Supervised Learning and Generalisation

James O'Reilly
james.oreilly@student.kuleuven.be

Introduction

This report follows my first coursework as part of the 'Neural Networks and Deep Learning' course at KU Leuven. This coursework focuses on supervised learning in simple feed-forward networks. The report is split into three parts:

- Comparing the performance of different learning algorithms for a simple curve fitting problem, and testing how well these algorithms generalise to noisy data.
- Approximating a 2D function using a feed-forward network. This section focuses on how data is divided before training, as well as selecting network hyper-parameters.
- Finally, following from section two, we look at the Bayesian treatment of neural networks and Bayesian inference of network hyper-parameters.

1 Comparison of Different Learning Algorithms

A number of neural networks were trained for a simple function approximation task with different learning algorithms. The underlying function is given by

$$y = sin(x^2), \quad 0 \leq x \leq 3\pi$$

Data points were sampled at intervals of 0.05. A list of the different training algorithms that were compared is given below:

- Gradient descent (GD)
- Gradient descent with adaptive learning rate (GDA)
- Conjugate gradient backpropogation with Fletcher-Reeves updates (CGF)
- Conjugate gradient backpropagation with Polak-Ribiére update (CGP)
- BFGS quasi-Newton backpropogation (BFG)
- Levenberg-Marquardt backpropogation (LM)

1.1 Network Configuration and Architecture

When comparing the learning algorithms, network hyper-parameters such as the number of neurons, the number of layers, initialisation weights, and transfer function were kept constant. The network has a single dense hidden layer with 30 neurons, each with a `tanh` activation function. It will be clear later that 30 neurons is not sufficient to accurately approximate the function, however limiting the neurons, and therefore the performance, allows for a much clearer comparison of the algorithms. If 50 or 60 neurons are used in the hidden layer, then the quasi-Newton and LM algorithms both reach a very low error on the test set, and it becomes difficult to see the difference in performance. The data set was divided randomly with 70% used for training and 15% for both validation and testing. Early stopping is implemented by stopping training if the MSE on the validation set increases for six epochs consecutively.

1.2 Comparison

A number of metrics were used to compare the performance of the learning algorithms:

- Mean-squared error (MSE) on the test set
- Regression between the network outputs and targets
- Time taken to converge (number of epochs). Convergence is determined by the best performance on the validation set

The performance metrics were measured for networks trained for 1, 14, and 100 epochs. Each metric was measured by running the network 50 times and then averaging the results.

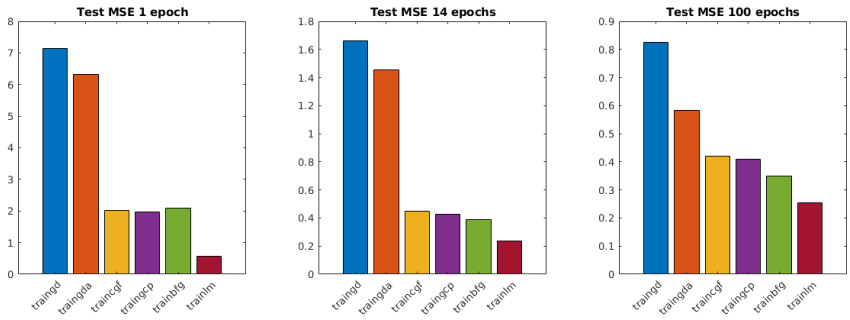


Figure 1: Test set MSE for different numbers of epochs. Note the different y-axis ranges.

It is clear that basic gradient descent performs the worst across the range of epochs, while Levenberg-Marquardt consistently outperforms all other learning algorithms. GDA gives only a slight improvement over standard GD. CGF, CGP, and BFG far outperform GD across the range of epochs. The performance of BFG relative to CFG and CGP improves as the number of epochs increases.

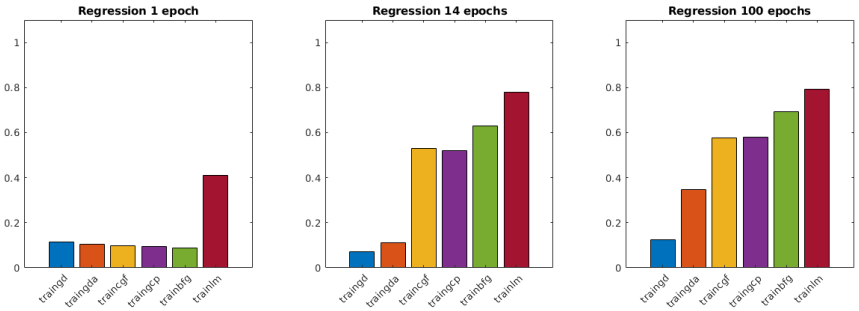


Figure 2: Regression between approximation and target for different numbers of epochs. Note the different y-axis ranges.

As would be expected, the regression values reflect the MSE results for the relative performance of the different learning algorithms. It is interesting to look at the average time taken to converge for the algorithms. This is given by the epoch at which the MSE on the validation set is minimised.

Note that LM converges most rapidly, while BFG is slower to converge on average. Naturally, Levenberg-Marquardt will converge faster than gradient descent in most cases as the varying dampening parameter μ allows it to interpolate between Gauss-Newton method and gradient descent. Note also that as LM converges rapidly and consistently within the maximum number of epochs, it is possible to increase the maximum value of the dampening parameter μ as we can afford slower convergence to the optimum. Increasing the upper range of μ could potentially give better performance, at the cost of speed. Both GD and GDA are slow to converge, with GDA converging due to the adaptive learning rate.

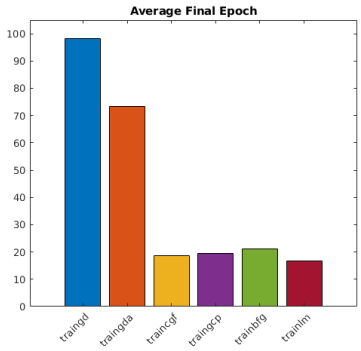


Figure 3: Number of epochs until convergence for each learning algorithm. The maximum number of epochs is 100.

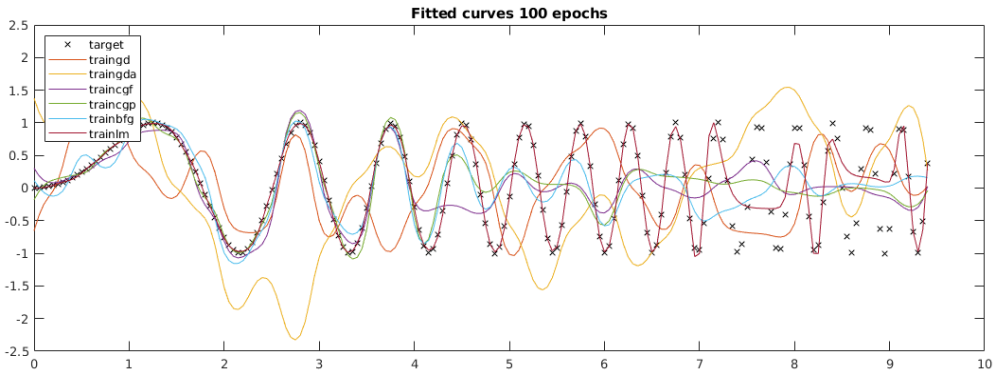


Figure 4: The fitted curves for 100 epochs.

1.3 Noisy Data and Generalisation

To test the performance of the learning algorithms with noisy data, additive white Gaussian noise was added to the original function with a signal-to-noise ratio of 10. The networks were then trained with under the same conditions as before and the average MSE, correlation, and time to convergence was recorded for each of the algorithms.

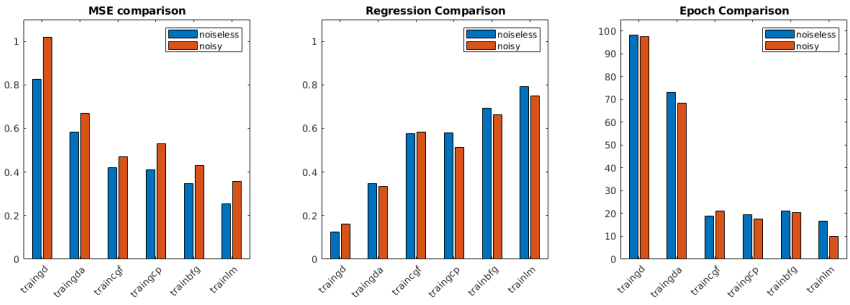


Figure 5: Comparison of performance on noiseless vs noisy data. These results are averaged over 50 runs.

Comparing the results of the noisy and noiseless function approximation show that in general both the test MSE and the regression were slightly worse for the noisy data, as would be expected. This was the case for every learning algorithm except for CGF, which had a better regression score with the noisy data. On average, the networks converged faster when trained on the noisy data, again with the exception of CGF.

2 Approximating a 2D Function using a Feed-forward Network

Here we approximate a 2D function using a multi-layer feed-forward network. This section focuses on data division and how network hyper-parameters are selected for.

2.1 Data Division

The data was sampled randomly from the dataset without replacement and split into three evenly-sized sets for training, validation and testing. What is the rationale behind random division? It is instructive to look at the alternative: dividing the data into contiguous blocks. The issue with dividing the data into contiguous blocks is that if the structure of the data is not consistent throughout the entire dataset, the network will not generalise well. The data is sampled without replacement because if there is an overlap between the training set and the validation or test set, the performance metrics for the validation and test set will be biased as they will be measured with data points on which the metric was trained. See Figure 6 for a visualisation of the training and test set used.

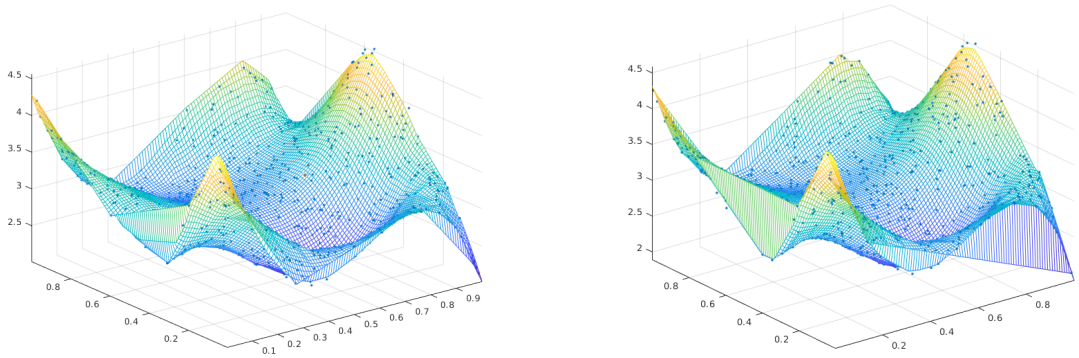


Figure 6: A visualisation of the training set (left) and test set (right) surfaces. The actual data points which were used to form the mesh are shown as blue dots. The roughness at the edge of the plot is from the interpolation for forming the mesh, and not from the dataset itself.

2.2 Building the Network and Choosing the Hyper-parameters

Based off the results from the function approximation in the previous section, I decided to use Levenberg-Marquardt as the learning algorithm. Before training the network, it's also important to consider the nature of the input data and the function that we are trying to approximate. The function has only positive values for both inputs and outputs, with inputs X_1 and X_2 both in the range $[0, 1]$, and so data-scaling is not needed as the input data is already properly scaled between 0 and 1.

2.2.1 Activation Function

For the activation function, I decided to use ReLU. Generally the two main benefits of using ReLU are that it avoids the vanishing gradient problem and is computationally more efficient. As there are only a few layers in the network (discussed below), the vanishing gradient problem is not a huge concern and the primary reason for using ReLU is computational efficiency. The decision must then be made between using standard ReLU or some ReLU variant such as leaky ReLU, parametric ReLU, or ELU (exponential linear unit). This decision is determined by whether or not one is concerned with the 'dying ReLU' problem. The aforementioned extensions to ReLU each relax the non-linear output of the function to allow small negative values in some way, and therefore solve this problem. It was decided to use standard ReLU and to instead try to initialise weights and biases in a way which avoids the dying ReLU problem as much as possible.

Firstly, a smaller bias input value of 0.1 was used. This makes it likely that the units will be initially active for most inputs in the training set and allow the derivatives to pass through. Secondly, a different weight initialisation scheme should be used when using ReLU. If the weights are initialised to small random values centered on zero, then by default half of the units in the network will output a zero value. One initialisation scheme which could be used is 'Kaiming initialisation'. This initialisation scheme causes each individual ReLU

layer to have a standard deviation of 1 on average. keeping the standard deviation of layers' activations around 1 allows stacking of several more layers in a deep neural network without gradients exploding or vanishing. Ultimately I couldn't effectively implement these weight and bias initialisation schemes in Matlab, but I think its important to mention them nonetheless. However, after training the network multiple times using ReLU layers, I found that with my (likely faulty) implementation it consistently performed worse than with a *tanh* activation function, and so *tanh* is used in the sections that follow on hyper-parameter tuning and Bayesian regularisation.

2.2.2 Determining Neural Network Topology

Determining both the optimal number of hidden layers and neurons is very important. One way to determine these values manually is to perform a grid search over the hyper-parameter space. First the number of neurons was first kept constant and the number of layers was varied. Fixing the number of neurons at 60, networks with one up to five hidden layers were trained. The performance of these networks over many runs was averaged to determine the effective number of layers. Increasing the number of layers improved performance on the validation set, but the performance began to drop once the 60 neurons were split between more than four layers. Splitting the 60 neurons between two or three layers gave the best performance. I chose to use two hidden layers.

No. of Hidden Layers	1	2	3	4
Average Test MSE ($\times 10^{-4}$)	0.1223	0.0468	0.0487	0.0730

Table 1: Average test MSE for different numbers of hidden layers. The results were averaged over 10 runs and the networks were each trained for 200 epochs.

Once then number of hidden layers was decided, this number was kept fixed while the number of neurons was increased. The goal of this was to try to determine at which point the network became over-parametrised and would not generalise well. The performance of networks with two hidden layers were evaluated for each layer having from 10 to 30 neurons. The performance of the network generally increased as the number of neurons increased, however there were outliers, likely due to favourable or unfavourable initialisation of weights and biases. The results of this grid search are given below. The networks performance still improved as the number of neurons increased, and did not appear over-parametrised with 30 neurons in each layer. The final architecture that was used is given below

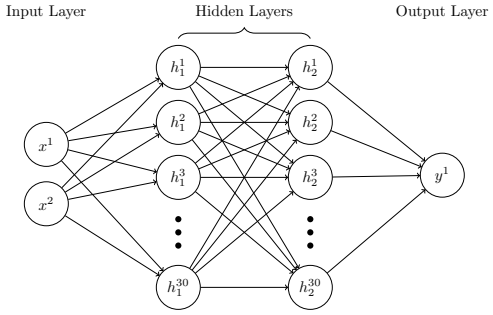


Figure 7: The final architecture for the 2D function fitting network determined through grid search.

Once the architecture was determined, the network was trained for 800 epochs with early stopping. If the MSE on the validation set did not improve for 10 epochs, training would stop. The approximation of the function given by the network is shown in Figure 8.

The error level curves for the network trained with standard Levenberg-Marquardt are given on the left of Figure 9. Looking at these error level curves, the training performance diverges massively from both the test and validation performance, implying that the network has overfitted to a large degree. This overfitting could be avoided by using Bayesian regularisation or dropout regularisation. The second graph in Figure 9 shows the improved performance of the network with Bayesian regularisation. Note also the shape of the error curves, with the test and validation error not diverging as much from the training error.

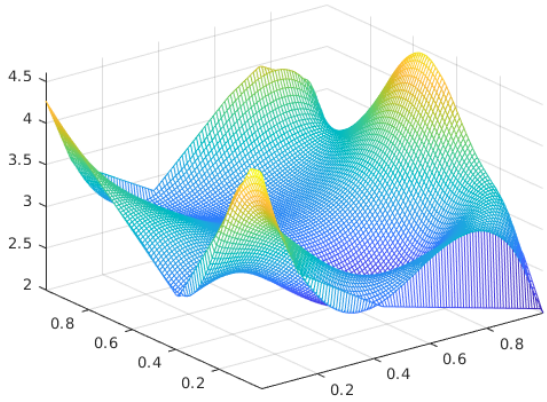


Figure 8: Approximation for the network.

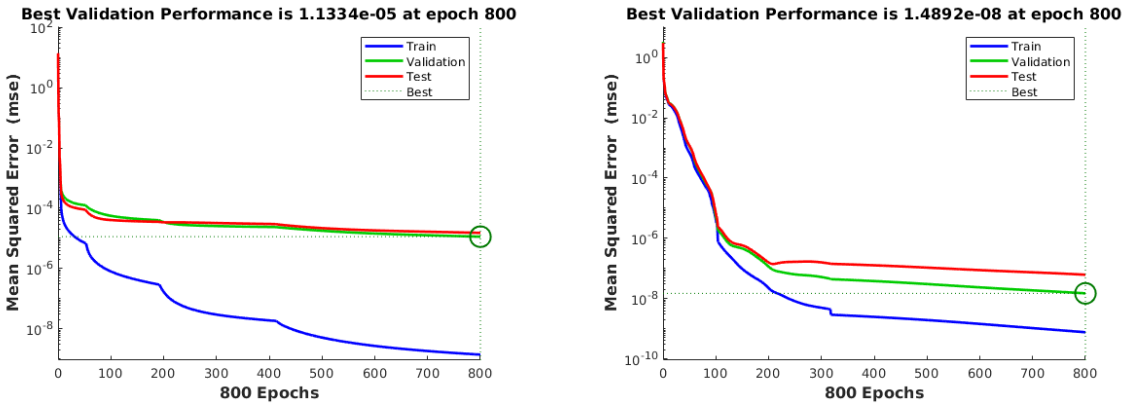


Figure 9: Performance of the network on the training, validation, and test sets. Without regularisation on the left, with regularisation on the right.

3 Bayesian Inference of Network Hyper-parameters

Here we return to the 1D function fitting exercise discussed in the first section. Generalisation can be improved by giving the network hyper-parameters a Bayesian treatment and effectively modifying the prior distribution over the weight-space and bias-space. Doing so can reduce overfitting in over-parametrised networks. To test this, the performance of standard Levenberg-Marquardt (`trainlm`) was compared with Levenberg-Marquardt with Bayesian regularisation (`trainbr`) for a network with a single hidden layer of 30 neurons, both for noisy and noiseless data. In theory, the learning algorithm with Bayesian regularisation should generalise better and reduce overfitting, especially in the case of over-parametrised networks.

Table 2 shows that Levenberg-Marquardt with Bayesian regularisation performs better on the test set than standard Levenberg-Marquardt. In particular, the results show that the network with Bayesian regularisation generalises much better with noisy data, although the effective number of parameters is higher when trained on noisy data. This clearly shows how Bayesian regularisation improves generalisation and performance by reducing overfitting. The results for the same experiment performed with a massively overparametrised network (400 neurons) are shown in Table 3.

In the case of the overparametrised network, the results show the same pattern as before but are just slightly worse. In the overparametrised network, standard Levenberg-Marquardt converges very rapidly.

	Test MSE	Regression	Convergence	Effective # of Parameters
`trainlm`	0.2263	0.7924	14	91
`trainbr`	0.0852	0.9655	100	54
`trainlm` (noisy)	0.3216	0.3216	15	91
`trainbr` (noisy)	0.0766	0.9585	89	65

Table 2: Average test MSE, regression, number of epochs until convergence, and effective number of parameters for Levenberg-Marquardt with and without Bayesian regularisation for both noisy and noiseless data. Maximum number of epochs is 100 and the number of parameters is 91.

	Test MSE	Regression	Convergence	Effective Number of Parameters
`trainlm`	0.1147	0.9679	6	301
`trainbr`	0.04	0.9872	54	110
`trainlm` (noisy)	0.1838	0.9392	4	301
`trainbr` (noisy)	0.1626	0.9563	80	151

Table 3

Recurrent Neural Networks

James O'Reilly
james.oreilly@student.kuleuven.be

Introduction

A recurrent neural network (RNN) is a class of artificial neural networks where connections between nodes form a directed graph along a temporal sequence. This allows it to exhibit temporal dynamic behaviour. Derived from feedforward neural networks, RNNs can use their internal state (memory) to process variable length sequences of inputs. In this assignment we will first discuss Hopfield networks, before exploring the use of MLPs and long short term memory networks (LSTMs) for time-series prediction.

1 Hopfield Networks

Hopfield networks serve as content-addressable memory systems with binary threshold nodes. They are often used as a model of associative memory. The network consists of a single layer of fully interconnected neurons. These neurons update synchronously, with the output of timestep t as the input in timestep $t+1$. A single input therefore gives a series of output. After initialisation, the network will evolve dynamically toward a stable state (an attractor). One can choose the attractor states of the network, and then model the dynamic behaviour of the network with different initialisation states. Sometimes the network has attractors which are not explicit. Here we explore the convergence and attractors of simple Hopfield networks with two or three neurons.

1.1 A Two-Neuron Hopfield Network

We create a two-neuron Hopfield network with three explicit attractors given by the matrix

$$T = \begin{bmatrix} 1 & 1 \\ -1 & -1 \\ 1 & -1 \end{bmatrix}$$

where each row is an attractor. The network was then initialised with other vectors to understand the dynamics of this network – including the presence of other attractor states that were not explicitly set, and also the number of timesteps taken to converge. To fully explore the space of initial vectors, a grid vectors (x, y) was used, with $x, y \in [1, -1]$. The use this grid has benefits of random initial vectors for finding implicit attractor states as using highly-symmetric points allows us to see saddle points which are situated exactly halfway between the explicit attractors. Using this grid, another set S of attractor states was found:

$$S = \begin{bmatrix} -1 & 0 \\ 1 & 0 \\ -1 & 1 \\ 0 & -1 \\ 0 & 0 \\ 0 & -1 \end{bmatrix}$$

At each timestep, the percentage of initial vectors which had reached one of the stable states given in S was calculated. The results of this are shown in Figure 1.

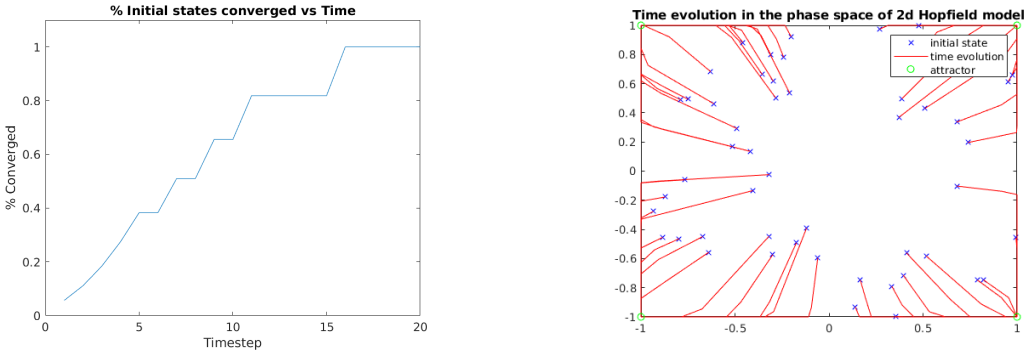


Figure 1: Convergence for the two neuron Hopfield network, along with the trajectories for random initial states.

1.2 A Three-Neuron Hopfield Network

Similarly, we create a three neuron Hopfield network with three explicit attractors given by the matrix

$$T = \begin{bmatrix} 1 & 1 & 1 \\ -1 & -1 & 1 \\ 1 & -1 & -1 \end{bmatrix}$$

Initialising the network with vectors from a three-dimensional grid, no implicit attractor states were found. However, looking at the graph for convergence over time (see Figure 2), it shows that there were a number of initial states for which it took over 200 iterations to converge to one of the explicit attractors. Also given are the trajectories of ten randomly generated initial states.

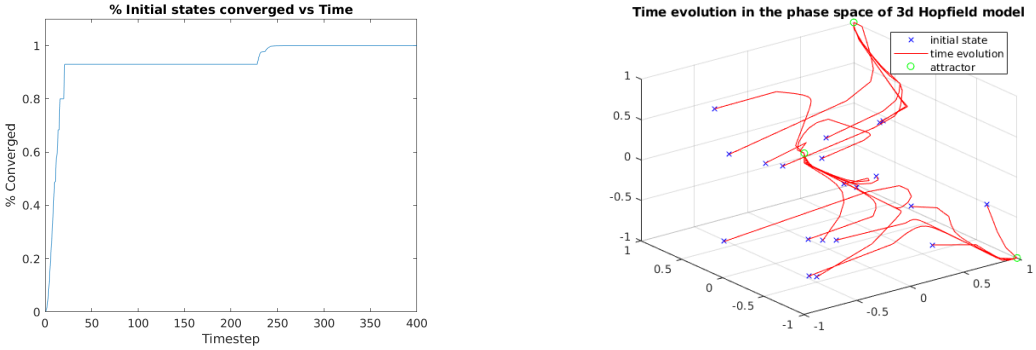


Figure 2: Convergence for the three neuron Hopfield network, along with the trajectories for ten random initial states.

1.3 A Hopfield Network for Handwritten Digits

We create a Hopfield network which has as attractors the handwritten digits $0, \ldots, 9$. Then to test the ability of the network to correctly retrieve these patterns some noisy digits are given to the network. As the noise is increased, the ability of the network to correct corrupted digits falls off quite rapidly, particularly in the case of numbers which are similar to others (both numbers have loops, a vertical line, etc.), and so the network is not always able to reconstruct corrupted digits. Increasing the number of iterations improved performance but even with more than 1000 iterations, some digits could not be reconstructed for high levels of noise. The network would likely perform better with higher resolution images (more pixels), as this effectively increases the 'distance' between the attractor states, which would mean more noise is necessary to move from one state to another. Figure 3 shows the attempted reconstruction of corrupted digits.

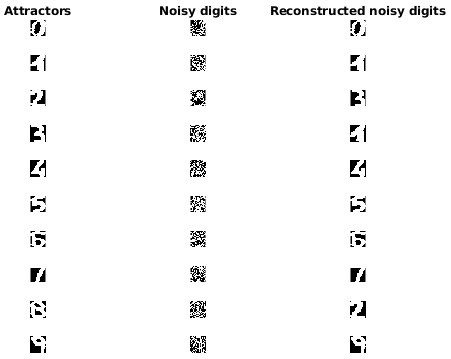


Figure 3: Attempted reconstruction of corrupted digits with a noise level of 6 and 10000 iterations.

2 Long Short-Term Memory Networks

In this section I give a brief introduction to time-series prediction, before trying make predictions about series data using a simple multi-layer perceptron with one hidden layer and then using an LSTM.

2.1 Time-Series Prediction using a Multi-layer Perceptron

The data being used is from a chaotic laser which can be described as a non-linear dynamical system. Given 1000 training data points, the aim is to predict the next 100 points. The training and test data are given in Figure 4.

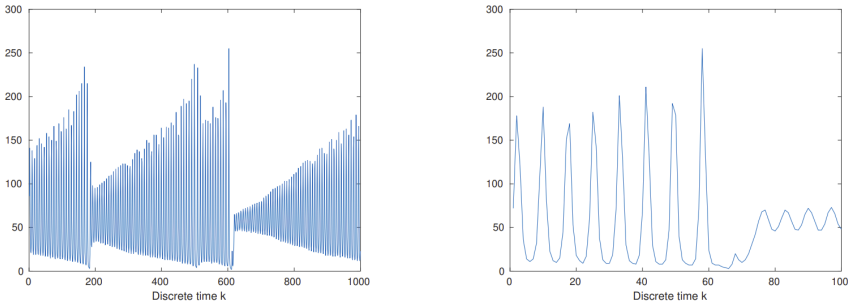


Figure 4: Training and test data from the chaotic laser.

First the data is standardised so that it has zero-mean and unit variance. After the dataset is standardised, an MLP with one hidden layer is trained. Training is done in feedforward mode:

$$\hat{y}_{k+1} = w^{\mathrm{T}} \tanh\left(V\left[y_k; y_{k-1}; \ldots; y_{k-p}\right] + \beta\right)$$

where p is the lag – the number of previous timesteps considered. In order to make predictions, the trained network is used in an iterative way as a recurrent network:

$$\hat{y}_{k+1} = w^{\mathrm{T}} \tanh\left(V|\hat{y}_k; \hat{y}_{k-1}; \ldots; \hat{y}_{k-p}\right] + \beta\big)$$

A single-layer MLP with was trained on this data using Levenberg-Marquardt with Bayesian regularisation. The network was trained for 100 epochs with different lag values and different numbers of neurons in the hidden layer. While the performance of the network improved both as the number of neurons and the size of the lag was increased, the network originally could not capture the qualitative behaviour of the laser (see Figure 5). However, after increasing the lag to 30 and the number of hidden neurons to 50, the network was able to predict

the initial 'drop' after a series of increasing spikes (see Figure 5). The qualitative behaviour of the laser after the drop, however, was not predicted properly and behaved far too erratically. Perhaps this could be solved by either increasing the number of neurons, epochs or inputs, but I was reaching the limits of my laptop and didn't pursue this any further.

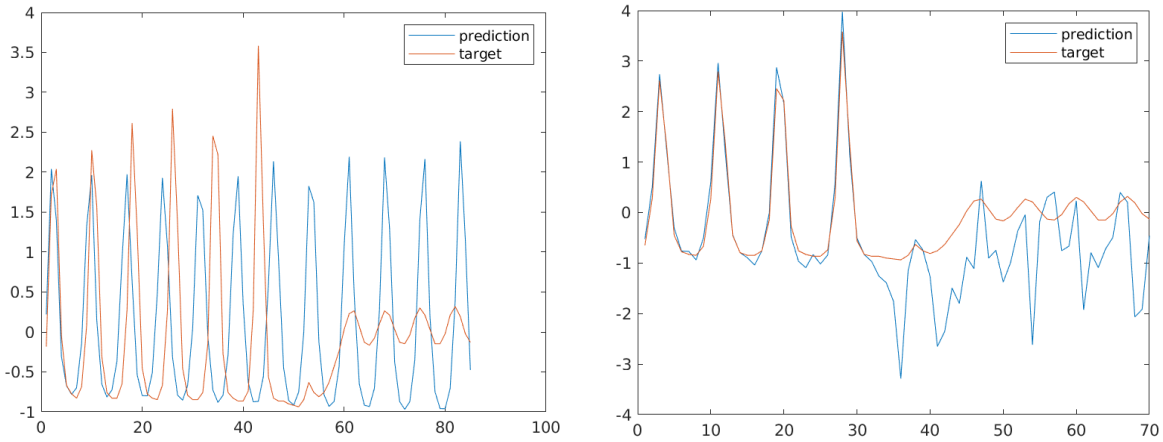


Figure 5: Target vs prediction plots. In the first plot, the network was trained for 100 epochs with a lag value of 15 and 30 hidden neurons. In the second plot, the network was trained for 100 epochs with a lag value of 30 and 50 hidden neurons.

2.2 Introduction to Long Short-Term Memory Networks

Long Short Term Memory networks, usually just called LSTMs, are a special kind of RNN, capable of learning long-term dependencies. LSTMs contain information outside the normal flow of the recurrent network in a gated cell. Information can be stored in, written to, or read from a cell, much like data in a computers memory. The cell makes decisions about what to store, and when to allow reads, writes and erasures, via gates that open and close. Those gates act on the signals they receive, and similar to the neural networks nodes, they block or pass on information based on its strength and importance, which they filter with their own sets of weights. Those weights, like the weights that modulate input and hidden states, are adjusted via the recurrent networks learning process. That is, the cells learn when to allow data to enter, leave or be deleted through the iterative process of making guesses, back-propagating error, and adjusting weights via gradient descent.

2.3 Training an LSTM on the Laser Data

The network architecture consists of a sequence input layer, an LSTM layer with 200 hidden units, a fully connected layer, and a regression layer. The network was trained with Adam for 500 epochs, and the learning rate is decreased after 250 epochs. To prevent gradients from exploding, a gradient threshold of 1 is used. The network was first trained with a lag of 1. That is, at each time step of the input sequence, the LSTM network learns to predict the value of the next time step. This network was then used to forecast the next 100 timesteps, without updating with observed values. The prediction and errors are shown in Figures 6a and 6c. Note that it is not able to accurately predict the qualitative behaviour of the laser. The network was then used to predict the next 100 timesteps, except the network state was updated after every prediction. The prediction and errors are shown in Figures 6b and 6d. By updating the states after every prediction, we are able to accurately predict the 'drop' in the signal given by the laser.

Ultimately I could not evaluate the effect of using different lag values for the LSTM as I had trouble formatting the data and training the LSTM with lag. My intuition would be that the RMSE for the prediction would be lower as the lag increased, and that in the case of forecasting future timesteps without updating the network state, it might have been possible to predict the qualitative behaviour correctly with sufficient lag.

Comparing the results from the LSTM to the recurrent neural network, it is clear that the LSTM performs better. Furthermore, the LSTM took less time to train – in order for the recurrent network to adequately capture the qualitative behaviour of the signal, it must be trained with a large number of inputs (large lag

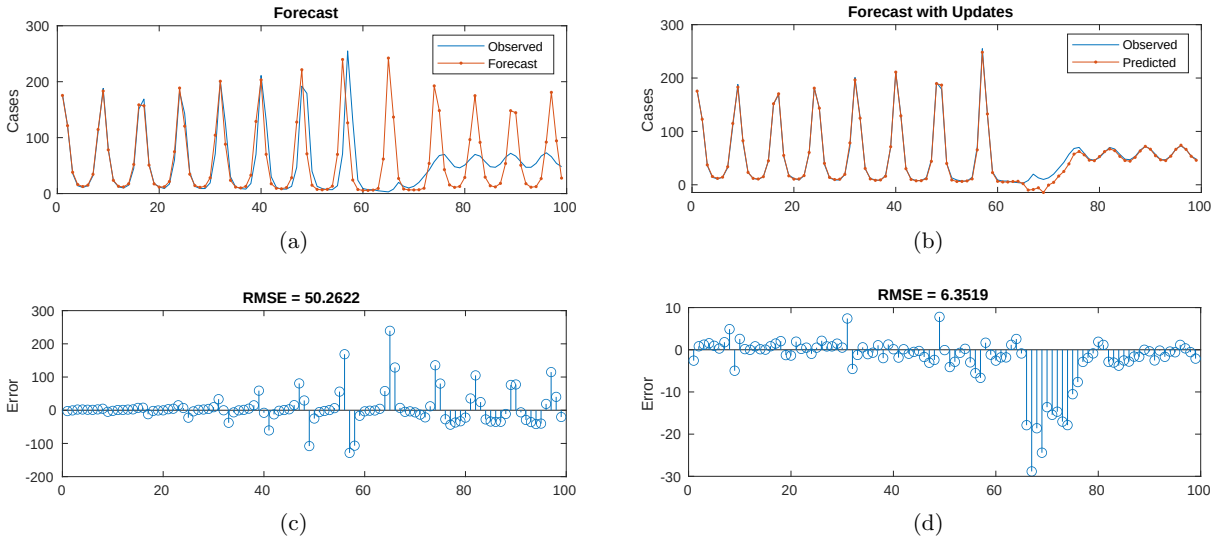


Figure 6: Predictions and RMSE for both the forecast *(left)* and updating the network with observed values *(right)*.

value). Whereas, in the case of the LSTM, we were able to accurately predict the signal using a lag value of one, and so the training time was much shorter.

Deep Feature Learning

James O'Reilly
james.oreilly@student.kuleuven.be

Introduction

In this assignment I discuss different methods for learning features from and classifying image data. I will first discuss principle component analysis, before briefly discussing stacked autoencoders, and then finally I will discuss convolutional neural networks using the AlexNet architecture as an example.

Principal Component Analysis

Principal component analysis is a dimensionality reduction methods which projects data from a higher dimension onto a lower dimension subspace such that the maximal amount of variability in the data is retained. The idea is that the low dimension representation will capture the structure of the data and the most salient features. A higher dimensional representation can then be reconstructed (imperfectly) by projecting from this lower dimensional subspace. I don't have space to full explain the mathematics behind PCA here, or the steps involved in the PCA algorithm, so I will include a link to a detailed review by Jolliffe.[1]

PCA for Random and Correlated Data

Here we compare the reduction of random and highly-correlated data. Intuitively, with randomly generated data, the distribution of eigenvalues should follow a Gaussian distribution, and so it is unlikely that the majority of variance in the dataset will be contained in only a small fraction of the dimensions. A 50×500 matrix of Gaussian random numbers was generated and PCA was used to try to reduce the dimensions. From this reduced representation, we tried to reconstruct the original dataset and measured the error of this reconstruction. The error is calculated as the RMSE between the original and reconstructed dataset. The percentage of variance captured in the lower dimensional representation was also calculated. Figure 1 shows their relationship.

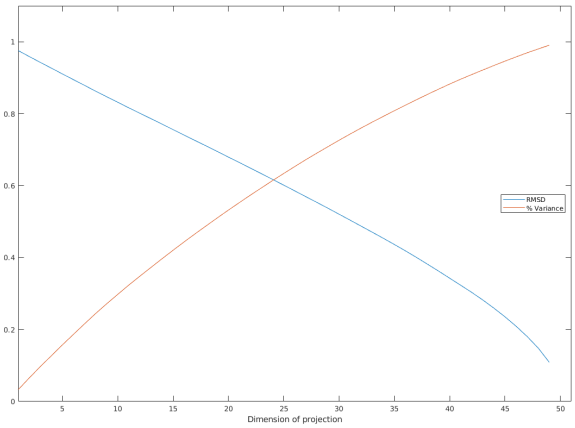


Figure 1: Caption

The same procedure was also performed on the p-component of the `choles_all` data, which is a 21×264 matrix. This data is highly correlated. Plots for the reconstruction error and the percentage of total variance contained in the lower dimensional representation are given in Figure 2. Note that the reconstruction error only

marginally improves after the dimensionality of the subspaces increases past 1. This is due to the fact that over 95% of the variance in the dataset is contained in this single dimension, as can be seen in the second plot.

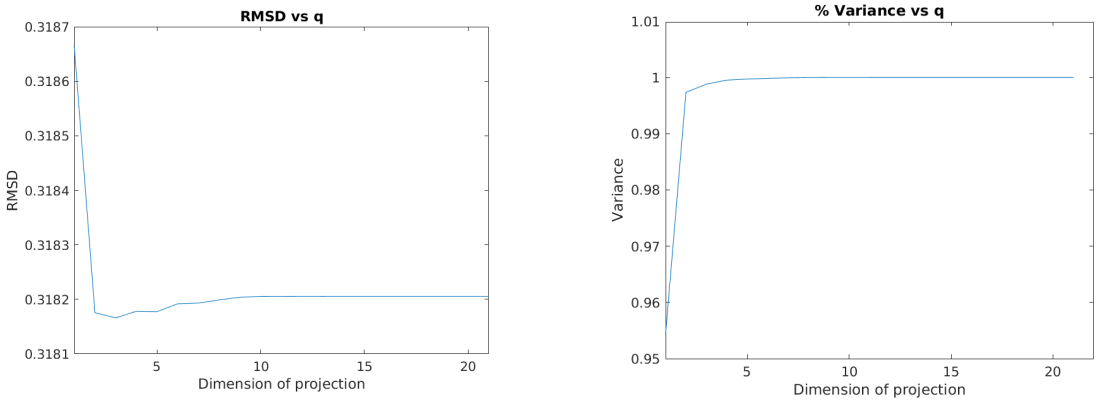


Figure 2: RMSD and variance for the `choles_all` vs dimensionality of subspace.

Principal Component Analysis on Hand-written Digits

Here we perform PCA on handwritten images of the digit '3' taken from the US Postal Service database. Displayed in Figure 3 is the mean '3', along with the reconstructed 3's from dimensionality reduction to one, two, three, and four dimensions.

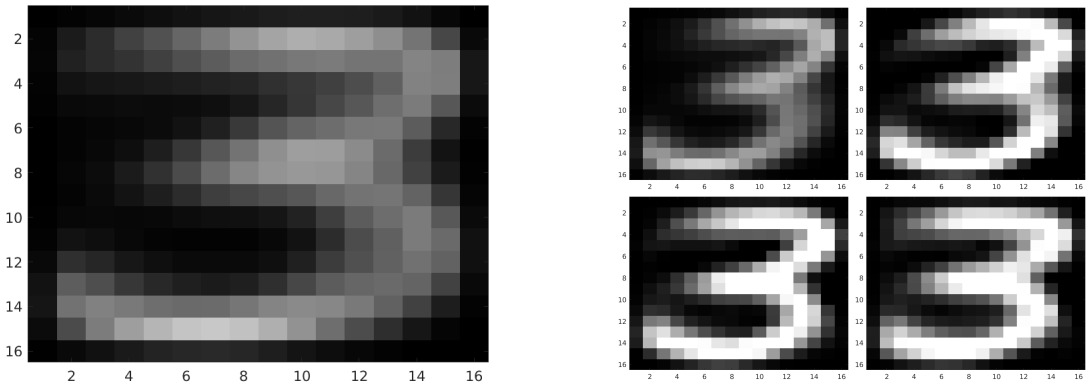


Figure 3: The mean '3' is given on the left. On the right are the reconstructions from one, two, three, and four dimensions (top left through bottom right).

The entire dataset was then compressed onto q principal components for $q = 1, \ldots, 50$ and the RMSE was plotted. The percentage of total variance was also plotted. The inverse relationship between these two values is clear in Figure 4.

The previous graph should illustrate an important fact – the squared reconstruction error induced by not using a certain principal component is proportional to its eigenvalue (eigenvalues are a measure of variance). That is why if the eigenvalues fall off quickly then projecting onto the first few components gives very small errors because the sum of the eigenvalues that we are not using is not very large.

Stacked Autoencoders

In this section we compare the performance of a stacked autoencoder to a normal mulltilayer neural network. The default stacked autoencoder (with which I will be comparing both the multilayer network and other stacked autoencoders) consists of two autoencoders and a softmax layer. The first autoencoder has 100 hidden units and is trained for 400 epochs. The second autoencoder has 50 hidden units and is trained for 100 epochs. The

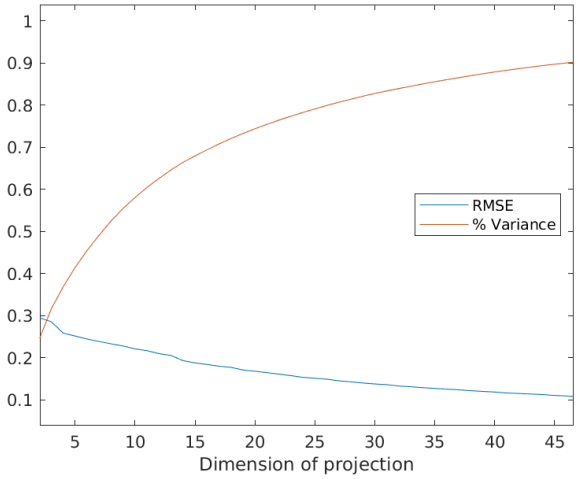


Figure 4: RMSE and % of total variance for the first 50 eigenvalues of the `threes` dataset. Note the inverse relationship between their slopes and values.

default stacked autoencoder has a classification accuracy of 85%. I will first try to optimise the hyperparameters of this stacked autoencoder (number of epochs, number of hidden units, number of layers).

Naturally, time is one major constraint, and so I first observed that the first autoencoder barely improves its performance after 250 epochs. Setting the number of epochs to 250 affords me some extra time which can be used elsewhere. I then observed that the second autoencoder was still improving at 100 epochs. I therefore increased the number of epochs to 200. These two changes increased the classification accuracy to 96 percent (without fine-tuning). Next I looked at changing the number of neurons in each layer. Changing the number of neurons in each layer affects the dimensionality of the low-dimension representation formed by the autoencoder. Increasing the number of hidden units in either the first of second layer had little impact on the classification accuracy. I then decreased the neurons in each layer to see if a lower-dimensional representation could still yield the same classification accuracy. Reducing the number of hidden units in the second autoencoder to 25 significantly impacted classification accuracy, reducing it to 77%. With this information, I was content with 100 hidden units in the first autencoder and 50 in the second.

Lastly, I needed to analyse the impact of additional layers on performance. Introducing a third autoencoder raises a number of questions, the primary question being how should I change the dimensionality of the other autoencoders in the network. I firstly tested adding a third autoencoder with 30 hidden units. Doing so reduced the classification accuracy. I then experimented with three autoencoders, with 150, 100, and 50 hidden units respectively. Doing this reduced the classification accuracy to less than 50%. Autoencoders with many hidden layers seldom work well because if the initialisation weights are too large, backpropogation finds a poor local minima. If the initialisation weights are too small, it takes a long time to optimise. If the initial weights are close to the solution however, then finetuning with backprop works well, however. Good initialisation weights can be learned using restricted Boltzmann machines.

The final architecture consists of two autoencoders and a softmax layer. The first autoencoder had 100 hidden units and was trained for 250 epochs. The second had 50 hidden units and was trained for 200 epochs. After fine-tuning, the network has a classification accuracy of 99.2%. The normal feedforward neural network, with two layers and the same number of neurons in each layer as the stacked autoencoder, had a classification accuracy of 95%.

Finetuning

The results for the stacked neural network can be improved by performing backpropagation on the whole multilayer network. This process is often referred to as fine-tuning. You fine-tune the network by retraining it on the training data in a supervised fashion. Why does this improve performance? Before finetuning, each of the autoencoders have been pre-trained individually: the weights for layer one and layer two and first learned and then frozen. In practice, initialising the weights in this manner has been shown to decrease training times and

start the fine-tuning stage much closer to good minima than random weight initialisation. The global fine-tuning uses backpropagation through the whole autoencoder to fine-tune the weights for optimal reconstruction.

Convolutional Neural Networks

In this section, a convolutional neural network (CNN) is trained on the Caltech 101 dataset for feature extraction. The network uses the AlexNet architecture, with five convolutional layers, each followed by a ReLu layer and a max-pooling layer.[2] Following the these layers are three fully-connected layers. A convolution is the simple application of a filter to an input that results in an activation. Repeated application of the same filter to an input results in a map of activations called a feature map, indicating the locations and strength of a detected feature in an input, such as an image.

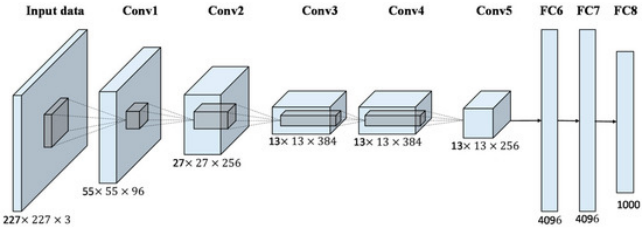


Figure 5: A simplified schema of the AlexNet architecture.

What do the weights in the first convolutional layer represent?

A visualisation of the weights in the first convolutional layer is given in Figure 6. This figure shows the 96 convolutional kernels learned by the network. Many of these kernels are frequency and orientation selective, selecting for edges in different orientations at different frequencies. Note also that some of the kernels select for edges with specific colours, or instead select for coloured blobs. Originally I was confused as to why this was only the case for some of the kernels. However, according to the paper on which this architecture was based, this network was trained with two different GPUs. The kernels on GPU1 are colour-agnostic, while the kernels on GPU2 are largely colour-specific. It is for this reason that only some of the kernels contain information about colour, and not due to the actual architecture of the network itself.

Inspect layers 1 to 5. If you know that a ReLU and a Cross Channel Normalization layer do not affect the dimension of the input, what is the dimension of the input at the start of layer 6 and why?

There is a simple formula for calculating the dimension of output from a convolutional or max pooling layer. The dimension is given by:

$$\text{output width} = \frac{W - F_w + 2P}{S_w} + 1$$

$$\text{output height} = \frac{H - F_h + 2P}{S_h} + 1$$

where W amd H are the width and height of the input, respectively. F is the size of the filter, P is the padding, and S is the stride size. The dimension of the original input into the network are $227 \times 227 \times 3$. The first convolution layer uses 96 kernels of size 11×11 with a stride of 4 and a padding of 0. As the inputs have the same width and height, we can use the same formula for both:

$$\frac{227 - 11 + 2(0)}{4} + 1 = 55$$

The output of the first convolution layer is therefore $55 \times 55 \times 96$. As both the ReLU layer and cross channel normalisation layer do not affect the dimension of the input, we next need to look at the max pooling layer. The same formula can be applied, giving

$$\frac{55 - 3 + 2(0)}{2} + 1 = 27$$

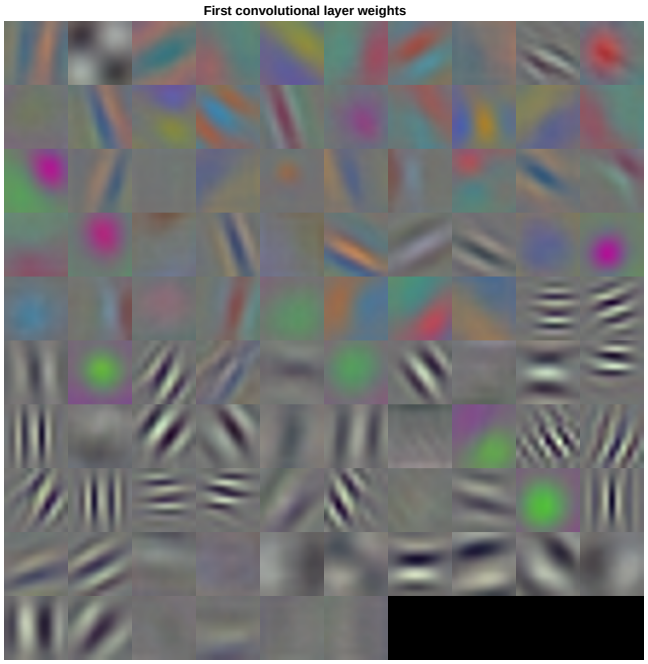


Figure 6: 96 convolutional kernels of size $11 \times 11 \times 3$ learned by the first convolutional layer on the $224 \times 224 \times 3$ input images.

The input to the second convolutional layer is therefore $27 \times 27 \times 96$.

What is the dimension of the inputs before the final classification part of the network (i.e. before the fully connected layers)? How does this compare with the initial dimension?

Applying the same formula for each of the convolution and max-pooling layers, we find the dimension of the inputs to the first fully-connected layer is $6 \times 6 \times 256$. The initial dimension was $227 \times 227 \times 3$. If we compute the number of actual input neurons in each case, there are 154587 initial input neurons, and 9216 input neurons to the first fully-connected layer. Through the convolution and max-pooling layers, the input size has been down-sampled by almost a factor of 15.

Briefly discuss the advantage of CNNs over fully connected networks for image classification.

The three primary advantages of CNNs for image classification come from local connectivity, parameter sharing, and translation invariance:

- **Local Connectivity:** Convolutional neural networks exploit spatially local correlation by enforcing a sparse local connectivity pattern between neurons of adjacent layers – each neuron is connected to only a small fraction of the input neurons. As each of the neurons are only locally connected, they can take advantage of local spatial coherence of images. This allows for a significant reduction in the number of operations needed to process an image. If the network was fully connected, there would likely be problems with both storage and computing the linear activations of hidden units.
- *Parameter Sharing:* This is the idea that units in the same convolution kernel share some parameters across layers. This further reduces the effective number of parameters, and therefore improves computational efficiency.
- **Translation Invariance:** Translation invariance (aka shift invariance) in a CNN refers to the fact that the CNN's output will be the same regardless of the position of the object in the image – this is useful if you just want to classify the image and don't care about the position. This is achieved in two ways. Firstly, due to parameter sharing, the weights of the filter are shared across patches of the image, and so the weights learnt will be invariant to position. Secondly, by performing max pooling we approximate translation invariance since subsequent layers of the CNN don't care about the specific position in the patch that the max value was in.

References

[1] Jolliffe, I. T., and Cadima, J. Principal component analysis: a review and recent developments. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 374*, 2065 (2016), 20150202.

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

Generative Models

James O'Reilly
james.oreilly@student.kuleuven.be

Restricted Boltzmann Machines

Introduction to RBMs

A detailed introduction to restricted Boltzmann machines (RBMs) is given in Fischer and Igel.[1] In short, an RBM is a generative stochastic artificial neural network that can learn a probability distribution over its set of inputs. After successful learning, an RBM provides a closed-form representation of the distribution underlying the observations. It can be used to compare the probabilities of (unseen) observations and to sample from the learned distribution (e.g., to generate new images), in particular from marginal distributions of interest. Below we investigate the effects of changing different training parameters when training RBMs (number of epochs, number of components, Gibbs sampling steps, etc). A practical guide for training RBMs is given by Hinton.[3]

Exercises

In this section, we investigate the effect of changing training parameters for an RBM that is trained on the MNIST dataset. Increasing the number of iterations (number of forward and backward passes through the network) or increasing the number of components (hidden units) should increase the performance of the network. Note that RBMs are particularly difficult to train. As the partition function which normalises the probability function is intractable, we cannot estimate the log-likelihood $\log(P(x))$ during training, and therefore have no direct metric for choosing network hyper-parameters. Instead, more tractable functions can be used as a proxy to the actual likelihood. One example is the pseudo-likelihood, which will be used here.

As the number of iterations was increased, there was a steady increase in the pseudo-likelihood score. This increase eventually plateaued after certain number of iterations. Figure 1 shows the pseudo-likelihood over 30 iterations, measured for a network with 20 hidden units and a learning rate of 0.01. The pseudo-likelihood also increases as the number of hidden units increases.

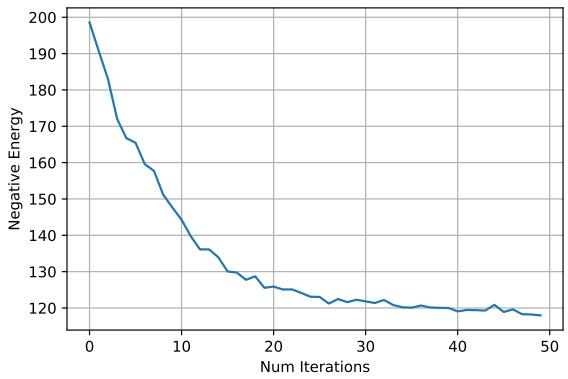


Figure 1: Negative energy vs iterations. Here the negative of the pseudo-likelihood score is used for clarity of presentation. Note the plateau after a certain number of iterations.

Samples are obtained from the test image using Gibbs sampling. The results for different numbers of Gibbs

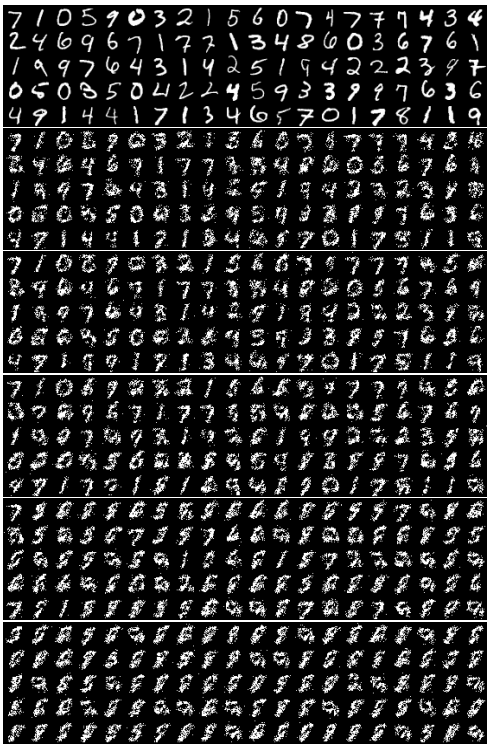


Figure 2: Results for different Gibbs sample steps. The first figure is the original test image. The images below are for steps of 1, 10, 100, 1000, and 10000, respectively.

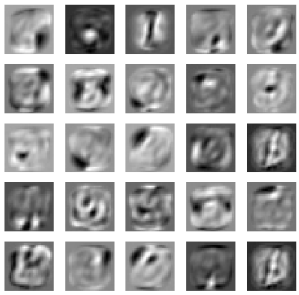
sample steps are displayed below, and compared with the original test data. Looking at 2, it can be seen that as the number of sample steps increases, the figures become less clear and more homogeneous. I'm not sure why this is the case.

Reconstruction of Unseen Images

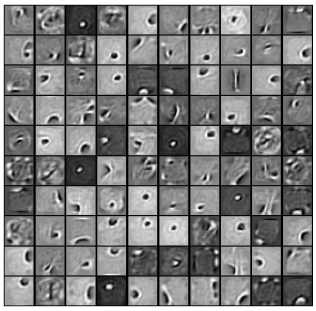
Here we remove pixel rows from certain images. Using the recently trained RBM, we will then try to reconstruct the original images. By changing the number and position of the rows that are removed, we can observe differences in the ability of the RBM to reconstruct the original images. Increasing both the number of hidden units and the number of iterations improved the ability of the network to reproduce the original images. Even with very few rows removed, reconstruction with the shallow RBM with 20 hidden units trained for 30 epochs was rather poor. Removing more than seven rows in any position resulted in poor reconstruction. Reconstruction was improved if the rows were removed from the center of the image, perhaps because the defining characteristics some digits are located at the top and bottom of the image.

Deep Boltzmann Machines

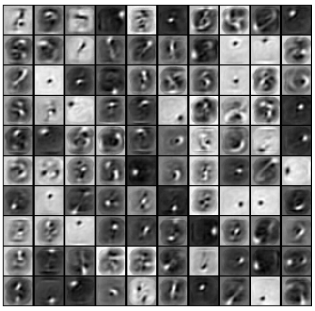
Deep Boltzmann machines can be understood as a series of restricted Boltzmann machines stacked on top of each other.[5] The hidden units are grouped into a hierarchy of layers, such that there is full connectivity between subsequent layers, but no connectivity within layers or between non-neighbouring layers. A DBM that was pre-trained on the MNIST database was used in this assignment. The interconnection weights from the pre-trained DBM can be extracted and compared with those from the RBM we trained previously. In the first layer of the DBM, the filters focus heavily on identifying loops and teh position of these loops. The area within these loops is black. While in the RBM, the filters contain larger, more complex structures with loops, lines, and differences in brightness throughout the image. The RBM filters are aslo much more noisy. This is the case because the DBM has multiple layers, and so the earlier layers can focus on identifying smaller, mores specific features which can then be combined in the later layers. This is not possible in the RBM, as it only has one layer, and so it must attempt to capture lots of different features (lines, loops, curves, etc) in the this single



(a) RBM hidden layer



(b) DBM layer one



(c) DBM layer 2

Figure 4: Sample from the DBM with Gibbs = 100.

layer of filters.

When comparing the first layer of the DBM to the second layer, we can see that the filters contain more complex structures, with combinations of loops and some small lines and edges. Most notably, the many of the interconnection weights filter for a circular area in the center of the image, showing the darker area with which the digit would be surrounded.

Samples were then taken using this deep Boltzmann machine (see Figure 4). When compared to the samples taken earlier from the RBM, the quality is markedly better and there is less noise.

Generative Adversarial Networks

In this section we train a deep convolutional generative adversarial network (DCGAN) on the CIFAR dataset. The architectural guidelines for DCGANs set out by Radford & Chintala[4] suggest that:

- Any pooling layers be replaced with strided convolutions (discriminator) and fractional-strided convolutions (generator)
- Use batch normalisation in both the generator and the discriminator
- Remove fully connected hidden layers for deeper architectures
- Use ReLU activation in generator for all layers except for the output, which uses Tanh
- Use LeakyReLU activation in the discriminator for all layers

I chose to use class 3 of the CIFAR dataset (cats), and then trained the DCGAN using the recommendations given above.

GANs and Stability

I struggled to work with DCGAN notebook provided for this assignment – in particular I could not plot the loss or accuracy. Ideally, I would plot the generator and discriminator loss over time, to view the stability of the network. It can be challenging to strain a stable GAN. The reason is that the training process is a dynamical system and inherently unstable, resulting from the simultaneous training of two competing models. The generator model and the discriminator model are trained simultaneously in a 'game', and so any improvement to one model comes at the expense of the other. The ultimate goal of training is therefore to find an equilibrium where the models perform equally well. The concern is that the networks don't converge. GANs will often become stuck in a state where the generator oscillates between generating different kinds of samples.[2] One type of model failure, known as 'mode collapse', occurs when multiple inputs to the generator result in the generation of the same output.

It is also difficult to evaluate whether a GAN is performing well during training. Looking at the loss in the models is not sufficient and often the best metric is a subjective review of the generated images. In the paper "Improved techniques for training GANS", Goodfellow notes

> *Generative adversarial networks lack an objective function, which makes it difficult to compare performance of different models. One intuitive metric of performance can be obtained by having human annotators judge the visual quality of samples.*[6]

Understanding that the loss is not the best metric of model performance during training, it is still interesting to note that the discriminator loss varied between 0.45 and 0.85 during training, and did not converge by the end of training. The generator loss also did not converge, and so the GAN was ultimately unstable.

Optimal Transport and the Wasserstein Metric

In statistics, the Wasserstein metric or Earth Mover's Distance (EMD) is a measure of the distance between two probability distributions over a region D (note that this is simplified, I'm not going to give a formal definition or discuss metric spaces.) Informally, the Wasserstein metric can be interpreted as the minimum cost of transforming one probability distribution into the other, for some defined cost function C. Naturally, this definition is only The above definition is valid only if the two distributions have the same integral.

Optimal Transport and Colour Swapping

One simple application of Wasserstein metric is for the optimal transport (OT) of colour between two images using their normalised colour histograms. Each normalised colour histogram defines a probability distribution. Here we use OT to transfer colour between the two images given in Figure 5.

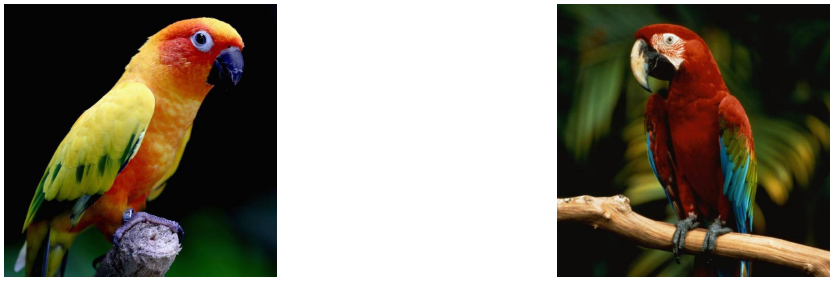


Figure 5

I'm not going to go into detail about how exactly the optimal transport algorithm works. If you're interested in a gentle introduction to OT and then I'd suggest this video series by Marco Cuturi. The results of the optimal transport are given in Figure 6.

Figure 6: Images with colours swapped using optimal transport.

Looking at the first two columns of images in Figure 6, we can see that the colour histograms of each image have been transformed into the other, but not simply by swapping pixels. For example, note the blue on the wings of the parrot in Image 2. When transporting this colour to Image 1, we want to transport it to the pixel that is closest in terms of colour (assuming there is no other transportation which minimises the cost further). Looking at the adapted version of Image 1, we can see this blue was transported to beak, the pixels of which were blue-purple in the original image.

Wasserstein GANs

Two common divergences used in generative models are the Kullback-Leibler (KL) divergence and the Jensen-Shannon (JS) divergence. These divergences give a measure of 'distance' between two probability distributions. For two normal equivariant probability distributions p and q with mean μ and τ respectively, we can plot the KL and JS divergence as the two distributions as τ moves further from μ. As τ increases, the curve will flatten and therefore the gradient of the divergency will diminish, which means that the generator learns nothing from gradient descent.

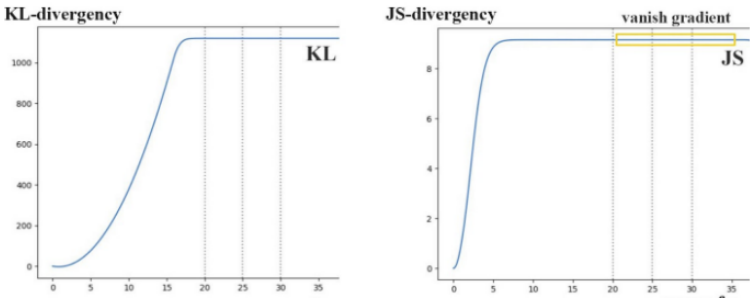


Figure 7: A figure showing the diminishing gradient problem for both the KL and JS divergence. The x-axis is the difference between the τ and μ for the equivariant probability distributions p amd q.

Wasserstein GANs (WGANs) use the Wasserstein metric as a cost function as it has a much smoother gradient which does not diminish as the distributions are separated. This means that there won't be the same vanishing gradient problem and the generator can learn even at the beginning when it is not producing good images and the distributions are very far apart. Without going into the details, the equation for the Wasserstein metric is highly intractable. Using the Kantorovich-Rubinstein duality, the calculation can be simplified to

$$W\left(\mathbb{P}_r, \mathbb{P}_\theta\right) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{x \sim \mathbb{P}_r}[f(x)] - \mathbb{E}_{x \sim \mathbb{P}_\theta}[f(x)]$$

where the function f is a 1-Lipschitz function following a constraint called the Lipschitz constraint. In order to enforce this constraint, Wasserstein GANs apply clipping to restrict the maximum weight values in the discriminator. This is called weight-clipping. However, using weight-clipping to enforce the Lipschitz constraint can lead to problems with convergence and stability. Instead of using weight-clipping, Wasserstein GANs can instead use a gradient penalty to enforce the Lipschitz constraint. Using a gradient penalty makes training more computationally intensive, but can significantly improve the stability of the GAN. As WGANs with gradient penalty are more stable, they can be trained for a much longer time on more powerful networks as we know they will converge.

For this assignment, A standard GAN, a Wasserstein GAN with weight clipping, and a Wasserstein GAN with gradient penalty were each trained on the MNIST dataset. These networks are not convolutional and so performance is not optimal, but we are mostly interested in the relative stability and performance of these different GANs. The standard GAN was unstable and did not converge. The Wasserstein GAN with weight clipping converged. Shown in Figure 8 are the final generated images from the standard GAN and the Wasserstein GAN with weight penalty.

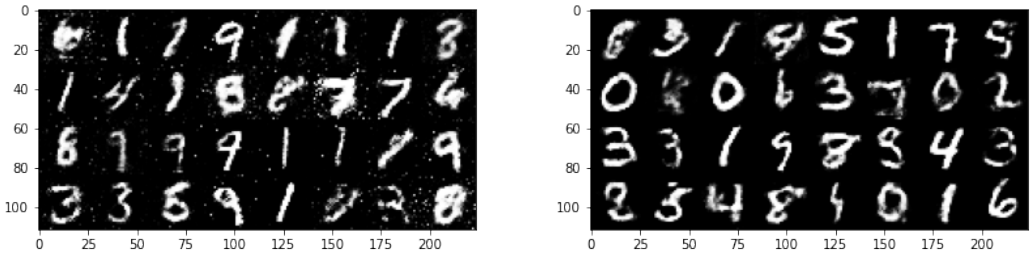


Figure 8: Generated images from the standard GAN *(left)* and Wasserstein GAN with weight penalty *(right)*

References

[1] Fischer, A., and Igel, C. An introduction to restricted boltzmann machines. In *Iberoamerican congress on pattern recognition* (2012), Springer, pp. 14–36.

[2] Goodfellow, I. Nips 2016 tutorial: Generative adversarial networks. *arXiv preprint arXiv:1701.00160* (2016).

[3] Hinton, G. E. A practical guide to training restricted boltzmann machines. In *Neural networks: Tricks of the trade*. Springer, 2012, pp. 599–619.

[4] Radford, A., Metz, L., and Chintala, S. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434* (2015).

[5] Salakhutdinov, R., and Larochelle, H. Efficient learning of deep boltzmann machines. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (2010), pp. 693–700.

[6] Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., and Chen, X. Improved techniques for training gans. In *Advances in neural information processing systems* (2016), pp. 2234–2242.