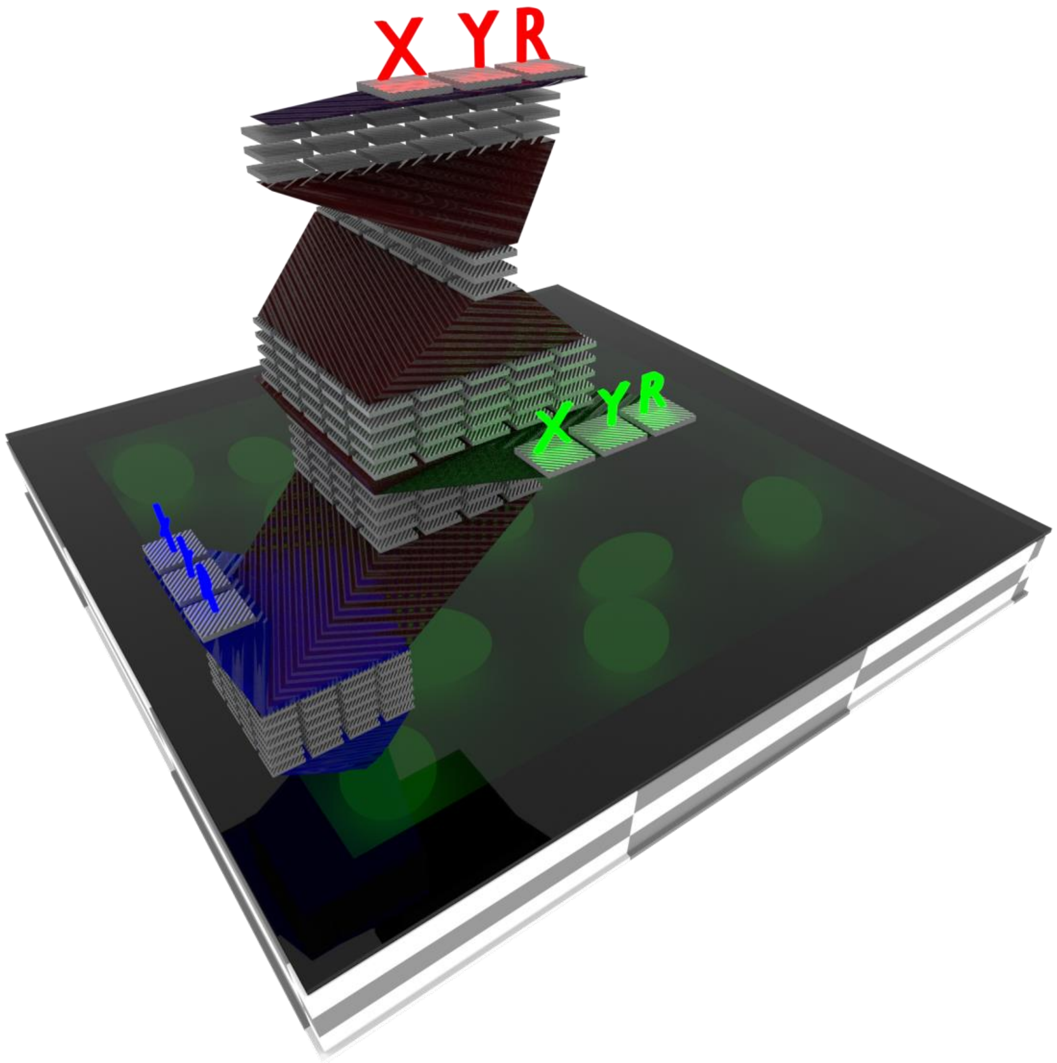
CHALMERS
UNIVERSITY OF TECHNOLOGY



Growing Artificial Neural Networks

Novel approaches to Deep Learning for Image Analysis and Particle Tracking

Master's thesis in Applied Physics

MARTIN SELIN

Department of Physics
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2019

MASTER'S THESIS 2019

Growing Artificial Neural Networks

Novel approaches to Deep Learning for Image Analysis and Particle Tracking

MARTIN SELIN

Department of Phyics
Soft Matter Lab
Chalmers University of Technology
Gothenburg, Sweden 2019

Growing Artificial Neural Networks
Novel approaches to Deep Learning for Image Analysis and Particle Tracking
MARTIN SELIN
© MARTIN SELIN, 2019
Supervisor : Giovanni Volpe, University of Gothenburg
Examiner : Giovanni Volpe, University of Gothenburg

Master's thesis 2019
Department of Phyics
Soft Matter Lab
Chalmers university of Technology
SE-412 96 Gothenburg
—-
Telephone +46 730 68 18 93
Cover: Multistage neural network in action. (Artistic rendering by Martin Selin,

2019)
Typeset in LaTeX
Gothenburg, Sweden 2019

Abstract

Deep-learning has recently emerged as one of the most successful methods for analyzing large amounts of data and constructing models from it. It has revolutionized the field of image analysis and the algorithms are now being employed in research field outside of computer science. The methods do however suffer from several drawbacks such as large computational costs.

In this thesis alternative methods for training the underlying networks are evaluated. These methods are based on gradually growing networks during training using layer-by-layer training as well as increasing network width. These training methods lend themselves to easily implementing networks of tune-able size allowing for trade-off between high accuracy and fast execution or the construction of modular networks in which one can chose to execute only a small part of the network to get a very fast prediction at the cost of some accuracy. The layer-by-layer method is applied to multiple different image analysis tasks and the performance is evaluated and compared to that of regular training. Both the layer by layer training and the breadth training are comparable to normal training in performance. The modular nature of the networks make them suitable for applications within multi-particle tracking.

Keywords:**Deep learning, ANNs, DVM, image analysis, particle tracking**

Acknowledgements

I am very grateful for the help and advice I have received while working on this thesis. I would like to thank Saga Helgadóttir and Aykut Argun for always being available to help whenever I got stuck. Giovanni Volpe for guiding me through the work steering the work away from several potential pitfalls and bringing the final report to a level I am very satisfied with. Henry Yang and Bernhard Mehlig for countless of interesting and rewarding discussions on the topic of deep learning which helped highlight what is novel in this work and how to develop it. The Soft Matter Lab group for being impeccable company. Lastly I also want to thank Adam Liberda and Theo Berglin for many refreshing walks and fascinating conversations and also for patiently listening to my poor jokes and poorer puns.

Contents

Glossary

Activation function function used by nodes to introduce nonlinearities in a neural network. 4

Batch Subset of training data used in SGD. 8

bias Parameter added to the dot product between weight vector and node input. 4

CNN Convolutional neural network. 8

DeepTrack Deep learning framework for particle tracking[1].. 8

epoch A epoch is when the full dataset is passed thorough the ANN for training. 8

Hyperparameters Parameters describing the network architecture and training details, such as number of nodes in each layer and batch size.. 12

Layer A set of nodes with the same input vector(s) which also send their output to the same layers.. 2

LBL Layer-by-layer training. 15

MAE Mean Absolute Error. 32

MSE Mean squared error. Loss function commonly used for estimation tasks.. 6

Node Computational unit of neural network. 4

Optimizer Algorithm for updating network parameters. 8

RMSprop Optimizer using a variant of gradient descent with momentum. 8

SGD Stochastic Gradient Descent, a standard optimizer for neural networks.. 8

SNR Signal to Noise Ratio. 30

weight Parameters used for dot product with inputs to node. 4

Chapter 1

Introduction

In the last decade there has been a massive growth in interest of AI with the number of published articles and money invested per year growing contiguously [2]. One often cited reason for this is the increase in data available to researchers and the increased computational performance of computers [3]. The fastest evolving branch of AI is deep learning. This has lead to deep learning algorithms finding use in several widely different fields ranging from climate science to material science, microbiology to more engineering oriented ones such as autonomous vehicles as well as areas in economics such as product recommendation [4]–[8].

Also contributing to the increased interest is that deep learning has made several breakthroughs in the last couple of years which has gotten plenty of media attention. In 2015 deep mind beat professional go-player [9], match humans in image recognition [10], [11]. The progress is rapid and this reflects on the investments made in the technology.

1.1 Deep learning in image analysis

One of the most widespread applications of deep learning is within the field of image analysis. Deep-learning has been especially successful compared to other approaches in tasks which are generally considered difficult for machines, such as object detection and image classification where 'super human' performance has been achieved [10], [11].

The progress has been rapid and so has the number of applications which utilize the techniques. They are now being employed in for instance health care to help doctors diagnose diseases [12].

Deep learning is starting to achieve widespread use also within the scientific community. For instance they have been employed to distinguish between quark and gluon jets [13].

1.1.1 Digital video microscopy

Digital video microscopy is a common tool for researchers studying dynamical processes at the microscale. Deep learning has been shown to be a viable method for analyzing the data accurately, more so than the standard methods [1], [14], but little attention has been paid to the processing time which risks limiting the use of deep learning for real-time applications. For instance for steering the microscope during an experiment. This is especially difficult to handle since the network are oftentimes optimized by the research team for a specific task meaning that creating a highly well tuned network might demand too much resources to be worth the effort.

1.2 Difficulties and drawbacks with deep learning

Deep learning is however by no means a magic toolbox for solving all problems. A disadvantage with deep learning models is that they have very large computational costs associated with them, the largest models can have several billion parameters [15]. Not only when they are trained but also when they are used to make predictions. This complexity can be a hindrance. But the larger the network the larger is the range of functions it can replicate meaning that they can potentially be more accurate than a small network[16]. Furthermore the algorithms are often subject to problems such as overfitting meaning that it is difficult to know how much to trust the predictions they make. These are some of the main issues which limits their usability.

Several methods exist to help alleviate these problems such as data augmentation, shuffling and dropout, some of these are briefly described in section 2.4.2. Combating overfitting and developing methods for measuring the trustworthiness network predictions are active areas of research.

One can often both save training time and improve performance of convolutional neural networks by using the "convolutional base" from another network trained on a different but larger dataset. Which is known as transfer learning. That is reusing the first convolutional Layers of a preexisting model as the base for a new one. The borrowed convolutional base do not need to be retrained to perform well on the new problem. A recent variant of transfer learning is progressive learning which has emerged in the last couple of years as an efficient way to transfer knowledge between machine learning problems. [17] used it to learn multiple different videogames and showed that the networks performed superior if they had previously learned to play other games.

1.3 Growing neural networks

Here we focus on the problem of choosing between accuracy and computational cost and a method which might make this trade-off easier to make while not compromising performance. Our approach to this is based on various variants of network growing applied to the field of image analysis. First is the layer-by-layer training. LBL is tried on two separate image classification tasks, the well known MNIST and the former Kaggle competition *Cats and Dogs*. LBL is also evaluated on a task of particle tracking which is in essence an estimation task (finding the coordinates of a particle in an image).

The second growth algorithm is yet more novel and involves increasing the network width after finishing training, a training method we chose to call breadth growth. This is done without changing the weight already trained. The training method is used only for the particle tracking task as a proof of concept.

Both algorithms are compared in detail to regular training in order to find any performance differences to give some guidance as to when growing networks is a good option and when it is not.

These two training algorithms open up for the possibility to easily tune the network size. This has multiple potential applications not least for decreasing computational costs. One such application is demonstrated, multi particle tracking, for which the modularity of the networks allow for greater execution speed.

Chapter 2

Deep learning

Deep learning has emerged as one of the most promising group of algorithms for applications in artificial intelligence and data analysis. Performing state of the art in fields as various as speech recognition, game playing and image analysis [18]. This chapter aims to give the reader a brief overview of deep learning and enough background to understand the methods and results presented later in the thesis.

The focus of this thesis is on the applications of deep learning to data analysis and more specifically to image analysis. Within image analysis deep learning has already started to transform the field with deep learning algorithms already outperforming humans in certain tasks [10], [11]. It is important also from a physicist's perspective in enabling large scale analysis of for instance image data sets from experiments to be automated. One application which special focus is put upon in this thesis is that of digital video microscopy (DVM).

2.1 Basic principles of deep learning

To understand deep learning one first needs a basic understanding of artificial neural networks (ANNs). These consists of a set of layers each having a number of Nodes (sometimes called neurons), see fig. 2.1. Typically all the nodes in a layer are connected to all the nodes in the previous layer. The connections consist of scalar multiplication of the inputs to a node, $\mathbf{x}$, with the weight vector, $\mathbf{w}_i^k$, of that node. A bias term b is added to result of the scalar multiplication which is then sent into an Activation function A to produce a new output. A node performs the operations

$$N_i^k(\mathbf{x}) = A(b_i^k + \mathbf{w} \cdot \mathbf{x}) = A(b_i^k + \sum_j w_{i,j}^k x_j) \tag{2.1}$$

Thus each layer transforms the input before passing it onto the next layer. Here we have introduced indices such that the weights upper index, k in eq. (2.1), indicates which layer the weight is in, the first of the lower indices, i in eq. (2.1), indicate which node it belongs to and the second lower index , j in eq. (2.1), shows which input it is connected to. The values for the weights $w_{i,j}^k$ are along with the bias term

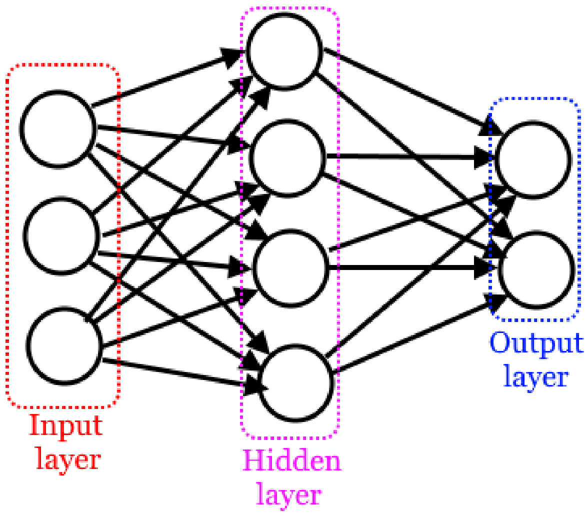


Figure 2.1: *Illustration of a basic artificial neural network with a single hidden layer and dense connections. The network has three layers, input hidden and output. The layers have 3,5 and 2 node respectively. The first layer feeds the second with inputs, the second feeds the third etc, propagating the signal through the network.*

b_i^k what is "learned" during training. I.e values of the parameters are found such that the network manages to perform the desired task. Note that each node has a set of parameters of its own. The activation function A introduces nonlinerities in the network data processing which is necessary for learning of non linear functions, [16]. There are very many different activation functions found in the literature. In general one wants the derivatives of the activation functions to be easy to compute since this simplifies the learning, see section 2.1.1. Common activation functions are tanh, sigmoid and ReLu (rectified linear unit). The latter is defined as

$$ReLu(x) = max(0, x) \tag{2.2}$$

Relu is often used in the hidden layers of models. Sigmoid is defined as

$$Sigmoid(x) = \frac{e^x}{1 + e^x} \tag{2.3}$$

The sigmoid activation is often used in output layers when the task is binary classification [19].

An activation function commonly used for the output layer is the softmax defined as

$$Softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{2.4}$$

The softmax function has the property that the sum of all the output channels i equals 1 meaning that the output may be interpreted as a probability distribution. Which activation function to use depends on the task at hand, especially in the case of the output layer[16], [20]. It is important to use one which covers the full range of the possible outputs.

Another central component to deep learning is the loss function. The loss function is used to gauge how close the network was to the desired output. It is important for the learning process but is not a part of the actual network during execution. It is discussed in somewhat more detail in section 2.1.1 below.

Deep learning is artificial neural networks with many hidden layers. For a more thorough introduction to deep learning the reader is referred to [16], [21] and [20].

2.1.1 Learning through gradient descent

The way a deep learning algorithms learns is by finding appropriate values for the many parameters (weights and biases) in the networks. This is generally done using gradient descent[16], [20], [21]. To be able to use gradient descent one first needs a function to calculate the gradient of which measures how close the networks output was to the target output. The function that calculates this is called the loss function (or occasionally the cost function). The lower the value of the loss function the closer the networks prediction is to the true value. The idea being to update the weights so that the next prediction produce a smaller loss.

There are multiple different loss functions, which loss function one should choose depends on the problem. Designing and choosing loss functions can be difficult and is a key challenge in successfully applying deep learning algorithms[22]. For estimation tasks a common choice is the mean squared error (MSE) defined as

$$L(t,y) = \frac{1}{n}\sum_{i}^{n}(t^i - y^i)^2 \tag{2.5}$$

Where t^i is the true value and y^i is the estimation made by the algorithm for training example i. We want the network to perform well on all the training examples which is why the loss function is calculated as a sum over the whole training set.

For classification problems cross-entropy is often used. The binary cross entropy is often used in two class classification problems when you have a single output predicting which of two classes,a and b, an input belongs to. The output should be 0 for class a and 1 for class b. The binary cross entropy is defined as:

$$L(y,t) = \frac{-1}{N}\sum_{i=1}^{N} t^i \log(y^i) + (1-t^i)\log(1-y^i) \tag{2.6}$$

As before t^i is the true class of sample i, either 1 or 0 while y^i is the prediction made. The sum is over the training examples.

For multi class classifications it can be generalized to the categorical cross entropy

$$L(y_i,t_i) = \frac{-1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} t_i^j \log(y_i^j) \tag{2.7}$$

Where j indexes the different classes in the problem. In practice one often use one-hot encoding to describe the target vectors in multiclass classification [20].

The gradient is calculated on the loss function with respect to the various parameters. To be able to calculate the gradient we first need and expression for the operations the network performs. Each node takes the output from the previous layer, computes the scalar product with the weight vector, add the bias term and use this as input to the activation function. Thus the networks computation can be written as a series of nested activation functions. In short this means that x_i in eq. (2.1) is replaced with a sum over the outputs from the previous layer, which are also given by an expression of the form in eq. (2.1). As an example the expression for a two layer network can be written as

$$y_i = A^2 \left\{ \sum_j W^2_{i,j} A^1 \left\{ \sum_k W^1_{j,k} x_k - b^1_j \right\} - b^2_i \right\} \tag{2.8}$$

Where as before x_k are the inputs, y_i are the outputs, $W^1_{i,j}$ and $W^2_{j,k}$ are the weights in the first and second layer respectively.

Now we are prepared to compute the gradient with respect to the parameters. The weights are updated as

$$\delta W^k_{i,j} = -\eta \frac{\partial L(y,t)}{\partial W^k_{i,j}} \tag{2.9}$$

Where the parameter η is known as the learning rate. The learning rate is often tuned for the problem at hand. Occasionally it is changed during the course of training, it is then decreased as the network learns more and more. The idea being that the network should learn fast in the beginning when the weights are far from optimal and then slower once it is closer to its optimum. Thus the new weights are calculated as

$$W^k_{i,j} := W^k_{i,j} + \delta W^k_{i,j} \tag{2.10}$$

Where the operator := denotes that we reassign the variables value, similar to the "=" operator as used in programming languages such as python. The gradient is calculated using the chain rule. For a scalar valued function z of nonscalar functions $\mathbf{y}$, $z = z(\mathbf{y}(\mathbf{x}))$ the chain rule tells us that

$$\frac{\partial z}{\partial x_i} = (\frac{\partial \mathbf{y}}{\partial \mathbf{x}})^T \nabla_{\mathbf{y}} z \tag{2.11}$$

Where $(\frac{\partial \mathbf{y}}{\partial \mathbf{x}})^T$ is the Jacobian and $\nabla_{\mathbf{y}} z$ is the gradient of z w.r.t $\mathbf{y}$. The learning algorithm is often termed back-propagation in the context of deep learning since the weight updates are propagated backwards from the output towards the input. The full back-propagation for larger networks consists of multiple multiplications with such Jacobians. As is evident from eq. (2.11) many of the factors will be the same for many of the nodes meaning that one can save runtime by storing the values of the matrices and their products. This does however come with extra memory requirements meaning that there is a tradeoff between computational time and memory requirements. Equation eq. (2.11) can easily be generalized to tensors. For further details see [16].

Calculating the loss on the whole training set can be computationally as well as prone to getting stuck in local minima demanding making the learning process slow. Therefore it is common to split the training set into a set of "batches". The loss function is then calculated only on the samples in each training Batch, rather than the full training set, before being used to update the parameters. Sending all the training batches through this process is known as one training epoch. To ensure stability the order of the training samples are shuffled at the beginning of each epoch meaning that the same subset is not reused for the batches. Due to this shuffling this version of gradient descent is known as Stochastic Gradient Descent (SGD).

Other optimizers

Various different versions exist of SGD which seek to counteract some of its weak points, such as the risk of getting stuck in local minima. A common method for combating this is to use momentum. Then the weight updates are changed to

$$\delta W_{i,j}^{k} := -\eta \frac{\partial L(y,t)}{\partial W_{i,j}^{k}} + \alpha \delta W_{i,j}^{k} \tag{2.12}$$

Meaning that the weight updates has 'memory' and the parameter $0 < \alpha \leq 1$ is called momentum constant. We will mostly make use of the Optimizer RMSprop, mainly because it is the default in DeepTrack which is used extensively in this thesis, see section 3.5.1. RMSprop utilize a variant of momentum.

2.2 Convolutional Neural Networks

One of the largest subsets of deep learning architectures are Convolutional Neural Networks (CNNs). Their perhaps biggest advantage being that they effectively combat one of the weakest points of dense layers have when used for image processing, namely that the weights are tied to specific locations in the images. Convolutional networks are mostly used for image processing and consist of a set of convolutional operations. And consists of convolutional operations in series, often with pooling in between [16].

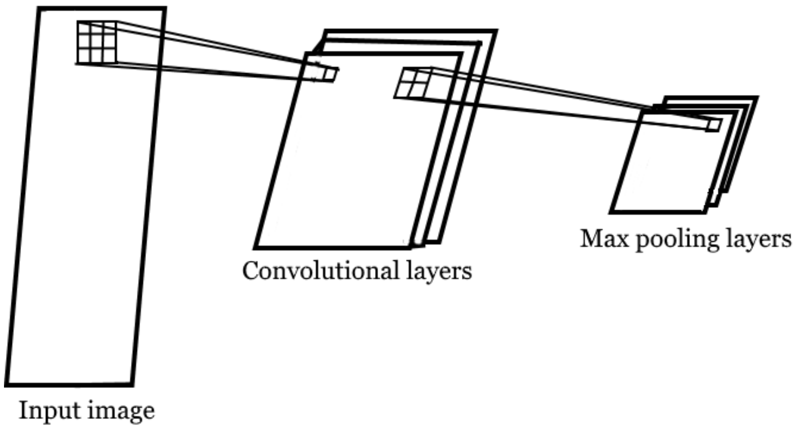


Figure 2.2: *Illustration of the convolutonal operation followed by a pooling operation. The grid represents the weight matrix being multiplied with the pixel values for the convolution. In the pooling operation the max value of the pixels in the grid are instead used.*

The convolutional operation can be described as

$$V_{i,j} = A(\sum_{p=1}^{3}\sum_{q=1}^{3} w_{p,q}x_{p+i-1,q+j-1} - b), i \in [0, N], j \in [0, M] \tag{2.13}$$

Where A is the activation function and $w_{p,q}$ the weight matrix of what is called a kernel, which here is of size 3×3 which is a quite typical kernel size [19]. N and M are the number of pixels in the input image horizontally and vertically respectively. Naturally to get the full output on combines all the outputs to a single image. The operation is illustrated in fig. 2.2. For kernels larger than 1×1 the boundary will have to be treated differently from the interior of the image. This is because the kernel would need to go beyond the image to calculate its features, that is $p + i - 1 \notin [0, N]$ or $q + i - 1 \notin [0, M]$ in eq. (2.13). The most common ways to threat the boundary is to either use padding with zeros or to ignore it and not compute $V_{i,j}$ there which leads to a slight decrease in image size. We will not use padding but let the images decrease slightly in size.

The stacking of many convolutions on top of another one can gradually distill more and more information since each kernel in a new layer is applied to more of the pixels in the original image through the downsampling performed in the previous layers. One can increase the downsampling rate of the convolution by increasing the stride, that is how much the kernel should be shifted as it scans the image [20]. Doubling the stride along one axis will therefore halve output image size. We will stick to using stride 1.

There are other reasons to use convolutional layers. One of the most important being that the number of parameters in a convolutional layer is small relative to that of a dense layer since one has only a single kernel per input image and channel rather than individual weights for each input signal. For an input signal of width W, height HWC. As an example a RGB image of size 256×256 we have a total size of $256 \times 256 \times 3 = 196608$ weights (factor 3 from the three different color channels) if connected to a dense node yet only $3 \times 3 \times 3 = 27$ weights if connected to a convolutional node (of kernel size 3×3).

2.2.1 Pooling

Convolutional layers are often combined with pooling layers. The most commonly used of which is the maxpooling, but other types such as mean pooling are occasionally used [20]. Max pooling consists of taking the maximum value of the pixels in a certain window, see fig. 2.2. It can be described as

$$V_{m,n} = max(x_{m+i-M/2,n+j-N/2}) i \in [0, N], j \in [0, M] \tag{2.14}$$

The purpose of the pooling layers is to reduce the dimension of the input and try to extract the most important features. Therefore a stride length larger than one is most often used. Here a pooling window of 2×2 will be used with a stride of 2 meaning that the images will be down sampled by a factor two. Down sampling has the added benefit of reducing the dimension of the data making for fewer parameters in later layers meaning a lesser complexity of the overall model.

2.2.2 Dense top

Most convolutional networks start with a series of convolutional layers with pooling layers in between. On top of these one often adds a series of dense layers to make the actual prediction. The outputs from the previous convolutions is put into a 1D array before being fed into the dense layers.

2.3 Applications

Deep learning has found applications in a variety of fields such as language processing, customer recommendations and image analysis. The number of applications for deep learning is growing rapidly and is expected to do so in the coming years as more and more fields make use of machine learning [2].

In this thesis the focus is on image analysis some of it's subfields. Mainly image classification and object tracking.

Possible that some of the methods discussed here has greater use in other areas than image analysis but investigating whether this is the case is outside the scope of this thesis.

2.3.1 Digital video microscopy

The application of deep learning that we will look closest at is to digital video microscopy (DVM). Deep learning is becoming more and more common in DVM not least for particle tracking with several groups working on the algorithms [1].

2.4 Difficulties with training deep learning networks

There are many difficulties that can arise when trying to train a deep learning network[16], [21]. This section seeks to give a brief overview of some of the most common problems which may arise during training. It is not a universal guide to combating these issues but should serve as a short introduction to the area.

2.4.1 Vanishing and exploding gradient

The vanisning gradient problem is caused by the partial derivatives, mostly in early layers, being very small. The error signal is multiplied by the partial derivatives of the weights of each layer. This means that the error signal, in the worst cases, decrease exponentially with the distance in number of layers from the output. This slows the training to a crawl and can in some cases completely prohibit it.

Exploding gradient is the opposite. If the partial derivatives are very large then the errorsignal will grow exponentially with with the number of layers it passes on its way from output layer to input layer. Which makes it "explode" in magnitude thus also making the weight updates unreasonably large.

Greater hardware in combination with suitable weight initalization techniques have helped with these problems.

2.4.2 Overfitting and unstable training

Overfitting is caused by the network learning the training data too well. This means that the network performs very well on the training data but fails to generalize due to the weights being tuned very specifically for the training data.

Multiple methods exist to deal with overfitting and increase network performance [20]. Some of the more popular ones are listed below

- **Early stopping** - Abort training before the network has had a chance to memorize the training data.
- **Pruning** - Weak connections are removed after training.
- **Dropout** - during training a fraction of the connections between nodes are set to zero. Which are set to zero is chosen at random and thus one effectively trains a various subnetworks which prevents overfitting [16].
- **Data augmentation** - The training is distorted randomly in various ways, images can for instance be elongated or compressed along an axis. It is now standard practice for many image recognition tasks [23], [24].
- **Weight normalization** - Large weights in the networks can lead to very large signals drowning other signals. Apply some kind of normalization to prevent the weights becoming too large. Such as a penalty in the loss function for large weights.

Whether to use these normalization techniques or not effectively adds more hyperparameters to the training. This makes it more difficult to make a comparison between various training techniques. To simplify the comparison we therefore abstain from using most of the normalization techniques with the exception of data augmentation and to a certain degree early stopping. Furthermore in the estimation task simulated data is used for training which means that there should be little risk of overfitting anyhow.

2.5 Computational cost

With computational cost we refer to the amount of computing power needed to execute an algorithm, such as train a neural network, in practice. Deep learning networks are notorious for requiring large amounts or training to perform well. For instance the famous example of AlphaGo which beat world champion at the game Go was trained on a cluster with hundreds of GPUs for several months [9]. Computations of this scale are also associated with large economical costs. Both in terms of hardware and power consumption. The cost of a Nvidia Tesla 100V, a recent high performance GPU, is around 10000$ [25].

The large computational costs makes it difficult to optimize Hyperparameterss, doing a search in the hyperparameter space is in general unfeasible. Therefore one needs to have a quite good idea of what one is after already before starting training a big network so that one can choose reasonable hyperparameters. This is an active area of research [26].

The issue of how the architecture affects the computational cost is discussed somewhat more in depth in section 2.6 below.

2.6 Computational complexity

Closely related to the issue of computational cost is the computational complexity. Although not quite equivalent they are similar enough for one to be able to get a decent estimate for the scaling of the computational cost by calculating the computational complexity.

There have been plenty of research into how one can reduce the computational complexity of deep learning networks, see for example [27]. But also into how one can choose between the higher accuracy of large models and the fast execution of small [28].

In this section the scaling relations for various hyperparameters of deep learning are derived.

2.6.1 Dense layers

The computational cost of a dense layer is quite straightforward to estimate. Assuming that the layer has N nodes and an input of size M then the complexity is calculated by noting that each node takes the whole input vector M and multiplies it with corresponding weights for each node and then adds the result together. The addition of bias terms is constant in N and is thus a lot smaller in magnitude, order N and so is the activation function. The computational cost,K, thus scales as

$$K_{dense} = \mathcal{O}(2MN + N + NA) \tag{2.15}$$

It is worth noting that dense layers are mostly what is used as output layers.

2.6.2 Convolutional layers

Convolutional layers are generally somewhat more demanding to compute than dense layers. They consist of an array of image convolutions. Each convolutional map has a set of $d \times d$ parameters (on occasion a non quadratic map is used). Typically d is 1 to 5 [20]. These maps are applied to the input a number of times equal to the size of the output,width times height ($H' \times W'$). For each time they are applied both a multiplication and an addition is made for every element of the mapping. The input is an image with width W, height H and C different channels. The channels correspond to colour in normal images and the outputs of different convolutional maps in intermediate layers in models. The full map is applied to each channel of each image. The total computational cost is thus

$$K_{conv} = \mathcal{O}(2Nd^2H'W'C + NH'W') \tag{2.16}$$

There are also convolutions of different dimensions from the 2D described above, such as 1D convolutions. Their computational cost can be estimated in a similar fashion.

2.6.3 Pooling layers

The complexity of pooling layers is (generally) small compared to that of dense or convolutional layers. It is calculated by noting that each time the map is applied one comparison is made for each element in the map in order to choose the largest element. Assuming a pooling map of size $d \times d$ and a stride s then the computational complexity when applied to an input of size $H \times W \times C$ is proportional to the number of times the map is applied times the map size.

$$K_{pool} = \mathcal{O}(\frac{CHWd^2}{s^2}) \tag{2.17}$$

2.6.4 Forward call

The execution of a neural network in training is often referred to as the forward call. In it the input is propagated through all layers to compute an output. We have already presented the computational costs for the dense and convolutional layers but in practice many network has other operations built in as well such as maxpooling and batch normalization. It is therefore difficult to put a general number on the complexity of the execution of a network. The complexity can be calculated as the sum of the computational complexity of the individual layers.

$$K_{network} = \sum_{l \in network} K_l \tag{2.18}$$

During training one forward call is made for each sample in each epoch. Large datasets such as ImageNet contains millions of images and the networks can be trained for thousands of epochs meaning that the number of forward calls during training can easily be in the billions.

2.6.5 Back-propagation

The back-propagation is yet more complex than the forward call. As discussed in section 2.1.1 it consists of taking multiple matrix products. The size of the matrices being multiplied is proportional to the number of nodes in each layer. For simplicity we assume that each layer has n nodes making all the Jacobians square matrices. The computational complexity of multiplying two $n \times n$ matrices in the standard way is $\mathcal{O}(n^3)$ but faster implementations do exist [29]. Assuming our network has l layers then the total computational complexity for the back-propagations to all nodes is $\mathcal{O}(n^4 l)$. However as mentioned in section 2.1.1 one can greatly reduce the computational cost by saving the matrices and their products bringing the final complexity closer to

$$\mathcal{O}(n^3 l) \tag{2.19}$$

Equation eq. (2.19) needs some modification to hold for convolutional networks. Lastly we note that there is a strong dependence of the performance on the specific implementation. And that it is easier to parallelize operations within a layer than those acting between layers due to their interdependence.

2.7 Keras

Keras is a high level API used for machine learning tasks, especially deep learning [19]. It is the framework in which all models investigated here are designed,trained and evaluated in and is used with a TensorFlow back-end. Keras was chosen for a variety of reasons but mostly because it allows for easy implementation of networks and is widely used by the deep learning community.

Chapter 3

Layer by layer training

The first training method that is studied is layer by layer,LBL, training. LBL training is performed by, as the name suggests training the models layer by layer. In practice this means that one starts out with a network consisting of solely two layers (with weights), one input layer and one output layer. One then gradually grow this small network by adding more and more layers while not changing those already in the network, with the exception of the output layer.

3.1 The layer-by-layer training method

In LBL training first a small network with only an input layer and an output layer is trained. The output layer is needed partly to provide the network with a usable loss-function. The outputs of the output layer has the same structure and activation function in all the LBL models. It tries to solve the full problem the final network will address. This means that the same loss function can be used unaltered as for a regular network.

Once the first simple network is trained one freeze the weights of the input layer (the first layer with adjustable weights). They are then the used in the final network without any change. This layer is then saved in what is to become the final model. Next one saves the intermediate output of the first layer. Then one creates a new two layer network using as input the output from the previous "mini-network". One trains the new mini-network. Once this is done the input layer of this mini-network is attached to the final model (on top of the previous one). The principle of LBL training is outlined in fig. 3.1. Saving of intermediate outputs is done to limit the computational demands of training as much as possible, it does not influence performance in any way and is therefore not strictly necessary.

Others have used similar algorithms for deep learning before. The algorithm used here for layer-by-layer training is very similar to that of [30] but differ from for instance [31] in that it only utilize one output layer and not a full set of layers, for further details see section 3.3.

As discussed in chapter 2 deciding on the details of the training such as the number of epochs, batch size and optimizer to use is far from trivial. This is true also for

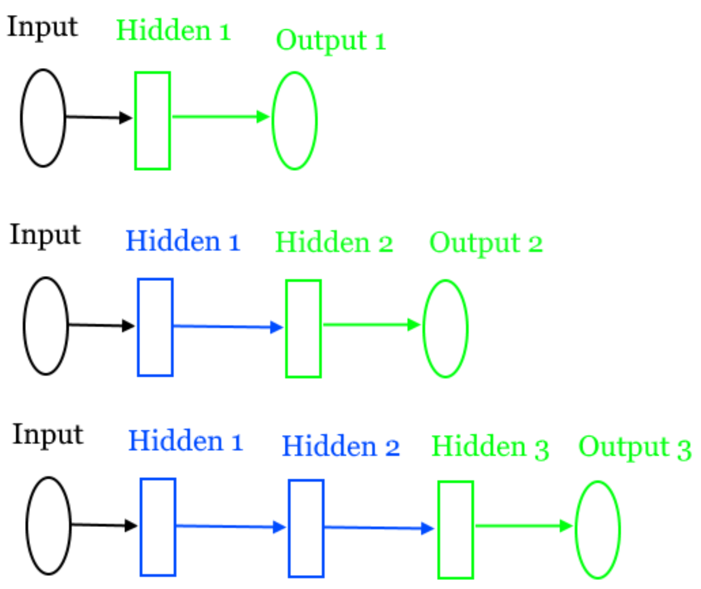


Figure 3.1: *Simplified schematic of how the layer by layer training progresses. The layers in green are being trained at each stage while those in blue have their weights frozen (prevented from updating). First only input layer is trained along with an output (top). Thereafter the input layer is frozen and a hidden layer is added on top of it (middle). The hidden layer is frozen and a new hidden layer and output is attached to it and trained (bottom). Notice that there is always a new output layer being added and trained.*

these LBL networks. Some of the most important parameters are the number of nodes in the various layers, a parameter set which still has to be chosen. This is a set of parameters which we do not vary.

3.2 Potential advantages of LBL training

Layer by Layer training has many potential advantages making the method worth investigating further. Not least in that can reduce the number of layers one needs to propagate the training data through both in the forward and backward call. Which is something that greatly influence the training time, see section 2.6.

LBL essentially avoids the vanishing gradient problem which otherwise limits the depth of networks. Meaning that it may be possible to use it for training very deep architectures which technically can represent a larger set of different functions. This does not necessarily mean that larger networks will perform better.

LBL also enable one to easily create networks of different sizes which share many weights. This means that one can choose to use one of the smaller networks if execution time is of great importance or the larger ones if accuracy is of higher priority.

These advantages needs to be compared to any eventual performance losses or gains. It is not beforehand obvious whether or not this training scheme will impact performance significantly.

3.3 Related work

Similar schemes were popular for pretraining of large deep learning networks a couple of years ago [32]. What is investigated here is however not pretraining but the possibility of doing actual training of the networks using LBL methods. They were mainly used to initialize the networks weights in a stable way by preventing or limiting the vanishing gradient problem. We will not dig deeper into this application of LBL.

There exists variants of LBL training. One of these variants is kernel similarity used for classification problems. In kernel similarity one tries to maximize the difference of the layer output vectors for different classes while maximizing the similarities. It was used successively to train networks for the MNIST task [33]. Another variation of LBL using not only a single output layer on top but a whole set of layers was shown to perform well on the imagenet task [31].

LBL also carries some resemblance to weight sharing.

Progressive learning also has some similarities to LBL training. It is used to combine the learning of multiple tasks. It has been successfully applied to for instance video games [17].

3.4 Image classification with layer-by-layer

Layer by layer training has previously been successfully applied to image classification task such as image-net and MNIST [30], [31]. Here we make the comparison between LBL training and normal training to try and discern if there are notable differences in performance if so what might be the cause of these.

Two different datasets are tested, the MNIST and the "Cats and Dogs". The first, MNIST can be considered a relatively easy task of classifying grayscale images of handwritten digits of size 28×28 while the latter is more challenging and there the task is to tell whether an image,150×150 RGB, contains a cat or a dog [20]. For the MNIST a both a convolutional and a purely dense network is used while for the "Cats and Dogs" a convolutional network is used.

These two datasets should together give an indication of when one can expect LBL may be used successfully and when not. It may be noted that neither dataset is exceedingly large with a couple of thousand training images in each.

3.4.1 MNIST digits

The MNIST digits (Modified National Institute of Standards and Technology database) is a dataset consisting of 60000 handwritten digits. Examples of which can be seen in fig. 3.2.

The task is to separate the images into 10 different classes each corresponding to one digit $0, 1, 2..9$. There is a separate test set of 10000 images. Classification accuracies of 99% and above are common in literature and relatively easy to achieve [34]. The best results are over 99.5% accuracy and are achieved by a wide range of architectures most of which being based on some type of convolution. MNIST is

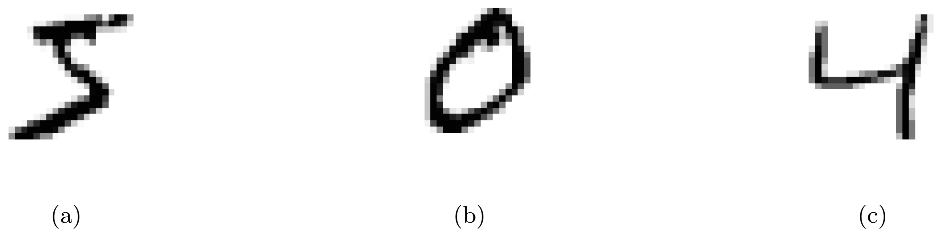


Figure 3.2: *Example images from the MNIST digits dataset. In fig. 3.2a a 5, fig. 3.2b a 0 and fig. 3.2c a 4 is shown. The images are grayscale and size* 28×28 *pixels.*

commonly used to test machine learning methods and verify that they are built on a working principle [20].

We initially only investigate the performance of LBL training on a convolutional network. This architecture has 3 convolutional layers of size $32, 64, 64$ each followed by maxplooling. On top of the convolutional layers is a dense layer of size 64 and a output layer, size 10. All layers used ReLu activation with exception of the output which uses sigmoid. The results can be seen in fig. 3.3 below. The LBL and normal training has similar distributions but that of the LBL training is shifted to slightly higher accuracies. Both methods reliably gave high accruacies.

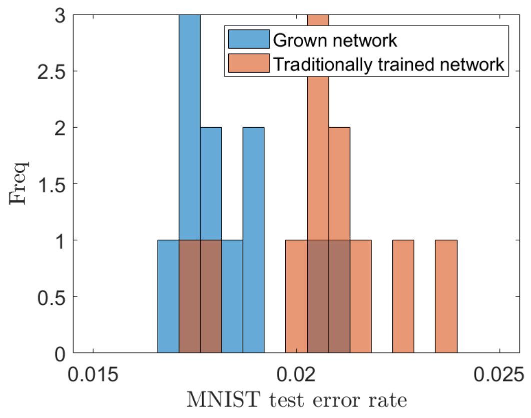


Figure 3.3: *Comparison of the performance on the MNIST dataset for layer by layer training and normal training. 5 epochs where used for training with a batch size of 5. The validation accuracy did not improve after 5 epochs for either training method.*

We are not technically limited to training small networks with LBL. We can in principle also train very deep ones, the final architecture of such very deep networks is outlined in fig. 3.4.

We build such large networks with up to 100 layers both with LBL and normal training, see fig. 3.5. The two training methods perform comparably for small networks but for larger the normal training starts struggling severely. A smaller number of training epochs seems favourable for the LBL method.

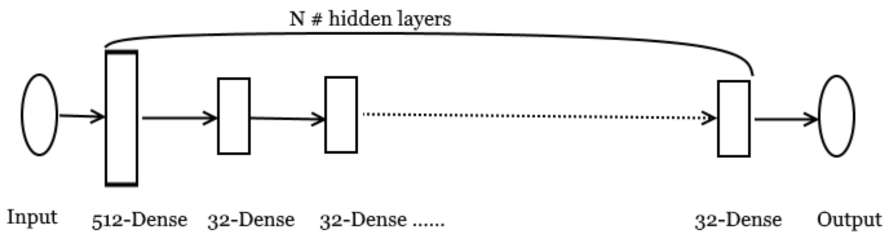
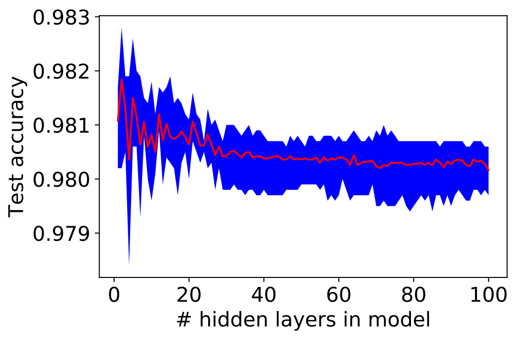
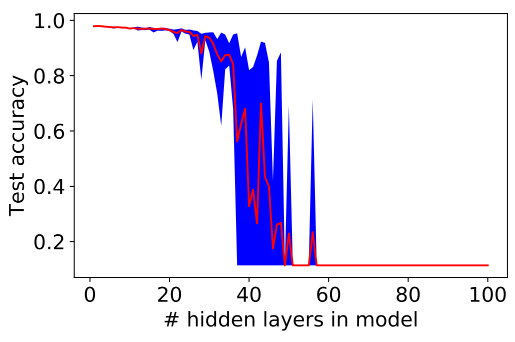


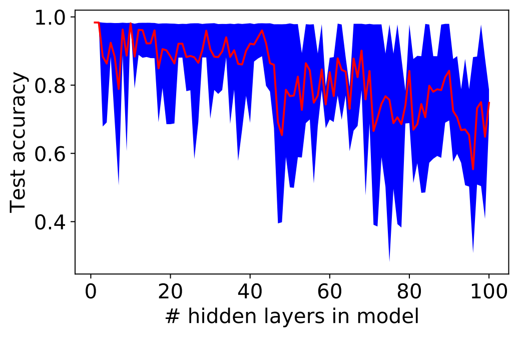
Figure 3.4: *The architecture used for testing large layer by layer networks on MNIST. The input is the images reshaped to be (784,1) rather than (28,28,1) which the original images are to better suit the dense layers. The number of hidden layers ,N, is grown.*



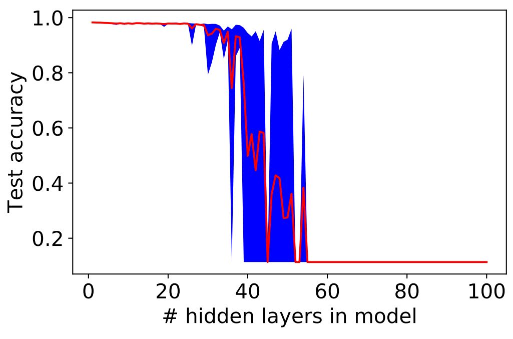
(a) Layer by layer training 5 epochs per layer.



(b) Normal training 5 epochs per network.



(c) Layer by layer training 20 epochs per layer.



(d) Normal training 20 epochs per network.

Figure 3.5: *Plot of the test accuracy as function of number of layers in model for the MNIST digits dataset. Both for normal training and LBL. The architectures are the same for the two different training methods and consist of a first layer with 512 nodes connected to the input followed by a number of dense layers each with 32 nodes. All hidden layers are dense with ReLU activation. Errorbars in blue represent the best and worst performance achieved by any model of that particular size sampled from five independent runs. The red lines are the averages of the five runs.*

As can be seen from fig. 3.5 LBL training enables one to train networks with significantly more layers than regular methods allows for. Inspired by this a single run up to 500 hidden layers is performed, see fig. 3.6, and indeed the network still manages to perform the task well.

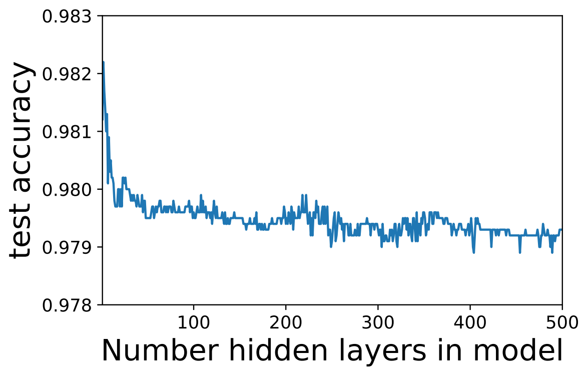


Figure 3.6: *Plot of test accuracy on the MNIST dataset for LBL training with up to 500 hidden layers. The network had 512 nodes in the first hidden layer and 32 in subsequent layers all of which are dense with ReLu activation.*

3.4.2 Probability of activations

The structure of the information flow in the network may be differ greatly between a regularly trained network and a network trained by means of LBL. We study this by looking at the activation probability of the nodes in each layer. More specifically we look at the likelihood of each node giving a positive output. Since ReLu activations are used all the local fields of the hidden units are non-negative.

In the case of normal training it is possible that the results varies depending on how many layers the model has. Because of this we plot it for a range of different sizes. The result can be seen in fig. 3.7 below. The nodes in the early layers of the LBL model are in general more likely to activate than those in later. For the normal training the smaller networks show a similar behaviour.

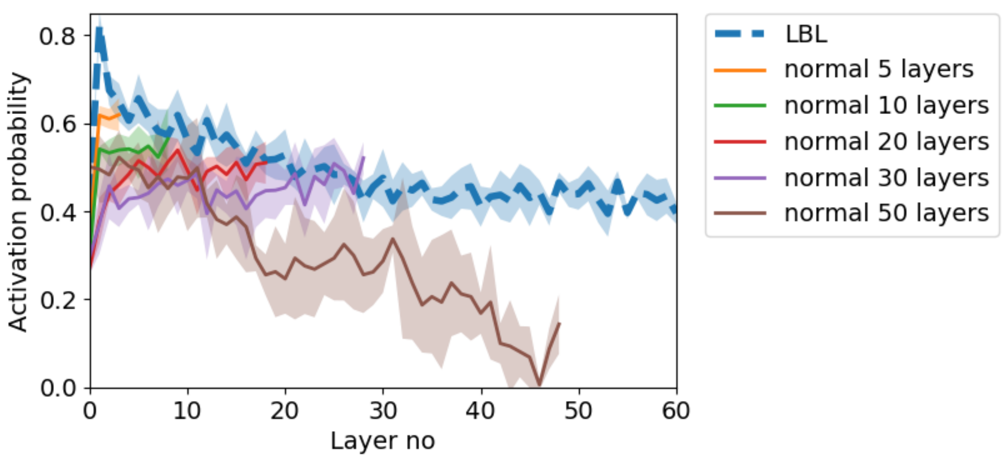


Figure 3.7: *Likelihood of neuron activations in the various layers of the MNIST networks. Various sizes where used for the normal scheme. Five separate runs. Evaluated on the 10 000 test images.*

On top of looking at the presence of a positive local field we also look at the magnitude of these local fields. The trend there is similar to that of the activation fractions, see fig. 3.8.

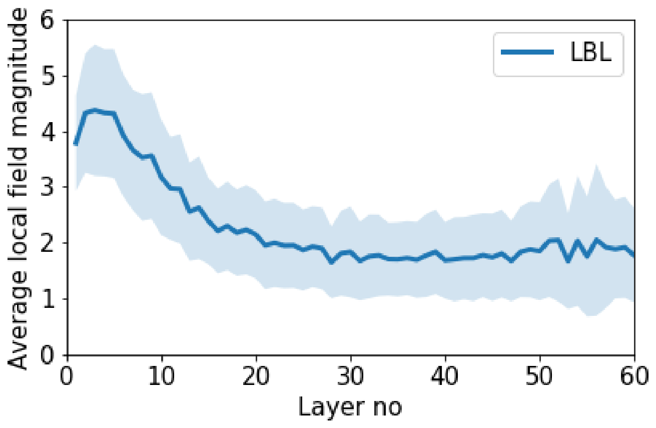


Figure 3.8: *Average local fields (activation magnitudes) in each layer of MNIST networks calculated over 10 000 test images. The shaded area represent the standard deviation calculated on the test images. Calculated as the average over 5 independent runs.*

3.4.3 Cats and dogs

The cats and dogs dataset consists of a grand total of 25000 images and is originally from a kaggle competition, .Two example images can be seen in fig. 3.10. In the following a small subset of the total dataset consisting of 2000 images has been used to evaluate the performance of LBL training on small yet moderately complex images. The images are rgb and are resized to 150×150 before being fed into the network.

We utilize a convolutional network which architecture is outlined in fig. 3.9. As loss function we choose the binary cross entropy described in eq. (2.6). The optimizer we use is the RMSprop with a fixed learning rate of 10^{-4}.

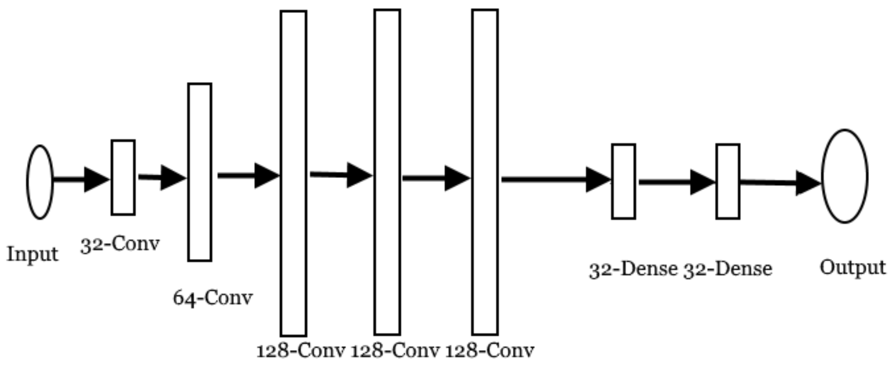


Figure 3.9: *Schematic of the architecture used for the Cats and Dogs task. Each of the convolutional layers is followed by a max-pooling. All the hidden layers utilize relu-activation and the output has sigmoid activation.*

Figure 3.10: *Example images of cat (left) and dog (right) used for training. The images are resized to* 150×150 *pixels before being fed into the network.*

The larger image size in combination with the variety in the dataset makes it a notably more challenging task than the MNIST, explored in section 3.4.1. It is not beforehand obvious whether the LBL training scheme will work on such a complex dataset. This is because it is commonly thought that the early convolutional layers in a network learns to detect rough features, such as edges or corners, while the later rather detect more abstract features, which in the case of cats and dogs, might include ears and noses [20]. Therefore the fact that LBL performs well on a very simple dataset such as MNIST is no guarantee that it will work well also for more advanced images for which larger and more abstract features may need to be learned.

3.4.4 Results

The networks are trained for 80 epochs on 2000 images and then then evaluated on a separate set of 1000 images. This is repeated five times both for LBL and normal training. Normal training of networks of equal size is used as reference. The performance improve as more layers are added see fig. 3.11.

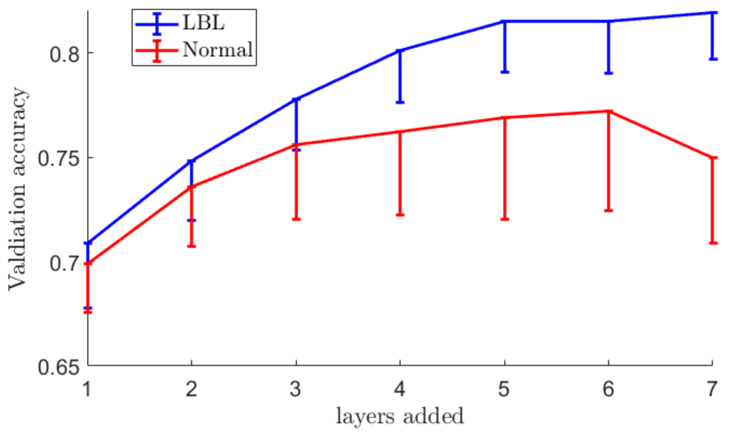


Figure 3.11: *The maximum validation accuracy achieved for various numbers of layers in the model for classification of cats and dogs. The validation accuracy was calculated at the end of each training epoch using 2000 separate test images (1000 of each class) and the max performance of achieved is plotted. The errorbars represent the average max performance achieved by each of the five models. Each model was trained for 80 epochs. The dataset used is the one described above with the numbers of images limited to 2000.*

Judging from the accuracy as function of training epochs both models have saturated before the full 80 epochs, see fig. 3.12. Suggesting that further training would not increase the accuracy more.

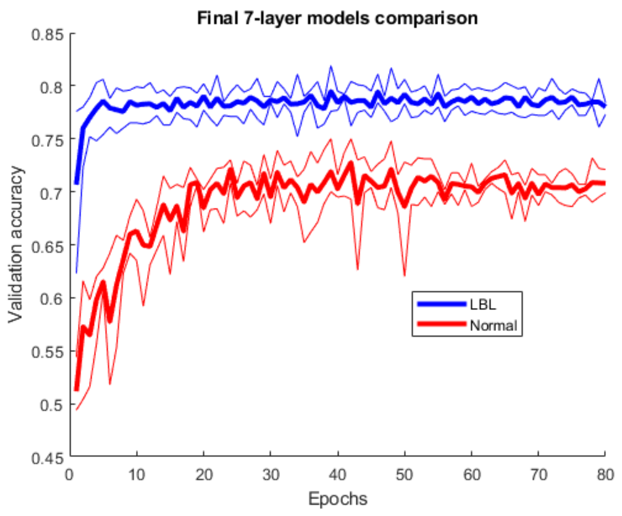


Figure 3.12: *Performance of the final model in the cats vs dogs task both for LBL and normal training as function of the number epochs the networks has been trained (final model in the case of LBL). In bold is the average performance as measured in each epoch over five separate runs. The narrow lines indicate the best and worst performance respectively.*

As previously mentioned, see section 2.4.2, a common method for improving the results when little data is available is to use data augmentation. Because of this the task is also tried with the use of data augmentation both for the LBL and the regular training scheme. We once again make a comparison when each layer is trained for 80 epochs. Then the normal training sees a large gain in performance relative to LBL fig. 3.13. But both training methods benefit from data augmentation, see fig. 3.15.

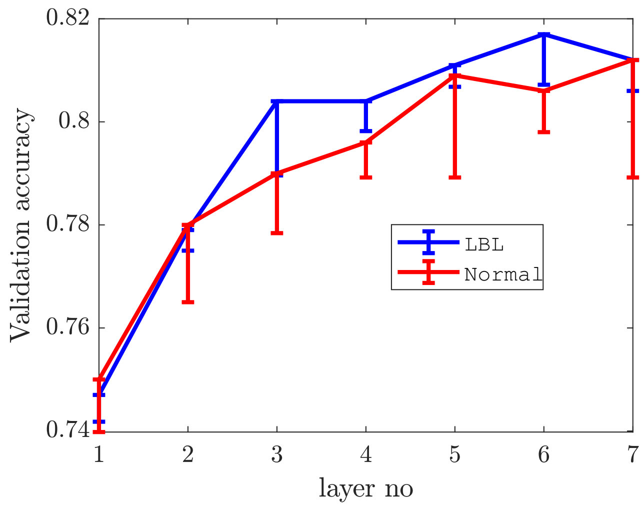


Figure 3.13: *The maximum validation accuracy achieved for various numbers of layers in the model for classification of cats and dogs when the models where trained on augmented data. The validation accuracy was calculated at the end of each training epoch using 1000 separate test images (500 of each class) and the max performance of achieved is plotted. The errorbars represent the average max performance achieved by each of the five models. Each model was trained for 80 epochs. The dataset used is the one described above with the numbers of images limited to 2000.*

When data augmentation is used it does not seem to be sufficient with 80 training epochs, see fig. 3.14. After 80 epochs the normal training method is still improving.

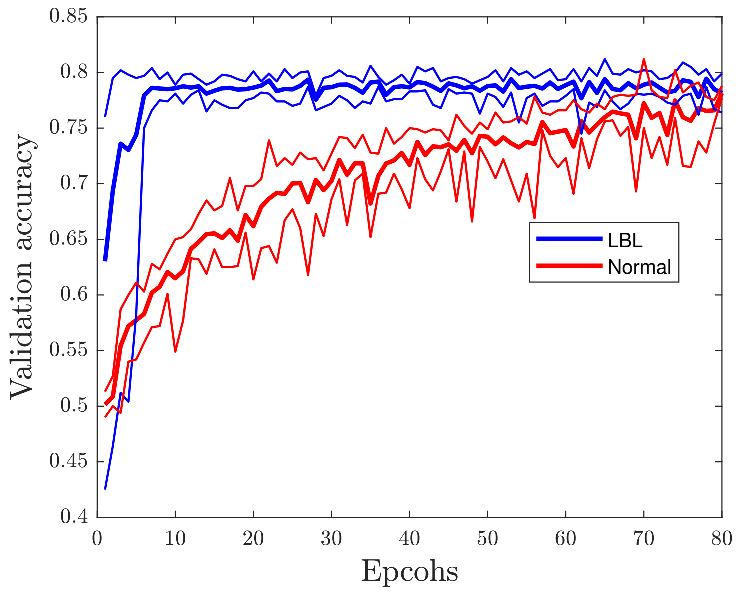


Figure 3.14: *Validation accuracy of the full model in the cats vs dogs task both for LBL and normal training when data augmentation was used. Plotted as function of the number epochs the networks has been trained (final model in the case of LBL). In bold is the average performance as measured in each epoch over five separate runs. The narrow lines indicate the best and worst performance respectively.*

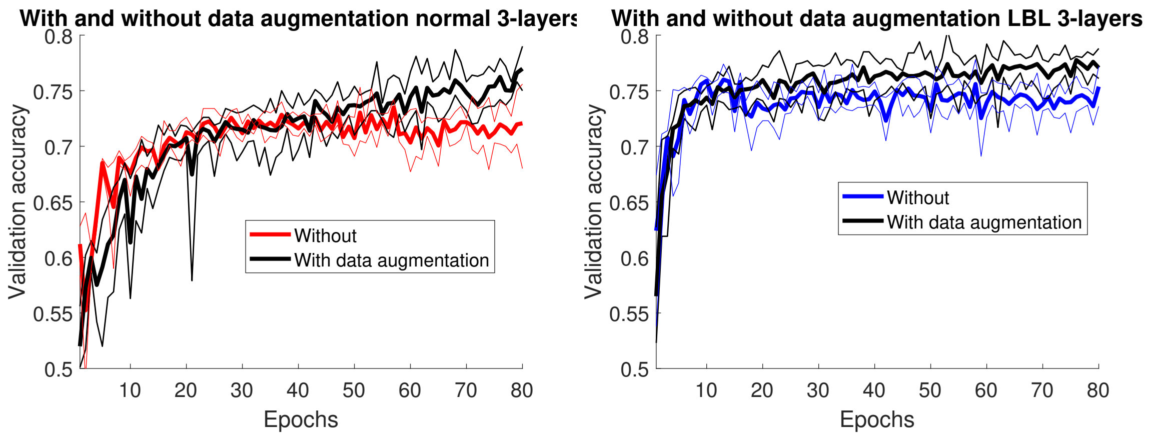


Figure 3.15: *Comparison of performance for augmented and non augmented data. In section 3.4.4 the comparison is made for normal training and in section 3.4.4 it is done so for LBL training. In both cases a small network of three hidden layers is used (only the first three convolutional layers and an output layer, see fig. 3.9). The thick lines are the medium performance and the thinner lines represent max and min performance respectively.*

Because 80 epochs may not be enough to saturate the models a run is also made with 200 epochs of training for LBL and the regular method. In fig. 3.16 we see the result of this. Now, for the first time, normal training clearly outperforms LBL.

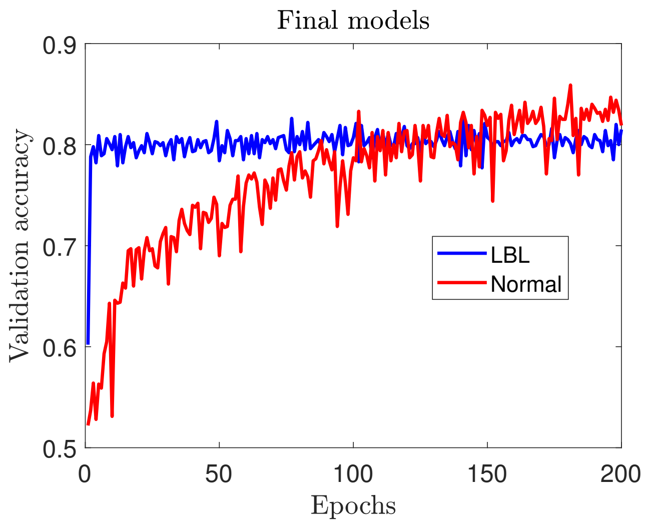


Figure 3.16: *Final models when trained for 200 epcochs, both LBL and normal training. Both perform slightly superior to when trained for 80 epochs, see fig. 3.14.*

Intermediate activations in LBL training

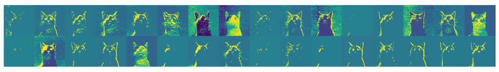
It is interesting to understand how the networks produced by the LBL training method differ from those trained in a regular fashion. Put differently, how do we influence the transformations performed by the layers in the model by only adding a single layer at a time?

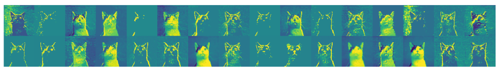
For convolutional networks there are some standard ways of investigating this as described in [20] including "intermediate activations", "visualizing convnet filters" and "visualizing heatmaps of class activations in an image". Here we will employ only the first method "intermediate activations".

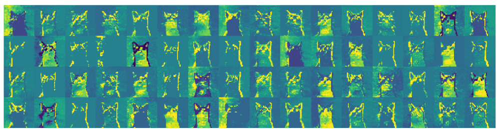
The "intermediate activations" consists of running a sample image through the network and as the name suggests looking at how the intermediate (convolutional followed by maxpooling) layers transforms the input. As previously mentioned in section 3.4.3 there is a common notion that early layers learn to detect rough features and that later ones learn more abstract and class specific features [20]. This works since all the layers are trained at once meaning that the different layers may take on different parts of the analysis.

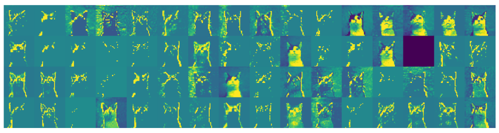
In LBL training each convolutional layer is trained to maximize the prediction of the subsequent dense prediction layer. This means that early layers learn completely independently from later ones and that they seek to maximize the information available for the output. Which also means that the hierarchical way in which normal convolutional networks transform the data likely is not how LBL trained networks do so. Visualizing the intermediate outputs is a way to compare the information flow for the two different methods.

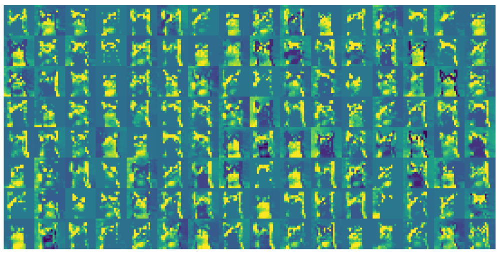
In fig. 3.17 the intermediate activations of all the different pooling layers are shown for a randomly chosen example image, (the example image in fig. 3.10). The last layer is essentially impossible to interpret since by then the image has been down sampled to be only 2×2 pixels. However there are some noteworthy general differences in the earlier layers between the LBL and normal, LBL seem to detect more features, i.e the activation images have less blank areas, compare fig. 3.17e and fig. 3.17f. The dark squares in some of the activations are caused by the local field being too small to activate the node.

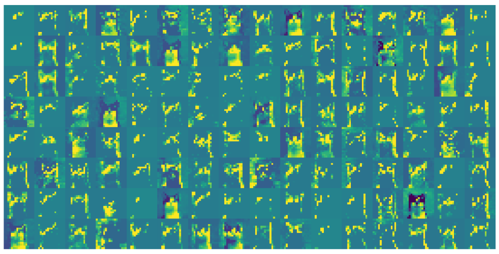

(a)

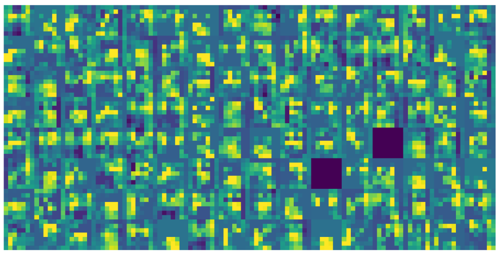

(b)

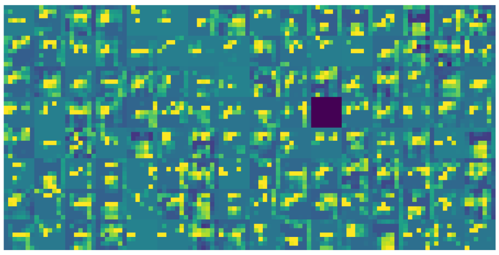

(c)

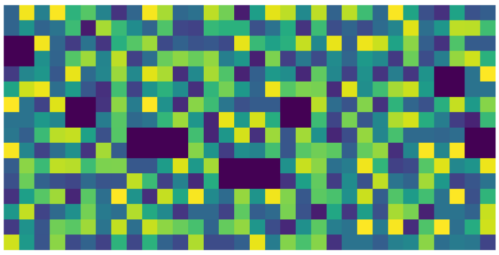

(d)


(e)


(f)


(g)


(h)


(i)

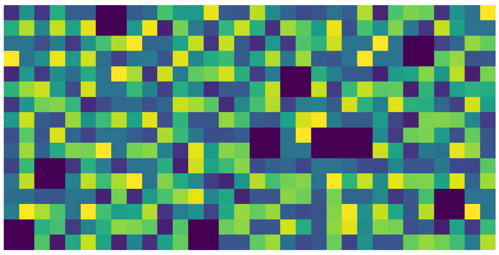

(j)

Figure 3.17: *Example of the activations of each feature map in the convolutional layers after going through the following maxpooling. Both the models were trained for 200 epochs. Figures a,c,e,g and i are from the LBL model and b,d,f,,h and j are from the normally trained one. Figures a and b are from the first layer, c,d the second etc. The image used is the cat from fig. 3.10. The different node activations are displayed on a grid 16 images wide and with height adjusted to fit the number of nodes in the layer.*

Local field activations

In section 3.4.2 it was shown that LBL training may increase the number of nodes having positive local fields. We perform a similar comparison for the Cats and Dogs network by looking at the average number of active nodes, or equivalently inactive, when the models are fed the test data of 1000 images. A node being inactive means that its activations is zero which in the visualization in section 3.4.4 fig. 3.17 would correspond to a dark image. As can be seen in fig. 3.18 LBL once again has slightly larger number of activations.

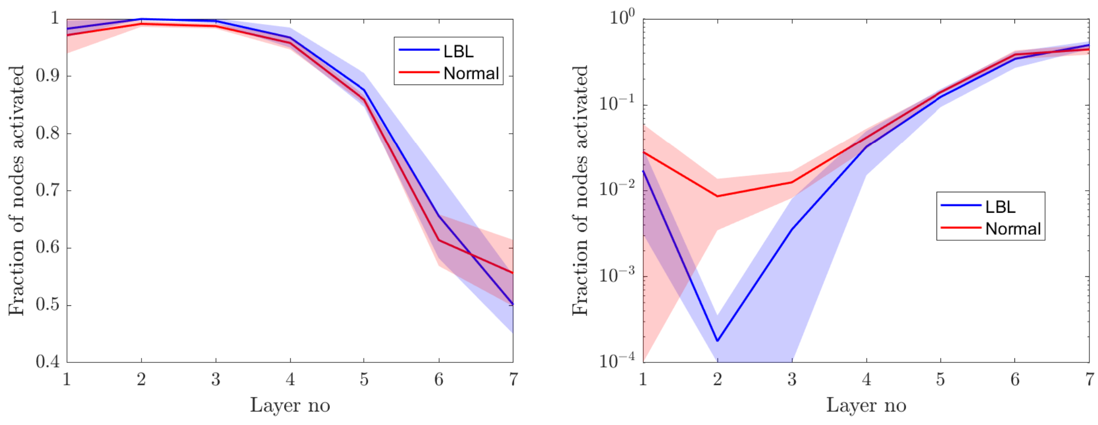


Figure 3.18: *Activation likelihoods for the various layers, excluding output, of the final models trained for the cats and dogs task. Both in linear scale section 3.4.4 and logarithmic section 3.4.4.*

3.5 LBL for estimation

Estimation differs from classification in that there is a quantity that is to be estimated from the data rather than a label to be assigned. The estimation task investigated is that of particle tracking in images. For this the framework *DeepTrack* developed in [1] is used.

3.5.1 DeepTrack

DeepTrack is a deep learning framework for particle tracking developed by [1]. It is useful for both single and multi-particle tracking and allows for easy training of networks for specific tasks. It does so largely by providing a image generator for simulating training data but also a framework for which to create and train deep learning networks for particle tracking.

The parameters of the generated images can be easily tuned. These include signal to noise ratio, particle size, background gradient and the area in which the particle is able to appear in. The particle shape is described by a Bessel function since it is identified as diffraction patterns of points sources. Typically these parameters are not fixed but rather chosen from a distribution for which one choose the parameters. The order of the Bessel function is also used as a parameter to further widen the range of possible input data. This means that the training data can be automatically varied while still resembling the experimental input data. Example images can be seen in fig. 3.19 below.

The networks are trained to predict particle positions within the images. Also images without particles may be included for which the network is trained to predict that it is outside the image. This is so that one does not need to know beforehand if there is a particle in the image or not. Which is useful especially when tracking multiple particles in a large area. The loss function used is the MSE see eq. (2.5).

DeepTrack is quite well suited for evaluating performance on estimation tasks because it enables both changing the parameters of training and also evaluation of performance in different conditions, such as low and high noise levels or very large amounts of training data. Perhaps more importantly it is also a highly relevant framework in the field of digital video microscopy in which particle tracking is of key importance for several types of experiments, such as diffusion coefficient measurements [35], [36].

The networks trained using DeepTrack are (typically) convolutional without activation function in the output layer. The output layer is just a scalar product between the weight vector and the output from the previous layer. In this thesis we will look at alternative training methods for the networks such as LBL.

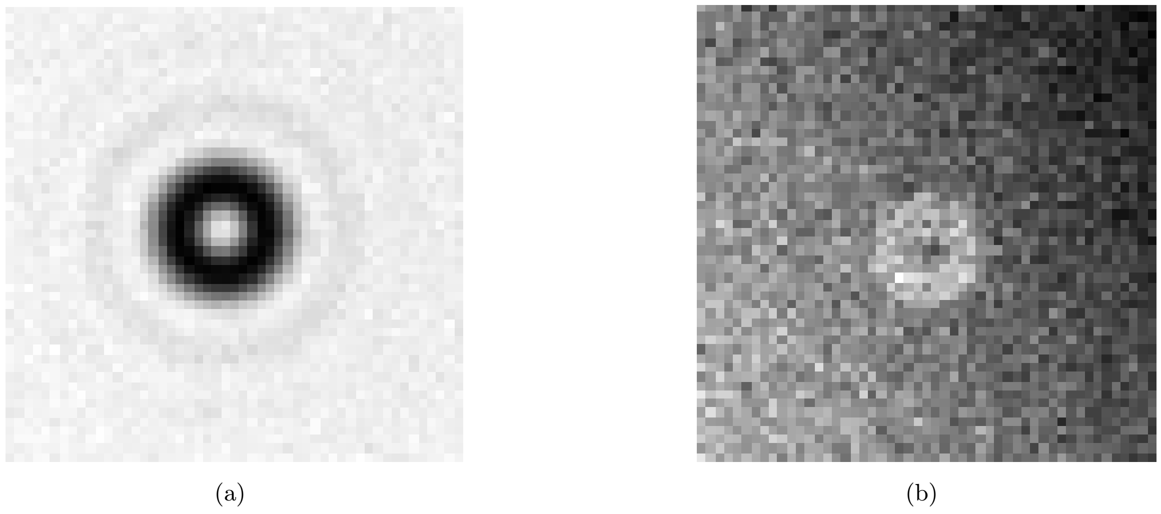


Figure 3.19: *Example images sampled from typical training images of DeepTrack. In fig. 3.19a with a SNR of 100 and in fig. 3.19b with a SNR of 10. The task is to locate the center of the particle. The images are* 51×51 *pixels.*

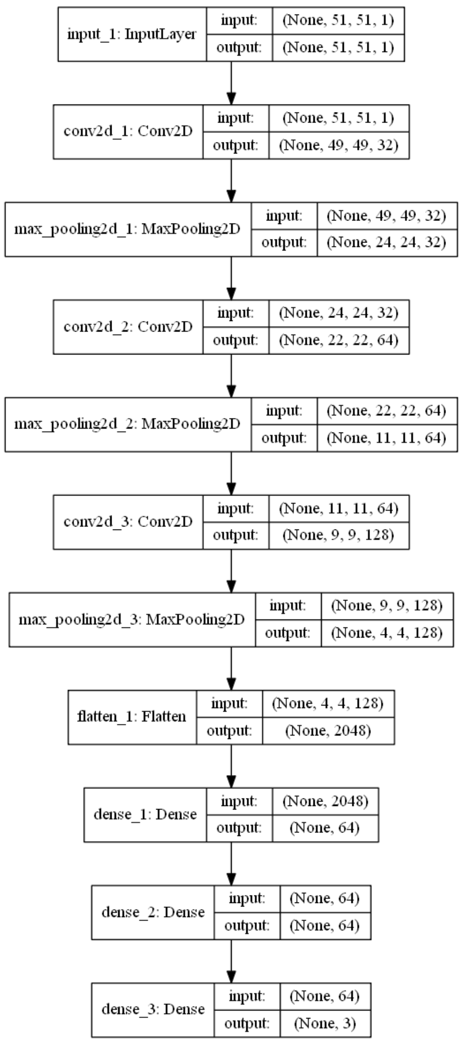

input_1: InputLayer
input: (None, 51, 51, 1)
output: (None, 51, 51, 1)
conv2d_1: Conv2D
input: (None, 51, 51, 1)
output: (None, 49, 49, 32)
max_pooling2d_1: MaxPooling2D
input: (None, 49, 49, 32)
output: (None, 24, 24, 32)
conv2d_2: Conv2D
input: (None, 24, 24, 32)
output: (None, 22, 22, 64)
max_pooling2d_2: MaxPooling2D
input: (None, 22, 22, 64)
output: (None, 11, 11, 64)
conv2d_3: Conv2D
input: (None, 11, 11, 64)
output: (None, 9, 9, 128)
max_pooling2d_3: MaxPooling2D
input: (None, 9, 9, 128)
output: (None, 4, 4, 128)
flatten_1: Flatten
input: (None, 4, 4, 128)
output: (None, 2048)
dense_1: Dense
input: (None, 2048)
output: (None, 64)
dense_2: Dense
input: (None, 64)
output: (None, 64)
dense_3: Dense
input: (None, 64)
output: (None, 3)

Figure 3.20: *Architecture used in DeepTrack summarized using the Keras utils function plot_ model [1]. It is the same architecture that is used as basis for the growing networks techniques.*

Since the data is generated we can in principle train on arbitrarily large number of images. In order to limit the number of parameters to vary we set a fixed number of images delivered in a fixed way for training. We use $4000, 3000, 2000, 1000, 500$ batches of $8, 32, 128, 512, 1024$ images in each for all training unless otherwise specified. This is the same as used in [1]. For similar reasons we fix the values which the other parameters of the images can have, such as range of particle radii, to also be the same as those used by [1]. All the images for training and evaluation are generated upon runtime so that the same image is not fed to the network for training twice. This also guarantees that when the network is later evaluated it is done so on a completely new set of images which practically eliminates the risk of the network parameters being accidentally fitted to the validation data, which can become a problem if the when evaluating multiple times on the same data [20].

Results

The DeepTrack networks are evaluated by measuring the MAE of their predictions. In general LBL training shows results comparable to those of regular training, see fig. 3.22. There is also shown to be a strong dependence on the number of layers added to the models, see fig. 3.21.

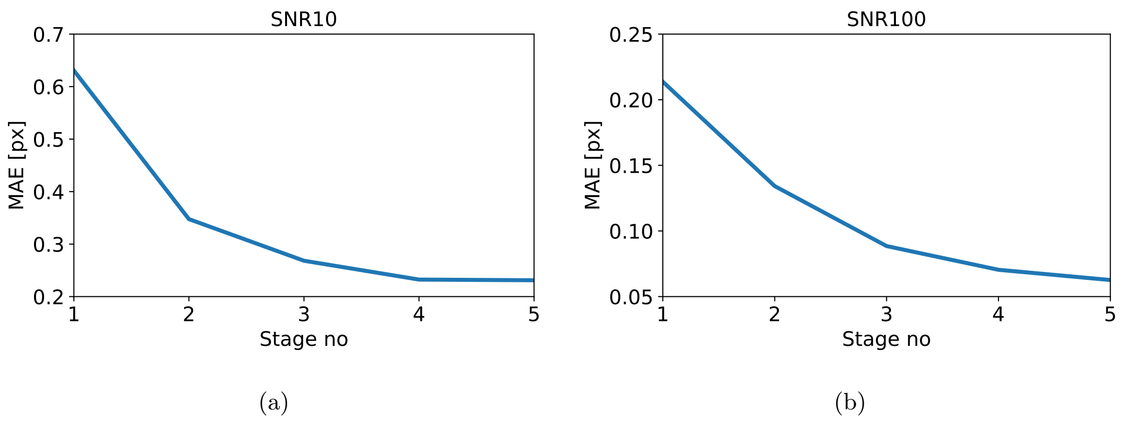


Figure 3.21: *Performance measured by MAE of LBL training as more layers are added to the model evaluated on two different signal to noise ratios. In fig. 3.21a there is a signal to noise ratio of 10 while in fig. 3.21b the signal to noise ratio is 100. The full architecture, reached at 5 hidden layers, is the same as for DeepTrack which is shown in fig. 3.20.*

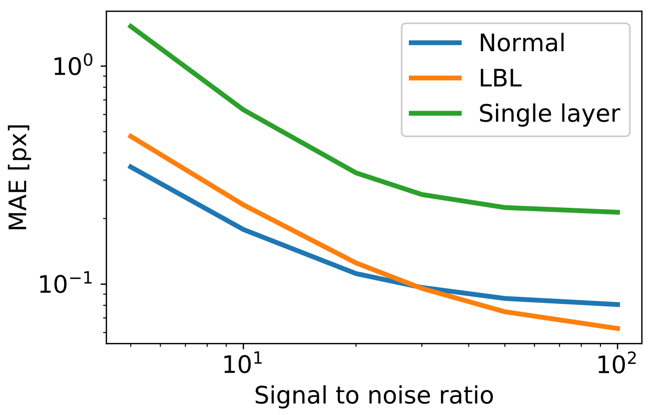


Figure 3.22: *The performance for different SNR of a LBL trained network and a regularly trained network. Plotted is also the first single layer network for comparison. The architecture of the final networks is that shown in fig. 3.20.*

3.5.2 Results on experimental data

To further demonstrate the effectiveness of the DeepTrack networks they are also presented to experimental data. More specifically they are tasked to track an optically trapped particle, see fig. 3.23. Two different videos are used one with rather good illumination and one with rather poor. These videos are the same as used by [1]. The original videos are 120×120 pixels but are down sampled to 51×51 so that they can be fed to our already trained networks without alteration of the network structure. We use the 100 first frames of each video for evaluation.

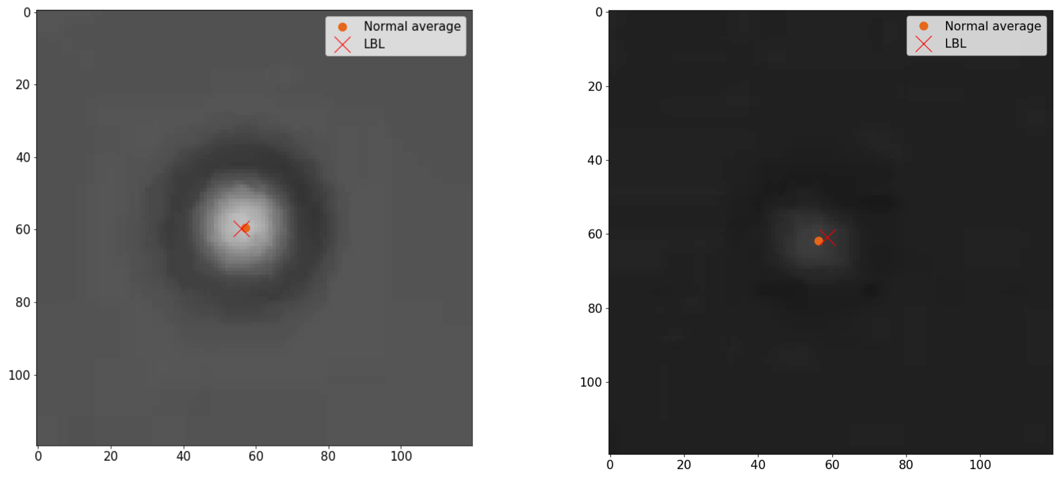


Figure 3.23: *Example images from the experimental data. In section 3.5.2 with good illumination and in section 3.5.2 with poor illumination. The red cross represents the prediction made by a single layer LBL network and the orange dot the prediction from an ensamble of normal networks.*

Due to that we now are looking at experimental data this means that we do not have access to the ground truth. Which in turn makes it non-trivial to evaluate the performance of the different algorithms. However an ensemble of different networks tend to perform better [37], [38].

Because of this we evaluate the LBL networks by looking at how close they come to the ensemble prediction from 10 normal networks, standard DeepTrack architecture and training. Thus we look at the average deviation from the ensemble average as a metric of performance. Graphically this corresponds to the average distance between the cross and the dot in fig. 3.23. The result of this is shown fig. 3.24 below. Both for good and poor illumination the performance seem to increase as more layers are added until it is comparable to the standard networks. This can be seen from that both for the good and for the poor illumination the average deviation from the ensemble come very close to the average deviation as measured by the normally trained networks.

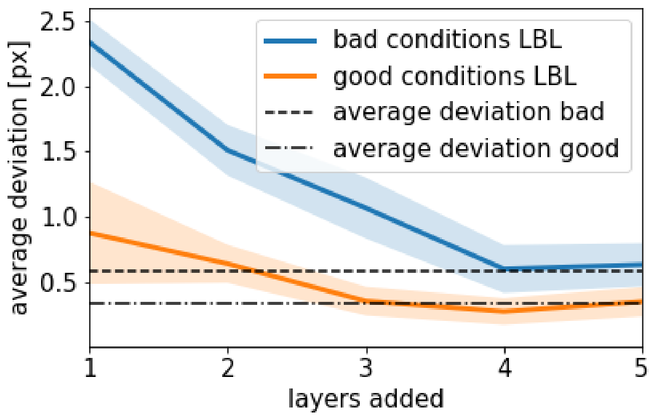


Figure 3.24: *The performance of the LBL networks on experimental data as measured by the average distance between prediction from the individual LBL networks and the ensemble average of ten full networks trained normally. The shaded areas represents the standard deviation as calculated over the ten separate LBL networks. The two horizontal lines represent the mean deviation from the ensemble by the normal networks and act as a benchmark for the LBL algorithms.*

Chapter 4

Modular networks

As discussed in section 2.5 and section 2.6 a drawback with ANNs is that they are quite demanding to execute and train. Smaller networks are less demanding to execute but also less accurate due to their smaller representation space. This means that there is a tradeoff between accuracy and computational complexity. Ignoring for the moment the instabilities which can arise when training very large networks.

This opens up a potential application for layer by layer training as a way to bridge the gap between computational complexity and prediction accuracy. Limiting the need to have multiple different networks for similar tasks with different demands. With early layers used for fast execution and later ones for more accurate one.

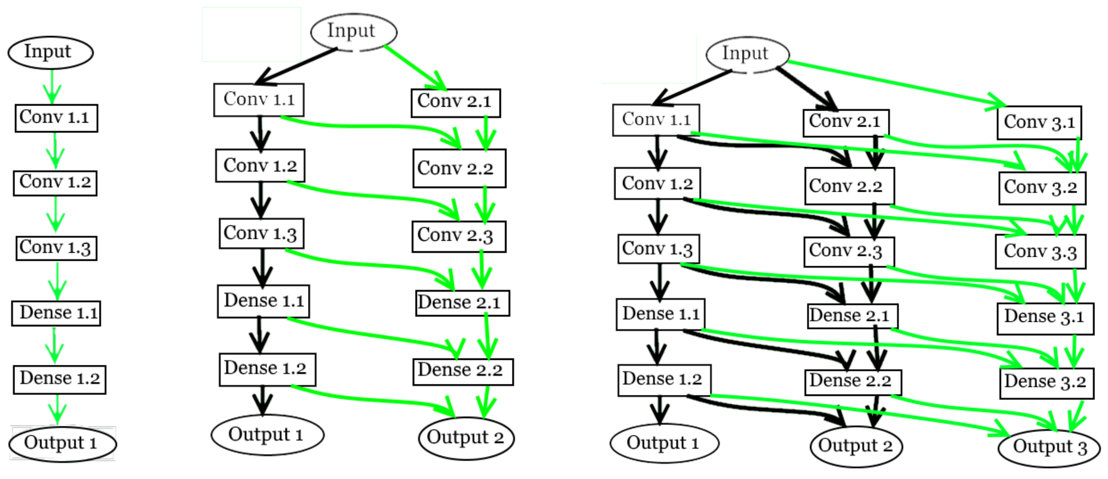
One is not strictly limited to using layer by layer training for this. Another way to increase the network size is to increase the network width. How to do this is non-trivial due to the many different possible methods which achieves this. One can for instance use an ensemble like approach in which one trains multiple networks of the same size and combine their predictions using averaging or add more nodes in each layers keeping the weights of the connections already trained. An approach which lies somewhere in between the variations described here is explored in section 4.1.

There are many possible applications for networks which are tuneable in size. For instance in video microscopy one might want to follow a single particle, or a set of particles, by moving the microscope relative to the sample. To do this in real time requires a fast but not necessarily extremely accurate algorithm. But a more accurate but slower algorithm could be used to create a accurate description of the particle(s) trajectories.

4.1 Breadth-wise growth of networks

There are multiple ways in which one may expand a ANN. In chapter 3 LBL training was introduced which increase the network depth by adding more layers. A natural alternative to this is to expand the networks width. How this is best done is not trivial since there are very many different options.

The algorithm we have opted for is designed to keep the modularity, i.e not changing the weights already optimized while still not compromising performance.



(a) (b) (c)

Figure 4.1: *Schematical representation of the training of the first three stages in (modular) breadth wise growth. The arrows in green represent the connections being trained while those in black represent connections which has already been trained and are therefore kept fix. In fig. 4.1a the first network with few nodes are trained. In fig. 4.1b another network is attached to the first and trained and lastly in fig. 4.1c a third network is attached and trained. The procedure can be repeated for an arbitrary number of new networks.*

4.1.1 Implementation

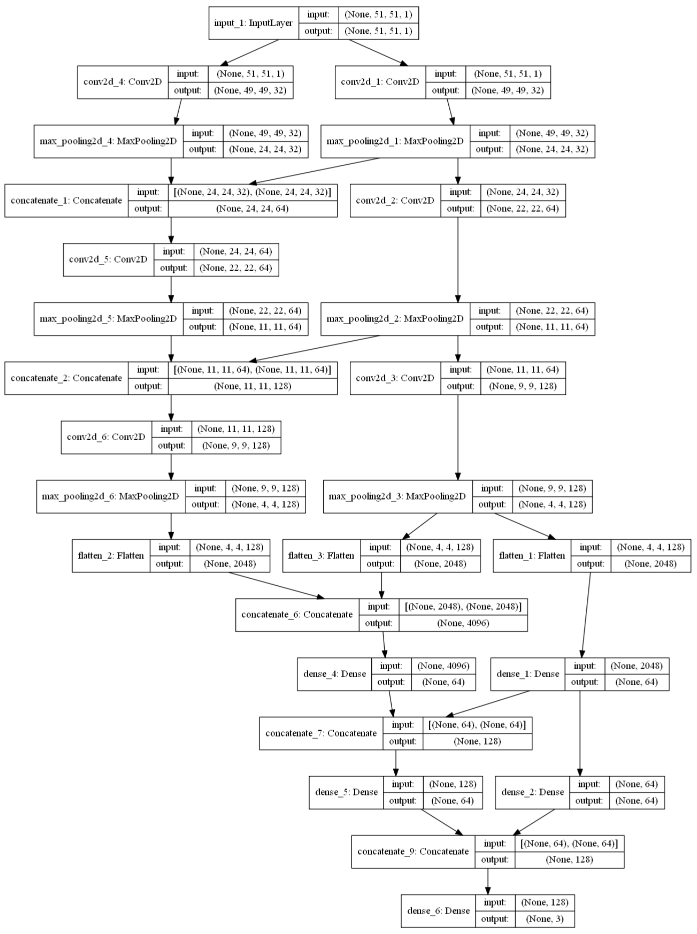
We start with a narrow network, that is one with relatively few nodes per layer, and then gradually expand it by adding more nodes to each layer as well as a new output layer. The new nodes are connected to all nodes in the previous layer, including the newly added ones. How this is done for first three steps with DeepTrack networks is shown in fig. 4.1. Note that one can still use the narrow, less computationally demanding network even after having attached the new ones.

The breadthwise training scheme resembles that of LBL training in that first a small network is trained. There are still a number of parameters to choose for the networks. Most importantly perhaps the number of nodes in each layer of each network. To simplify the analysis somewhat we try and keep the architecture structurally similar to that used before for DeepTrack. Specifically the same ratio between the sizes of the different layers is used. For the same reason the new networks attached have the same number of nodes in each layer as the first network. For detailed information of the number of nodes in each layer see table 4.1.

Plotting the the network when it has been grown one stage reveals the connections used, see 4.2.

Network	Conv 1	Conv 2	Conv 3	Dense 1	Dense 2
DeepTrack	16	32	64	32	32
Base 32	32	64	128	64	64
Base 16	16	32	64	32	32
Base 8	8	16	32	16	16
Base 4	4	8	16	8	8

Table 4.1: *The number of hidden nodes in each subnetwork used in the breadth growth scheme for particle tracking. See fig. 4.1 . Using even fewer nodes in the network would pose a risk of causing an information bottleneck greatly inhibiting learning.*



input_1: InputLayer | input: (None, 51, 51, 1) | output: (None, 51, 51, 1)
conv2d_4: Conv2D | input: (None, 51, 51, 1) | output: (None, 49, 49, 32)
conv2d_1: Conv2D | input: (None, 51, 51, 1) | output: (None, 49, 49, 32)
max_pooling2d_4: MaxPooling2D | input: (None, 49, 49, 32) | output: (None, 24, 24, 32)
max_pooling2d_1: MaxPooling2D | input: (None, 49, 49, 32) | output: (None, 24, 24, 32)
concatenate_1: Concatenate | input: [(None, 24, 24, 32), (None, 24, 24, 32)] | output: (None, 24, 24, 64)
conv2d_2: Conv2D | input: (None, 24, 24, 32) | output: (None, 22, 22, 64)
conv2d_5: Conv2D | input: (None, 24, 24, 64) | output: (None, 22, 22, 64)
max_pooling2d_5: MaxPooling2D | input: (None, 22, 22, 64) | output: (None, 11, 11, 64)
max_pooling2d_2: MaxPooling2D | input: (None, 22, 22, 64) | output: (None, 11, 11, 64)
concatenate_2: Concatenate | input: [(None, 11, 11, 64), (None, 11, 11, 64)] | output: (None, 11, 11, 128)
conv2d_3: Conv2D | input: (None, 11, 11, 64) | output: (None, 9, 9, 128)
conv2d_6: Conv2D | input: (None, 11, 11, 128) | output: (None, 9, 9, 128)
max_pooling2d_6: MaxPooling2D | input: (None, 9, 9, 128) | output: (None, 4, 4, 128)
max_pooling2d_3: MaxPooling2D | input: (None, 9, 9, 128) | output: (None, 4, 4, 128)
flatten_2: Flatten | input: (None, 4, 4, 128) | output: (None, 2048)
flatten_3: Flatten | input: (None, 4, 4, 128) | output: (None, 2048)
flatten_1: Flatten | input: (None, 4, 4, 128) | output: (None, 2048)
concatenate_6: Concatenate | input: [(None, 2048), (None, 2048)] | output: (None, 4096)
dense_4: Dense | input: (None, 4096) | output: (None, 64)
dense_1: Dense | input: (None, 2048) | output: (None, 64)
concatenate_7: Concatenate | input: [(None, 64), (None, 64)] | output: (None, 128)
dense_5: Dense | input: (None, 128) | output: (None, 64)
dense_2: Dense | input: (None, 64) | output: (None, 64)
concatenate_9: Concatenate | input: [(None, 64), (None, 64)] | output: (None, 128)
dense_6: Dense | input: (None, 128) | output: (None, 3)

Figure 4.2: *Architecture when a second network has been attached to the first one, shown in fig. 3.20. Summarized using the Keras utils function plot_model.*

4.1.2 Computational complexity

Highly relevant if one is to use the modular network for tasks with high demand on execution time is that their computational complexity is not significantly greater than that of regular networks of similar performance. Because of this we look at the difference in computational complexity between normal networks and breadth-grown networks.

The final network resembles that of one trained using conventional methods. But is slightly smaller for a network with the same number of nodes due to the lesser number of connections (and therefore also weights), nodes can only be connected to nodes already in the network or which are added at the same time, see fig. 4.3 and fig. 4.1. This means that the number of connections, and thus also the computational complexity, behaves as an arithmetic sum as more networks are connected. The number of connections will thus be roughly half that of an ordinary network with the same number of hidden nodes. The exact complexity depends on the width of the individual networks.

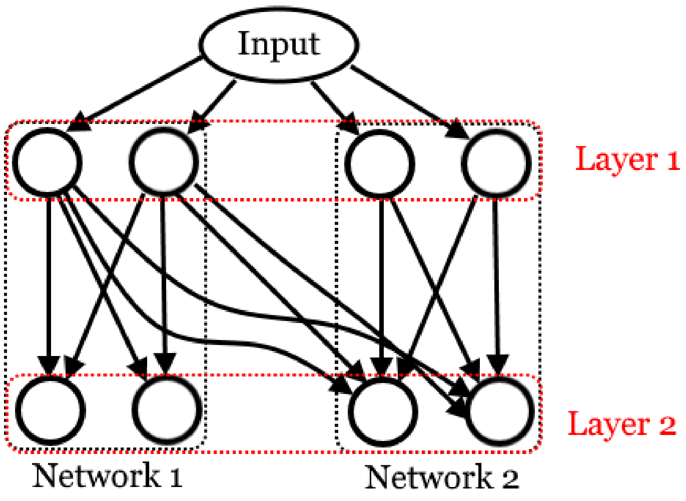


Figure 4.3: *Schematic of the connections in the first two layers of the first two networks when expanding network width.*

Assuming that all the networks $n = 1, 2, 3...M$ are of the same size and that layer L_i^n has N_i nodes. Then the computational complexity of the new layer i is proportional to $i * N_{i-1}$. Thus the computational complexity of all layers at stage i is

$$K_{i,M}^{breadth} \propto N_i \sum_{n=1}^{M} nN_{i-1} = N_i N_{i-1} \frac{M(M+1)}{2} \tag{4.1}$$

Where we in the last stage used that we have an arithmetic sum. Remember that for a regular network the computational complexity is proportional to the number of connections, which would with the same notation be

$$K_{i,M}^{breadth} = MN_{i-1}MN_i = M^2 N_i N_{i-1} \tag{4.2}$$

In theory then a modular network should be close to half as computationally complex as a regular network with the same number of nodes in the layers, assuming that M is large. However this may not directly translate into significantly faster execution unless the underlying implementation, in this case Keras, is optimized for this architecture type.

4.1.3 Results

For the breadth growth there is one important parameter which vary namely the width of the individual networks. As an alternative to breadth growth we look also at the performance of normally trained convolutional networks of various network sizes (nodes per layer), see fig. 4.4 below. There is no clear trend towards greater performance for larger networks. The larger configurations struggle with performing well.

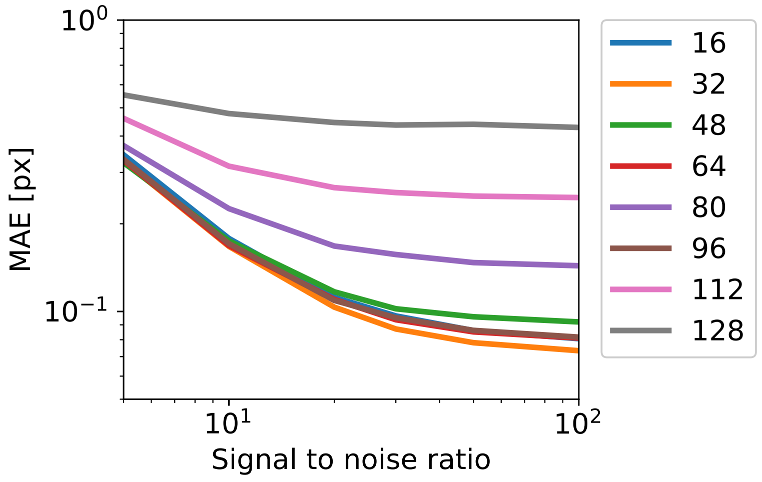


Figure 4.4: *MAE as function of signal to noise ratio for various sizes of networks trained normally. The results are the mean performance of 9 independently trained networks of each size. The training procedure was the same as for normal DeepTrack.*

Next we do a similar comparison for breadth grown networks while also including the first network i.e normal networks of various size for comparison see fig. 4.5. There is a clear decrease in the MAE for all networks when they are grown. The test is configured so that the final (grown) model of each different base-width has the same number of nodes in each stage.

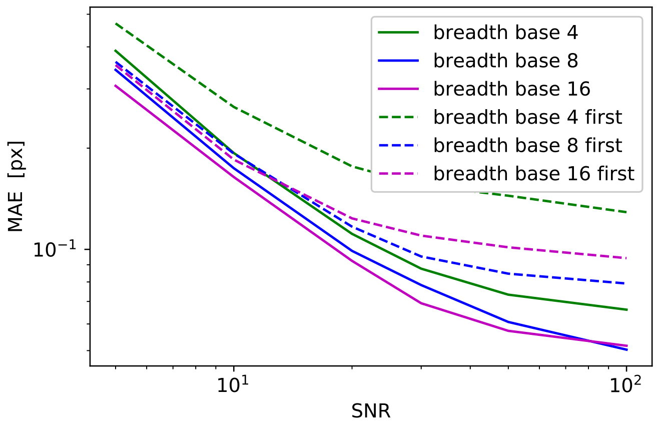


Figure 4.5: *Figure comparing the MAE as function of signal to noise ratio (SNR) of breadthwise growth for various widths of the networks. Both the first (narrow) network and the final network are displayed. The final networks are all with the same number of hidden nodes. The 32 width first network is the same architecture as used by in DeepTrack [1]. There is a slight variation in performance when the experiment is repeated.*

We look also at the improvement as more networks are added, see fig. 4.6 and fig. 4.7. In order to more easily compare the different base widths we plot this as a function of the number of nodes in the first stage. Which would correspond to the number of nodes in the first layer in a normal convolutional network. In all cases there is some decrease in the MAE as the networks are grown.

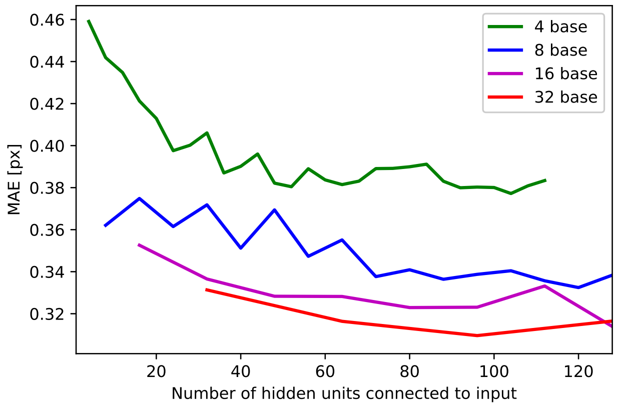


Figure 4.6: *The performance of breadthwise growth as more layers are added for a signal to noise ratio of 5. The x-axis indicates the number of nodes directly connected to input, corresponding to the size used to label the conventional networks in fig. 4.4.*

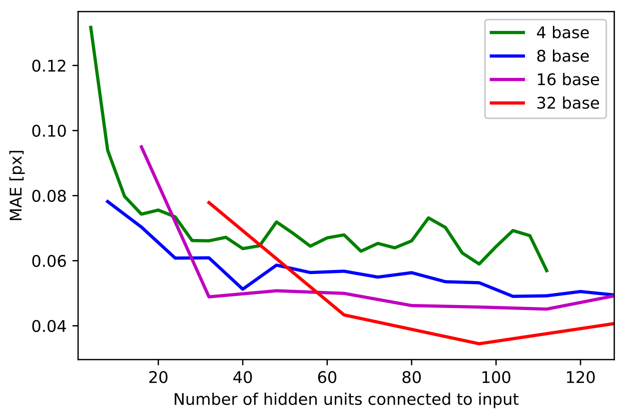
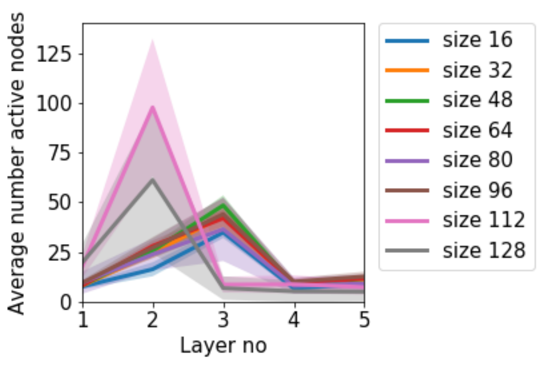


Figure 4.7: *The performance of breadthwise growth as more layers are added for a signal to noise ratio of 100.*

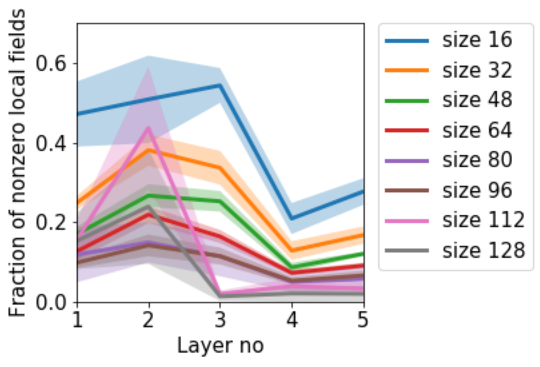
Number of active nodes

To compare the connections in the final architectures we look at the number of nodes in each layer which has positive local fields. The procedure is very similar to that in section 3.4.2 and section 3.4.4.

We start by looking at conventional convolutional networks of various sizes to have a baseline to compare to. The networks are trained using the standard parameters and all that is varied is between them is their number of nodes in each layer. The result is shown in fig. 4.8a. The average number of active nodes in each layer seem to be more or less independent on the network width. Meaning that the fraction of nodes which are activated become smaller and smaller which is clear also from fig. 4.8b. Once again the largest networks deviate, probably due to them failing to properly converge.



(a)

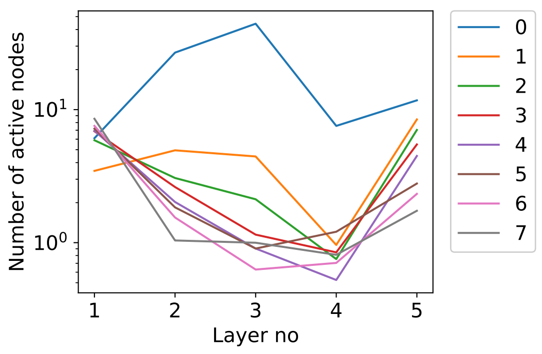


(b)

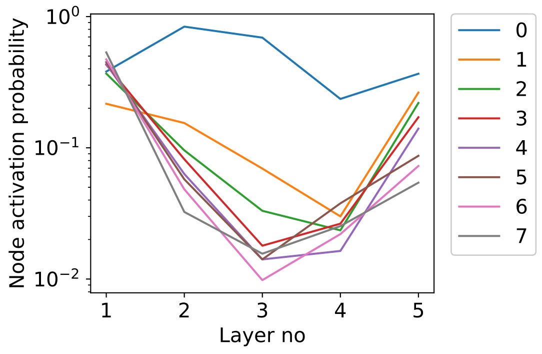
Figure 4.8: *Node activation for convolutional newtorks trained on DeepTrack. The networks were trained with the standard DeepTrack method as described in section 3.5.1. Average number of nodes being active in each layer for various network widths (nodes per layer) is shown in fig. 4.8a and the activation probability is shown in fig. 4.8b. The sizes indicate the number of nodes in the first layer. The average is taken over 10 000 images sampled from the same distribution as the training data and also averaged over 9 independent networks. The shaded areas indicates the standard deviations as calculated over the 9 different networks.*

In the case of layer by layer followed by breadth growth the first network is clearly the one which has the most activations, see fig. 4.9 below. So is also the case when only breadth growth is applied (i.e the first network is a trained all layers at once). Then the subsequent layers have even fewer activations.

In both cases the number of activations in each layer decrease as more layers are added through the breadth growth, especially for the convolutional layers (layer 1,2 and 3 in fig. 4.9 and fig. 4.10).

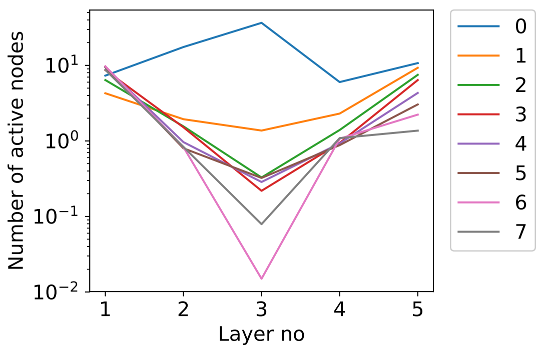


(a)

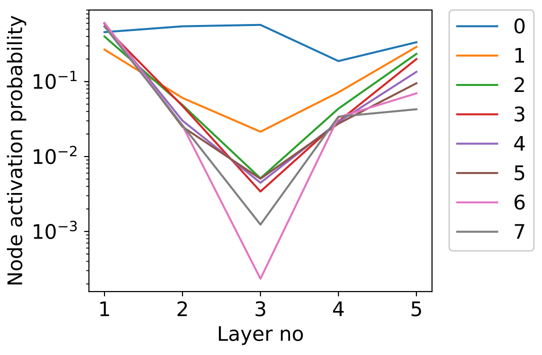


(b)

Figure 4.9: *Activations of nodes in the case of layer by layer training followed by breadth growth. The activations are shown for the different networks connected as a function of the layer number (i.e how far down the information pipeline the layer is). The zeroth model was trained using LBL. The first three layers are convolutional and the last two are dense layers. The number of nodes in the individual layers are the same as in DeepTrack.*



(a)



(b)

Figure 4.10: *Activations of nodes in the case of only breadth growth. The activations are shown for the different networks connected as a function of the layer number (i.e how far down the information pipeline the layer is). The first three layers are convolutional and the last two are dense layers. The number of nodes in the individual layers are the same as in DeepTrack.*

4.2 Fast multi-particle tracking

Multi-particle tracking consists of tracking an unknown number of particles in a video or image file, often particles of different types. It is a significantly more challenging problem than single particle tracking due to that it is not (in general) beforehand known how many particles, or more generally objects, there are to track. But also due to the large area in which they can appear in an the fact that the particles can overlap. Multi-particle tracking is something that the modular networks might be well suited for performing quickly by combining the speed of a small network with the accuracy of a big. DeepTrack has built in support for particle tracking which is why the frameworks is used also here for the multi particle tracking [1].

4.2.1 Sliding window with modular networks

Multi particle tracking is often performed by dividing the images which are to be tracked into overlapping segments and for each image segment detect if and where there is a particle in it, used by for instance [1]. This is called the sliding window approach and is used partly because it's ease of implementation. It is very demanding from a computational viewpoint since it requires running the network multiple times for each image and the same pixels are processed multiple times in various image segments. Also combining the results from the segments can be computationally demanding especially if the overlap is large. Applying the modular networks could use the faster less accurate models to detect if there is a particle in the segment and then the more accurate networks only when they are really needed then reusing the convolutional output's already calculated.

This approach is most useful when the particles are sparsely relatively distributed in the image. If they are very densly packed then most of the scanning boxes will contain particles and the large network will need to be used most of the time anyhow. Here a proof of concept of this faster approach is demonstrated.

4.2.2 Implementation of enhanced detection algorithm

We use LBL networks to implement a quick multi-particle tracking using the sliding window algorithm as base. A small single hidden layer network is used to make a first prediction on the images. Then a threshold T_1 is applied to the r value (distance from center) prediction to determine when a more second prediction is merited. This threshold is chosen such that the number of extra predictions is roughly three times the number of particles.

If the predicted r value is within the threshold then the second prediction is made. In order to determine which of the predictions were of an actual particle and which were caused by some sort of noise a second threshold, T_2, is applied to the r values of the predictions from the second network. This is needed because we want to use the larger network to decide if a particle was detected or not. The remaining predictions after applying the second threshold are considered to be detected particles.

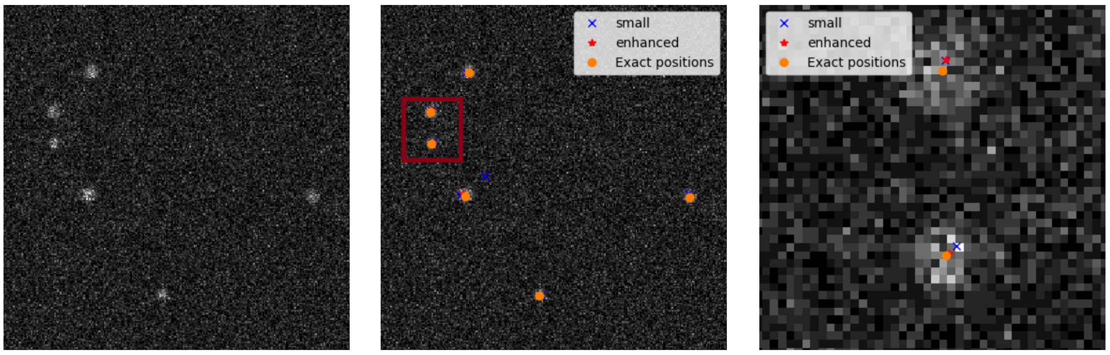
Since the prediction boxes overlap, which is necessary since we allow the particles to come close to one another, the same particle can be detected multiple times. To prevent this a new threshold is introduced on the distance between particles, D_p. If two predictions fall within D_p from another then they are deemed to belong to the same particle and the prediction with the smallest r value is chosen since the network is most confident on this prediction.

4.2.3 Performance

To evaluate the performance images of size 251×251 with 5 to 6 particles randomly distributed in the image are generated, a example image can be seen in fig. 4.11a. A scanning step ("box to box" distance) of 10 pixels is used both horizontally and vertically and each box is 51×51 pixels leading to a total of $21 \times 21 = 441$ boxes. The same training parameters as used in [1] for vesicle tracking are employed. The SNR fixed to 3 and the background intensity is kept at 0.2 this along with the particle positions and image size are the only distribution parameters changed from training and evaluation.

The prediction procedure described in section 4.2.1 and section 4.2.2 is used. Performance is measured by calculating the distances between the true particle positions and the corresponding closest predictions. We also count the false positives as the number of predictions made that are not the closest to any of the particles. The number of missed particles is also counted using a threshold on the minimum distances between true position and prediction. If the closest prediction is more than twice the particle radius from the particle center then the particle is not considered to be detected.

To confirm that network do indeed manage to replicate the performance of the large LBL network the performance of the single layer network and using only the full network is also measured. The predictions of these are made using the same algorithm as described in section 4.2.2 but without the extra step of making a second prediction with another network. Therefore these use only one threshold for the r value. Example of predictions made by the small and the enhanced network can be seen in fig. 4.11b. The close up in fig. 4.11c reveals how the larger networks serves to bring the prediction closer to the true value.



(a) (b) (c)

Figure 4.11: *fig. 4.11a shows an example of an evaluation image, with SNR of 3. fig. 4.11b shows an example of tracking of multiple particle with the prediction from the small single layer network, the prediction when enhanced with a larger network and the true positions marked. The red square to the left of the image fig. 4.11b is displayed in more detail in fig. 4.11c. There it is apparent how the enhanced version manages to improve upon the prediction of the smaller network.*

To compare the accuracy we look at the distributions of the errors, see fig. 4.12.

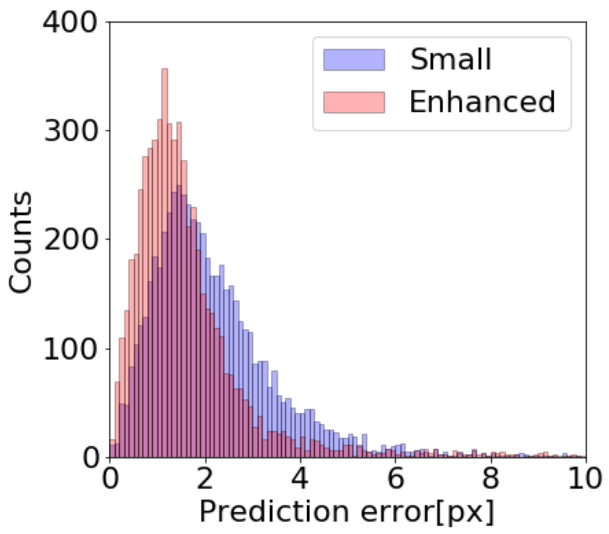


Figure 4.12: *The distribution of the errors of the predictions for multi particle tracking when using only a small network and when enhancing it with a larger network. The shift to smaller errors in the enhanced network indicates a greater accuracy.*

We quantify the performance by looking at the median and mean errors along with the number of false positive and false negative, see table 4.2. Also the results if only using the full network for all predictions is included for comparison. The false negative rates is small (missed predictions) for all the networks, roughly $1 - 2\%$. However the number of false positives is somewhat larger, especially for the small network, roughly 1 per 5 real particles and one per 20 real particles for the full network. In an experimental setting post processing can counteract the false positives looking at how the predictions differ from frame to frame.

Network	mean	median	false negatives %	false positives%
Small	2.32	1.91	1.2	17.0
Enhanced	1.74	1.36	1.6	4.2
Full network	1.71	1.36	1.3	6.8

Table 4.2: *Mean and median error for the three different tracking networks, the enhanced network combines the small and the large network predictions. The percentage of false negatives corresponds to the fraction of particles the networks has failed to detect. The false positive rate is the fraction of predictions which do not correspond to a actual particle. The evaluation was performed on 1000 images with a SNR of 3 each with 5 to 6 particles, with a total of 5865 particles to detect.*

4.3 Execution times and computational complexities

The execution time is measured by letting the networks make predictions on 200000 images and timing this. Only the prediction step is timed. From this we calculate how long time it took for the network to process a single image on average. These tests are performed both on a GPU, more specifically a NVIDIA Quadro K620, and on a CPU 10 threads on a cluster compute node, with Intel Xeon E5-2650v3 CPUs.

From eq. (4.1) we expect the computational complexity to increase with the square of the number of nodes in each layer. That is doubling the layers widths should quadruple execution time. In practice how the network is implemented and especially for the GPU the memory management is very important for the performance so we make the following anzats for the execution time

$$T(N) = a_0 + a_1 N + a_2 N^2 \tag{4.3}$$

Where N is the number of nodes in the first layer a_i are coefficients which we fit to the measurements. The relative size of the layers is the same for all the networks.

The result can be seen in fig. 4.13 below. Evaluating the performance of a algorithm accurately is difficult but these experiments should give a indication as to how the performance scales with the number of node in each layer. This is also why we use also a regular network rather than a just the modular to investigate the dependence.

We can expect the implementation of convolutional networks to be highly optimized for in TensorFlow and by extension Keras due to the large userbase of tensorflow. The modular networks, albeit technically less complex, may be less well optimized and therefore behave unexpectedly.

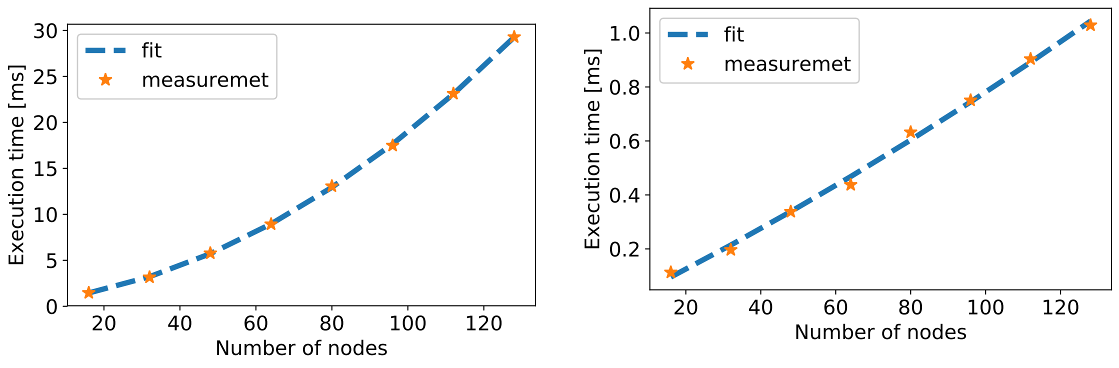


Figure 4.13: *Execution time per image of networks as function of network size when run on CPU,section 4.3 and when run on GPU,section 4.3. The GPU is a NVIDIA Quadro K620 and the CPU 10 threads on a Intel Xeon E5-2650v3. The fits are least square fits of second degree polynomials made using the scipy function curve_fit.*

For the LBL network we also perform an estimate of the computational complexity using eq. (2.16), eq. (2.15) and eq. (2.17) in combination with eq. (2.18). These estimations do not include the computational cost of the activation function. The contribution from these should however be small since the number of calls to the activation functions is proportional to the number of nodes in the layers rather than the input size. As can be seen in section 4.3 the execution time and computational complexity agree well with the exception of an offset.

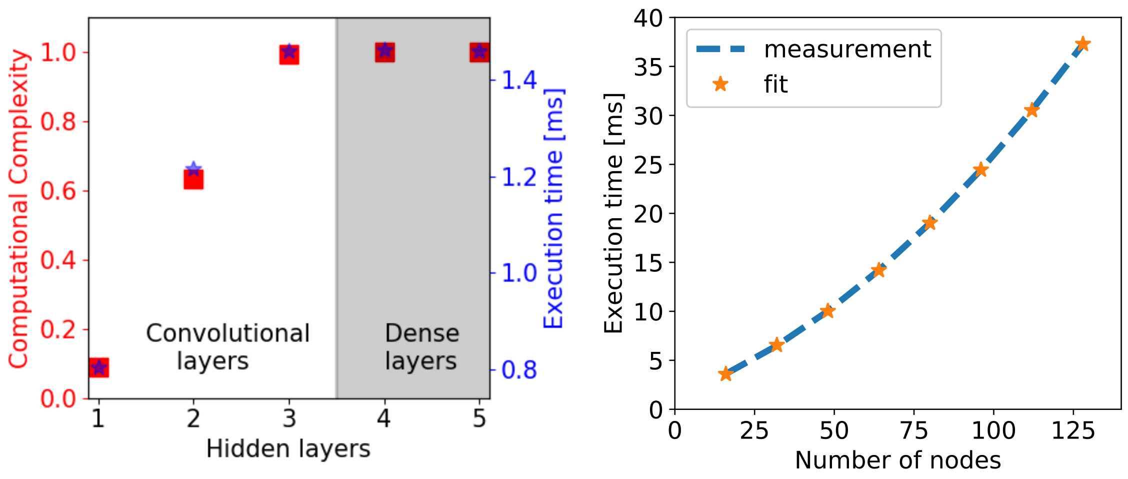


Figure 4.14: *The execution time per image for the layer by layer models section 4.3 and the breadthwise model section 4.3. The red squares in section 4.3 represents the estimated computational complexity and the blue stars the measured execution time. The computational complexity is measured relative to the full model. The DeepTrack architecture described in table 4.1 was used for the LBL and as base in the breadthwise growth. Measured when run on 10 threads on a Intel Xeon E5-2650v3 CPU. The x-axis in section 4.3 shows the number of nodes in the first layer.*

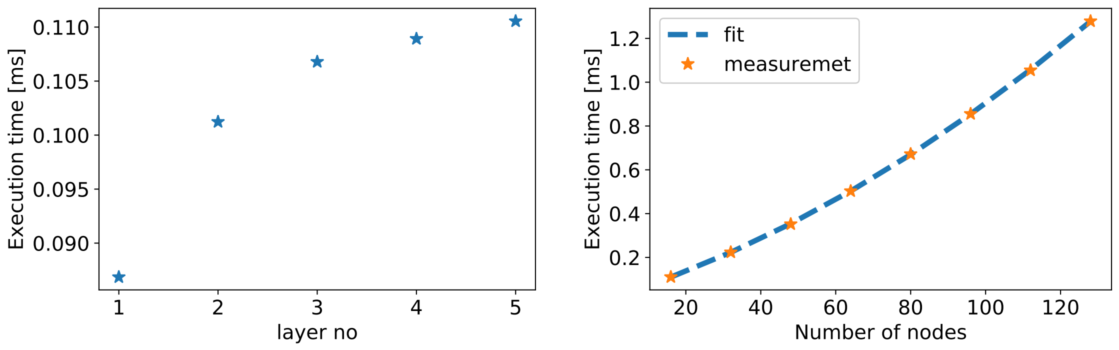


Figure 4.15: *The execution time per image for the layer by layer models and the breadthwise model. The DeepTrack architecture described in table 4.1 was used as base and for the LBL. Measured when run on a NVIDIA quadro K620 GPU.*

It is worth noting however that if execution and training speed are of large importance then TensorFlow may not be optimal and other frameworks such as Theano and Pytorch are faster [39]. This may however change as the soft-wares matures.

Chapter 5

Discussion

Both the two tested training methods, LBL and width growth, where shown to be similar in performance to standard training. Sometimes performing slightly better other times slightly worse.

In this chapter we seek to bring some clarity into the origin of the performance differences and by extension when such are to be expected. Furthermore we discuss the implications of our results to future applications of modular and grown networks such as for network architecture optimization.

5.1 LBL training applied to classification tasks

The LBL training performs well in both the *Cats and Dogs* and the MNIST datasets. Actually it even performs notably better than regular training in the case of *Cats and Dogs* when no data augmentation is used and for the MNIST especially for larger networks. As can be seen from fig. 3.12 this is not due to insufficient amounts of training epochs nor overfitting.

It must be noted here however that this is without the use of data augmentation which is known to significantly improve performance of image recognition models, [23], [24].

When data augmentation is added to the training the normal methods sees a significant boost in performance while the LBL method only gets a slight performance increase see fig. 3.15. The regular training method surpasses LBL when data augmentation is used and training is sufficiently long fig. 3.16. However the difference is not exceedingly large. It is likely that also the MNIST would benefit from data augmentation but whether this would mean that normal training outperforms LBL remains to be investigated.

There are also other regularization methods such as dropout and pruning. These will likely affect the performance of both methods but a thorough investigation of it is beyond the scope of this work. But the large fractions of active nodes see fig. 3.7 and fig. 3.18 suggest that there might be little room for pruning.

Visualizing the intermediate activations for the *Cats and Dogs* showed strengthen the hypothesis that LBL training in general produce more activations and more aggressively transform the data. The explanation for this lies within the fact that each convolutional layer in LBL training is optimized for making it easy for the subsequent output layer to make a classification. This greedy training favours passing through much information while also transforming the data to extract as much features as possible. Which runs the risk of getting stuck in local minima and failing to efficiently make use of large scale features which need many layers to work together. More of the later activations still resemble the original image, at least more so than in normal training, compare fig. 3.17f to fig. 3.17e. However here only a single image is used for the visualization, although randomly chosen, is displayed so even though most of the support this claim it may not be true for all networks.

All in all it seems like LBL performs comparable to normal training and that which is best depends on the setting. Because of this one must once again be careful in generalizing the results to other problems. The choice of parameters such as layer size, kernel size etc tend to affect model performance and by extension which training method that is most suitable. So rather than proving that LBL training is inherently better (or worse) than normal training it has been shown that it can be so depending on the application. More investigations are needed to tell, or provide guidance of, when LBL is going to perform better or worse than normal training.

5.1.1 Large networks by means of LBL

Layer by layer training has previously been applied to the MNIST dataset by using convolutional networks of a fixed final size achieving state of the art performance [30].

From fig. 3.5 we see that it is possible to train very large networks using the LBL method. Normal training fails to produce anything meaningful once the networks are above a certain size, see fig. 3.5. This is due to the vanishing gradient problem discussed in section 2.4.1. However even though LBL training makes very large networks possible making the network very large does not seem to improve performance, at least not on MNIST. A bigger network has in theory greater prediction capabilities due to the increased number of functions it can approximate but in practice this does not always translate into greater performance.

The results of the LBL training where notably more stable when fewer epochs were used per layer. This may be due to some sort of overfitting. Remarkable also that the networks can recover its prediction accuracy once more layers are added.

Each new layer can be seen as a transform of the output from the previous one. Since this transform can be the identity one can prove that adding a new layer can not decrease the networks performance (assuming training is successful) [31]. This might be what is happening for the later layers, they essentially only reshuffle the information already there without contributing significantly to the data processing.

5.2 Potential extensions of LBL training schemes

Considering how well reusing of convolutional bases works for various image analysis tasks when the base has been trained on a large dataset one can easily imagine extending this and combining it with layer by layer training. For instance in a scheme in which the early layers of a model are trained for a very wide general task on a large dataset. Then implementing a layer wise scheme, possibly with a set of output layers rather than just one to prevent early compression, in which layers are trained for tasks ever more similar to the target task. This would potentially enable the layers to learn ever more specific features while still having the flexibility which comes with LBL training and some of the advantages which comes with training a model on a large dataset. This may be further extended by branching the upper layers for different tasks keeping the early ones in common. One can thus for instance have one branch for finding the breed of dogs and one for finding the breed of cats. Would however demand large amounts of data and be more or less as demanding as regular training.

The most common method today for training very deep networks is by using residual connections. These are essentially skip connections. They were first described in [40] and have been highly successful. One could try using a variant of LBL to train certain connections first for instance the longest and then train the shorter ones keeping the long connections fix. The idea being that the later added layers will act just as minor corrections to the ones already trained in the network.

5.3 Breadthwise growth

Breadthwise growth explores a novel way of expanding already trained networks, which to the best of our knowledge, has not been used before. It does have some similarities to other techniques. It is somewhat similar to progressive learning discussed in section 3.3 in that it reuses the old weights.

It manages to produce high accuracy while maintaining modularity. It does in fact beat the regular DeepTrack architecture, compare the dotted lines to the solid in fig. 4.5. The difference is bigger for large SNR than for small.

From fig. 4.7 and fig. 4.6 it is evident that adding more networks (generally) increase performance. However how much one gains upon adding new networks to the architecture seems to depend on the setting. Furthermore the first couple of networks make the biggest difference in performance. This is unsurprising considering that the relative size of the network increase the most for the early networks. The biggest relative difference when a new network is connected seems to be for larger networks at high SNR, see fig. 4.7. It is known from the literature that using a very wide network risks making the learning process unstable and may actually hinder performance [16]

There are notable variations in performance when a new newtwork is added, see fig. 4.6 and fig. 4.7. These may be explained in part by the fact that the initial weights of the networks are random and may be more or less close to the optimum. Thus the network can get stuck in a local minima during training. Nonetheless as previously discussed adding more networks do in general improve performance.

Using wider networks as bases seems to improve performance. This difference is somewhat bigger for small SNR on an absolute scale but smaller on a relative scale for high SNR, compare fig. 4.6 to fig. 4.5. A plausible explanation to this is that more nodes are needed to accurately make sense of high noise data because high noise levels require looking at larger structures in the images. However on a relative scale the differences are quite small as long as the network is big enough (roughly base 16 or larger it seems).

The smallest network performs relatively poorly compared to the other ones, see fig. 4.5. This may be because of an information bottleneck. Meaning that the number of nodes in the layers is so small that not enough information can be passed through to make an accurate prediction, nor for that matter compress the information efficiently. While the greedy optimization may hinder the learning of larger features which requires many nodes in each layer to be trained in unison. Which is to say that the networks may be getting stuck in local minima.

The normally trained networks do not seem to benefit significantly from having more parameters in the various layers, see fig. 4.4. In fact the two biggest networks perform very poorly. That there is no clear improvement in performance with larger networks may be due to the usage of a fixed training scheme which may not be as well suited for large networks. For instance larger networks do in general require more training to perform well. There are several other possible explanations as to why the large convolutional networks fail. With this in mind it is remarkable that the breadth growth performs as well as it does completely without fine tuning, either of network parameters such as layer sizes or activation functions or training parameters such as learning rate or optimizer. In theory larger networks should be able perform as good or better than smaller, which is easy to understand when one considers that smaller networks are subsets of the larger.

5.3.1 Node activations

The number of active nodes is essentially constant for the different sizes of the normal networks, with the exception of the two largest networks see fig. 4.8a. However considering the poor performance of the two largest networks it is likely that the deviating behaviour in the number of activations is closely linked to their poor performance. It seems like that very few nodes in the third convolutional layer are active meaning that this layer could be acting as an information bottleneck preventing accurate predictions and as a consequence of this also further learning.

The fact that there is such little difference in the number of active nodes and may be an indication of that many of the nodes are not doing anything, i.e are always zero. This is not necessarily the case, it could be that there are different nodes active for different images resulting in a constant average, but it would explain why there is so little change for the various network sizes.

5.4 Execution time

From fig. 4.13 it is evident that one stands to gain much by using a GPU. For the larger models there is roughly a factor 30 difference see fig. 4.13. This is to be expected considering that neural networks can be highly parallelized and GPUs are well suited for parallel workloads [20]. Also why one generally use GPUs rather than CPUs especially for training which is often very time consuming.

It also seems like the computational time indeed increase with the network width and can be described by a second degree polynomial, at least for the CPU. For the GPU it is more difficult to tell for sure whether a polynomial best describes the relation. It is possible that it also follows a second degree polynomial but is at a different scale meaning that it appears linear when looking at models of such limited size. Which can be explained by the fact that the GPU is considerably faster.

The modular networks execute similarly fast to the normal ones when run on CPU, only slightly slower, see fig. 4.14. As mentioned in section 2.5 and section 4.3 this is likely due to that TensorFlow and Keras is better optimized for regular architectures as used in the regular convolutional networks. Therefore giving an extra computational overhead.

Comparing section 4.3 with section 4.3 and it becomes apparent that the scaling as more layers are added is different. It seems like if the convolutional layers are greatly more expensive than the dense layers when run on CPU, but not so on CPU. This can be explained once again with the fact that the workload is highly parallelizable meaning that what limits the GPU performance might not be computational speed in this case but something else. Potentially the memory management which is essential to performance [41]. It might also be the reason as to why the increase in computational time for the different LBL models is so small when run on GPU. The memory is likely bottle-necking the computational pipeline. Most likely it is in loading the images from main memory to the GPU that most of the time is lost since this memory is relatively slow.

This is probably also what is limiting the performance on the CPU. The execution time scales almost perfectly with the computational complexity but with a constant offset suggesting that there is a constant memory cost involved with processing each image, see section 4.3. The small difference can be explained by a combination of a computational overhed from many layers increasing the length of the data pipeline and to a lesser extent neglecting the activation function in the cost calculation. Even though the cost of the small network is roughly one tenth that of the full network it only executes about twice as fast in practice. This is however something that is highly dependent on the implementation. And the difference may be smaller (or larger) for another set of networks.

5.5 Potential applications of modular networks

It is not difficult to come up with multiple applications modular networks. Guided by the results we elaborate further on some problems for which these are deemed especially suitable.

5.5.1 Multi-particle tracking

As illustrated in section 4.2 the modular networks can be used for multi-particle tracking. The main advantage with using the modular network instead of a small and a large normal network to make the fast and the more accurate prediction respectively is that the modular networks can be made even faster. This while not compromising the performance as was shown in section 3.5.1. The implementation here in Keras is not fully optimized since it uses two separate networks for the prediction rather than a single network which cancels it's execution if no particle is detected. This means that the computational overhead from loading data into the larger model might not be avoided. However here the aim was a proof of concept rather than a perfectly optimized implementation which was achieved. It is not inconceivable that the predictions between the small and the large network would differ so much that it would not be any significant improvement or that the small network would be too inaccurate to function as a useful first step.

The enhanced version manages to reach essentially the same performance as the full large network confirming the algorithms efficiency, see table 4.2 and fig. 4.12. The decreased number of false positives and the somewhat higher number of false negatives of the enhanced network can be attributed to the use of two rather than only one threshold. This means that there is a stricter requirement for when a particle is considered detected. These miss classification rates can be modified somewhat by changing the threshold. The lower the thresholds the fewer false positives and the more false negatives meaning that there is a tradeoff to be made.

One could also in principle save time on not recalculating the convolutions of the overlapping boxes. In the single layer network most of these are the same. In subsequent layers they do however differ. Also the breadth grown networks could be used in a similar way as the LBL network for multi particle tracking and the results in section 4.1 suggest they would perform even better in terms of accuracy albeit slower.

Other object tracking (or detection) techniques based on deep learning exist which seeks to tackle the same compromise of speed versus accuracy. For instance the recently developed the YOLO algorithm [42]. Which to the best of our knowledge has not yet been applied in the field of particle tracking but also has the potential to perform both fast and well. The key to YOLOs fast execution is that it only process each part of the image once, somewhat similar to what is done here using the modular networks.

5.5.2 Architecture optimization

The methods may also be used to find suitable architectures even in cases where regular training is preferred due to greater performance. This is because the performance for the same architecture is similar for LBL training and normal training as has been shown for both estimation and classification tasks.

Growing networks could be used to find a good model is then found simply by growing the network until either the improvement has diminished or performance has reached satisfactory levels. This can be done either through LBL or breadth growth or a combination of the two. Finding an optimal architecture is however significantly more difficult due to the vast number of parameters invloved in designing and training a network.

This is closely related to the area of automatic hyperparameter optimization which is an active area of research [26], [43].

Bibliography

[1] S. Helgadottir, A. Argun, and G. Volpe, "Digital video microscopy enhanced by deep learning", *Optica*, vol. 6, no. 4, pp. 506–513, 2019.

[2] Y. Shoham, R. Perrault, E. Brynjolfsson, J. Clark, J. Manyika, J. C. Niebles, T. Lyons, J. Etchemendy, and Z. Bauer, *The ai index 2018 annual report*, 2018.

[3] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. U. Khan, "The rise of "big data" on cloud computing: Review and open research issues", *Information systems*, vol. 47, pp. 98–115, 2015.

[4] P. Mamoshina, A. Vieira, E. Putin, and A. Zhavoronkov, "Applications of deep learning in biomedicine", *Molecular pharmaceutics*, vol. 13, no. 5, pp. 1445–1454, 2016.

[5] Y. Liu, E. Racah, J. Correa, A. Khosrowshahi, D. Lavers, K. Kunkel, M. Wehner, W. Collins, *et al.*, "Application of deep convolutional neural networks for detecting extreme weather in climate datasets", *arXiv preprint arXiv:1605.01156*, 2016.

[6] K. T. Butler, D. W. Davies, H. Cartwright, O. Isayev, and A. Walsh, "Machine learning for molecular and materials science", *Nature*, vol. 559, no. 7715, p. 547, 2018.

[7] A. E. Sallab, M. Abdou, E. Perot, and S. Yogamani, "Deep reinforcement learning framework for autonomous driving", *Electronic Imaging*, vol. 2017, no. 19, pp. 70–76, 2017.

[8] B. Hidasi, A. Karatzoglou, L. Baltrunas, and D. Tikk, "Session-based recommendations with recurrent neural networks", *arXiv preprint arXiv:1511.06939*, 2015.

[9] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search", *nature*, vol. 529, no. 7587, p. 484, 2016.

[10] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification", in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

[11] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", *arXiv preprint arXiv:1502.03167*, 2015.

[12] J. Ker, L. Wang, J. Rao, and T. Lim, "Deep learning applications in medical image analysis", *Ieee Access*, vol. 6, pp. 9375–9389, 2017.

[13] P. T. Komiske, E. M. Metodiev, and M. D. Schwartz, "Deep learning in color: Towards automated quark/gluon jet discrimination", *Journal of High Energy Physics*, vol. 2017, no. 1, p. 110, 2017.

[14] Y. Zhong, C. Li, H. Zhou, and G. Wang, "Developing noise-resistant three-dimensional single particle tracking using deep neural networks", *Analytical chemistry*, vol. 90, no. 18, pp. 10 748–10 757, 2018.

[15] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean, "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer", *arXiv preprint arXiv:1701.06538*, 2017.

[16] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, `http://www.deeplearningbook.org`.

[17] A. A. Rusu, N. C. Rabinowitz, G. Desjardins, H. Soyer, J. Kirkpatrick, K. Kavukcuoglu, R. Pascanu, and R. Hadsell, "Progressive neural networks", *arXiv preprint arXiv:1606.04671*, 2016.

[18] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning", *nature*, vol. 521, no. 7553, p. 436, 2015.

[19] F. Chollet *et al.*, *Keras*, `https://keras.io`, 2015.

[20] F. Chollet, *Deep learning with python.* 2017.

[21] B. Mehlig, "Artificial neural networks", *arXiv preprint arXiv:1901.05639*, 2019.

[22] K. Janocha and W. M. Czarnecki, "On loss functions for deep neural networks in classification", *arXiv preprint arXiv:1702.05659*, 2017.

[23] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks", in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[24] L. Perez and J. Wang, "The effectiveness of data augmentation in image classification using deep learning", *arXiv preprint arXiv:1712.04621*, 2017.

[25] Nvidia. (2019). Nvidia tesla v100 overview, [Online]. Available: `https://www.nvidia.com/en-us/data-center/tesla-v100/` (visited on 04/24/2019).

[26] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy, "Progressive neural architecture search", in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 19–34.

[27] M. Jaderberg, A. Vedaldi, and A. Zisserman, "Speeding up convolutional neural networks with low rank expansions", *arXiv preprint arXiv:1405.3866*, 2014.

[28] J. Huang, V. Rathod, C. Sun, M. Zhu, A. Korattikara, A. Fathi, I. Fischer, Z. Wojna, Y. Song, S. Guadarrama, *et al.*, "Speed/accuracy trade-offs for modern convolutional object detectors", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 7310–7311.

[29] H. Cohn, R. Kleinberg, B. Szegedy, and C. Umans, "Group-theoretic algorithms for matrix multiplication", in *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)*, IEEE, 2005, pp. 379–388.

[30] C. Hettinger, T. Christensen, B. Ehlert, J. Humpherys, T. Jarvis, and S. Wade, "Forward thinking: Building and training neural networks one layer at a time", *arXiv preprint arXiv:1706.02480*, 2017.

[31] E. Belilovsky, M. Eickenberg, and E. Oyallon, "Greedy layerwise learning can scale to imagenet", *arXiv preprint arXiv:1812.11446*, 2018.

[32] D. Erhan, P.-A. Manzagol, Y. Bengio, S. Bengio, and P. Vincent, "The difficulty of training deep architectures and the effect of unsupervised pre-training", in *Artificial Intelligence and Statistics*, 2009, pp. 153–160.

[33] M. Kulkarni and S. Karande, "Layer-wise training of deep networks using kernel similarity", *arXiv preprint arXiv:1703.07115*, 2017.

[34] R. Benenson, *Classification datasets results*, Accessed: 2019-04-29. [Online]. Available: `https://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html#4d4e495354`.

[35] L. Guo, J. Y. Har, J. Sankaran, Y. Hong, B. Kannan, and T. Wohland, “Molecular diffusion measurement in lipid bilayers over wide concentration ranges: A comparative study”, *ChemPhysChem*, vol. 9, no. 5, pp. 721–728, 2008.

[36] A. J. Berglund, “Statistics of camera-based single-particle tracking”, *Physical Review E*, vol. 82, no. 1, p. 011 917, 2010.

[37] G. Hinton, O. Vinyals, and J. Dean, “Distilling the knowledge in a neural network”, *arXiv preprint arXiv:1503.02531*, 2015.

[38] B. Clarke, “Comparing bayes model averaging and stacking when model approximation error cannot be ignored”, *Journal of Machine Learning Research*, vol. 4, no. Oct, pp. 683–712, 2003.

[39] S. Bahrampour, N. Ramakrishnan, L. Schott, and M. Shah, “Comparative study of deep learning software frameworks”, *arXiv preprint arXiv:1511.06435*, 2015.

[40] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition”, in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[41] A. R. Brodtkorb, T. R. Hagen, and M. L. Sætra, “Graphics processing unit (gpu) programming strategies and trends in gpu computing”, *Journal of Parallel and Distributed Computing*, vol. 73, no. 1, pp. 4–13, 2013.

[42] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You only look once: Unified, real-time object detection”, in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[43] T. Domhan, J. T. Springenberg, and F. Hutter, “Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves”, in *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

Appendix A

Github

For the interested reader there is a github with code for modular networks, see `https://github.com/Martin-c-lin/Growing-deep-nets.git` .