MSc thesis

Master's Programme in Computer Science

Multi-task learning in Computer Vision

Konsta Kutvonen

May 31, 2020

Faculty of Science

University of Helsinki

Supervisor(s)

Associate Professor Laura Ruotsalainen

Examiner(s)

Associate Professor Laura Ruotsalainen, Professor Jyrki Kivinen

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Master's Programme in Computer Science

Tekijä — Författare — Author

Konsta Kutvonen

Työn nimi — Arbetets titel — Title

Multi-task learning in Computer Vision

Ohjaajat — Handledare — Supervisors

Associate Professor Laura Ruotsalainen

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
MSc thesis	May 31, 2020	67 pages

Tiivistelmä — Referat — Abstract

With modern computer vision algorithms, it is possible to solve many different kinds of problems, such as object detection, image classification, and image segmentation. In some cases, like in the case of a camera-based self-driving car, the task can't yet be adequately solved as a direct mapping from image to action with a single model. In such situations, we need more complex systems that can solve multiple computer vision tasks to understand the environment and act based on it for acceptable results. Training each task on their own can be expensive in terms of storage required for all weights and especially for the inference time as the output of several large models is needed. Fortunately, many state-of-the-art solutions to these problems use Convolutional Neural Networks and often feature some ImageNet backbone in their architecture. With multi-task learning, we can combine some of the tasks into a single model, sharing the convolutional weights in the network. Sharing the weights allows for training smaller models that produce outputs faster and require less computational resources, which is essential, especially when the models are run on embedded devices with constrained computation capability and no ability to rely on the cloud.

In this thesis, we will present some state-of-the-art models to solve image classification and object detection problems. We will define multi-task learning, how we can train multi-task models, and take a look at various multi-task models and how they exhibit the benefits of multi-task learning. Finally, to evaluate how training multi-task models changes the basic training paradigm and to find what issues arise, we will train multiple multi-task models. The models will mainly focus on image classification and object detection using various data sets. They will combine multiple tasks into a single model, and we will observe the impact of training the tasks in a multi-task setting.

ACM Computing Classification System (CCS)
Computing methodologies → Artificial intelligence → Computer vision
Computing methodologies → Machine learning → Learning paradigms → Multi-task learning

Avainsanat — Nyckelord — Keywords

computer vision, deep learning, convolutional neural networks, multi-task learning

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Algorithms study track

Contents

1 Introduction

Since 2012 computer vision research has been increasingly dominated by approaches using deep Convolutional Neural Networks (CNNs). These CNN based techniques have allowed for achieving significantly better results in computational image and video understanding compared to previous state-of-the-art approaches. Nowadays, there are many interesting avenues of computer vision research, such as image segmentation, object detection, object recognition, image captioning, pose detection, and many more. Since the task of computational image understanding is quite complex, the CNNs used to solve these problems often require dozens, if not hundreds of millions of parameters, in order to achieve sufficiently reliable accuracies. As the number of parameters is high, the networks require large computational resources to find optimal values for all of the variables in all of the hundreds of layers of matrices. Graphics Processing Units (GPUs) or Tensor Processing Units (TPUs) are the tools that people use to train the parameters in the networks since the training operations are highly parallelizable, allowing for the training of networks with hundreds or even thousands of layers. The problem often is that when training, one wants to increase the batch size, but at the same time, the computations must fit in the memory of the processing unit, which can be difficult to increase. Also, as the number of parameters in the model increases, the memory required for training the networks, and the time to produce outputs increases. To combat these problems, we often have to get better or more hardware, which can be expensive or compromise in the size of the model, which may lead to unreliable performance.

Embedded devices can use these CNN based algorithms to analyze their surroundings in an environment where they can't rely on external predictions from the cloud. One of the most significant issues is that achieving scene understanding requires multiple classifiers to solve various tasks. For example, a camera-based self-driving car has numerous things it is interested in resolving using the video output of its cameras. The tasks could be, for example, finding and classifying other vehicles, traffic signs, road markings, predicting paths of vehicles, pedestrians, and other actors, understanding where it can move, and finally deciding the steering angle and acceleration for the car with the information. All these tasks have to be solved using the limited computing capability onboard the actor, and the inference time for them can't be too long in order to react to everything within a reasonable time frame. Here we have many tasks, and if every task requires its independent

network to produce the outputs, and we cut down on the network sizes, the classifiers might be powerful enough to produce safe outputs. On the other hand, if we don't reduce the network sizes, our inference time might be too long for a real-time system. One way to possibly get a good compromise between inference time and model complexity is to use multi-task learning and weight sharing within the model to get a single model with multiple outputs and similar or better performance to individual classifiers.

Multi-task learning proposes some solutions to these problems but makes the training process more difficult. Still, in some cases, it is something that has to be done to get a good enough inference time and accuracy on the problem at hand. Many modern computer vision models already feature an ImageNet backbone as a part of the classifier, so this is a good part of the network to consider for sharing. However, finding which tasks and how much can be shared is a difficult problem to solve.

1.1 Scope of study

In this thesis, we will present state-of-the-art solutions to fundamental computer vision problems, like image classification and object detection. We will present the architectures of these models and show how the ImageNet backbones are an essential part of solving these problems. Using the presented solutions, we will make use of various data sets to train multi-task models and evaluate the effects of training multi-task models.

1.2 Structure of thesis

Chapter 2 will present some basic terminology for describing convolutional neural networks that will be used throughout the thesis. We won't cover the very basics of neural networks or the exact inner workings of convolutional neural networks, and refer to (Goodfellow et al., 2016) for studying those details.

Chapter 3 covers the modern image classification models, describing their architecture and how to train them. We will also describe why the ImageNet is such a valuable data set to so many problems. We will demonstrate the significance of the ImageNet models by defining transfer learning and describing how it is instrumental in solving small-scale problems with the ImageNet models as backbones.

Chapter 4 covers the basics of multi-task learning in terms of how it can be applied to

computer vision problems. Multi-task learning is also a useful technique in other domains of machine learning, but we focus only on computer vision. We will define multi-task learning and see how it relates to transfer learning. Also, the basic differences of multi-task training to a single-task setting are described, and some examples of multi-task models are shown. Finally, the theoretical benefits and issues that may arise are talked about and demonstrated using some examples.

Chapter 5 describes what object detection is and how to evaluate object detection models. We will also briefly discuss the basic parts of an object detector and the difference between single and two-stage object detectors. Finally, we will cover the EfficientDet architecture in detail and explain how to train an object detector by using it.

In chapter 6, we will cover the results of our experiments on training multi-task models. The first experiment is about training image classifiers on plant imagery. The second experiment is about training multi-label classifiers. The third experiment covers training object detectors with multiple datasets. The fourth experiment uses a single EfficientDet to do image segmentation and object detection. Finally, we will summarize our experiences in training multi-task models and explain what problems we encountered.

2 Neural networks

The topics presented in this thesis are based on neural networks, especially convolutional neural networks, which are a distinct neural network variation for image processing using filters. Here we provide a very brief high-level overview of some of the concepts that are important for the later chapters, but a more detailed overview of deep learning and neural networks can be found, for example, in (Goodfellow et al., 2016).

2.1 Convolutional Neural Networks

The most significant component of Convolutional Neural Networks (CNNs) is the convolutional layers that are their primary building block. Using convolutional filters allows neural networks to extract useful information from images to accomplish various tasks. The convolutional blocks are based on the mathematical convolution operation of capturing the signal of pre-determined window size. A CNN classifier often consists of multiple convolutional layers with other layers in between. Successive convolutional layers build on top of one another, and the deeper layers get high activations on more specific features. In many final models, the earlier layers are thus somewhat similar and re-usable as they often detect relatively primitive shapes, like diagonal or horizontal lines. In this thesis, we often talk about the parameters or the weights of a network. The number of parameters in a network tells how complex it is and how expensive it is to store. The floating-point numbers in the various layers are these weights, which are generally 32-bit numbers, but their precision can vary.

2.1.1 Parameters

To get the total number of parameters required for a single convolutional layer, we need to determine the input size, convolution window size, and the number of filters. For example, if the input is a 256x256 RGB image, the input size is 256x256x3. Now, if we apply 16 filters with a window size of 5, each of the filters requires 5x5x3 parameters, bringing the total to 1200 parameters. A fully connected layer with 1000 hidden units would require a matrix with around 200 million parameters, so it would not really be possible even to

apply multiple of these layers while maintaining the image size. The convolution allows for keeping the dimensions between successive layers and only modifying the number of channels between the layers. Often convolutional models reduce the first two dimensions, and the number of channels increases in the deeper layers. For this, the convolutions can use a stride parameter to skip some positions for the convolution. The pooling layer is another way to reduce the size. Generally used pooling layers are average and maximum pool layers. They are applied similarly to a convolution but capture the average or the maximum of a window. Neither of these approaches requires any extra learnable weights.

2.1.2 Embedding

Generally, the role of the convolutional layers in a CNN model is to act as a feature extractor. As can be seen in the number of parameters of the fully connected layer on a full-sized image, the image needs to have a smaller dense representation in order to connect a linear layer to it. This representation can be generated with the convolutional layers. The final representation that can be pushed into a linear classifier for some final inference is called an image embedding. Embeddings are representations of things in relatively small continuous numbered vectors, here we consider embeddings of images, but embeddings are used, for example, for words and sentences as well. Besides just doing classification, the embeddings can represent things, for example, faces or objects. When a general embedding is generated it can be used to for example to evaluate if someone is the same person as the embedding or to find similar images.

2.1.3 Backbone

The feature extraction part of a convolutional classifier is called the backbone. The backbone is annotated in Figure 2.1. Finding better features is often correlated with more parameters and a larger model, but various architectural decisions can also help. The largest convolutional models have thousands of layers. Best models often use as large backbones as is possible within the constraints of the specific deployment. The backbone is thus responsible for creating the image embedding that is used to produce an output. Some tasks require feature maps taken from the middle of the backbone instead of only relying on the final layer.

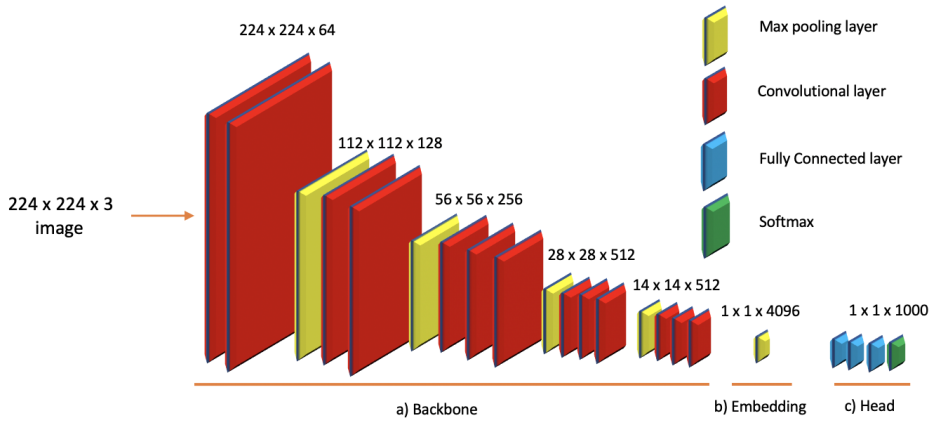


Figure 2.1: Image of an example of a convolutional neural network architecture with backbone (a), image classifier (b) and task head (c) labeled. The architecture in this image is the VGGNet (Simonyan and Zisserman, 2015).

2.1.4 Head

The final layers of a classifier are called the head. The head is responsible for producing the output, and its exact architecture varies from task to task. It could be just a single linear layer connecting the image embedding to final class probability distribution or an image segmentation model classifying each pixel in the image. The head can thus contain whatever layers are needed to get the output, combining batch normalization, dropout, convolutional, recurrent, and linear layers depending on what is needed to solve the task. An example visualization of how the different parts of a convolutional model are connected is shown in Figure 2.1. The architecture would vary on a task-by-task basis, and the figure shows a basic image classifier architecture.

3 Image classification

Image classification is one of the essential modern computer vision problems, where the goal is to create a model that can classify an input image into one of a set of pre-defined classes. Before the popularization of applying large Convolutional Neural Networks for this task, the most successful way of solving the problem was to use some algorithm for finding feature descriptors in a set of images to construct a Visual Bag of Words (Faheema and Rakshit, 2010). A linear classifier, like a Support Vector Machine, would then do the classification using the Bag of Words representation of an image. These days nearly all approaches are based on using deep CNNs, and working CNNs were deployed already in 1998 on character recognition in the form of LeNet (LeCun et al., 1998). In this chapter, we will present what ImageNet (Deng et al., 2009) is and how the models trained with it are important when applying transfer learning, and we will show an example of how it can be done.

3.1 ImageNet

ImageNet (Deng et al., 2009) has been one of the most significant data sets for image classification. It is has enabled the development and served as a benchmark for several significant improvements to modern computer vision techniques. The ImageNet challenge (*ImageNet Large Scale Visual Recognition Competition (ILSVRC)* 2020) is a competition consisting of a selected collection of 1000 classes and a total of 1.2 million training images and 150 thousand test images from the ImageNet data set. Winning models from the ImageNet competition have provided some of the essential architectures modern computer vision algorithms use. In 2012 the winning model, AlexNet (Krizhevsky et al., 2017), showed that it was possible to train deep CNNs efficiently using GPUs. Since 2012 all top-performing models showed some new improvements on how to create the most performant network architecture, for example, VGGNet (Simonyan and Zisserman, 2015) from the year 2014 and ResNet (He, X. Zhang, et al., 2016) from 2015, both of which have been popular models to use for Transfer Learning since.

Human accuracy in image classification on the ImageNet challenge is about 5.1% (Russakovsky et al., 2015), ResNet achieves a top-5 error rate of 3.57% (He, X. Zhang, et al.,

2016) and newer architectures even lower, but this still does not mean that image classification is a solved problem. The human performance experiment found that many of the human errors are caused by not having expert information in, for example, identifying animal species or not even being aware of the existence of a class (Russakovsky et al., 2015). ObjectNet (Barbu et al., 2019) is a dataset designed to test image classifiers with a focus on their generalizability. It contains many classes that also exist in the ImageNet dataset. However, the objects in the images are in unexpected locations or have an unexpected pose, causing the high accuracy image classifiers trained on ImageNet to experience a 40-45% accuracy drop when evaluating them on the ObjectNet images of classes shared between ImageNet and ObjectNet. This kind of adjustment is relatively easy for a human, and it shows that while the classifiers are good, they are by no means perfect.

3.2 Transfer learning

Transfer learning is a powerful technique to obtain results quickly when using deep CNNs. Here we will only take a look at transfer learning within the domain of deep CNNs, but it is a technique that has been successfully applied to many other domains of machine learning as well.

The idea behind transfer learning is to train on a task that would produce useful features in solving some other task. Then the network weights in the model for the actual task are initialized to those of the model we are transferring from, so the training of the original model is a pre-step to the real task. Since training the models on ImageNet scale datasets is not generally feasible due to their large number of parameters and long training time, one of the pre-trained models is picked and then fine-tuned. Fine-tuning a classifier means that some existing model is used as a weight initializer, and the network is then trained on the data of another task, updating the weights of the original classifier to better align with the new task. The transfer learning approach differs significantly from the traditional learning model, where each task requires a separate model that learns from the given data using random weights. Since image inputs are often very high dimensional, the traditional approach may not work in many cases. The pre-training allows for focusing on data that provides answers to the actual task and not on learning low-level features, which the ImageNet classifiers would already have learned.

To give a formal definition of transfer learning, we will follow the definitions provided in (Pan and Yang, 2010). Let $\mathcal{D}$ be a domain, which consists of a feature space $\mathcal{X}$ and a

marginal probability distribution $\mathrm{P}(\mathcal{X})$. For a given domain, a task $\mathcal{T}$

$$\mathcal{T} = \{\mathcal{Y}, f(X)\}$$

consists of a label space $\mathcal{Y}$ for the inputs and of a predictive function $f(x)$ which produces predictions for all pairs $\{x_i, y_i\}$ where $x_i \in \mathcal{X}$ and $y_i \in \mathcal{Y}$. Given a source domain $\mathcal{D}_S$, a source task $\mathcal{T}_S$, a target domain $\mathcal{D}_T$ and a target task $\mathcal{T}_T$, where target and source are disjoint, transfer learning tries to improve the performance of $f_T(x)$ using $\mathcal{D}_S$ and $\mathcal{T}_S$.

Unlike the ImageNet challenge, most real-world tasks do not have such an abundance of data for all possible classes. Still, to achieve the highest accuracies, they require models that are equal in terms of complexity to those that have top accuracies problems on the scale of the ImageNet classification. For this reason, many CNN classifiers, irrespective of the problem, feature one of the ImageNet classifiers in the model architecture. Even though some datasets may contain a large number of images per class, using a pre-trained classifier as a basis often produces a better final classifier by applying fine-tuning (Kornblith et al., 2019).

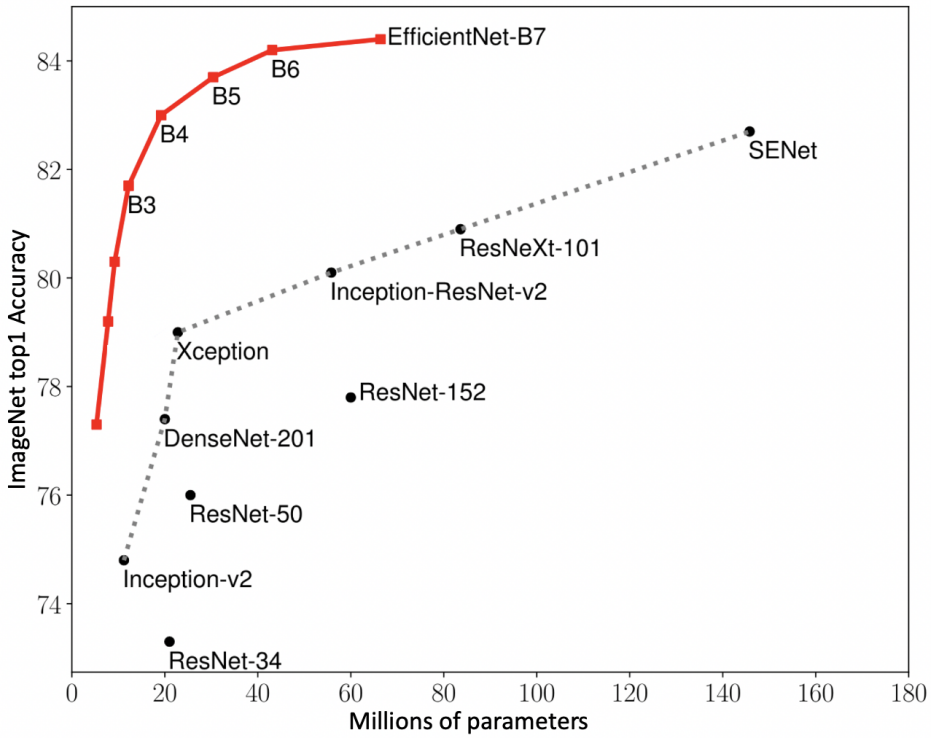


Figure 3.1: Number of parameters in popular ImageNet classifiers. Figure adapted from (Tan and Le, 2019).

Picking which classifier to use as a base is often problem-dependent. As it is not possible to declare one network structure to be the best at all tasks, picking the best model to

start with usually requires the user to compare different architectures and weighing the requirements for the problem at hand. Often though, the larger models will perform better, and there exists a correlation between performing well on ImageNet and being a good transfer learning model (Kornblith et al., 2019). Though as can be seen in figure 3.1, better performance often comes at the cost of many more parameters, requiring more memory to train the model. Though, as the results in (Lee et al., 2020) show, just the number of parameters is not the only thing to compare as the throughput of a Resnet50 turns out to be about three times as large as the throughput of an EfficientNet-B4 (Tan and Le, 2019) even though they have a similar amount of parameters. So in the case of time-constrained inference environments, also the time required for an inference has to be taken into account when picking models. Often though, the number of parameters is a relatively good proxy for the inference time. The throughput is likely due to the differences in the internal implementations of the EfficientNet and normal CNNs.

If there is enough data, it turns out that using a pre-trained network does not provide any benefits in terms of the converged model accuracy, but it is not detrimental to performance either (He, Girshick, et al., 2019). When training sufficiently long on a sufficient amount of data, the pre-trained and randomly initialized networks converged to similar accuracies but required significantly different amounts of training resources. Still, this does not mean that pre-training is useless by any means as the saved resources and getting models to converge faster are essential factors for progress. And of course, there is not enough data in many cases, and training from scratch will not provide satisfying results due to the large number of parameters that would need to be optimized. In these cases, it would then be a decision to apply transfer learning or to get a model that is not good enough to produce reliable results.

3.3 ResNet

ResNet is one of the first effective and very deep Convolutional Neural Network architecture that was presented in 2015 and won the ImageNet challenge. Prior to the publication of ResNet, the most powerful networks were relatively shallow, like VGGNet (Simonyan and Zisserman, 2015), which has only 19 layers. One of the big issues relating to training deep networks is vanishing gradients, where gradients disappear when they are backpropagated through many layers (Zagoruyko and Komodakis, 2016). ResNets utilize residual connections around bottleneck building blocks, which allow for the networks to contain

many more layers than those without them, the largest network presented in the original ResNet paper was 152 layers, totaling for around 8x increase in the number of layers when compared to earlier networks (He, X. Zhang, et al., 2016).

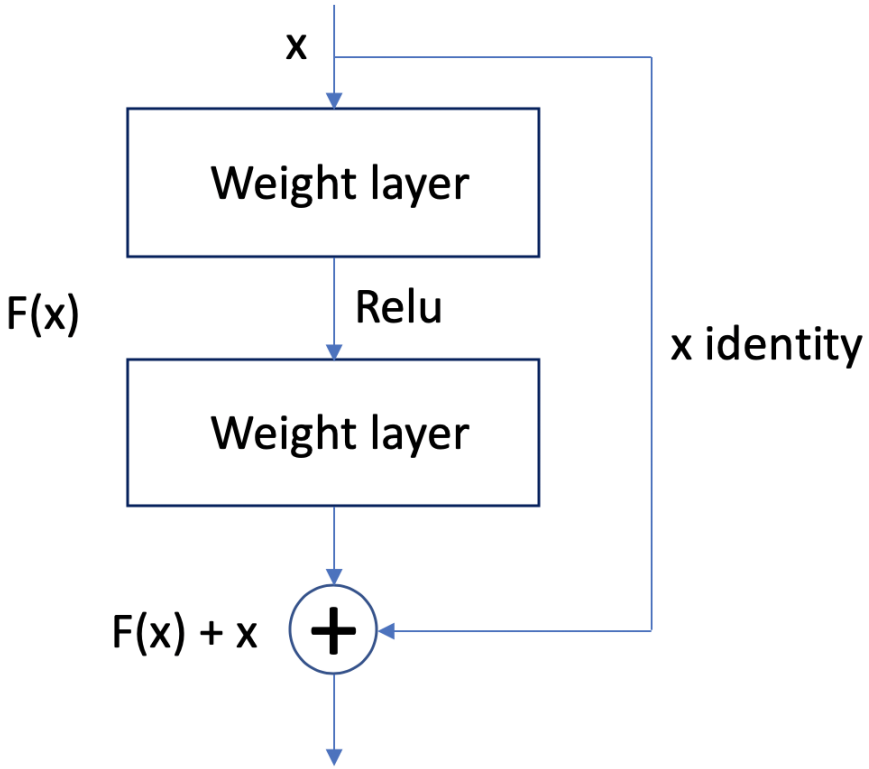


Figure 3.2: Resnet building block for learning the residual (He, X. Zhang, et al., 2016)

The ResNet block architecture allows for the blocks to learn the identity function more efficiently by trying to learn the residual function instead of the direct mapping. The residual can be learned by adding the unchanged weights of an earlier layer to the output of some transformations on it. The residual block is shown in Figure 3.2. Normally a neural network layer learns the mapping between x and y using a function $H(x)$, but by the same token, we can learn a residual transformation. Learning these residuals using the identity mapping skip-connections is the revolutionary idea introduced by the ResNet. Since these skip-connections are only connections and not extra layers, they do not require extra parameters that would need to be learned. The residual transformation is

$$F(x) = H(x) - x$$

where H is the mapping of two or more network layers. Both of these approaches should approximate the same functions; the difference is in how easy it is for the network to learn. Learning a transformation of $F(x) = 0$ would intuitively seem easier for a neural network than learning $F(x) = x$. As can be seen from the success of the ResNets compared to non-residual networks, this is what allows for creating very deep networks. If the transformation changes the input size, a matrix W is necessary to map the input to the same dimensions, generating the final formula for a residual block.

$$y = F(x, \{W_i\}) + W_s x$$

where W_i and W_s are the the weight matrices to be applied to x.

Many variants of ResNet exists, such as wide ResNets (Zagoruyko and Komodakis, 2016), ResNeXt (Xie et al., 2017) and others. Also the DenseNet (G. Huang et al., 2017) is heavily inspired by ResNets. ResNet-50, ResNet-34, ResNet-101, and ResNet-152 are still some of the most popular models to use when a pre-trained ImageNet trained backbone is needed for some part of a classifier as they produce good results and do not contain too many parameters compared to some other architectures.

3.4 Improving model performance

Improving model performance is not easy, but using the various types of skip-connections, such as the ones used in ResNet residual blocks with the identity transform, it is possible to increase the size of the network to massive sizes. A 557 million parameter model called GPipe (Y. Huang et al., 2019) takes the model scaling to the extreme and requires some unique parallelism libraries to train the model. It is still only slightly better than previous models, showing that size is not the only thing that matters.

There are three main ways to scale up a network, as shown in figure 3.3. Scaling by depth means adding more layers to the model, allowing for more complex dependencies to be captured by the model. For example, ResNet-1000 is a very deep type of ResNet, but it has similar performance to a ResNet-101, so there are diminishing returns when trying to scale up by depth (Tan and Le, 2019). Scaling by width means increasing the number of channels in the layers, and it is especially popular when optimizing smaller size models, such as MobileNet (Howard et al., 2017). Wide ResNets (Zagoruyko and Komodakis, 2016) increase the width of the ResNet blocks and allow for better features and easier training. The third way is to increase the resolution of the input. A higher resolution

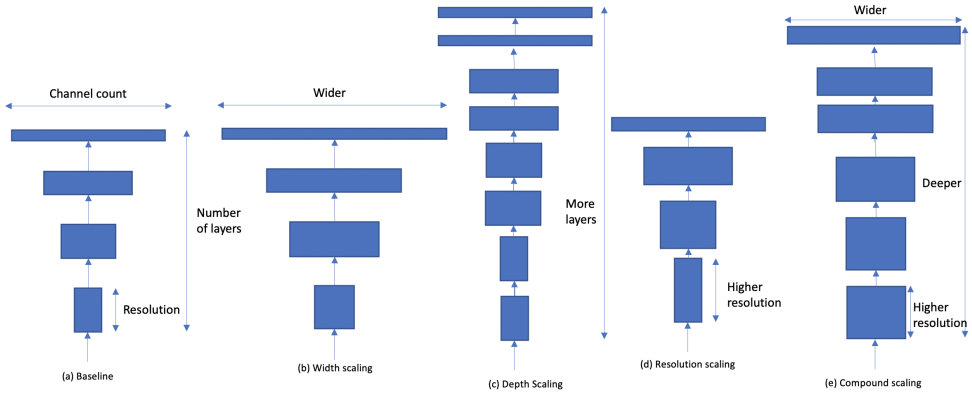


Figure 3.3: Various ways of scaling network architectures (Tan and Le, 2019). a) Shows the baseline network. b) Shows how to scale by the number of channels. c) Shows how to scale the number of layers. d) Shows how to scale by resolution. e) Shows how to scale all the metrics with a compound factor.

allows for the network to find more specific features as the level of detail increases, and often specific network architectures work best at one particular resolution.

The other approach to finding a better architecture is to use different kinds of blocks that are better. One way of finding better building blocks is by manual experimentation. The other, like in the case of MnasNet (Tan, Chen, et al., 2019), is to pose the problem as an optimization problem and to use Neural Architecture Search (Elsken et al., 2019) to find the optimal solution. In the case of MnasNet, the goal is to maximize the model architecture m in

$$ACC(m) \times [\frac{LAT(m)}{T}]^w$$

where T is the target, by searching in a constrained space. So the goal is to optimize the accuracy (ACC) of the model with a given latency (LAT) on a specific device. The search produces small but very efficient models when the target device is a mobile phone.

3.5 EfficientNet

EfficientNet models form a family of models ranging from EfficientNet0 to EfficientNet7 that were generated by smartly scaling existing convolutional models to optimize them for efficiency in a similar way to MnasNet (Tan and Le, 2019). The more complex models are created by using compound scaling on the EfficientNet0 model, shown in Figure 3.3 (e), where the width, depth, and the resolution of the network are scaled using a factor ϕ. As the cost of neural architecture search grows exponentially as the size of the network grows,

the search is only done on the base model. The search of ϕ is a constrained optimization problem, where width, depth and, resolution are constrained such that $d^{\phi} \times w^{\phi} \times r^{\phi} \approx 2$ and each increase in ϕ increases network FLOPS by 2^{ϕ}. The architecture search for the base network is similar to that of MnasNet. The target of EfficinetNet is to maximize the model m

$$ACC(m) \times [\frac{FLOPS(m)}{T}]^{w}$$

so the goal is to optimize for both accuracy (ACC) and number of floating-point operations (FLOPS) instead of latency as in MnasNet (Tan and Le, 2019).

As seen in figure 3.1, these optimizations allow for the EfficientNet to form a very performant architecture using a low amount of parameters that does well on ImageNet and is good at Transfer Learning also. Interestingly though, the reduction in FLOPS and parameters do not directly translate to low real-world inference time. As is compared in the paper, an EfficientNet-B1 gets a 5.7x speedup over a ResNet-152, while the EfficientNet scores significantly higher on ImageNet. So in terms of the time and accuracy ratio, the EfficientNet seems to do well. When looking at the inference time in terms of FLOPS and parameters, we can find out that the ResNet would be more efficient, as the ResNet has 7.6x as many parameters and 16x FLOPS when compared to the EfficientNet, but the inference time multiplier is 5.7x. From this comparison, we can easily see that the number of FLOPS is not a direct proxy for inference time. Instead, the optimization should be for latency directly as in MnasNet if that is the goal.

The proposed EfficientNet models use mobile inverted bottleneck (MBConv) blocks used in MobileNetV2 (Sandler et al., 2018) in constructing the base model. The MobileNet uses a Depthwise Separable Convolution, where a depthwise convolution and a pointwise convolution are applied in sequence. The depthwise convolution applies d_i kxk filters to the input, where d_i is the input channels and k is the kernel size leading to an output channel count $d_j = d_i$. A normal convolutional layer would apply multiple filters having M channels with the computational cost of $h_i w_i d_i d_j k^2$, whereas the depthwise convolution only has a cost of $h_i w_i d_i (k^2 + d_j)$, reducing the cost by k^2. The result of the depthwise convolution then runs through a pointwise convolution, where a $1 \times 1 \times d_j$ 1d convolution is applied to get the final output as a linear combination of the channels. The inverted residuals in MBConv are connections similar to ResNet, but in this case, they are connected between the low dimensional layers.

3.6 Training image classifiers

An image classifier's basic function is to predict which of some set of classes exists in an image. So, to train a model to answer such a question, a data set with images and accompanying labels are needed. For example, the Oxford-IIT Pets dataset (Parkhi et al., 2012) consists of images of 37 different cat and dog species and labels for those images signifying which of the 37 classes an image represents. The dataset consists of about 200 images per class, so it is relatively small and an excellent example of a situation where transfer learning is instrumental in getting good results.

After the data of the images and labels are gathered to a data loader, that outputs small batches for training, the model needs to be constructed. For the backbone, any ImageNet classifier will do. For example, EfficientNet-B4 could be a good decision. So we will initialize the backbone of our model with the weights of a pre-trained EfficientNet that was trained on the ImageNet data set by someone with access to large computational resources. After the model has been initialized, the head has to be changed to be compatible with our new task. By default, the model will be expecting to classify images to the thousand classes of the ImageNet. In this case, we want to solve the 37 cat and dog species problem, so the final fully connected layer has to be changed. It will be initialized as a fully connected layer with 37 output units, representing each of the classes, connecting the embedding to the outputs. The head does not need to be just a single, fully connected layer, but it could also contain more fully connected layers, dropout layers, and batch normalization layers and more. Adding more than a single fully connected layer might be a good idea if the backbone remains frozen and is not fine-tuned. That way, the head can learn some useful combinations of the image embedding instead of optimizing the features in the backbone.

Once all the layers are in place, the model needs to be optimized with a training loop. Every iteration, a loss needs to be calculated for the batch that has been sampled from the data loader. To be able to calculate the loss, the outputs of the final fully connected layer need to be normalized into a distribution of probabilities with softmax. The softmax is defined as

$$Softmax(x_i) = \frac{exp(x_i)}{\sum_j exp(x_j)}$$

For each class i, there is an output score of x_i, for which we will calculate the proportion of its score with respect to the sum of all the classes' scores x_j, creating the probability distribution. Since we have a single correct class, we would want to maximize the probability of the correct class and minimize the probability of the other classes. The loss function to

do exactly this optimization is called categorical cross-entropy, and it is defined as follows.

$$CE = -\frac{1}{N}\sum_{i=1}^{N}(y_i * log(p_i) + (1 - y_i) * log(1 - p_i))$$

Where N is the number of classes, y_i is either 1 or 0 depending on if it is the correct label and p_i is the probability for that class. This formula does exactly what we previously described that we wanted to do. To get the loss for a single class, if $y = 1$, the value will be the first part of the sum, if $y = 0$, the value will be the second part of the sum, since $(1 - y_i)$ is now 1. When we optimize the model, in the case of the chosen architecture, the model will reach around 95% accuracy on this data set (Tan and Le, 2019).

4 Multi-task learning

Multi-task learning is a generalization of Transfer Learning to training a single model where the goal is to optimize for multiple objectives. Training a multi-task model requires a set of distinct tasks to be posed as a multi-task learning problem. It can have many benefits when applied correctly, such as making the models more general by regularization and reducing computational requirements. On the other hand, if multi-task learning is applied to problems that don't train well in a multi-task setting, the resulting models can be significantly worse than the single-task counterparts. Here we look at examples of multi-task learning in computer vision. Especially recently, with the introduction of transformers like BERT (Devlin et al., 2019), multi-task learning and transfer learning have gained popularity within natural language processing, but to restrict our scope, we only consider how to apply multi-task learning in computer vision.

4.1 Definition

Multi-task learning is quite similar to transfer learning. However, the main difference lies in the fact that the goal is to generalize to solve and improve performance in all tasks using some shared representation. In contrast, transfer learning aims to optimize a single new task, ignoring the performance on the original task entirely. Often in multi-task learning, the shared layers are initialized using some ImageNet model and transfer learning as the ImageNet backbone is an architectural feature shared in many models.

The formal definition for multi-task learning follows the definition given in (Y. Zhang and Yang, 2018). A learning problem consists of m related learning tasks $\{T_i\}_{i=1}^{m}$ that are trained together. Each task has a dataset D_i with n_i pairs $\{x_j^i, y_j^i\}_{j=1}^{n_i}$, where x_j^i refers to the input of task T_i and y_j^i is the label corresponding to the input vector. Let $X^i = (x_1^i, ..., x_{n_i}^i)$ be the input matrix for task T_i. In a multi-task setting there can be x such that $x \in X^i$ and $x \in X^j$ or $X^i = X^j$ for some $i \neq j$, meaning that a single image has multiple labels or an entire data set has labels for multiple tasks.

To train the network, each task T_i needs to have its loss function defined. A commonly used way to get the total loss of the input is by using the weighted loss functions for all

tasks, resulting in a formula

$$L_{total} = \sum_i w_i L_i$$

, where L_i is the loss function for task T_i and w_i, is the weight that specifies the sensitivity of the task (Cipolla et al., 2018). The sensitivity multiplies the loss of the task. Multiplying the loss has the effect of making the error on the task with higher sensitivity to have a larger impact on the adjustment of weights. So the sensitivity should be used to either pick tasks that are important or to normalize the losses if they are very different. There can be significant differences in the scale of the losses if different tasks use different loss functions. The sensitivity of the tasks is an essential parameter to get right as choosing a too high parameter for some task might lead to a solution that is optimal for only the most sensitive task, starving the others (Standley et al., 2019).

Once the loss function is specified, the actual training is quite similar to training a single-task model. The one new thing to consider in a multi-task setting is the sampling ratio of the different tasks as some tasks can be easier to learn or contain significantly more or less data compared to other tasks. Like in regular training, the models are trained one batch at a time. In a multi-task setting, we have datasets for each of the tasks from which we can pick at a time. Now, we can define an epoch e in the training process as the total number of batches from all tasks. The epoch can be formalized as

$$e = \sum_i \frac{|X_i|\alpha_i}{B_i}$$

batches over i tasks, where $|X_i|$ is the training set size, B_i is the batch size, and α_i is the scaling factor for each task, telling how important the specific task is for the learning process. The sampling ratio is one of the new parameters that is required when doing multi-task learning on multiple datasets. When training models, some value has to be picked for the sampling ratios, and picking wrong values leads to worse models. Precisely what is a good sampling ratio is complicated to tell beforehand, and generally, the chosen values need to be evaluated by experimentation. Iterating through all batches can be done in a round-robin fashion, by going through all batches of each data set at a time or randomly sampling the sets with probabilities respective to their magnitude and scaling factor.

4.2 Latent image representations

For a multi-task model, the optimal situation is such, where all classifiers would use a single shared latent representation of the image to predict all attributes. This representation is an image embedding, that would capture all valuable information of the image for our tasks. The embedding could be the vector represented by the final layer of an ImageNet classifier flattened to some n-dimensional vector that then solves all the tasks instead of just a single task. However, such a universal image representation can be challenging or

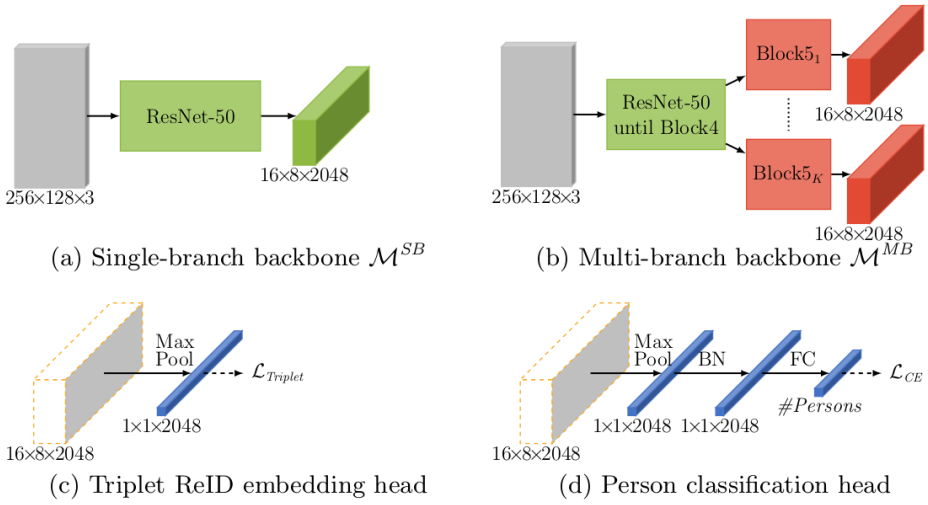


Figure 4.1: Example of partly and completely shared backbones. Figure from (Pfeiffer et al., 2019).

impossible to learn, and instead, only a part of the network is shared between the tasks. Here, this shared structure is called the backbone of the multi-task classifier. Often the backbone is in the form of some ImageNet classifier architecture, and it can be the entire classifier or some layers up to some arbitrary limit. As the amount of layers shared is picked for each task separately, some tasks can share the entire classifier as a backbone, and others might only share some amount of layer that produces desirable results.

Besides just deciding the network branching, each task requires an independent head that outputs prediction for that task. These task heads are similar to those used in transfer learning, where only a single task is solved using the ImageNet backbone. Especially in a multi-task setting, these heads can be more than just a single linear layer calculating the softmax over all classes. A single head can contain any combination of several linear layers, dropout layers, batch normalization layers, convolutional layers, or it could be a Recurrent Neural Network creating a caption for the image embedding, depending on the

task at hand. In Figure 4.1, we can see an example of the flexibility of the sharing as the model can use a completely shared backbone or a branching backbone, which has a unique head for each task, solving the person re-identification and classification tasks.

4.3 Benefits

Depending on how and where multi-task learning is applied, it can provide a multitude of benefits to the model. Many of these benefits stem from the fact that the model gets to see more data, and the various tasks can make it easier to find useful features. Different kinds of data sets have different kinds of noise, learning multiple tasks makes it easier to distinguish which features are beneficial and detrimental some features may be difficult to learn on a dataset but can be useful and borrowed from another and learning multiple tasks forces the model to not overfit on one of them (Ruder, 2017).

These benefits come up when the tasks are compatible and allow the model to learn more general features, generally leading to better performance on the distinct tasks. When the needed features between tasks are conflicting, the model performance tends to go down (Kokkinos, 2017). The problem of deciding what to share is not easy to solve.

The benefits of multi-task learning come up, especially when dealing with limited amounts of data, in which case, particularly finding the features that matter without overfitting can be complicated. For example (W. Zhang et al., 2020) found that multi-task learning could improve gene expression pattern classifiers when trained in a multi-task setting. The multi-task classifier was significantly better than the one only using transfer learning, showing that the features learned were more general.

Another significant boon of multi-task models, especially in embedded domains, is the reduction in model size and inference time. As many classifiers are dependent on an embedding of an image to produce results, using a shared embedding of an image for multiple tasks means that the model requires only a single partly or completely shared backbone. For example (Kocabas et al., 2018) uses a shared backbone to detect people in an image and to detect keypoints on their body and then finally to do a semantic segmentation of the image while also improving performance on most of the tasks.

Finally, a model using multi-task learning can take advantage of the different losses to produce a more optimal loss weighing strategy compared to constant weighting. Picking the weights in the total loss function is very important as with invalid weights, the op-

timization can be difficult or even impossible (Gong et al., 2019). The weights become even more critical if the losses for various tasks are different, for example, one task might use mean squared error as a loss function and another cross-entropy, and the resulting loss values might differ by orders of magnitude. The total loss function can be modified by adding an uncertainty weighting to each of the tasks by considering the uncertainty of the prediction (Cipolla et al., 2018). Depending on the task, the benefit compared to an unweighted loss can be significant (Cipolla et al., 2018) or, in some cases, only small (Gong et al., 2019).

4.4 Hard parameter sharing

A hard parameter sharing setting is one where the same convolutional weights in the intermediate layers solve multiple tasks, like in Figure 4.1. Hard parameter sharing is the primary way of reducing total model size and inference time when solving multiple tasks simultaneously. While sharing the weights can provide many benefits to the end model, as seen in the previous chapter, it comes at the cost of an increased number of tunable parameters when training the models. As we have previously seen, multi-task models introduce new constant factors to the training process beyond the normal hyperparameters for learning rate, dropout rate, and others. These include the hyperparameters for weighing the loss functions and determining the task sampling ratios, but also the expensive to evaluate architectural decision of which tasks should share representations and how much should be shared.

In Figure 4.2, there are multiple ways of configuring a multi-task network in terms of what to share. The results for the different configurations were quite varied. However, even the worst multi-task Model A did better on every task when compared to the single-task models, and accuracy improvement from the single-task case to the best multi-task Model F was roughly 20% (Park et al., 2019). These results are quite validating for the presumed benefits of multi-task models. Since these results show that it is possible to share the majority of the network parameters and, at the same time, increase the performance, even with the most greedy share-all model. Though, at the same time, it shows that multiple compute-expensive experiments have to done to find the most optimal sharing structure for the backbone network.

In the experiments done by (Park et al., 2019), they utilized auxiliary training to great effect. They had two original features of interest that were sugar and alcohol level in the

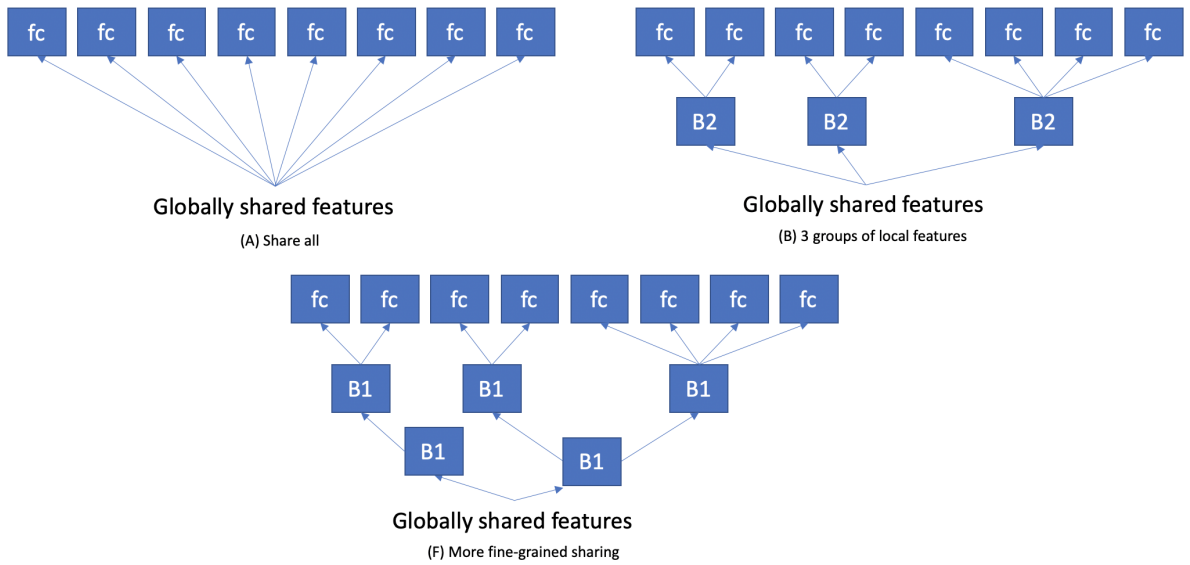


Figure 4.2: Some ways of sharing features between tasks. Here the global features are from resnet. The fc blocks are fully-connected heads. B1 is one ResNet block and B2 is two ResNet blocks. Out of these model F was the best. Figure adapted from (Park et al., 2019).

drinks. Other features were added as auxiliary training features to improve the performance on the two original features (Park et al., 2019). This auxiliary training can also be done by, for example, using ImageNet to keep the original classifier accurate on ImageNet while training on the new data set like in (W. Zhang et al., 2020); this way, the model won't be able to overfit on the new data so easily. In the drink classification experiment, the auxiliary tasks were picked in a way that they might help in finding the correct features to solve the actual tasks (Park et al., 2019). Adding auxiliary tasks is an interesting way of forcing features into the model that the developer thinks could be useful for the model to find, in that sense, they are hand-crafted features that get augmented in the model. In practice, this means that if we are training a model e.g., to predict the steering angle given an image, we might want to add an auxiliary task that predicts the lane markers to make it easier to learn these features as they should be a significant feature for producing correct outputs.

A similar multi-task model in (Pfeiffer et al., 2019), also partly visualized in Figure 4.1, solves 6 different tasks in a single model. The tasks in that model are very different from one another, including person re-identification, pose estimation, and image segmentation. The interesting result in their search for related tasks is that while a pair of tasks don't work well together, they can work well when combined with other tasks. For example, they found that just pairing the pose estimation head with other tasks significantly reduces

its accuracy, but when all tasks use the single shared backbone, it does just as well as if trained in a single-task setting. These results show that very different tasks can combine to a single model to provide a very significant reduction in model size while producing about as good or better accuracies.

4.5 Soft parameter sharing

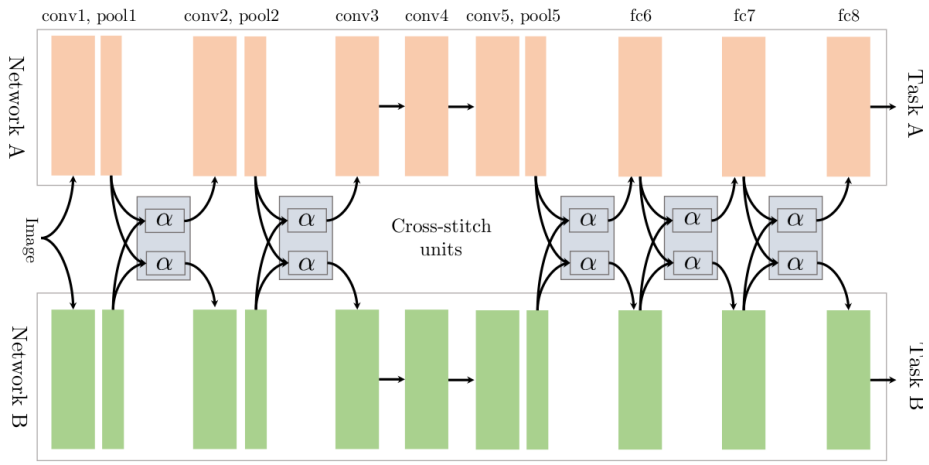


Figure 4.3: Cross-stitch architecture. Figure from (Misra et al., 2016).

Soft parameter sharing is quite similar to learning each task on its own since each task has separate weights. Soft parameter sharing happens by picking some layers, where some metrics constrain the parameters to be similar, like l_2 distance (Ruder, 2017).

The sharing functionality can be more complicated than just basic regularization. For example, the Cross-stitch units (Misra et al., 2016) are a particular type of soft sharing, where the stitch units combine tasks A and B using a linear combination of the activations, visualized in Figure 4.3. The benefit here is that the user does not need to specify how much should be shared between the tasks, but it is instead learned in the cross-stitch unit. If nothing needs to be shared, then the network can learn to assign the weight for the other task to zero.

This kind of sharing does not scale very well as the number of tasks increases since calculating the relation between various tasks is quadratic. Since the joining logic requires extra parameters, it means there is even more to learn than just learning all the tasks separately. As soft sharing does not provide many of the benefits that are gained by hard

sharing the parameters, it is not as popular. However, it is useful in adding some extra information between the tasks to gain extra performance by utilizing the latent representations between them in various ways. Also, in the basic case of using a soft parameter share to constraint, the layers can improve the generalizability of the model.

4.6 Attention augmented Multi-task learning

In a multi-task setting, the model ends up having a representation of the image from each task's point of view, and it is possible to use attention (Vaswani et al., 2017) to augment the predictions. Especially in the case where the tasks are heavily correlated, this can be quite useful. For example, when doing weather recognition, it would make sense that some of the features are correlated, and this correlation can be used by adding an attention layer between the different tasks, like in (Zhao et al., 2019). The Multi-Task Attention Network (MTAN) (Liu, Johns, et al., 2019) is an example of a more advanced application of attention in multi-task models. In the MTAN, each task gets its own attention module, and they are used to determine correlations of the tasks at specific layers of the network using the attention operations. It is similar to cross-stitching networks in that each task has to learn its attention module features. However, there is only a single backbone for the shared features, and the attention modules are responsible for producing the task level outputs. The number of parameters is still significantly reduced as the task specific-parts are not complete ImageNet classifiers, resulting in a significantly more efficient and accurate classifier for image segmentation.

4.7 What and when to share

What makes or breaks a multi-task model is the decision on what should be shared, but determining what and how much should be shared is exponentially more expensive as the number of tasks grows. Currently, there is no definitive way of theoretically evaluating which tasks should have a shared representation and just how much can be shared. As experimentation is expensive, a good starting point is to try to find similar tasks that should do well together. However, there is no guarantee that even all the tasks within a single family of problems are beneficial for joint training, so some experimentation has to be done. The desired result in a multi-task setting would be to share all the parameters between all the tasks, but as can be seen in (Kokkinos, 2017), that approach often signif-

icantly reduces performance. If sharing an entire network does not produce good results, it can be a good idea only to share some part of it as the lower level features tend to be more general (Oquab et al., 2014). By sharing only a part of the network, it may be possible to strike the right balance between performance and model complexity.

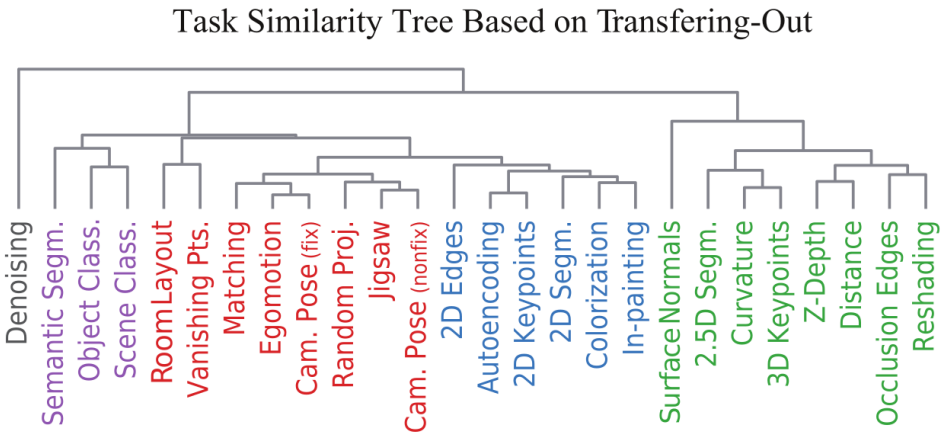


Figure 4.4: The taskonomy task similarity tree. Figure from (Zamir et al., 2018).

An attempt to create a taxonomy of related tasks called Taskonomy to find out what tasks are beneficial for transfer learning exists in (Zamir et al., 2018). The Taskonomy experiments pre-trained networks on tasks and then evaluated whether the features would transfer well to other tasks to end up with a hierarchical categorization of related tasks, shown in Figure 4.4. Still, the authors noted that, depending on model architecture and data set, the results could be different.

It would make sense that the results of the Taskonomy would be easily transferable to the multi-task setting. When evaluating the multi-task vs. transfer affinity, it turns out that they are negatively correlated, at least in the case of the five tasks that were the focus of the experiments in (Standley et al., 2019). Based on this observation, it can be beneficial to train some non-related tasks together. The authors suggest that the different tasks act as a good form of regularization as the models need to generalize to multiple types of inputs. It could also be that some of the learned features work very well for the other task, but can't be easily learned with the dataset of the other task and vice versa. These results are aligned with the empirical results that we saw when we looked at the compatibility of classifiers using hard weight sharing. Currently, it seems not to be possible to determine the compatibility of different tasks purely on a theoretical grounding. Instead, experiments need to be run, and the results of pairs of tasks do not always generalize to a set of all other tasks when using shared representations in models (Pfeiffer et al., 2019).

5 Object detection

Object detection is another prevalent task in the domain of computer vision. An object detection task is one where the goal is to localize one or many different classes of objects using bounding boxes. Before current deep learning-based techniques, a popular way to solve the problem was to use handcrafted features like in image classification and to use a sliding window over the image for localizing objects. These previous techniques include, for example, the Viola-Jones detector (Viola and Jones, 2001), which uses a sliding window and AdaBoost for features, and another popular method was using histograms of oriented gradients (Dalal and Triggs, 2005) to find where the boundaries of objects exist. These days there exist various ways of using the feature maps of neural networks for learning the filters to get much better results. The various architectures can be split into two approaches, one-stage detectors, and two-stage detectors. The two-stage detectors require region proposals, based on which the object detection is done. An often-used family of these kinds of methods is the R-CNN classifiers, for example, the faster R-CNN (Ren et al., 2015). In the one-stage detectors, features from various layers of the classifier are used to get the predictions. For example, the YOLOv4 (Bochkovskiy et al., 2020) is the 4th iteration of the single-stage YOLO family of detectors that are very popular due to the good balance between fast speed and good accuracy.

Figure 5.1: Object detection image labeled for people and cars.

5.1 Metrics

The training of object detection models requires data sets that have been labeled for that purpose. Generally, the data sets contain bounding box annotations for each of the classes in the data set, such as seen in Figure 5.1. For this reason, object detection models can't use a simple metric like the basic accuracy in image classification. As the images are often manually labeled, the boxes are most likely not completely consistent. So most likely, the predictions are never going to align with the labeled boxes perfectly. Consequently, the method for evaluating object detection performance is Average Precision (AP) or mean Average Precision (mAP) or one of their variants. These metrics use intersection over union (IoU) score to evaluate how incorrect the predicted bounding box is when compared to the actual label. Intersection over union is visualized in Figure 5.2.

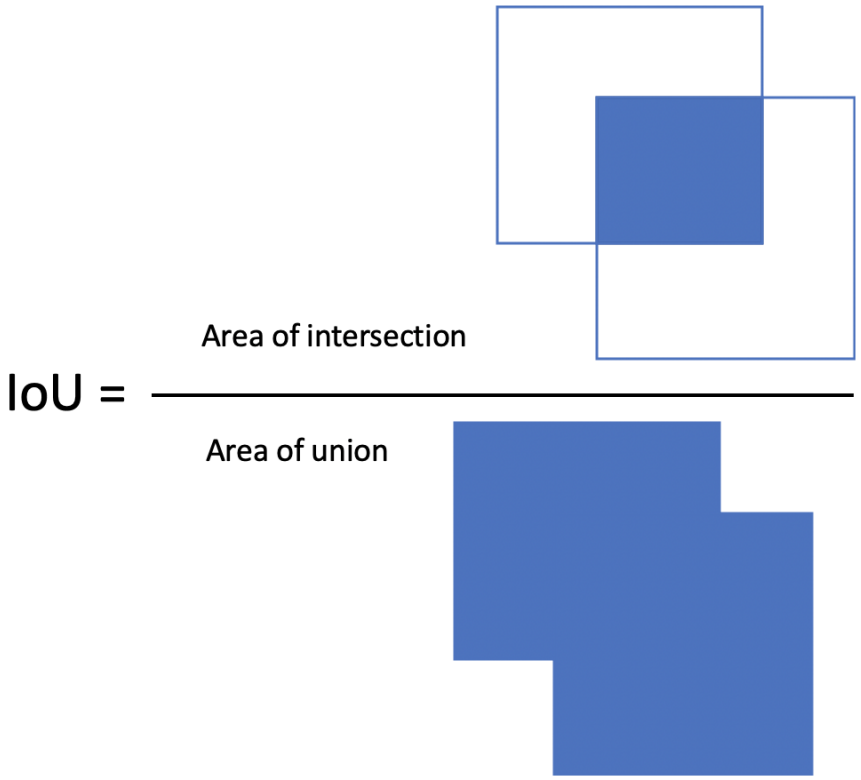


Figure 5.2: The visual formula for calculating intersection over union.

To get to a final AP score, the first thing that has to be decided is the IoU score threshold for considering a prediction to be correct. For example, we could consider all predictions

that have over 50% or 75% overlap in the IoU. The threshold value for IoU that should be used is not entirely standardized, and there are multiple ways of calculating the AP score. For example, the popular COCO dataset (T.-Y. Lin, Maire, et al., 2014) uses an average of 10 IoU scores ranging from .5 to .95 IoU thresholds as the main metric. To get the AP for a class, we need to graph the precision-recall curve and then calculate the area under it. For example, the Pascal Visual Object Classes Challenge (VOC) (Everingham et al., 2010) recommends doing this by using 11 points of interpolated precision. So we get the following formula for Average Precision:

$$AP = \frac{1}{11} \sum_{r \in \{0, 0.1, \ldots, 0.9, 1\}} p_{interp}(r)$$

Where p_{interp} is defined as

$$p_{interp}(r) = \max_{\tilde{r}:\tilde{r} \geq r} p(\tilde{r})$$

Then to get the mAP score, we can average the AP score for each of the classes in the data set. As was mentioned, this way of calculating the AP is not always the same, for example, the COCO metrics (*COCO - Common Objects in Context* 2020) recommend using 101 points for integrating the curve instead of the 11 proposed in the VOC. This discrepancy in the integration means that not all AP scores are directly comparable.

5.2 Object detection model structure

Much like image classifiers and many other vision tasks, modern object detectors rely on pre-trained backbones to generate the features required for predicting bounding boxes. A classifier head is required to get these predictions, but in object detectors, this head is more complicated than in image classification.

For the backbone, most object detection architectures use one of the same ImageNet models as in image classification such as ResNet or VGG. However, the YOLO models use DarkNet, which is specifically designed for achieving efficient results in object detection (Bochkovskiy et al., 2020). The main reason for picking one backbone over another is often a function of both accuracy and inference time. As object detectors are often run on video data, inference time can be more significant than in image classification, as it is often desirable to be able to run them in real-time.

The head is where the architectures differ the most, and generally, any backbone could be combined with a specific detector head. The head comprises of a neck, which connects the intermediate features from the backbone to the final head. The intermediate features are generally connections from some specific layers of the backbone. Earlier single-stage detectors used the extracted features directly, but the current state-of-the-art methods use special feature pyramids and path aggregation to get the best results (Tan, Pang, et al., 2020). For example, the YOLO v4 uses spatial pyramid pooling (He, X. Zhang, et al., 2015) over the DarkNet features and path aggregation net (PANet) (Liu, Qi, et al., 2018) to concatenate the parameters for the classifier head (Bochkovskiy et al., 2020). Different ways of combining the feature pyramids are described in Figure 5.3. Like many other parts of the architecture, picking one is a trade-off of more parameters and inference time and accuracy.

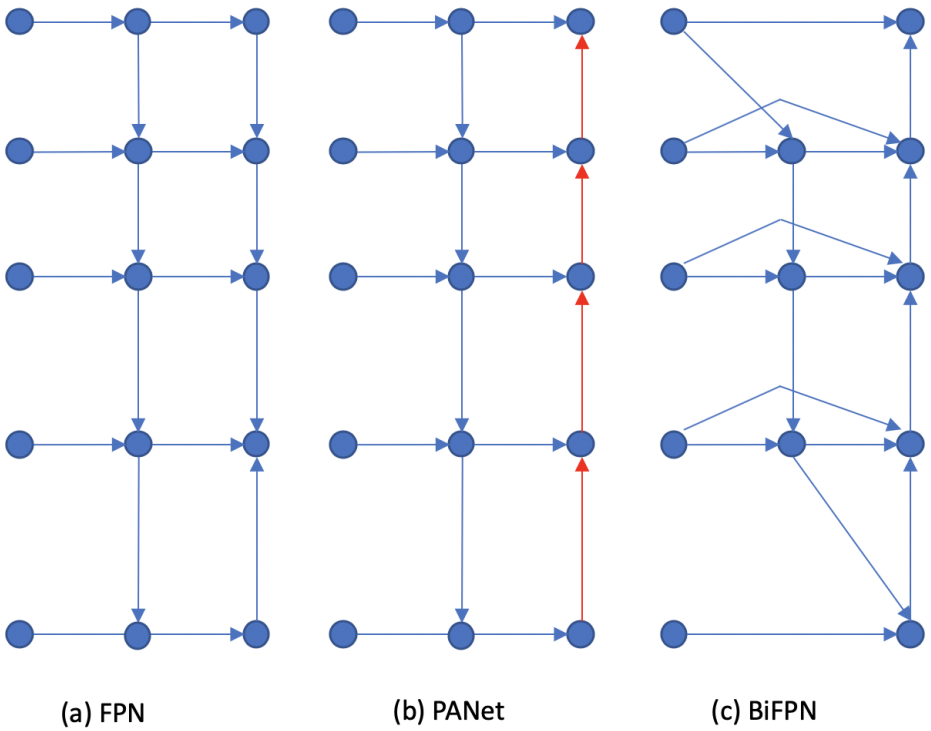


Figure 5.3: a) Basic FPN (T.-Y. Lin, Dollár, et al., 2017) where features are combined in only one direction. b) PANet modifies the basic FPN by adding a path from the lower level features to higher-level features. c) BiFPN tries to prune the connections to the most important ones. Image adapted from (Tan, Pang, et al., 2020).

In two-stage detectors like faster R-CNN (Ren et al., 2015), there is a region proposal network (RPN), which predicts region proposals using a sliding window for some anchor boxes. Running this RPN is expensive, and often the two-stage models are much slower

than the single-stage detectors. The benefit of the two-stage approach is generally a better accuracy. Still, recently the one-stage methods like YOLO v4 have achieved very similar accuracies to the two-stage ones with much faster inference times (Bochkovskiy et al., 2020). By utilizing new ways of using the feature maps, the object detectors have recently become significantly better over the years, as can be seen from the different iterations of the YOLO models, as they have been using very similar backbones over the years (Bochkovskiy et al., 2020).

5.3 Multi-dataset training

Often an object detection problem requires detecting multiple different classes in a similar context. For example, a self-driving car would need to detect various traffic signs, cars, pedestrians, road markings, cyclists, and many other things. Collecting a single dataset that has labels for all of the classes of interest can be a very daunting task. Even if it is feasible to create the dataset for the original classes of interest, this approach does not scale very well when new classes need to be recognized. As likely the original dataset might contain millions of images labeled for multiple classes, adding a new class would require going through the entire dataset again and labeling the new class as well. The new class may be relatively rare; for example, we may be interested in detecting emergency vehicles with sirens on. Here is where multi-dataset learning is highly beneficial as it only allows for collecting a specific class dataset. This type of cross-dataset learning is useful when we need to combine multiple distinct data sources to detect some union of the labels (Yao et al., 2020).

The main difficulty in combining multiple datasets for detection lies in the fact that they likely contain unlabeled overlapping classes. For example, given a dataset for detecting cars and another for detecting stop signs, we have two distinct datasets where only one of the classes is labeled. If we train this model, assuming that the labels are genuinely valid, we will end up unlearning the tasks due to the overlap of the classes. The problem lies in the fact that it is most likely that in the car dataset, we will find stop signs that are not labeled. Similarily we will find cars that are unlabeled within the stop sign dataset. When we naively train this model, we will end up detecting cars and stop signs that are actually correct but incorrect based on the labels. The model will be punished for detecting these non-labeled positive examples due to the absence of the labels.

One way to do this is by disallowing the labels from incorrect datasets affecting the total

loss. For example, in (Yao et al., 2020), the RetinaNet (T. Lin et al., 2017) model's loss function is modified to ignore the losses for tasks that are not a part of the dataset. All models using the same focal loss as RetinaNet can be trained with this adjusted loss function. This way, each class will get its own positive and negative dataset to train on and not affect the performance of the other classes. Still, this does not completely solve the problem as some of the classes might require conflicting features, as can be the case when very different tasks are combined. In that case, it could be smart to split the similar classes into different detectors and maybe share only the lower levels between all tasks.

Similar multi-dataset training can also be applied in multi-label image classification settings. A multi-label image classification problem is one where we want to assign multiple labels to an image. For example, we might want to classify whether it is raining, the sun is shining, the sea is visible, is there a dog in the image, and so on for all the classes of interest. Again, collecting a dataset containing all labels for all images is quite expensive, but we can train this with a separate binary classification head for each task. And as collecting a separate dataset for each task is relatively easy, it is possible to create a quite powerful model with relative ease.

5.4 EfficientDet

EfficientDet (Tan, Pang, et al., 2020) is an object detection model, that is based on the previously covered EfficientNet ImageNet model. The neck for the EfficientDet is a unique bidirectional feature pyramid network (BiFPN). The EfficientDet model also uses similar compound scaling as the EfficientNet models for the new object detection specific parts. The scaling factor is used for the BiFPN network, box and class predictor networks, and to find the optimal input resolution. The EfficinetDet shows that improving the backbone is quite useful for the entirety of the model. This can especially be seen when looking at the comparisons of using a ResNet vs an EfficientNet as the backbone in the same EfficientDet architecture (Tan, Pang, et al., 2020). The EfficientDet model architecture is shown in Figure 5.4.

The different feature maps in the BiFPN are combined using a weighted feature fusion. The feature maps can't be directly combined since they have different dimensionality, so they have to be up or downsampled depending on the direction of the connection. By weighing each feature map, the network can learn how important each feature map is. Since the dimensionalities differ from one another, they likely don't contribute equally to

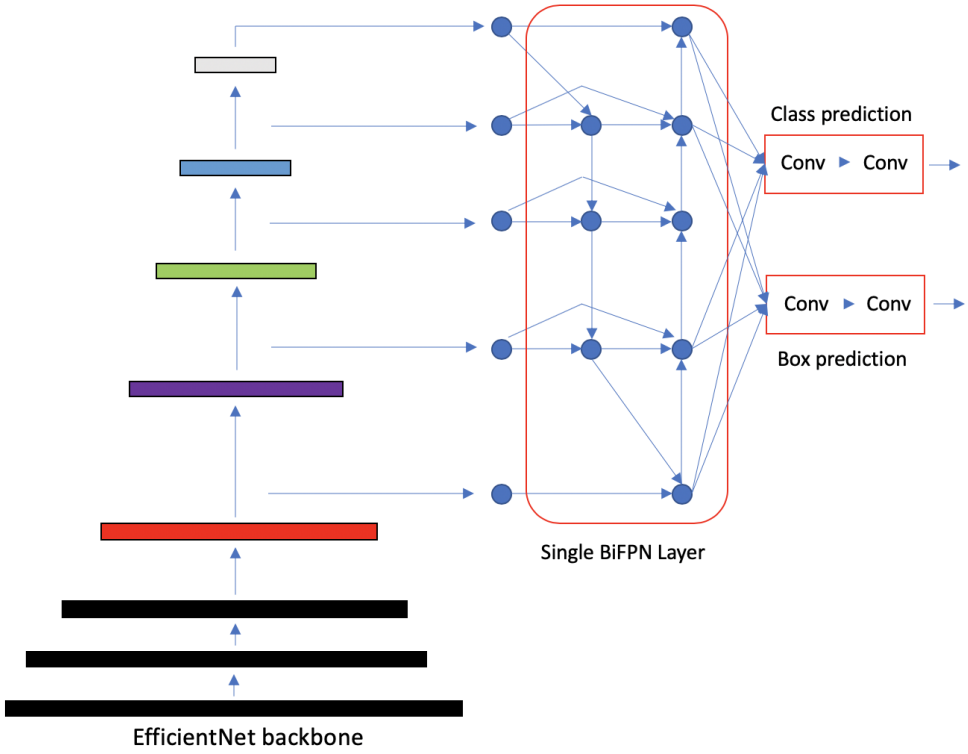


Figure 5.4: EfficientDet architecture comprises of the EfficientNet (Tan and Le, 2019) backbone, the bidirectional feature pyramid network neck and the class and box classifier heads. Here only one BiFPN layer is pictured, but there are multiple of them normally stacked in succession.

the output feature maps, which is signified by the learned weight. The BiFPN connection architecture is a simplified version of the PANet, as shown in Figure 5.3. The goal of the changes from PANet is to have a more high-level fusion of features while dropping the nodes that have little feature fusion and then to stack multiple of these layers to get the final model (Tan, Pang, et al., 2020).

The efficient scaling of the parameters and the BiFPN architecture allow EfficientDet to produce impressive results at relatively high framerates. The new YOLO v4 produces very similar scores at acceptable framerates also (Bochkovskiy et al., 2020). Still, some slightly better results are produced by the two-stage detectors, but those can't reach a 30 FPS inference time (Bochkovskiy et al., 2020). So in practice, using either the EfficientNet or YOLO is a good idea when video needs to be classified. The EfficientNet scales relatively well with larger EfficientNet backbones, but it is not possible to classify real-time video. This flexibility allows for picking the model with a suitable accuracy for the problem at

hand.

5.5 Training object detectors

As we described previously, object detectors are trained when there is a need to localize some set of classes within images by describing them with bounding boxes. One widely used format for labeling such a dataset is the format for the object detection data in the COCO data set. The COCO data consists of a set of images and a separate JSON file containing the labels. The label file contains a list of all the images for the data set and a unique identifier for each image. There is also an annotation list that contains a list of objects that specify the bounding box annotations. The annotations are specified as being linked to a single image by its image id and defining the bounding box annotation dimensions. The bounding box annotations are formatted by providing the coordinates of the top-left corner of the annotations and then the width and height of the box, so it is a tuple of four numbers (x, y, width, height). To train on new data sets, they should be formatted similarly to make the training as easy as possible.

Once the data set has been gathered and collected in a correct format for the object detection task, a model needs to be picked. Here, the final inference model's requirements need to be evaluated to decide whether a single-stage or a two-stage detector should be used. For example, when running the model in real-time, a single-stage detector might be better, and in the case of doing text detection, it seems to be more popular to use a two-stage detector. We will describe here how things work for the EfficientDet, which is a single-stage detector.

Once the exact architecture has been chosen, an implementation of it is needed to construct the model. Generally, all the popular architectures have open-source implementations on GitHub in both Tensorflow and Pytorch, which are the most popular deep learning frameworks, that can be used. Most of these implementations come quite readily wrapped in a way that given a correctly formatted data set, the models will be automatically initialized with the correct number of classes, and the training process is already specified by the input of a few necessary parameters. As mentioned in the case of image classification, it is extremely important to initialize with the pre-trained weights with an object detection model to achieve the best results instead of training from scratch. In an object detector, the problem-specific part is the number of classes in the classification head, which can be seen in Figure 5.4. For the rest of the network, pre-trained weights should be used.

Training the EfficientNet itself is a multi-task training problem. We have two losses, the regression loss signifying how close the bounding boxes are to the annotations by calculating IoU and classification loss, which tells how incorrect the predicted labels are. A major problem in optimizing anchor-based single-stage detectors like the EfficientDet is the fact that between the background no-class and the foreground classes of interest, there is a massive discrepancy in the number of instances to classify (T. Lin et al., 2017). The anchor-based detectors determine whether an anchor placed at any valid spot in the image contains a class of interest or the background. There are tens of thousands to hundreds of thousands of valid anchors to consider depending on the architecture. Due to this, we can have a class imbalance of 1:1000 between the positive and negative cases. For this reason, the recent single-stage detectors use a focal loss presented in the RetinaNet paper, to account for this class imbalance (T. Lin et al., 2017).

The focal loss introduces two new parameters to the cross-entropy formula to account for the class imbalance. The formula for focal loss (FL) is as follows.

$$FL(p_i) = -\alpha_i * log(1 - p_i)^{\gamma} * log(p_i)$$

Where α and γ are the new scaling factors and p_i is defined as p if y $= 1$ and $1 - p$ otherwise. The factor α is there to weight the loss for the positive and negative cases separately. It is a parameter that is recommended to be set at the inverse class frequency, but the value can be searched by experimentation as well (T. Lin et al., 2017). The factor γ makes sure that a large number of easy classifications do not dominate the total loss. For γ the recommended value is two (T. Lin et al., 2017). With this loss function, it is possible to have a very large number of anchors, where a large majority of them are always predicted as the background class.

With these, the object detection model can be trained. Still, just leaving all the extra parameters at the default values may not produce the best results. For example, the possible anchors that the model uses are specified using specific parameters. The parameters that can be modified are the anchor size, depending on whether large or small objects need to be detected, the anchor sizes can be modified to suit the task at hand. The anchors also have aspect ratios applied to them, which can be modified, the default ratios are 1:1, 2:1, and 1:2.

Finally, once a model has been trained with a suitable dataset, architecture, and hyperparameters, it can output predictions for new images. If the model is used without any modifications, the output boxes will likely look like the predictions in Figure 5.5. To fix this issue in the predictions, non-max suppression (NMS) should be used (Bodla et al.,

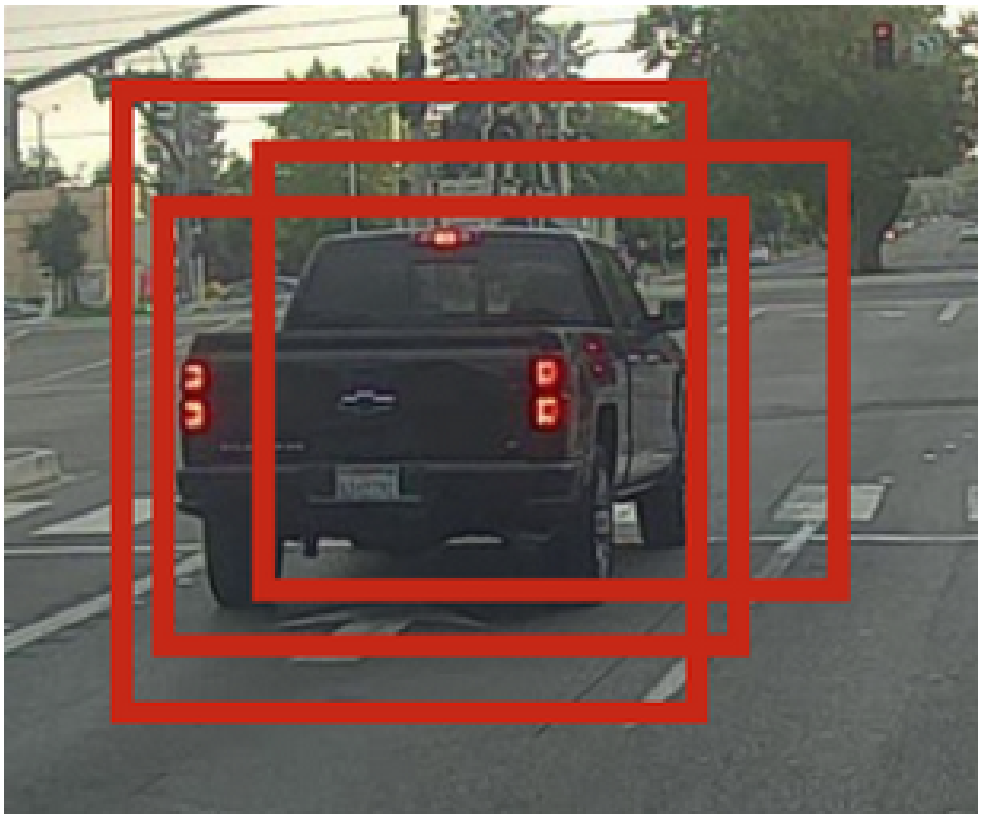


Figure 5.5: Object detection without non-max suppression. The same car is detected multiple times.

2017). With non-max suppression, we can achieve only a single bounding box that better represents what is most likely wanted. In some cases, using NMS can lead to not being able to predict heavily overlapping classes. Generally, though NMS should pretty much always be used to avoid the issue depicted in the Figure 5.5.

The object detectors are a good example of multi-task learning and how using the same feature maps, it is possible to solve multiple problems. In this case, only a single annotation is needed to label for the two different tasks that can be optimized together. Since we have two losses that are calculated, we can decide to focus on one of the losses by adding a multiplier to it and see if the higher loss could improve that task. While some of the things covered here will be similar in the two-stage detectors, the region proposal network approach is very different from the anchor-based single-stage detectors covered in this section.

6 Experiments

As we have covered, the effect of a Multi-Task problem setting is challenging to quantify theoretically without experiments. Here we will take various datasets for object recognition, object detection, and pose estimation and see what the effects of training them in a multi-task setting are. The metrics we are mainly interested in are the reduction in model size, inference time, and model accuracy on the different tasks compared to the single-task counterparts. We will also address the difficulty of the multi-task training process when compared to the single-task problem. These issues will include trying different sharing architectures and searching for the new hyperparameters that provide the best possible results. The experiments will focus on utilizing the previously presented EfficientNet and ResNet in various multi-task experiments to see how they compare. The actual models will be EfficientNet-B3 and ResNet-101 since they require a roughly equivalent amount of memory to train.

6.1 Training setup

All the experiments use the same computer, and the model training is done using an Nvidia GeForce RTX 2070 GPU that has 8 GB of GDDR6 VRAM (*Introducing NVIDIA GeForce RTX 2070 Graphics Card* 2020). Thus all the models will have to be small enough to train on this relatively small amount of memory. The deep learning framework that is used to implement the models in all the experiments will be PyTorch (Paszke et al., 2019) that is used inside Docker (*Empowering App Development for Developers — Docker* 2020) containers. Also, Nvidia Apex (*NVIDIA/apex* 2020) will be used to train models using mixed precision floating-point numbers to reduce the size of the models and to increase the speed of floating-point operations. The model weights will be 16-bit floating-point numbers where it is safe to use the lower precision representation. Besides the performance improvement, using lower precision floating-points can act as a regularizer as the model can't overfit on the high precision values and thus improve model accuracy as well (Micikevicius et al., 2018).

As a default for each task, we will use the following default procedures unless otherwise stated. For the Multi-task training process, we will sample the data sets for each task

with a probability respective to its size. The loss function for every task has the same weight multiplier of one. The batch size for each task will be maximized to fit within the memory requirement, giving a batch size of 32. For weight optimizer, we will use SGD with a cyclic learning rate scheduler (Smith, 2017).

6.2 Training loop

The base training loop for a multi-task training problem is relatively similar to that of a single-task model. Algorithm 1 shows the alterations that are needed for the training loop. The main difference is that instead of a single data loader, batch size, and loss function, there needs to be one of each for each task that is trained. Besides these, some values for the sampling ratios and possibly some values for loss weights are needed. The loss weights can be omitted if there is no need to diverge from having a constant 1 factor for each of the losses. One completely new part of the loop is the sampling of the task according to the ratios provided. All the parts of the standard training loop can be chosen from the provided lists of parameters, and the training is as in a single-task model.

Algorithm 1 Basic multi-task training loop

Input: Epochs e, Data loaders $data\_loaders$, Sample probabilities α, Batch sizes B, Training sets X, Model $model$, Loss functions $loss\_funcs$, Loss weights w

$batches \leftarrow \sum_i \frac{|X_i|\alpha_i}{B_i}$

for $epoch \in \{1, \ldots, e\}$ **do**
  for $i \in \{1, \ldots, batches\}$ **do**
    task_id $\leftarrow$ sample(α)
    (x, y) $\leftarrow$ next(data_loaders[task_id])
    y_hat $\leftarrow$ model(x, task_id)
    loss $\leftarrow$ loss_funcs[task_id](y, y_hat) $\times$ $w[task\_id]$
    loss.backward()
  end
end

The provided algorithm assumes that at a time, we are interested only in the labels of a single task per image. If there is a need to train multiple tasks from a single input, then the minor adjustment that is needed is to sum the losses of multiple tasks that are

backpropagated. Otherwise, the training loop remains the same, even if multiple tasks are optimized from a single batch.

6.3 Multiple object classification tasks

As we have previously covered, fine-tuning an ImageNet classifier often produces good results. Here we will take a pre-trained ImageNet backbone and learn multiple object classification tasks using the shared image embedding. First, we will use two datasets that both contain images of healthy and unhealthy plants.

Figure 6.1: Example images from the citrus and ibean data sets. The images in the top row are from ibean, and the bottom row is from the citrus data set.

The ibean dataset (Lab, 2020) contains 1296 images classified into three classes; Healthy, Angular Leaf Spot, and Bean Rust. The data set provides a split for the images that will be the split that is used for the experiments. The citrus leaves dataset (Rauf et al., 2019), on the other hand, contains 759 examples of citrus fruits and leaves with six different classes; Black Spot, Canker, Greening, Scab, Melanose, and Healthy. This set is quite tricky as it contains 609 images of leaves and only 150 images of actual citrus fruits, and some of the classes have very few images. Some examples from the data sets are in Figure 6.1. For the training split, the data will be split in 65/10/25 ratio for the training, validation, and test sets, respectively. The goal of this kind of split is to have some relatively reliable test results by having proportionally larger than regular test set since there is so little data. Both of these datasets have a relatively small amount of images available for training, so it should serve as a good basis for trying to train multi-task classifiers and give a glimpse into the adjustments that need to be made when training multi-task classifiers. The goal for this experiment is to get familiar with the basics of how to create the multi-task models, and how to do the actual changes to the training loop in the Pytorch code to accommodate for the multi-task training.

When the two closely related datasets are used to train the model, we could expect the model to generalize better on both of them, as hopefully similar features would benefit both tasks. But as the tasks are quite simple, the benefits can be relatively small as the likely accuracy for the single-task models should be quite high. In the case of the citrus data set, learning to classify the citrus fruits could prove quite difficult, and for that, the bean leaves likely won't be of much help. This model is depicted in figure 6.2.

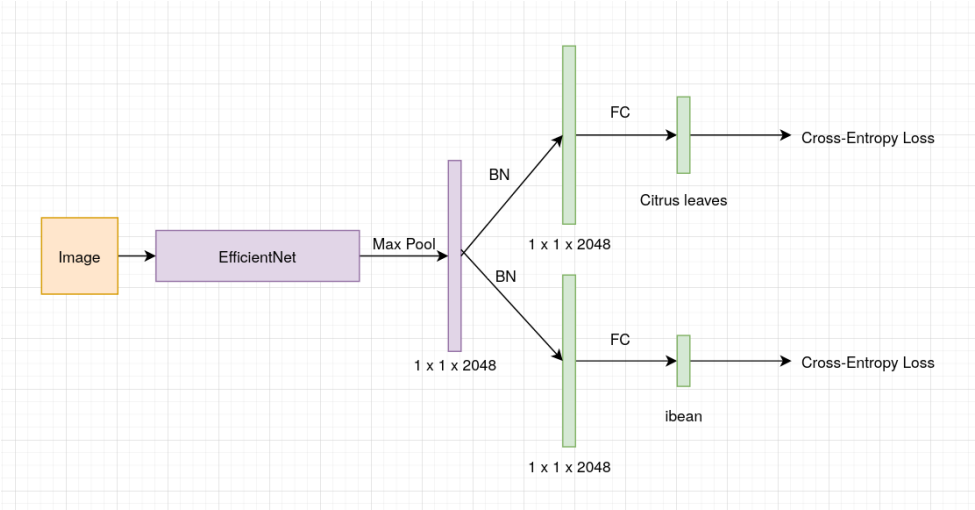


Figure 6.2: Network architecture for two image classification tasks.

To further extend the generalizability, we will try to add a third somewhat similar classification task using the Oxford flowers102 dataset (Nilsback and Zisserman, 2008), which contains 102 different types of flowers for classification. The assumption here would be that flowers could require similar features as the previous leaf classification tasks, thus when increasing the training data with a closely related task, we could increase the generalizability even further.

Multi-task and single-task models were created and evaluated for both EfficientNet and ResNet backbones. For the single-task performance, in this case, the ResNet model did slightly better on both of the tasks achieving 96% test accuracy on the beans dataset and 88% test accuracy on the citrus dataset, and the EfficientNet model achieved 94% and 84% test accuracies on the tasks. In terms of training, the most significant parameter is the sampling ratio of the two tasks.

The sampling was done with a random number to pick which data loader would be used for the current batch. If the sampling is done by going through each of the datasets entirely one at a time, the model ends up at a point where only one of the tasks has good accuracy. If the ratio is left as proportional to the data set size, the multi-task model ends

	ibeans	citrus	flowers
Single-task models			-
EfficientNet	94%	-	-
EfficientNet	-	84%	-
EfficientNet	-	-	97%
ResNet	96%	-	-
ResNet	-	86%	-
ResNet	-	-	97%
Multi-task models			-
EfficientNet	94%	88%	-
ResNet	96%	89%	-
ResNet	96%	91%	97%
ResNet	96%	-	97%
ResNet	-	85%	95%

Table 6.1: Accuracies on the data sets for both single-task and multi-task models with ResNet and EfficientNet.

up only learning to classify the ibean data set, and the accuracy for the citrus data will stay low. After some experimentation, the suitable probabilities for achieving good results with the EfficientNet multi-task classifier were 0.66 for the citrus and 0.33 for the ibean data. Interestingly, the same ratio does not directly transfer to the ResNet model. When we used the same ratio as with the EfficientNet, the model still seems to converge only on a good result for the ibean data set. For the final results with the ResNet classifier, the sampling probability for the citrus dataset needed to be raised, and the value we ended up getting the best results with was 0.73.

Training both tasks in a multi-task setting, we found that the accuracy on the citrus data set was increased, and ibean accuracy stayed the same. Specifically, the accuracy for the EfficientNet classifier went up to 88% and for ResNet up to 89%. The final model evaluated on the test set was the model with the highest average validation accuracy over the two tasks when trained for 25 epochs. All the results are displayed in table 6.1. The results show that even though these tasks were relatively simple and the accuracies for the single-task models were high, there was a benefit to be gained from multi-task training. Here we mainly focused on getting good results on a fully shared backbone, and more experiments could be done by having fewer parameters shared between the

tasks. As an introduction to multi-task learning, the experiments show that picking the correct sampling ratio is extremely important for achieving desired results. In that way, the sampling ratio seems quite unlike learning rate in that picking too small learning rate may lead to slow convergence but will most likely produce results in the end. With the sampling ratio, it seems that the correct value needs to be found by experimentation, as a too low ratio for the citrus data gives accuracies that can be as low as 60%.

Based on the results from the multi-task results on two tasks, we tried to add the third flower recognition task. The flower recognition task has about 10 000 images and 102 classes, whereas the other two tasks have significantly less data and fewer classes. On the two task experiment, it seemed that the sampling ratio should be relative to the task difficulty. Here it would seem that the 102 class flower recognition should be much more difficult than the other two tasks, and we tried using sampling ratios in the range of about 0.5 to 0.7 for the flower task. The best result achieved with these ratios was multiple percentage points below the single-task performance on all tasks. However, when the flower task ratio was 0.1, it was possible to surpass the single-task results on all the tasks. This training approach took significantly longer to converge to results. The original sampling ratios converged in about five epochs to the best model, and the low flower ratio took about 50 epochs. I started the training using the sampling ratio of 0.1 and decided to increase it up to 0.2 by the end of the training process.

When the training the model using the lower flower sampling ratio, it quickly converges close to the single-task performance on the two smaller tasks and slowly learns the flower task. It is hard to say exactly why such low sampling ratios seemed to work. Perhaps it has something to do with the difficulty of overfitting on a single task in a multi-task setting even though the small training set is seen so many times. So the features for the smaller data sets are not just memorizing the pixels of the training set, but rather the features that generalize well. Since the model already has good features for the smaller tasks and it is forced to find only a relatively minor shift in the parameters to learn the flower classification. And seeing as the other tasks are improved as well, it shows that some of the features of the flower data are useful in the easier tasks.

From the three task models, the final model was picked as the one having the best average validation accuracy on all the tasks. The final model wasn't just this model directly, though. To get the final model, we froze all the shared weights and then did fine-tuning similar to what one might do when training a transfer learning classifier. Having the model train on all the tasks using the frozen representations seemed to be useful. The

fine-tuning may be a good idea since, while training, the shared features are constantly shifting, and when one task is trained, it shifts the weights according to its gradients. The other tasks won't have adjusted to the new, slightly shifted shared weights, which can lead to incorrect classifications. When the shared weights are frozen, each of the task heads gets to learn the exact mapping from the shared representation to the outputs without them shifting during training.

All in all, these experiments validated the assumptions made about the multi-task training quite well. Still, finding the sampling ratio is very laborious and not very intuitive. This is why the results for the citrus and flower pair are worse since we spent less time finding the optimal ratios and mainly verified that it roughly works as expected. Perhaps it could be automated to get somewhat decent indications of what should be tried. The problem, in the end, is that even if some kind of grid search could be applied, it would take very long to predict the final metrics of the models, and as seen in the discrepancy of the flower ratios, it could take many epochs to get the final results. Perhaps some kind of dynamic sampling ratio, where the task lowest relative to the expected accuracy is sampled more, could be beneficial.

6.4 Multi-label classification

For multi-label prediction, there can exist a problem where not all classes are restricted to only their data sets. For example, a photo of a dog taken outdoors can contain other classes such as a person, sky, or sea. These may not be labeled for that set but could be classes of interest. If we train models on incorrect labels, it can lead to unlearning as the model may make correct predictions but is punished due to the absence of the labels for the images. However, if the problem is posed as a multi-task binary classification problem over several classes, each task can have its own distinct data set with labels for only the single task. Then upon training on an image containing other classes, there is no inverse learning as the image will not try to learn the other classes if there exists no certainty of that label. Still, the problem is not gone as often any dataset has at least some mislabeled examples.

We will run some experiments on a 14 class multi-label prediction with some overlap between the classes in the images. The 14 head multi-task model structure and the structure of the 14 identical single-task models are shown in Figure 6.3. The goal here would be to see how what kind of a difference there would be in training with a single-task

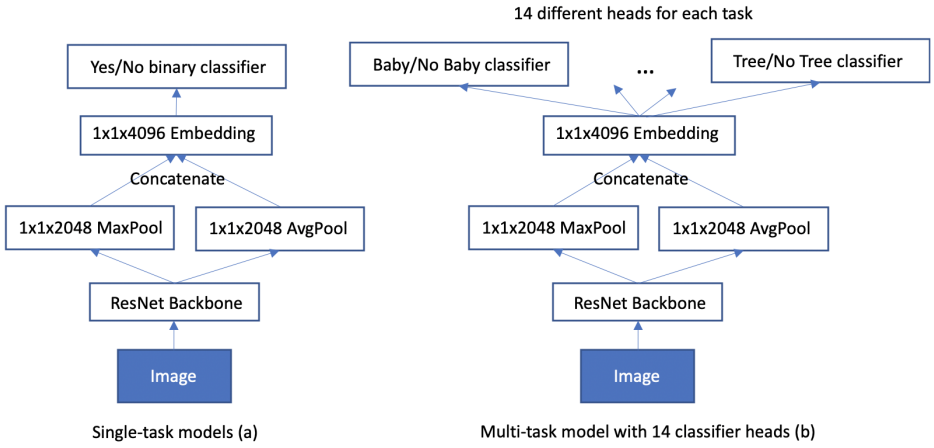


Figure 6.3: Architectures for both single-and multi-task models. The model (a) is the single-task model, which is the same for all 14 tasks. Model (b) is the Multi-task model, which contains 14 heads for each of the classification tasks. For these classifiers we took both max and average pool to create the embedding.

and multi-task model over all of the labels.

The dataset is a collection of 15000 images that have labels for some of the classes. The number of labels for each task is highly varied. The smallest tasks have dozens of images, whereas the largest ones have thousands. From this dataset, we will take for each class a 15% cut for validating the performance. In this case, we will pose our main problem as a binary classification problem. For each of the tasks, we will train a classifier telling if the label exists in the image or not. The classifier will use a ResNet-101 as a backbone, and the head will be a regular fully connected layer connected to a positive and negative class for each of the tasks. For the single-task models, we will have the same ResNet backbone as for the multi-task model. The main metric that we will compare will be the ratio of correct labelings per class. Since we pose this as a binary classification task, it is easy to guarantee that our test sets have an exact 50/50 split for positive and negative classes. Similarly, while training, we can oversample the positive classes in a way to guarantee that each batch will have half positive and half negative classes. This way, we won't learn classifiers that are biased toward predicting the absence of the label due to the small number of images for that class. The control over the sampling of the data is a very beneficial trait of the multi-task problem setting.

To train the multi-task model, we experimented on various sampling ratios to find the best

	Multi-task	**Single-task**
People	90%	89%
Man	87%	84%
Woman	86%	87%
Baby	87%	93%
Sea	89%	96%
River	89%	88%
Bird	88%	89%
Car	85%	94%
Clouds	93%	93%
Dog	87%	93%
Flower	85%	93%
Night	87%	92%
Portrait	89%	91%
Tree	87%	92%
Average	88%	91%

Table 6.2: Accuracies on the data sets for both single-task and multi-task models. The accuracy is the proportion for correct labelings in the balance evaluation set, so pure guess would result in 50% accuracy.

model. Manually giving a meaningful sampling ratio is quite difficult due to the relatively large number of tasks in this multi-task model. The various sampling ratios we tested for this data were equal sampling, data proportional sampling, and finally, some manual adjustments to the proportional sampling based on the results. The proportional sampling was the one that produced the final best model. It is quite an interesting result, considering just how small some of the classes are relative to the entire data set. We tried adjusting the proportional ratios based on the difference from the single-task accuracies but could not find ratios that would have improved the performance. Based on the previous results, we could say that there likely exists some ratios that could provide superior results, but finding them is too expensive. With the best model, which was chosen based on the average accuracy on the validation data, the model achieved an average of 88% accuracy in labeling all the tasks. By average, we mean taking the accuracy of each task separately and averaging those averages and not the total ratio of correct classifications.

One new thing that we noticed when training on this data set was that with too high learning rates, the experiments were not very repeatable. It seems that in a multi-task setting, it is not easy to tell that the learning rate is too high. In this case, when the learning process seemed to progress in a manner that we would call expected, the learning rate might actually be too high. With a slightly too high learning rate, it seemed like it is relatively random, whether the model reaches the best weights. Most likely, the sampling ratios matter here quite a lot, but with the same sampling ratios and a slightly too high learning rate, there didn't seem to be a guarantee that anything close to the best model would be reached. Maybe this has something to do with the large variance in the size of the datasets. With more dynamic sampling ratios, the problems might solve themselves, as then some tasks would hopefully not get stuck at low accuracies. We noticed here the same thing as in the leaves experiment that a long training process that slowly approaches the optimum seems to produce the best results in the end consistently. Maybe this means that finding the intersection that can solve all the tasks optimally is relatively small and easy to overshoot with too large learning rates. Similarly, with incorrect sampling ratios, the model would never be able to reach the weights needed for the shared optimal solution. To deal with the imbalanced classes, we also tried to add an exponent to the loss like in the focal loss, though it is unsure if it was useful or not. But in theory, it should allow for the network to learn more from the cases where it is unsure. Also, it would be yet another parameter to tune, as it is hard to say what exactly the exponent should be for the best results.

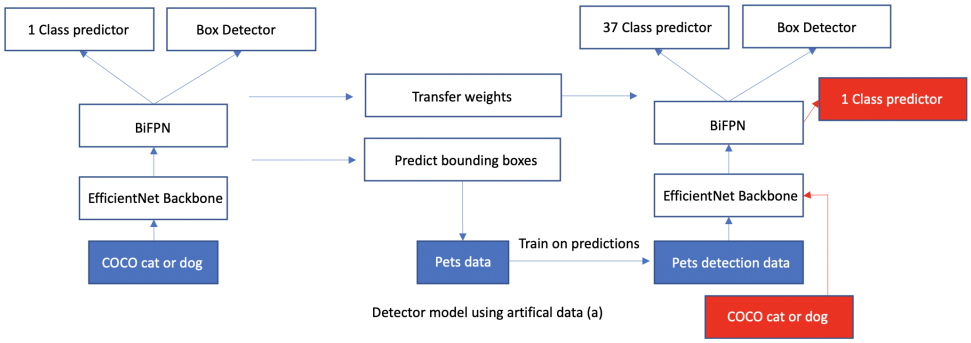
The single-task model training was done similarly to the multi-task model and by using the same data loading techniques. This way, we reached an average accuracy of 91%, beating the multi-task model slightly. All the task-specific results for both multi- and single-task models are collected in Table 6.2. Some part of the errors is due to the misclassification of the data, based on a random sample, around 2% of the labels are wrong. These results yet again show that it is possible to learn different tasks using the same representations. Yes, in this case, the multi-task accuracy didn't match the single-task counterparts, but they are relatively close. And all this is not really a fair comparison as the single-task models account for a total of around 14 times as many parameters. If the single-task models were trained on significantly smaller backbones, the multi-task model could be more accurate, showing the benefit of using larger models over multiple tasks. And even picking the smallest ResNets, the number of parameters over 14 tasks would be larger than a single ResNet101.

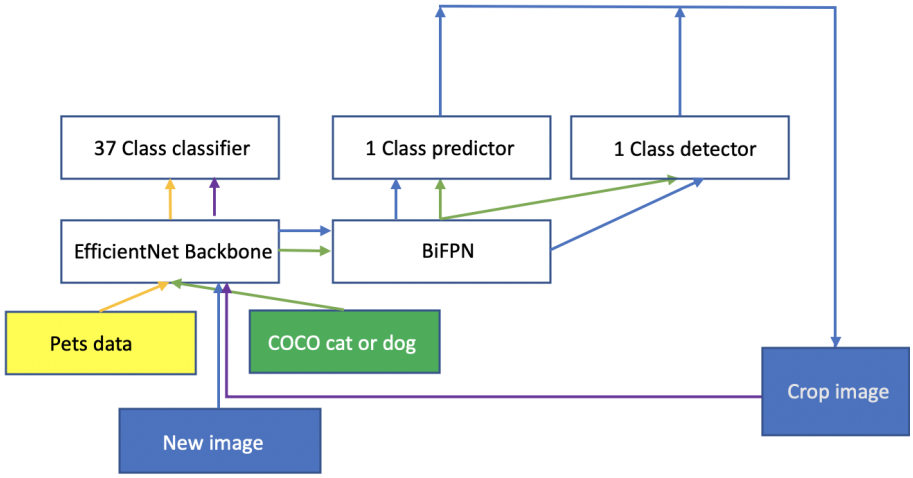
6.5 Object detection and image classification

In this section, we will use multiple data sources and combine them into a single powerful object detector. To be specific, our goal is to create an object detector for 37 different cat and dog species. The two datasets that we will use are the COCO dataset (T.-Y. Lin, Maire, et al., 2014) from which we will take the cat and dog classes and the Oxford-IIT Pet Dataset (Parkhi et al., 2012), which comprises of images with labels for the 37 different cat and dog species that we will detect. In these experiments, the backbone used will be EfficientNet-0, which will be used in the EfficientDet architecture as well as an independent EfficientNet backbone. The base for the code of the detector uses this Pytorch implementation of EfficientDet (zylo117, 2020). The base model is then modified to meet the needs of our experiments.

There are multiple ways to approach this problem. We will train two different models to see how the results differ. The first approach is to do a brute-force two-stage detector, where we share the common backbone. In this approach, the detector would try to learn to detect the dog and cat classes from the COCO dataset, while an object classifier is trained using the same backbone as the detector. The object classifier is then responsible for giving the actual class of interest. This way, the training is relatively simple, and the backbone would be used recursively. Specifically, the detector would generate the regions for the animals that would be cropped and pushed through the backbone, this time to the

classifier head. This recursive use of the backbone is efficient in terms of the parameters, but running images through the same backbone twice could turn out to be too expensive as it would double the inference time.





b) Two-stage recursive classifier with separate classification and detection

Figure 6.4: EfficientDet models for solving the object detection problem. In a), we have the basic object detector. We show how we train on the COCO data first and then transfer the weights to the model for the pets data with predicted bounding boxes. In red the additions needed to train on both the original data and artificially labeled are shown. Both tasks could share the weights for the detector but would need to have their own class head. In b), we have the recursive object detector. In yellow, we describe the path to train on the pets data. With green, we describe the path to train the cat or dog object detector. In blue, we show the first cycle of a new prediction. With purple, we show the second cycle for the new detection.

The second way of approaching this problem is to first train a model on the COCO classes

and to create a detector for the cat and dog classes. Using the high-level model, we can automatically create acceptable object detection labels for the 37 classes in the pets dataset. Then we can initialize a 37 class object detector using the weights of the two-class model. By having such a well-initialized model for the detection, the second part of the training focuses on learning the features to differentiate between the very similar classes. An interesting question here is just how well the object detector works when the classes are so similar. In an image classifier setting, it would make sense that the model could learn the subtle features that are unique to each of the classes. As we previously covered, the object detector is a multi-task problem. It follows that the network has to both decide the anchors most likely containing an object and also determining the class of the object. If we look at the classes for the COCO dataset for object detection, for example, we can see that there are 80 classes, but they are relatively difficult to mix up due to being quite distinct from one another. In Figure 6.5, we have an example of two images from the pets dataset showing two classes that would be relatively easy to mix up.

Figure 6.5: Example of two very similar classes in the pets dataset. On the left is an American bulldog and on the right is an American pit bull terrier.

To evaluate the effects of both approaches, we will look at multiple metrics. Since the two approaches are quite different, we can't make an entirely fair direct comparison between them. To get somewhat of an idea for how well the models detect our 37 different classes, we will use a slightly arbitrary metric. We will measure the accuracy of the detector on our test set for the pets dataset. A positive example is one where the detector gives the highest confidence to the correct class. This is a somewhat simplified version of measuring the detector performance. The baseline scores for our EfficientNet-0 model are 92% accuracy on the pets dataset and a 0.55 AP on detecting the cat and dog classes in the COCO

dataset. We will calculate these same scores for the recursive model to see how close the multi-task model can get. To compare the two approaches, we will calculate the accuracy of correctly detected species in a set of the pets dataset. The labels are not going to be perfect as they are generated by the model automatically, but should show if there are major discrepancies in the performance between the two approaches.

To train the single-stage model, we first created a new dataset in the COCO format from the dog and cat classes. Instead of having two classes, we combined both dog and cat into a single category as the goal is to detect them as well as possible, and we are not interested in identifying whether it is a dog or a cat. It is important to set the category id to be one instead of zero when training the object detector when there is only one class since otherwise, there will not be any predictions for the background. With this new dataset, we initialized the EfficientDet model and trained it to detect the single class from the COCO dataset. We first fine-tuned the head for ten epochs and then fine-tuned the entire network for 40 more epochs, and we picked the model with the lowest validation loss as the best one. The chosen model achieved 0.54 AP when tested. Then we created a new COCO style dataset out of the pets dataset. To get the annotations, we ran the images through our one-class object detector and labeled all found boxes with the class of the image from the pets dataset. The COCO format requires the labels as a tuple of four numbers signifying the x and y coordinates of the top left of the bounding box, followed by the bounding box width and the height. Since most of the images in the pets dataset contain only a single animal, we could tell that there were incorrect boxes as the number of annotations was about 10-15% higher than the number of images. The bounding boxes were most likely not perfect, but good enough to get decent results. We initialized the model with the weights of the original object detector, which meant that the regression loss for the model was consistently low. So the training mostly focused on learning to minimize the classification loss with the new 37 classes. Since this was a much more difficult problem than the original, it took a bit longer to reach the best model. Evaluating the model on the artificial labels of the validation set gave an AP score of 0.55.

There are a few ways this kind of training could be extended to possibly get even better results. We could attempt to train the largest possible model to get the best possible results on the original dataset so that the artificial labels for the unlabeled dataset would be the best possible. We could create two heads in the object detector to train the detector on both the original COCO images and the new images in a multi-task setting to hopefully get a more general detector. The training could get better if multiple cycles

Pets classification	
EfficientNet-B0 baseline	93%
Recursive detector task head	89%

Table 6.3: This table lists the accuracy for pets classification without localisation based on the baseline model and the classifier head in the recursive model.

Cat or dog detection	
EfficientDet-D0 baseline	0.54
Recursive detector	0.52

Table 6.4: This contains the AP score for detecting the single cat or dog class for the baseline and the recursive detector.

were completed, and the training would work in a semi-supervised way. So what we used here would be the first labels for the new dataset. The training would be done jointly on both datasets, as previously proposed. Every n epochs, we would generate new bounding boxes for the pets dataset, to finally reach the optimal bounding boxes for the classes in the pets dataset. Finally, we could do the label transfer also in the other direction. Since the COCO has the correct bounding boxes, we could run the image classifier trained on the pets dataset on those boxes to generate the actual animal species for them.

The recursive model heavily relies on sharing the EfficientNet backbone in the detector between the classifier head and the detector head. The model is recursive because, on the first pass, we will find the cat or dog regions from the model using the detector head. Thus, the detector head is predicting only a single class as the distinction between the cat or dog class is of no interest in the detector. Given these detected objects, we can crop them from the original image and pass through the other head, which is concerned with classifying the image into the 37 classes of interest.

For the recursive two-phase model, we trained it in a very similar fashion to the previous multi-task models. In this case, we ran the experiment with only a few sampling ratios, of which a 50/50 split produced the best results. The object detector only required adding the final classification head on top of the backbone, and then the model could be trained with a basic multi-task training loop since the EfficientNet backbone that we wanted to use is already a part of the EfficientDet model.

To compare the models, we have collected the various metrics in Tables 6.3 to 6.5, grouped

Custom pets detection metric	
EfficientDet-D0 trained with artificial data	85%
Recursive detector with separate heads	91%

Table 6.5: This table compares the EfficientDet trained on artificial data and the recursive model in detecting the 37 pets classes. Here the metric is the proportion of correct classifications in the most confidently detected box.

37 class object detection	
EfficientDet-D0 artificial data 37 classes AP	0.55

Table 6.6: The AP score on 37 class object detection for the model with artificial data.

by the comparable metrics. The pure object detector for 37 classes had a 0.55 AP score on the artificially labeled dataset of the pets, and this is mildly higher than the base AP for the cat or dog detector. While the number seems impressive given the number of highly similar classes, it is not exactly comparable to the baseline due to the fact that the pets dataset is not that varied in terms of having many or different sized animals. Furthermore, the bounding box labels are created by the initial state of the 37 object detector, which likely causes the accuracy for the bounding boxes is going to be very high. The regression loss of localizing the boxes stays very low and stable throughout all of the training, showing that likely the detection part does not change that much. In our artificial detection evaluation metric, this pure detector achieved an 85% accuracy. The accuracy is relatively high, given the difficulty of detecting and classifying such similar classes.

The recursive multi-task model reached again slightly lower accuracies than the single task counterparts. The AP score on the cat or dog detection task dropped by 0.02, and the accuracy on the pets classification dropped by 4% to 89%. Interestingly, the detector task had a higher accuracy than the classification head had on the pets test set. The better accuracy could result from the fact that classifying a cropped animal is easier than to classify it within a larger frame. And since the object detector is relatively decent at finding the cat or dog candidates as a combination, these achieve better results.

Based on the numbers, we could say that the two-stage detector seems to be the better one. The main downside to doing the two-stage detection is the fact that it would have to work one frame behind the video. Since on the first loop, we would detect our objects in the first frame. On the second frame, we would pass the second frame and the cropped

images that we found in the first frame. If the classification is done in this manner, the inference time cost won't be that much as it will be just the head to get the classification outputs. In the case that multiple videos would be run in parallel, this could possibly not be possible as at some point, the input could be too large to handle as a single batch. On the other hand, the pure detector doesn't need such considerations. The pure detector's problem is that it has some obvious downsides due to the artificial training set that was created for it. Whereas the recursive model does relatively as well on all kinds of data, the pure detector seems to have issues when there are multiple objects to detect, or the size varies. These issues likely stem from the fact that the pets dataset consists mostly of single relatively large animals, which is not always the case. Because the training and test sets are from the same source, they are relatively similar, so the performance is very transferable between those tasks. Some of the previously mentioned ways of training could alleviate this problem. Perhaps by having two classifier heads and doing auxiliary training on the COCO cats and dogs could make the detector detect the animals better, and hopefully, the classifier could solve the problem for all sizes from the single source. This modification is also pictured in Figure 6.4.

6.6 Detection and segmentation with EfficientDet

In the EfficientDet paper (Tan, Pang, et al., 2020), the authors point out that it is relatively simple to use the same EfficientNet backbone and BiFPN neck to do image segmentation as well. Here we will attempt to train such a model but in a multi-task setting. Our goal is to use the Kitti (Fritsch et al., 2013) road segmentation data set to train a segmentation model, and the COCO dataset to train an object detector. Since they both should be trainable on the same backbone and neck, we will see how good of a multi-task model we can create by sharing the entire detector.

Since we have not talked about image segmentation, we will briefly cover what it is about. Image segmentation is also a very popular and important task with applications in many domains. The goal for an image segmentation model is to determine the meaning of an image on a per-pixel level. For example, in our case, we are going to train a model on the road segmentation data where the goal is to predict where the current lane goes, so we would like to know where the pixels that represent the current lane the car is on are. So the output is a mask telling what class each pixel in the image belongs to. An example image and its segmentation mask are shown in Figure 6.6.

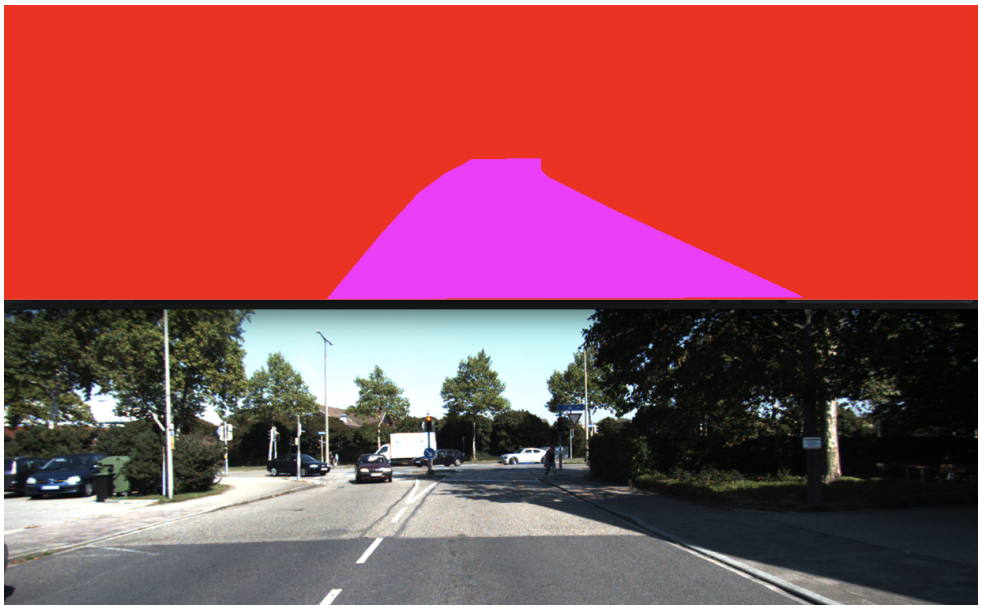


Figure 6.6: An example of the annotations and an input image from the Kitti (Fritsch et al., 2013) road segmentation data.

Here our goal is not to create a very accurate classifier but more to evaluate how difficult it is to take two very special problems and solve them by sharing as much as possible. To create the model, we could use a U-Net (Ronneberger et al., 2015), which is an architecture specially designed for image segmentation. We could adapt the U-Net architecture to EfficientNet and easily fit it with our backbone. However, by using the EfficientDet, we get a chance to share more, and it is a much more advanced way of doing image segmentation, producing better results (Tan, Pang, et al., 2020).

There is no complete implementation of the EfficientDet image segmentation head, and the authors don't really describe it too much in the paper. To create the model, we took the EfficientDet detector as a base and augmented it with a segmentation head. The segmentation head was based on the implementations of BiFPN segmentation from (Yakubovskiy, 2020). After creating multiple multi-task models, the creation and modification of the training loop were no longer so difficult to adjust to the new task. Figuring out exactly how to add the new task into the existing model architecture took a while to find the exact values that the segmentation model needs for creating outputs. After configuring data loaders and the model, the training was pretty much along the lines of the previous models we have covered. The results for our models were along the lines that we have seen previously, the multi-task model being a few points below the single-task models. It could be possible here that the segmentation model could get better if it had its

own BiFPN network and could only share the backbone with the detector, but we did not have enough time to run all possible experiments. Our modified EfficientDet architecture is shown in Figure 6.7.

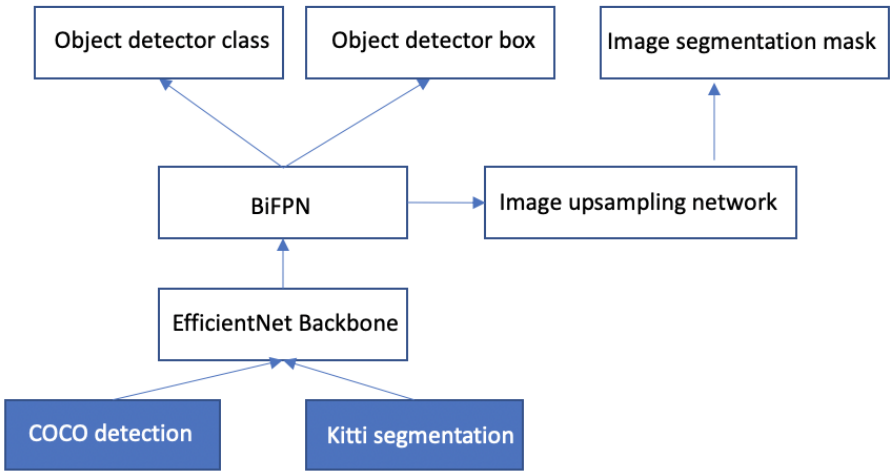


Figure 6.7: EfficientDet model with segmentation and detection heads.

With the segmentation head and object detector head configured, it would now be easy to extend this model further. We could easily add a new image classification head using the same backbone, like (Teichmann et al., 2018), where a single model is used for road segmentation, car detection, and road type classification. Adding more classes to the segmentation or to the object classifier would be relatively easy to do. Of course, there is a lot of sharing optimization to be done as well. In our implementation, we naively shared everything, which was likely relatively detrimental to performance. It would make sense to at least have the BiFPN be task-specific. And since everything is now configured in the same model, it would be relatively easy to chop half of the backbone for each task and possibly get better results. As we have covered over and over again, the space for possible improvements is huge.

6.7 Overview of the experiments

Based on our experiments, we can say that multi-task learning is hard when compared to training single-task models. Training multiple single-task models is easier and faster and requires less user input and parameter optimization. The effect of many parameters, like

the learning rate, can be evaluated relatively easily based on a few epochs in a single task setting. Perhaps to find the very best model, more experimentation and exploration is needed, but just some default values can generally provide good results.

For someone with little experience in training multi-task models, it can be difficult to get started. Whereas it is easy to find multiple examples of just how a training loop should be created, finding such examples for multi-task models is much more difficult. For experimenting with different ways of splitting the features into local and global features, it is important to have systems for easily doing experiments on different architectures. In our experiments, we created four points where it was possible to cut the last block in the ResNet, for example, so it was easy to create models that each had none to some local features. Devising ways to split on a more fine-grained level and grouping tasks when there are multiple would be extremely important when searching for the optimal architecture. Manually changing how the weights are shared would not be enough. In our experiments, this was also quite clear, as we did not experiment beyond having models that have some or none local features. We did not do any experiments where we would have grouped the features in a more well thought out way. Likely, most benefits are gained when multiple tasks share half of the networks instead of only having a head be local. This would make sense as the lower level features are more general and likely there would not be as many conflicts as in the deeper layers. The number of things to tune is also quite discouraging when the model does not train properly. Finding out which part of the modifications to the default process is wrong can be extremely difficult. Some of the mistakes we encountered were difficult and frustrating to solve. For example, we forgot to do a deep copy of the local features and just passed the same features as references to all tasks and wondered why the training of a single task was impacting others so much. Or when we froze all local features except the head when we meant only to freeze all the global features. Or mixing up the order of the tasks in the model task heads, so the final inferences ended up being complete garbage.

The impact of the various parameters that have to be picked seemed to be much larger. With an incorrect sampling ratio, the models could be extremely far from the good sampling ratios. It seems that finding the perfect sampling ratio might not be possible, and settling for a good enough one might be a good idea. It seems that it may be a good idea to pick a lower learning rate than normal to find the best intersection of features.

Finally, the model structure's changes to create the various classifiers' heads can be difficult to add. While adding multiple image classifiers is relatively trivial to do if one is

comfortable doing basic model definitions, it can be difficult to combine the various task heads for the more complex tasks. For example, object detector models are generally provided with very specific training loops, so getting an object detector that has a classifier head is much more difficult than just training the two separately. Without some easy way of abstracting the classifier heads and automatically injecting them into the training loop, it may not be possible for some to create multi-task models without extensive help. We considered relatively simple tasks that already had open-source implementations and mostly combined them into a single model. When dealing with some more advanced models like some of the ones presented in the multi-tasking chapter, it would be even more difficult to create the models.

Despite all the challenges, we did manage to create models that came close or were equal to their single-task counterparts. The multi-task models were smaller and faster in those cases, confirming that using the shared representations can be an excellent idea.

7 Conclusions

In this thesis, we have given an overview of fundamental computer vision tasks of image classification and object detection. We described the importance of the ImageNet data set and some of the most significant image classification architectures based on it. The importance of these models extend beyond just solving the ImageNet dataset, and we covered how it is possible to apply the trained models in new classification problems with transfer learning.

For object detection, we described how the problem and datasets differ from image classification ones. We also described the different parts of object detectors and how single and two-stage detectors differ from one another and when one might be preferred over the other. The object detectors also show how important the ImageNet backbones are, as they are an integral part of the object detection architectures also. The focal loss is an important loss function used by modern single-stage object detectors in allowing a large number of anchor points for localizing the objects in the images. Finally, we described one of the current state-of-the-art object detection architectures, the EfficientDet, and how it can be trained on a new problem using transfer learning like in the case of image classification.

Here we have also presented multi-task learning as a generalization of transfer learning. In a multi-task setting, the goal is to solve multiple tasks using some shared representation. It differs from transfer learning in that all tasks are optimized continuously, rather than just serving as a starting point. Doing transfer learning is an excellent way of obtaining good results, even on little data. With multi-task training, we can utilize the common model architectures as shared parameters for multiple tasks to allow the model to generalize better and run more efficiently. This is possible because the model is required to use the same capacity to solve multiple problems that may require different features, some of which can also be useful in other tasks. Still, sometimes the multi-task setting can be detrimental to accuracy. Despite the differences in performance, utilizing shared representations means that there are fewer parameters that the model needs to learn, leading to smaller models and shorter inference times.

We saw multiple examples where multi-task learning played an important part in getting models that are efficient and performant. One of these was the object detection, where the

model tries to classify and localize the objects. Both soft and hard parameter sharing for multi-task learning were covered, with some specialized architectures taking advantage of the fact that multiple tasks are solved at the same time. Finally, we found that multiple tasks can serve as auxiliary targets for training to aid the model in finding important features.

A multi-task problem setting modifies the training process and introduces some new parameters that can be tuned when training models. Task-specific sampling ratio during the data sampling process and giving task-specific loss weights were the two new factors that need to be tuned when different tasks are combined. These weights need to be tuned appropriately for the model to be able to find the features that work for all tasks. Beyond just tuning parameters, multi-task models may need some architectural fine-tuning in deciding which tasks should have shared representations and just how much should be shared.

The training process differences due to the multi-task problem setting were evaluated through multiple experiments on combining various datasets into a single model. With these experiments, we empirically verified some of the assumptions of training the models in a multi-task setting. We noted that it was difficult to find the very best models as different sampling ratios resulted in very different final models. As the model gets larger and the number of tasks increases, finding the correct ratios gets ever more difficult. After adding new tasks, the previous ratios might also need adjusting. Also, we discussed the difficulties of modifying the existing models to contain our desired output heads. This was most apparent in the case of object detectors. The supplied code is relatively prepackaged to run on the specified training loop, and modifying these model definitions to add new tasks takes some effort. We saw some improvements and reductions in the model performance, but in general, the multi-task models were close to the single-task counterparts so that reducing the model size might be a good trade-off.

In all of our experiments, we could have spent more time on them to do more trials trying over different parameters and combinations of parameters. Maybe by doing more thorough and systematic testing, it could be possible to evaluate exactly the effect of tuning a specific parameter. Doing all the experiments would take a long time, and it is questionable whether the results would generalize to all problems. The fact is that there are too many things that could be tuned, so it is difficult to determine which parameter should be tested.

Multi-task learning is a significant paradigm for creating large machine learning systems.

As it isn't really scalable, nor does it really make any sense to try to solve every task separately in a system that would be called intelligent. For example, Andrej Karpathy, the Director of AI at Tesla, has given multiple talks on how they apply multi-task learning and how it is an instrumental part of their convolutional models (Karpathy, 2019). The problems he describes their teams tackle are the same problems that we have encountered and tried to solve in this thesis. Like deciding what tasks should share and picking the correct sampling ratios are some of the problems Karpathy brings up as some of the most significant issues in training their models. It does make sense that a camera-based self-driving car is one of the largest current multi-task learning applications as it is a problem that benefits from all the boons of multi-task learning we have covered. It is important to have small models for running inference in constrained environments faster than single-task counterparts. By sharing weights between tasks that have been specially picked in a way that they can augment each other makes it easier to learn new tasks and improve performance on the existing ones. Having separate datasets for each task allows for the easy creation of new tasks in the model irrespective of others. The ability to modify the loss weights and task sampling ratios allows for focusing on the critical tasks that are not allowed to fail. So, while it likely is more difficult to train a model that solves all the tasks together, it is most likely unavoidable. Single-task models that would solve the same tasks with the same real-time requirements would likely have to be so small that they would be unusable. These are the kinds of problems where it is impossible to avoid using multi-task learning. With the correct techniques, it is possible to create systems that can reliably augment people's abilities.

In the end, what we learned is that the success of creating multi-task models out of some tasks can't be pre-determined currently. Some expensive experimentation is needed to arrive at the best multi-task architecture and to find the optimal parameters to train the model. Thus, multi-task learning can be considered as a trade-off of training time complexity for inference-time benefits in accuracy and speed.

Bibliography

Barbu, A., Mayo, D., Alverio, J., Luo, W., Wang, C., Gutfreund, D., Tenenbaum, J., and Katz, B. (2019). "ObjectNet: A large-scale bias-controlled dataset for pushing the limits of object recognition models". In: *Advances in Neural Information Processing Systems*, pp. 9448–9458.

Bochkovskiy, A., Wang, C.-Y., and Liao, H.-Y. M. (Apr. 2020). "YOLOv4: Optimal Speed and Accuracy of Object Detection". en. In: *arXiv:2004.10934 [cs, eess]*. arXiv: 2004.10934. URL: http://arxiv.org/abs/2004.10934.

Bodla, N., Singh, B., Chellappa, R., and Davis, L. S. (2017). "Soft-NMS — Improving Object Detection with One Line of Code". In: *2017 IEEE International Conference on Computer Vision (ICCV)*, pp. 5562–5570.

Cipolla, R., Gal, Y., and Kendall, A. (June 2018). "Multi-task Learning Using Uncertainty to Weigh Losses for Scene Geometry and Semantics". en. In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. Salt Lake City, UT, USA: IEEE, pp. 7482–7491. ISBN: 978-1-5386-6420-9. DOI: 10.1109/CVPR.2018.00781.

COCO - Common Objects in Context (2020). URL: http://cocodataset.org/#detection-eval (visited on 05/18/2020).

Dalal, N. and Triggs, B. (2005). "Histograms of Oriented Gradients for Human Detection". en. In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*. Vol. 1. San Diego, CA, USA: IEEE, pp. 886–893. ISBN: 978-0-7695-2372-9. DOI: 10.1109/CVPR.2005.177.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). "Imagenet: A large-scale hierarchical image database". In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee, pp. 248–255.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, pp. 4171–4186. DOI: 10.18653/v1/N19-1423.

Elsken, T., Metzen, J. H., and Hutter, F. (2019). "Neural Architecture Search". In: *Automated Machine Learning: Methods, Systems, Challenges*. Ed. by F. Hutter, L. Kotthoff,

and J. Vanschoren. The Springer Series on Challenges in Machine Learning. Cham: Springer International Publishing, pp. 63–77. ISBN: 978-3-030-05318-5. DOI: 10.1007/978-3-030-05318-5_3.

Empowering App Development for Developers — Docker (2020). en. Library Catalog: www.docker.com. URL: https://www.docker.com/ (visited on 03/25/2020).

Everingham, M., Van Gool, L., Williams, C. K. I., Winn, J., and Zisserman, A. (June 2010). "The Pascal Visual Object Classes (VOC) Challenge". In: *International Journal of Computer Vision* 88.2, pp. 303–338. ISSN: 0920-5691, 1573-1405. DOI: 10.1007/s11263-009-0275-4.

Faheema, A. and Rakshit, S. (2010). "Feature selection using bag-of-visual-words representation". In: *2010 IEEE 2nd International Advance Computing Conference (IACC)*, pp. 151–156.

Fritsch, J., Kuehnl, T., and Geiger, A. (2013). "A New Performance Measure and Evaluation Benchmark for Road Detection Algorithms". In: *International Conference on Intelligent Transportation Systems (ITSC)*.

Gong, T., Lee, T., Stephenson, C., Renduchintala, V., Padhy, S., Ndirango, A., Keskin, G., and Elibol, O. H. (2019). "A Comparison of Loss Weighting Strategies for Multi task Learning in Deep Neural Networks". In: *IEEE Access* 7, pp. 141627–141632. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2943604.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. http://www.deeplearningbook.org. MIT Press.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). "Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 37.9, pp. 1904–1916.

He, K., Girshick, R., and Dollár, P. (2019). "Rethinking imagenet pre-training". In: *Proceedings of the IEEE International Conference on Computer Vision*, pp. 4918–4927.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. (2017). "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". en. In: *arXiv:1704.04861 [cs]*. arXiv: 1704.04861. URL: http://arxiv.org/abs/1704.04861.

Huang, G., Liu, Z., Van Der Maaten, L., and Weinberger, K. Q. (2017). "Densely connected convolutional networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., and Chen, z. (2019). "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., pp. 103–112.

ImageNet Large Scale Visual Recognition Competition (ILSVRC) (2020). URL: http://www.image-net.org/challenges/LSVRC/ (visited on 01/26/2020).

Introducing NVIDIA GeForce RTX 2070 Graphics Card (2020). fi-fi. Library Catalog: www.nvidia.com. URL: https://www.nvidia.com/fi-fi/geforce/graphics-cards/rtx-2070/ (visited on 03/25/2020).

Karpathy, A. (2019). *Multi-Task Learning in the Wilderness*. SlidesLive. Library Catalog: slideslive.com. URL: https://slideslive.com/38917690/multitask-learning-in-the-wilderness?ref=og-meta-tags (visited on 05/31/2020).

Kocabas, M., Karagoz, S., and Akbas, E. (2018). "MultiPoseNet: Fast Multi-Person Pose Estimation Using Pose Residual Network". en. In: *Computer Vision – ECCV 2018*. Ed. by V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss. Vol. 11215. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, pp. 437–453. ISBN: 978-3-030-01251-9 978-3-030-01252-6. DOI: 10.1007/978-3-030-01252-6_26.

Kokkinos, I. (2017). "Ubernet: Training a Universal Convolutional Neural Network for Low-, Mid-, and High-Level Vision Using Diverse Datasets and Limited Memory". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6129–6138.

Kornblith, S., Shlens, J., and Le, Q. V. (2019). "Do better imagenet models transfer better?" In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2661–2671.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2017). "ImageNet classification with deep convolutional neural networks". en. In: *Communications of the ACM* 60.6, pp. 84–90. ISSN: 00010782. DOI: 10.1145/3065386.

Lab, M. A. (2020). *Bean disease dataset*. URL: https://github.com/AI-Lab-Makerere/ibean/.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11, pp. 2278–2324.

Lee, J., Won, T., and Hong, K. (2020). "Compounding the Performance Improvements of Assembled Techniques in a Convolutional Neural Network". en. In: *arXiv:2001.06268 [cs]*. arXiv: 2001.06268. URL: http://arxiv.org/abs/2001.06268.

Lin, T., Goyal, P., Girshick, R., He, K., and Dollár, P. (2017). "Focal Loss for Dense Object Detection". In: *2017 IEEE International Conference on Computer Vision (ICCV)*, pp. 2999–3007.

Lin, T.-Y., Dollár, P., Girshick, R. B., He, K., Hariharan, B., and Belongie, S. J. (2017). "Feature Pyramid Networks for Object Detection." In: *CVPR*. IEEE Computer Society, pp. 936–944. ISBN: 978-1-5386-0457-1.

Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollar, P., and Zitnick, L. (2014). "Microsoft COCO: Common Objects in Context". In: *ECCV*. European Conference on Computer Vision.

Liu, S., Qi, L., Qin, H., Shi, J., and Jia, J. (2018). "Path Aggregation Network for Instance Segmentation". In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 8759–8768.

Liu, S., Johns, E., and Davison, A. J. (2019). "End-To-End Multi-Task Learning With Attention". en. In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Long Beach, CA, USA: IEEE, pp. 1871–1880. ISBN: 978-1-72813-293-8. DOI: 10.1109/CVPR.2019.00197.

Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., and Wu, H. (2018). "Mixed Precision Training". In: *International Conference on Learning Representations*.

Misra, I., Shrivastava, A., Gupta, A., and Hebert, M. (2016). "Cross-Stitch Networks for Multi-task Learning". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3994–4003.

Nilsback, M.-E. and Zisserman, A. (2008). "Automated Flower Classification over a Large Number of Classes". In: *Indian Conference on Computer Vision, Graphics and Image Processing*.

NVIDIA/apex (2020). original-date: 2018-04-23T16:28:52Z. URL: https://github.com/NVIDIA/apex (visited on 03/25/2020).

Oquab, M., Bottou, L., Laptev, I., and Sivic, J. (2014). "Learning and Transferring Mid-level Image Representations Using Convolutional Neural Networks". en. In: *2014 IEEE*

Conference on Computer Vision and Pattern Recognition. Columbus, OH, USA: IEEE, pp. 1717–1724. ISBN: 978-1-4799-5118-5. DOI: 10.1109/CVPR.2014.222.

Pan, S. J. and Yang, Q. (2010). "A Survey on Transfer Learning". en. In: *IEEE Transactions on Knowledge and Data Engineering* 22.10, pp. 1345–1359. ISSN: 1041-4347. DOI: 10.1109/TKDE.2009.191.

Park, H., Bharadhwaj, H., and Lim, B. Y. (2019). "Hierarchical Multi-Task Learning for Healthy Drink Classification". In: *2019 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8.

Parkhi, O. M., Vedaldi, A., Zisserman, A., and Jawahar, C. V. (2012). "Cats and Dogs". In: *IEEE Conference on Computer Vision and Pattern Recognition*.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., pp. 8024–8035.

Pfeiffer, K., Hermans, A., Sárándi, I., Weber, M., and Leibe, B. (2019). "Visual Person Understanding Through Multi-task and Multi-dataset Learning". In: *German Conference on Pattern Recognition*, pp. 551–566.

Rauf, H. T., Saleem, B. A., Lali, M. I. U., Khan, M. A., Sharif, M., and Bukhari, S. A. C. (2019). "A citrus fruits and leaves dataset for detection and classification of citrus diseases through machine learning". In: *Data in brief* 26, p. 104340.

Ren, S., He, K., Girshick, R., and Sun, J. (2015). "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: *Advances in Neural Information Processing Systems 28*. Ed. by C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett. Curran Associates, Inc., pp. 91–99.

Ronneberger, O., Fischer, P., and Brox, T. (2015). "U-net: Convolutional networks for biomedical image segmentation". In: *International Conference on Medical image computing and computer-assisted intervention*. Springer, pp. 234–241.

Ruder, S. (2017). "An Overview of Multi-Task Learning in Deep Neural Networks". en. In: *arXiv:1706.05098 [cs, stat]*. arXiv: 1706.05098. URL: http://arxiv.org/abs/1706.05098.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., et al. (2015). "Imagenet large scale visual recognition challenge". In: *International journal of computer vision* 115.3, pp. 211–252.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L. (2018). "MobileNetV2: Inverted Residuals and Linear Bottlenecks". In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4510–4520.

Simonyan, K. and Zisserman, A. (2015). "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *International Conference on Learning Representations*.

Smith, L. N. (2017). "Cyclical Learning Rates for Training Neural Networks". In: *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pp. 464–472.

Standley, T., Zamir, A. R., Chen, D., Guibas, L., Malik, J., and Savarese, S. (May 2019). "Which Tasks Should Be Learned Together in Multi-task Learning?" en. In: *arXiv:1905.07553 [cs]*. arXiv: 1905.07553. URL: http://arxiv.org/abs/1905.07553.

Tan, M., Chen, B., Pang, R., Vasudevan, V., Sandler, M., Howard, A., and Le, Q. V. (2019). "MnasNet: Platform-Aware Neural Architecture Search for Mobile". In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2815–2823.

Tan, M. and Le, Q. (2019). "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks". In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by K. Chaudhuri and R. Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. Long Beach, California, USA: PMLR, pp. 6105–6114.

Tan, M., Pang, R., and Le, Q. V. (2020). "EfficientDet: Scalable and Efficient Object Detection". In: URL: https://arxiv.org/abs/1911.09070.

Teichmann, M., Weber, M., Zöllner, M., Cipolla, R., and Urtasun, R. (June 2018). "MultiNet: Real-time Joint Semantic Reasoning for Autonomous Driving". In: *2018 IEEE Intelligent Vehicles Symposium (IV)*. 2018 IEEE Intelligent Vehicles Symposium (IV). ISSN: 1931-0587, pp. 1013–1020. DOI: 10.1109/IVS.2018.8500504.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). "Attention is All you Need". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Curran Associates, Inc., pp. 5998–6008.

Viola, P. and Jones, M. (2001). "Rapid object detection using a boosted cascade of simple features". en. In: *Proceedings of the 2001 IEEE Computer Society Conference on Com-*

puter Vision and Pattern Recognition. CVPR 2001. Vol. 1. Kauai, HI, USA: IEEE Comput. Soc, pp. I–511–I–518. ISBN: 978-0-7695-1272-3. DOI: 10.1109/CVPR.2001.990517.

Xie, S., Girshick, R., Dollár, P., Tu, Z., and He, K. (2017). "Aggregated residual transformations for deep neural networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1492–1500.

Yakubovskiy, P. (May 31, 2020). *qubvel/segmentation_models.pytorch*. original-date: 2019-03-01T16:21:21Z. URL: https://github.com/qubvel/segmentation_models.pytorch (visited on 05/31/2020).

Yao, Y., Wang, Y., Guo, Y., Lin, J., Qin, H., and Yan, J. (2020). "Cross-dataset Training for Class Increasing Object Detection". In: *arXiv:2001.04621 [cs]*. arXiv: 2001.04621. URL: http://arxiv.org/abs/2001.04621.

Zagoruyko, S. and Komodakis, N. (2016). "Wide Residual Networks". In: *Proceedings of the British Machine Vision Conference (BMVC)*. Ed. by E. R. H. Richard C. Wilson and W. A. P. Smith. BMVA Press, pp. 87.1–87.12. ISBN: 1-901725-59-6. DOI: 10.5244/C.30.87.

Zamir, A. R., Sax, A., Shen, W., Guibas, L. J., Malik, J., and Savarese, S. (2018). "Taskonomy: Disentangling task transfer learning". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3712–3722.

Zhang, W., Li, R., Zeng, T., Sun, Q., Kumar, S., Ye, J., and Ji, S. (2020). "Deep Model Based Transfer and Multi-Task Learning for Biological Image Analysis". In: *IEEE Transactions on Big Data* 6.2, pp. 322–333.

Zhang, Y. and Yang, Q. (2018). "A Survey on Multi-Task Learning". en. In: *arXiv:1707.08114 [cs]*. arXiv: 1707.08114. URL: http://arxiv.org/abs/1707.08114.

Zhao, B., Li, X., Lu, X., and Wang, Z. (2019). "A CNN-RNN Architecture for Multi-Label Weather Recognition". In: *CoRR* abs/1904.10709. arXiv: 1904.10709. URL: http://arxiv.org/abs/1904.10709.

zylo117 (2020). *zylo117/Yet-Another-EfficientDet-Pytorch*. URL: https://github.com/zylo117/Yet-Another-EfficientDet-Pytorch (visited on 05/26/2020).