we have conducted. While we did not find any conclusive answers relating to the dysfunction of the first network, we have gotten several potential modification ideas (see Section (4.3)) from studying the Millidge implementation [15] and from constructing our second implementation.

Since our initial implementation did not turn out as we intended we pivoted into studying an existing implementation [15] with the motivation of being able to convert it into a functioning autoencoder. We proceeded to document the main differences of the Millidge implementation in comparison to the theory as it was presented in Bogacz's tutorial. It turned out to be a straightforward process to convert the existing library into an autoencoder with which we were able to obtain the results we had been anticipating. The Millidge implementation functioned as a library of all necessary pieces needed to construct the network we were looking to implement. We modified the layer structure of the network, making it symmetric and changed the target of the network to match the input signal (as the autoencoder requires). The results were then showcased and different configurations of the network were experimented with. The overall conclusion we draw from these results is that the model is capable of enabling learning in this task with good efficacy despite the suboptimal architecture which uses fully connected layers as opposed to convolutional ones. The experiments we conducted had to do with the effects of activation function choice, code layer width and network depth. With regards to these tests we conclude that: there is no clear indication as to which activation function is optimal for image quality, the width of the code layer is likely the most important factor when considering image quality, and that increasing the depth, at least somewhat arbitrarily, will not improve image quality even at the expense of greatly increased time complexity. We also replicated the configuration Millidge showcases in his report [19] for this network and conclude that the reuslts are very similar.

4.2. Project Evaluation

In the process of working on this project we have gained an appreciation for the practical side of predictive coding and have been positively impressed by its applicability to the chosen learning task. While the paradigm has, in its practical form at least, proven to not abide by the strict biological restrictions as closely as we had initially thought, it has remained a distinctive option to traditional neural networks. It has been an interesting process in its own way to observe how the implementations forego certain biologically based restrictions in order to optimize for performance in the task at hand.

Our biggest and only regret in the execution of this project is the miscalculation relating to the original implementation attempt. Had there been more thorough preparation or perhaps less insistence on our part we could have moved on to other aspects of the project more quickly instead of committing as much time on it as we did. While the documentation of the many problems we faced was interesting in its own way, it would have definitely been more so to conduct additional experiments with a functioning network.

What has made the implementation of the complex network difficult is the large amount of variables and parameters involved, which in turn makes analysing the network's function almost impossible. There simply is not a way to keep track of all the values involved and all the calculations that happen in parallel. After experimenting with the smaller test networks it would seem to be the case that the difficulties regarding diverging values has to do with the size of the matrices and vectors involved. The calculations might include some unwanted compounding effects which we have not been able to identify yet. For instance, when we multiply together the elements in the last term of equation (30) each element of the resulting vector consists of a sum of values where each individual element of the sum is the product of an element in Σ_i and $\vec{\epsilon_i}$. In the case of the input layer of our 784 dimensional network this sum will have 784 terms. It could be that these sums, which play a major role in forming the gradient for some node, are significantly larger in magnitude than the values they are meant to change. This might cause the system to quickly diverge. As mentioned earlier, this type of analysis is difficult to conduct in practise due to the reasons given above.

In addition to these computational complexities we have become aware of other problematic aspects of the original implementation. These findings are largely the result of working with the functional predictive coding library [15] and becoming more familiar with neural networks in general. For instance, we did not properly attach the output layer to the target (input signal) during training which causes obvious problems for the network's function. Additionally, we were uninformed on the specificity of the activation functions in the context of neural networks. The function we used for our original implementation was found to be inadequate for the application at hand. Lastly, it seems to be the case that the variance update can be left out of the netowork without sacrificing core functionality. This was the most persistent problem we struggled with during the entire project.

The pivot to the network based on the Millidge implementation was absolutely the correct choice as it allowed us to reach the goals we had originally set for ourselves and the results we were able to obtain were satisfying. The second network performs well in the task of recreating digits from the MNIST dataset and while no objective evaluation of the final images was constructed we are impressed with the network's performance. Additionally, being able to conduct experiments with the second network provided us with understanding of how a network constructed in this manner responds to the kinds of modifications we made. For instance, we showcase that the activation function does not seem to meaningfully impact performance regardless of the configuration otherwise when using fully connected layers. Also we demonstrate how the width of the code layer has significant impact on the quality of the output both in terms of image fidelity and proper classification. Overall we were pleased with the results we got and believe we were able to reach the goals we set even if the road there was not always as straightforward as we would have liked.

4.3. Future Work

While the preliminary results from our experiment are informative, there is surely much more potential for further study regarding these specific networks. Perhaps the most interesting addition one could make to the second implementation is the addition of convolutional layers to replace the currently fully connected ones. We hypothesize that this would yield greater performance in terms of image quality as this architecture has proven to be the superior one in traditional neural networks utilizing back-propagation. The Millidge implementation already contains the building blocks for this addition but to fully implement it would most certainly take time and careful study as the architecture does seem to become significantly more complicated.

A less ambitious but still interesting extension could be the enable experiments with alternative datasets such as CIFAR-10 or Fashion-MNIST. While we do not expect the performance to differ drastically between datasets it would still help to prove that a network built on these principles has the robustness that alternative approaches possess. Just as for the convolutional layers, the building blocks for extending the networks functionality to these alternate datasets already exist within the Millidge library.

Finally, we believe it is not inconceivable that the lessons learned through the Millidge implementation could be used to mend the original implementation in a meaningful way. We now have a much better understanding of how the mathematics needs to be configured in order for the network to function smoothly. We are aware of multiple aspects of the original network, each of which by themselves probably prevent proper function. These include the improper usage of activation functions, the improper setup of input and target values during training and the inclusion of variance which should be viewed as an optional addition which often causes malfunction. In addition to these we also know that it is possible to implement alternative approaches in the update rules for the variables as we saw in the case of the prediction error in Millidge's implementation. While pursuing the original network could end up being an exceptionally rewarding endeavour it might be more prudent to focus primarily on extending the functionality of the second implementation as it has already been shown to work. Attaining general results by way of said extensions is in all likelihood more important the pursuit of unproven novelties.

4.4. Contributions

Roni Latva:

- Main author on Chapters 2 and 4. Partial participation in writing rest of the chapters.

- Reading focused on the core texts relating to the predictive coding framework [1, 4, 21].

- Resoponsible for coding extensive experiments relating to the first implementation and documenting results of said experiments. Partial participation in building the second implementation and main responsibility in documenting its experiment results.

Santtu Orava:

- Main author on Chapter 3. Partial participant in Chapter 4.

- Base work on finding previous code for predictive coding implementations [15, 22] and building the second implementation based on them.

- Main responsibility on finding related works [3, 9, 10, 11] on relevant subjects.

Casimir Saastamoinen:

- Main author on Chapter 1. Partial participation in Chapters 3 and 4.

- Reading focused on related and similar works [1, 8, 11, 14, 23].

- Some experimentation with both implementations and testing with the Python code. A partial participation in building the implementation based on [15].

5. REFERENCES

[1] R. P. N. Rao and D. H. Ballard, "Predictive coding in the visual cortex: A functional interpretation of some extra-classical receptive-field effects," *Nature neuroscience*, vol. 2, no. 1, pp. 79–87, 1999.

[2] K. Friston, "Friston, k.j.: The free-energy principle: a unified brain theory? nat. rev. neurosci. 11, 127-138," *Nature reviews. Neuroscience*, vol. 11, pp. 127–38, 02 2010.

[3] H. Wen, K. Han, J. Shi, Y. Zhang, E. Culurciello, and Z. Liu, "Deep predictive coding network for object recognition," in *35th International Conference on Machine Learning, ICML 2018*, vol. 12, pp. 8369–8378, 2018.

[4] R. Bogacz, "A tutorial on the free-energy framework for modelling perception and learning," *Journal of mathematical psychology*, vol. 76, pp. 198–211, 2017.

[5] B. Millidge, "Applications of the free energy principle to machine learning and neuroscience," 2021.

[6] H. v. Helmholtz, "Concerning the perceptions in general.," 1866.

[7] K. Friston, "A theory of cortical responses," *Philosophical Transactions of the Royal Society B: Biological Sciences*, vol. 360, no. 1456, pp. 815–836, 2005.

[8] K. Friston, J. Kilner, and L. Harrison, "A free energy principle for the brain," *Journal of Physiology-Paris*, vol. 100, no. 1, pp. 70–87, 2006. Theoretical and Computational Neuroscience: Understanding Brain Functions.

[9] W. Lotter, G. Kreiman, and D. Cox, "Deep predictive coding networks for video prediction and unsupervised learning," 2016.

[10] T. Nguyen, R. Shu, T. Pham, H. Bui, and S. Ermon, "Temporal predictive coding for model-based planning in latent space," 2021.

[11] N. Alonso and E. Neftci, "Tightening the biological constraints on gradient-based predictive coding," in *ACM International Conference Proceeding Series*, 2021.

[12] J. Kolen and J. Pollack, "Backpropagation without weight transport," in *Proceedings of 1994 IEEE International Conference on Neural Networks (ICNN'94)*, vol. 3, pp. 1375–1380 vol.3, 1994.

[13] M. Akrout, C. Wilson, P. C. Humphreys, T. Lillicrap, and D. Tweed, "Deep learning without weight transport," 2019.

[14] "Intro to autoencoders." URL: `https://www.tensorflow.org/tutorials/generative/autoencoder`. Accessed 22.2.2022.

[15] B. Millidge and A. Tschantz, "Infer actively pypc." URL: `https://github.com/infer-actively/pypc#readme`, 2021. Accessed 22.2.2022.

[16] D. M. Blei, A. Kucukelbir, and J. D. McAuliffe, “Variational inference: A review for statisticians,” *Journal of the American Statistical Association*, vol. 112, p. 859–877, Apr 2017.

[17] L. Deng, “The mnist database of handwritten digit images for machine learning research,” *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[18] “Hebbian theory.” URL: `https://en.wikipedia.org/wiki/Hebbian_theory`. Accessed 19.6.2022.

[19] B. Millidge, “Implementing predictive processing and active inference: Preliminary steps and results,” 2019.

[20] J. Brownlee, “Gentle introduction to the adam optimization algorithm for deep learning.” URL: `https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/`. Accessed 19.6.2022.

[21] B. Millidge, A. Seth, and C. Buckley, “Predictive coding: a theoretical and experimental review,” 07 2021.

[22] S. Lippl, “Predicode: Hierarchical predictive coding in python.” URL: `https://sflippl.github.io/predicode/html/index.html`, 2019. Accessed 22.2.2022.

[23] K. J. Friston and K. E. Stephan, “Free-energy and the brain,” *Synthese*, vol. 159, no. 3, pp. 417–458, 2007.