Artificial neural networks - Exercise session 1

Supervised learning and generalization

• *Function approximation: comparison of various algorithms: Take the function y = sin(x^2) for x = 0 : 0.05 : 3π and try to approximate it using a neural network with one hidden layer. Use different algorithms. How does gradient descent perform compared to other training algorithms?*

In order to compare algorithms here we have trained each one 10 times with the specified configuration and taken the mean of their performance in various categories.

	traingd	traingda	traincgf	traincgp	trainbfg	trainlm
Speed (secs)	2.896	2.227	4.3162	3.381	10.269	6.312
R epoch 1	0.07607	0.0762	0.2495	0.2294	0.1076	0.8271
R epoch 5	0.07645	0.0797	0.7681	0.7674	0.8523	0.9951
R epoch 1000	0.668	0.889	0.999	0.999	1	1
MSE epoch 1000	0.29	0.1	9.28e-04	9.36e-04	1.67e-04	6.69e-08

Traingd and traingda are the quickest algorithms to train due to having only to specify search direction and step size. However their convergence rate is slow, as they always choose minus the gradient of the cost function as the search direction, which is not always the most efficient choice. This does improve with traingda, as the learning rate is adjusted during training, meaning that at each new epoch weights and biases are calculated using the new learning rate. Traincgf and traincgp perform similarly to each other, though the latter takes less time to train. They converge much more efficiently than traingd and traingda as via conjugate gradients, these algorithms seek directions conjugate to the previous line search direction, thereby not undoing progress. They also do not need to store hessian approximations, unlike the remaining algorithms. Trainbfg being a quasi-Newton algorithm performs very well, estimating the underlying function with an error rate of 0.09-0.10% after 1000 epochs. However it is computationally expensive due to needing to store approximated hessians. Trainlm, the Levenberg-Marquardt algorithm, is the best performing algorithm: in 7/10 cases, it arrives at R=1 after 5 epochs, and after 1000 epochs it has estimated the underlying function with an error rate of 0.09-0.10%. It is also considerably faster to converge than trainbfg in this case, likely because we are dealing here with a small-scale problem. However, low MSE and high R values for trainlm and trainbfg may indicate overfitting, so we need to keep this in mind, particularly for trainlm.

With noise (dx=0.01, x=0:dx:3*pi;y=sin(x.^2), sigma=0.2, yn=y+sigma*randn(size(y))

	traingd	traingda	traincgf	traincgp	trainbfg	trainlm
R epoch 1	0.0956	0.0956	0.0989	0.0989	0.0974	0.8597
R epoch 5	0.0950	0.1008	0.6782	0.6775	0.8933	0.9949
R epoch 1000	0.7276	0.8518	0.9948	0.9948	0.9859	0.9936
MSE epoch 1000	0.27	0.17	0.04	0.04	0.04	0.04

Adding noise, we see that traingd actually improves its performance, likely as a result of increasing the number of datapoints. Trainbfg and trainlm no longer reach R=1, which is related to the fact that when we add noise, we teach the network that it is less able to make high certainty predictions. This helps improve generalisation and avoid

overfitting. Overall the MSE counts are higher with noise than without, plateauing similarly for the 2 conjugate gradient algorithms, the quasi-newton as well as Levenberg-Marquardt.

Personal regression example

Here we performed some experiments – each averaged over 10 runs – to approximate an unknown nonlinear function using a feedforward artificial neural network. Of the 13,600 data points provided, we drawn 1000 samples for train, validation and test respectively. The points selected for train, validation and test are samples from the 13,600 size data-set and are saved as a .mat file. We included the validation set in the test set. The model is then optimised on the validation set which helps to avoid overfitting on the train set. Meanwhile the test set is used to evaluate the performance of the model when exposed to unseen data, with the goal able to generalise well. To keep the comparisons over several experiments standardized, the same variable is reused. The default parameters for the experiments are as follows:
[**number of neurons**=50, **number of hidden layers**=1, **epochs**=100, **transfer function**=logsig]. Weights are initialised randomly in order to see sufficient variance over multiple trainings of the same algorithm.

1. Computation speed (seconds)

Gd	**Gda**	**Cgf**	**Cgp**	**Bfg**	**Lm**
0.909	0.984	1.866	1.584	2.377	1.747

The above figures were noted averaging over 10 runs, keeping all parameters static.
Vanilla Gradient Descent was the fastest algorithm with an average time of 0.9 seconds to train for 100 epochs, while BFGS quasi-newton is was the slowest at 2.377. The average over all algorithms was around 1.6 seconds which is quite small. Training time can considered as not a decision critical parameter given the size of data and complexity of the function that we need to approximate.

2. Accuracy in terms of error rate

MSE at epoch	**Gd**	**Gda**	**Cgf**	**Cgp**	**Bfg**	**Lm**
1	5.542	5.277	3.147	3.518	2.451	0.021
5	2.730	2.395	0.305	0.375	0.2532	0.003
10	1.847	1.600	0.118	0.123	0.059	1.02E-03
20	1.178	1.005	0.048	0.0522	0.020	2.93E-04
50	0.722	0.342	0.020	0.023	0.010	6.29E-05
100	0.474	0.215	0.015	0.016	0.005	2.09E-05

Conjugate Gradient Fletcher-Reeves, Quasi-Newton and Levenberg-Marquardt algorithms demonstrate the best MSE at 100 epochs, the values of which range from 0.0002 to 0.02. However, very low MSE may indicate overfitting.

3. Memory (in terms of model complexity)

With this experiment, we wanted to investigate the memory efficiency of each algorithm. We aimed to achieve an MSE of 0.02 and find the most memory efficient configuration for each algorithm. The following results were recorded when a target MSE value of 0.02 was reached.

	Gd	Gda	Cgf	Cgp	Bfg	Lm
Hidden Layers	3	3	1	1	1	1
Neurons	[50,50,10]	[50,50,20]	[50]	[40]	[40]	[40]
Epochs required	652	239	54	48	28	2

In order to achieve a 0.2 MSE using the gradient descent algorithms, we needed to train for many epochs using three hidden layers. The other algorithms were all able to reach the same low error rates with much smaller and more memory efficient architectures. Even smaller numbers of neurons can also produce good results but we then run the risk of underfitting.

4. Efficiency

Here we measured efficiency in terms of epochs using the default parameters with unlimited epochs and results were recorded when an MSE value of 0.02 on the validation set was reached.

Gd	Gda	Cgf	Cgp	Bfg	Lm
8152	2593	63	56	19	2

The two gradient descent algorithms are the poorest performers in terms of memory efficiency. Levenberg-Marquardt is by far the most efficient, followed by the BFGS quasi-newton. The two conjugate gradient algorithms perform similarly as we would expect.

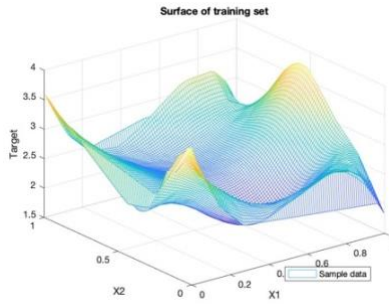
5. Transfer functions

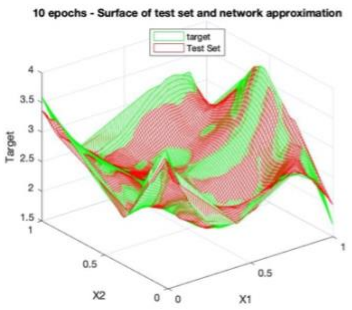
The below results were recorded training Levenberg-Marquardt with 40 neurons in a single hidden layer for 100 epochs.

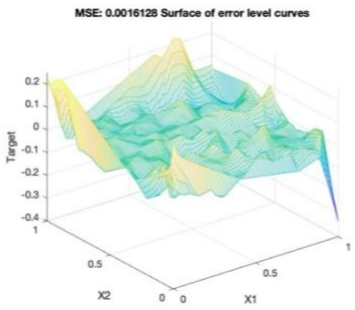
logsig	elliotsig	elliot2sig	netinv	radbasn	radbas	poslin
2.09E-05	0.0013	5.55E-04	0.54	9.88E-06	3.17E-05 (less consistent)	1.32E-03

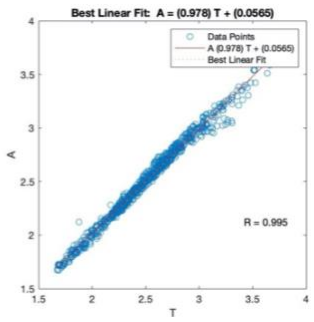
6. Best Model Selection

Following the above experiments, we chose the following parameters to approximate our unknown function. We selected Levenberg-Marquardt as our algorithm which had outperformed every other algorithm in all categories except training time. We trained the model using 40 neurons in a single hidden layer for 10 epochs using radbasn as the transfer function. Fewer neurons (e.g. 10) sometimes produced equally good results but had the tendency to underfit which we determined based on frequent subsequently failed validation checks. Similarly more epochs were shown to produce an overfitting effect as the R-value converged to 1, whereas with a limit of 10 epochs we observed that it never reached 1. Generally the performance is very similar across training, validation and test tests with only a very slight increase in error rates between train and test. This is corroborated by the graphs below which show that both the train and test set look very similar to each other, meaning that the train set was a good approximation of the test. Although we are already achieving very good results, one method to improve further would be adding an L2 regularisation term to the loss function which would reduce the impact of weights, thus reducing the tendency to overfit. This improves the ability of the model to generalise on new data by not overrelying on the train set. There are other techniques we could use if we had more data, such as k-fold cross validation, but since there are only a total of 3000 data points in the whole dataset we do not think this would lead to significant improvements in this case.









Bayesian Inference of Network hyperparameters

	Trainbr	**Trainlm**	**Trainbr**	**Trainlm**	**Trainbr**	**Trainlm**
	50 neurons		**150 neurons**		**200 neurons**	
	No noise (dx=0.05, x=0:dx:3*pi;y=sin(x.^2), sigma=0.2, yn=y+sigma*randn(size(y))					
R epoch 1	0.847	0.827	0.944	0.957	0.955	0.983
R epoch 5	0.988	0.995	1	0.962	1	1
R epoch 1000	1	1	1	0.962	1	1
Best perf epoch (MSE goal: 0.002)	6	28	3	2	2	2
	Noisy (dx=0.01, x=0:dx:3*pi;y=sin(x.^2), sigma=0.2, yn=y+sigma*randn(size(y))					
R epoch 1	0.801	0.860	0.985	0.991	.976	0.981
R epoch 5	0.995	0.995	0.991	0.987	.985	0.981
R epoch 1000	0.995	0.994	0.986	0.986	.981	0.983
Best perf epoch	169	1000	928	1000	970	1000
Best perf MSE	0.035	0.034	0.026	0.026	0.02	0.02

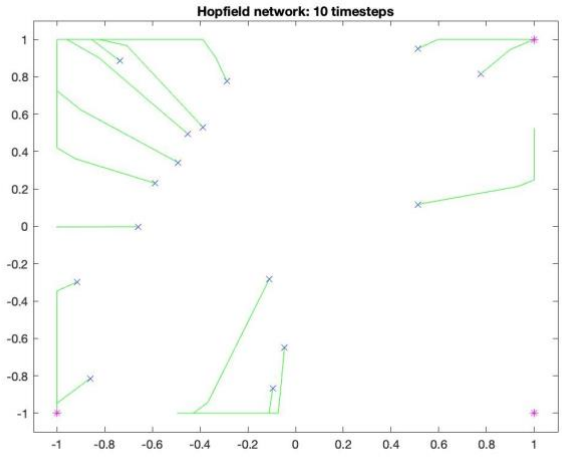
We compared trainbr to trainlm, both because the latter was the strongest performing from the previous exercise, and because trainbr updates weights and biases according to the Levenberg-marquardt optimization. In the case where we have a relatively small number of parameters (50 neurons), trainbr converges in fewer epochs than trainlm. This also occurs in all instances of noisy data. Trainlm still outperforms or matches in the noiseless cases with high neuron numbers, but may be overfitting. Regarding trainbr, due to the regularization inherent in the Bayesian inference of hyperparameters, the overparametrized networks (150 and 200 neurons) still perform well. However, considering the occam factor, it is important to consider that the strength of our performance depends on finding a suitable distribution for our number of inputs (189 in the noiseless case and 943 in the noisy case). Therefore, even given the weight decay term, it is recommended not to overparameterize. In general, trainbr works by inferring the most probable parameters for the model given the training data, and does not need a validation set as the regularization term included in the error function ensures the network will generalise well. It operates on 3 levels of inference, first inferring the optimal parameters (weights), then the optimal hyperparameters α and β, then the optimal model (simpler being preferred).

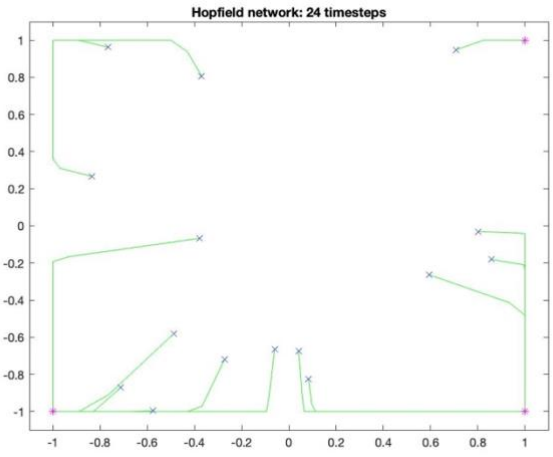
Artificial neural networks - Exercise session 2

Recurrent neural networks

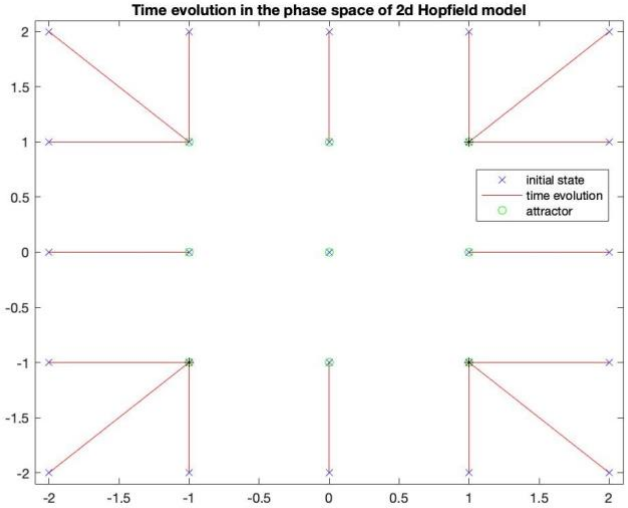
1. Hopfield Network

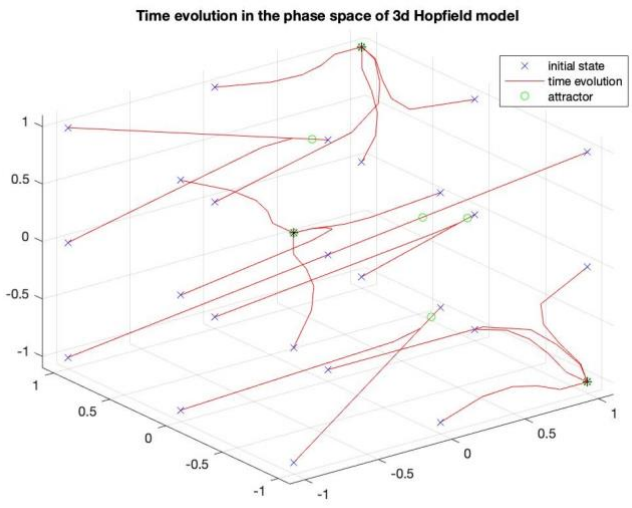
Starting with attractors $T = [1\ 1; -1\ -1; 1\ -1]^T$ and 15 random initialisations, we experimented with different numbers of timesteps. Training involves the use of an energy function which gradually evolves to reach the lowest state of energy. The energy is lowest when the current state has high overlap with the synaptic drive to the next state, in other words, when it has reached an attractor, which is a point of stability. When all vectors have reached an available attractor, training is complete. After 10 timesteps, we observed that not all points had reached an attractor. 24 seems to be the minimum number of timesteps required before all initialisations reach an attractor. All attempts result in a fourth attractor – [-1, 1] – appearing which we did not encode. This is called a spurious state. These can be composed of various combinations of the original patterns or simply the negation of any pattern in the original pattern set. [-1, 1] makes sense here as a particular kind of negation of one of our encoded attractors. After 24 iterations, all initial states reach one of the four attractors.



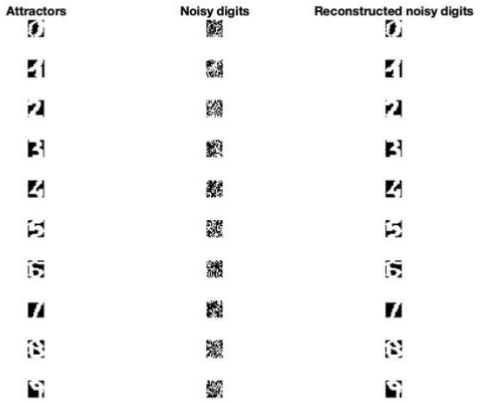


We then modified the rep2 and rep3 scripts to implement initial states of high symmetry. In the 2D case, our initial attractors triple themselves to create 6 spurious states, each of which is some combination of our initial attractors plus 0. In the 3D case, the spurious states – for example [0.3667, -0.3667, 0.3667] – are linear combination of the initial attractors $T = [1\ 1\ 1; -1\ -1\ 1; 1\ -1\ -1]^T$.

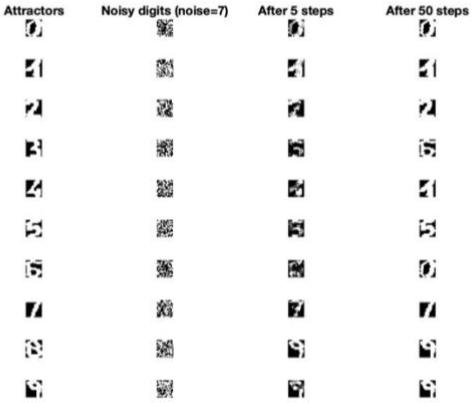




Experimenting with the hopdigit function reveals the hopfield network to be excellent, up to a certain noise level, at retrieving patterns when provided with noisy digits. When no noise is introduced, the network will reach the correct stable state in one iteration. More noise requires more iterations, but over a certain threshold, we are no longer guaranteed to reach the correct pattern. Up to a noise level of 4 (meaning that a random amount of noise is added to each pixel between the values of -1 and +1 and then multiplied by 4), the network is able to retrieve the correct patterns with around 90% reliability. Beyond 4, reliability drops off considerably. A nice illustration of how the Hopfield energy function works is to notice that the number of iterations has no effect on which pattern the noisy images resolve to. The more iterations, the more noise-free the final reconstructed image is, but we can observe that even from very early iterations, the noisy image has already 'chosen' which pattern towards which to move. In the below figure (b), the noise level is 5, which is too much for the network to reliably resolve towards the correct numbers in every case.



a) Noise=3, Iterations=25



b) Noise=5, Iterations=50

The hopfield network is always able to reconstruct noisy digits, however, over a certain noise level, they are not always the correct digits. The reconstructions show how the patterns always resolve to one of the available attractors – with no spurious ones in this case – even in very noisy conditions. This is not surprising, as associative memory is essentially a form of noise reduction, leading the network to a state of least energy. Noise only reduces reliability for arriving at the correct digit, but not reliability that some reconstruction will occur.

2. Long Short-Term Memory Networks

Time-series Prediction

The Santa Fe data set is a univariate time series prediction problem derived from laser-generated data recorded from a Far-Infrared-Laser in a chaotic state. We have 1000 training samples and 100 test samples which we have to predict after learning the pattern in the training data over time. For this, we statically train a model as a feedforward neural network with backpropagation, then apply recurrent neural networks to predict the next position. We will below compare the results from vanilla recurrent neural networks (RNNs) and long short-term memory networks (LSTMs).

Vanilla RNNs

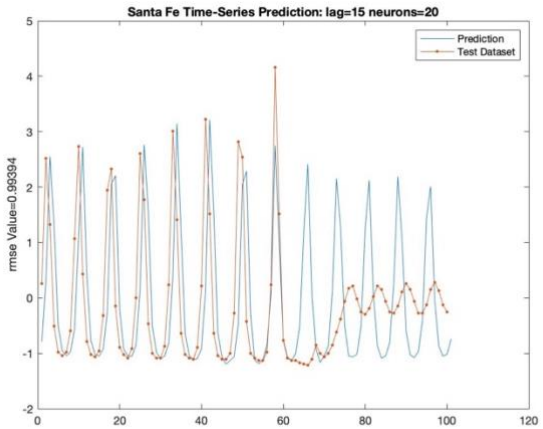
The most important parameters to tune for here are the number of lags (p) and the number of neurons in the hidden layer. The number of lags decides how many points in the history of the time-series upon which we can base our predictions. We first standardise the data as a pre-processing step in order to get a better fit and prevent divergence in training. This involves subtracting the mean of the dataset from each datapoint and dividing by the standard deviation of the dataset, in order to have zero mean and unit variance.

We then use the getTimeSeriesTrainData function which returns a matrix of [pxm], m being p subtracted from the number of training steps (1000). Thus, the input matrix – which we will use as the training set – m samples, each with a dimensionality of p.

The below chart maps the RMSE of various combinations of lags and neurons. One might wonder why simply increasing the number of lags would not lead to a better performance. This is because a weakness of vanilla RNNs as that they face difficulties in learning long-term dependencies. In our train set we have areas with sudden drop outs, which also happens to occur in our test set. Therefore, if our model cannot 'remember' previous occurrences of drop-out – that may have happened many timesteps previously – it will likely not perform well when it needs to predict over similar drop-outs. Setting the lag term too high will result in vanishing or exploding gradients. Below are the results of averaging performance over 10 runs.

		Number of neurons					
		5	**10**	**15**	**20**	**30**	**50**
Number of lags	**5**	1.7942	1.766	1.885	2.849	5.292	3.563
	10	4.093	2.303	1.917	1.873	2.757	4.939
	15	1.579	2.552	1.983	1.5423	3.915	2.283
	20	3.458	1.915	2.239	1.627	1.821	2.253
	30	1.795	1.641	2.772	2.804	2.597	2.651
	40	1.940	1.743	1.864	1.806	2.332	2.213

The below figure is taken from the single best performance from using the best-performing parameters from the above results (lag=15 and neurons=20):



We can see exemplified here the way in which the vanilla RNN struggles to 'understand' the drop-out information, as it continues predicting on what has come immediately before. So, in order to improve our performance, we will need to be able to learn long-term dependencies.
Increasing the lag and/or neuron numbers has a negative effect on the speed of computation. We prefer to keep these values as small as possible – but not too small as to introduce underfitting – as long as they produce results comparable to those of more complex models.

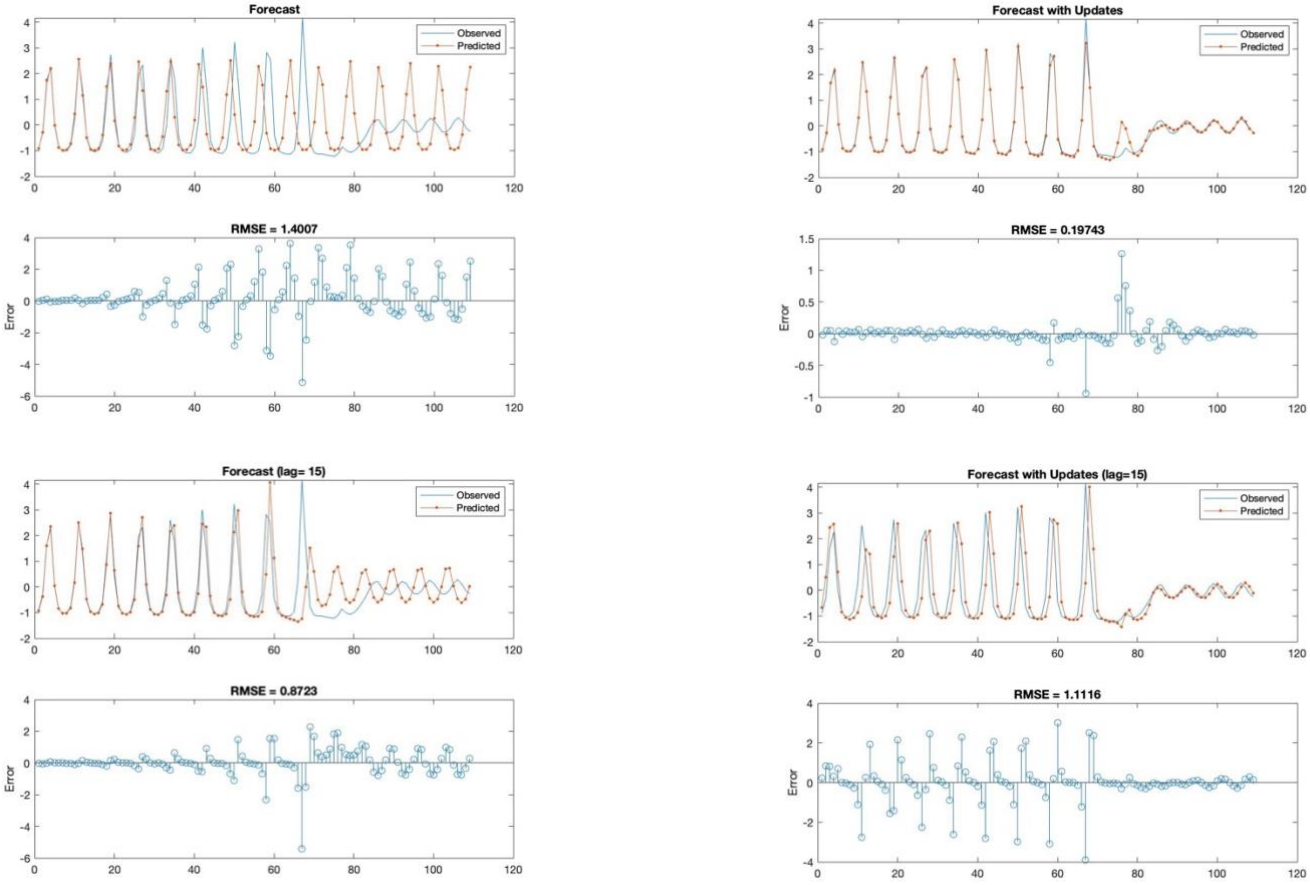
LSTM

As we have seen, traditional RNNs do not do well at predicting regions where the periodicity changed, even though such changes had occurred previously in the training data. In theory, these models should be able to harness past information regardless of how long ago it was captured, but due to the problem of vanishing gradients, they can only remember narrow contextual information. LSTMs fix this by being able to capture long-term dependencies. To be able to do this, they contain information outside the normal flow of the recurrent network in the form of a gated cell. Information can be stored in, written to, or read from these cells. Each gated cell has their own weight which is determined during training.

To effectively capture the sequencing of this time series, we need to tune the LSTM's parameters for our specific case. The most important parameters that were considered during training were the number of hidden neurons (H), the initial learning rate (initLR), factor for reducing the learning rate (dropFactor), and the number of iterations after which we should reduce the learning rate (dropPeriod).

Increasing the number of H will increase the complexity of the model and the more hidden units, the more likely overfitting is to occur. Too few however, will not capture enough information which can lead to underfitting. From the remaining parameters we can see the importance of controlling the learning rate – the aim should be to reduce it after a determined number of epochs so that we do not risk overshooting the global minima. For our parameters, we were inspired by the matlab tutorial called 'Time Series Forecasting Using Deep Learning', but changed the number of hidden units to 25 – as similar or even better results are achieved with this less complex model – and the LearnRateDropFactor to 0.2.

What we notice about LSTM is that it performs much better than the vanilla RNN case, but is much more computationally expensive, and training takes a long time comparatively (minutes as opposed to seconds). Adding lag of 15, forecast with updates (previously observed network states) becomes less accurate while closed loop forecast becomes more accurate, showing that more information is not always helpful in the case of LSTMs.

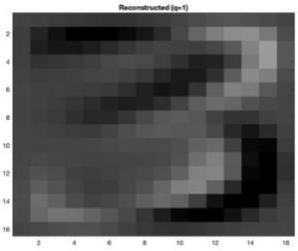


Artificial neural networks - Exercise session 3

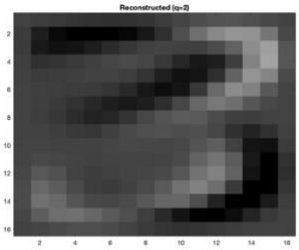
Deep feature learning

1. Principle Component Analysis

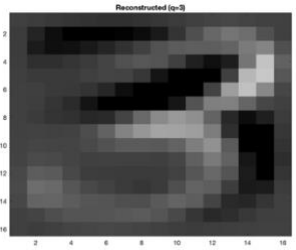
Principal Component Analysis is an example of an unsupervised learning algorithm. It has the ability to learn data structures autonomously. It does this through dimensionality reduction, transforming large datasets into smaller ones retaining the most important and descriptive information. Shrinking the number of variables in a data set engenders an accuracy tradeoff, but the effectiveness of dimensionality reduction lies in the fact that it greatly improves simplicity with only a small reduction in accuracy. The way this is done autonomously is by projecting the original data onto the eigenvectors of their covariance matrix and then picking q – corresponding to the number of dimensions to which we would like to reduce the data – of the largest eigenvalues of the correlation matrix. As we will see from subsequent experiments, as most of the important information is found in the largest eigenvalues, the original data can be reconstructed with relatively small error using only a small number of components.



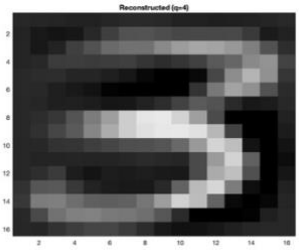
1 principle component



2 principle components

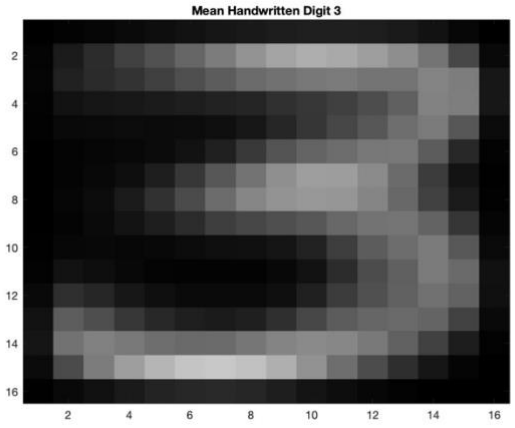


3 principle components

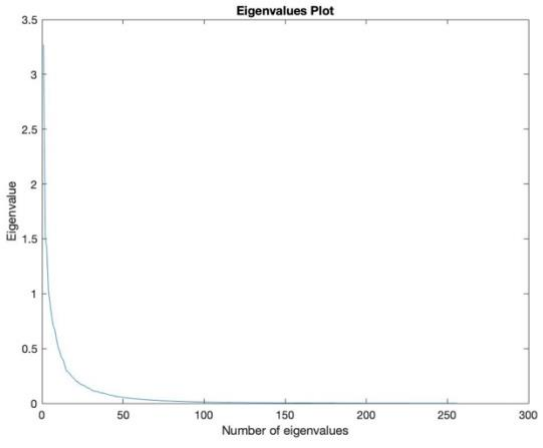


4 principle components

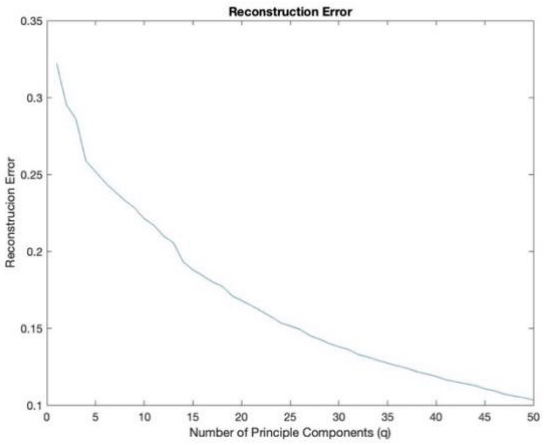
The above images are reconstructions after having compressed the data onto 1, 2, 3, and 4 principal components. It is interesting that using only 4 principal components out of the original 256 produce a distinctive 3. This is the case because the principal components are constructed so that they represent – in descending order – the largest amount of variance in the data. This means that, while it is true that the quality of the reconstructions depends on how close the sum of the largest q eigenvalues is to the sum of all p eigenvalues, it is also true that the most important information is captured in the first and largest principal components. This is illustrated in the Eigenvalues plot below, where we notice an immediate and dramatic drop in values from q=1 through to q=20 before the plot begins to level off with the remaining values being close to 0. At a q value of 256 – the p value of the original dataset – we might expect the reconstruction error to be 0. However, we found a tiny error – 5.4019e-16 – which comes about because there is learning involved – the algorithm still needs to learn the identity map**.**



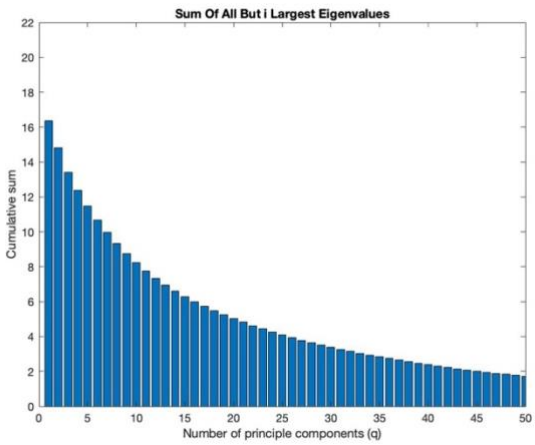
Mean 3



Eigenvalues



Error Plot



Sum of all but i largest Eigenvalues

Above, we see 1) an error plot and 2) a corresponding plot of the unused eigenvalues cumulatively summed at each step. We can observe that the squared reconstruction error induced by not using a particular number of eigenvalues is proportional to the cumulated sum of those unused eigenvalues. Adding more components will always reduce the error, but beyond the point at which the sum of the unused eigenvalues is not very large, we will no longer be able to reduce our error rate significantly.

2. Stacked Autoencoders

An autoencoder is a kind of artificial neural network which works in an supervised manner which, like PCA, aims to learn informative and (usually) dimensionally reduced data representations. The amount of data compression – of dimensionality reduction – depends on the number of hidden units. Undercomplete autoencoders for example aim to reduce dimensionality by constraining the number of hidden layers so that the number of neurons is less than the number of inputs.

Generally, sparsity constraints are introduced in order to force the hidden layers to extract the most informative and influential feature representations. A term is added to the cost function which constrains the values of the average activation of a given hidden unit to be low. This encourages the autoencoder to map a representation wherein each neuron in the hidden layer 'specialises' by responding to some feature only present in a small number of training examples, as opposed to responding to every training example. The goal here is to highlight unique signals across features, useful as a lead up to using downstream supervised classifiers to apply labels. It also ensures, when incorporating hidden layers with more neurons than inputs (overcomplete autoencoders), that the autoencoder does not simply copy the input to the output, i.e. learn the identity function.

Stacked autoencoders (SA) comprise multiple layers capped with a final layer which uses labelled data to classify outputs in a supervised manner. The output for each layer is used as the input for each subsequent layer. Each layer is trained individually while freezing parameters for the remainder of the model, before then applying backpropagation to fine-tune the entire model – all layers simultaneously – in a supervised manner.

We performed some experiments to compare a number of different stacked autoencoder architectures to a normal multilayer neural network. We used the rng('default') command to set the random number generator seed, ensuring the same weight initialisations for each run.

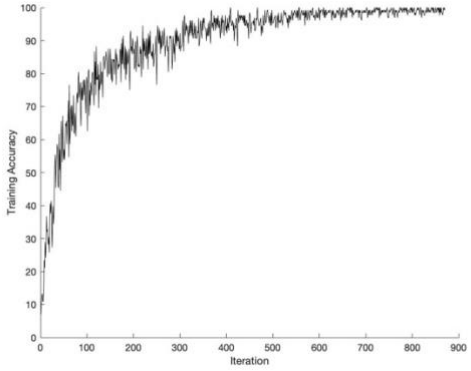
Highlighted Parameters	Network	Network architecture	Classification Accuracy
Default	Pre-trained SA	784/100/50/10	82.16
	Fine-tuned SA	784/100/50/10	99.64
	Normal multilayer NN	784/100/10	95.84
	Normal multilayer NN	784/100/50/10	96.58
Higher degree of sparsity [SparsityProportion = 0.4]; higher impact of sparsity constraint in cost function [SparistyRegularization = 6] (same for each hidden layer)	Pre-trained SA	784/100/50/10	83.80
	Fine-tuned SA	784/100/50/10	99.70
	Normal multilayer NN	784/100/10	95.84
	Normal multilayer NN	784/100/50/10	96.58
1 fewer SA layer	Pre-trained stacked SA	784/100/10	98.50
	Fine-tuned stacked SA	784/100/10	99.04
	Normal multilayer NN	784/100/10	96.06
	Normal multilayer NN	784/100/10	96.62
I autoencoder/1 layer	Pre-trained stacked SA	784/10	49.90
	Fine-tuned stacked SA	784/10	98.20
	Normal multilayer NN	784/10	70.34
	Normal multilayer NN	784/10	73.61
1 extra SA layer	Pre-trained stacked SA	784/400/100/10	48.70
	Fine-tuned stacked SA	784/400/100/10	99.62
	Normal multilayer NN	784/100/10	97.98
	Normal multilayer NN	784/100/50/10	96.70
Max Epochs = 750 (Layer 1)	Pre-trained stacked SA	784/100/50/10	88.62
	Fine-tuned stacked SA	784/100/50/10	99.64
	Normal multilayer NN	784/100/10	95.84
	Normal multilayer NN	784/100/50/10	96.58
More neurons in hidden layers	Pre-trained stacked SA	784/400/200/10	99.06
	Fine-tuned stacked SA	784/400/200/10	99.38
	Normal multilayer NN	784/200/10	96.16
	Normal multilayer NN	784/200/75/10	97.60

Better results were obtained by changing the sparsity parameters, removing one layer from the stacked autoencoder, increasing the maximum number of epochs and particularly by adding more neurons to the hidden layers, which brought the pre-training classification accuracy to a comparable level to the fine-tuned result. A better performance can be achieved over a normal neural network as long as pre-training is applied, otherwise without fine-tuning, two layers seem to be required. Fine-tuning improves the performance so much because it applies backpropagation to the whole network, which is applied by design in normal neural networks, while having the added benefit of having optimised each layer individually in the pre-training phase. Normal deep neural networks are more prone to experiencing vanishing gradients, as all layers are tuned simultaneously only.

3. Convolutional Neural Networks

The weights of layer 2 – the first convolutional layer – appear as 11x11x3x96. These represent 96 11x11 convolutional filters of 3 colour channels: red green and blue, roughly corresponding to the colour receptors in the human eye. Applying RELU and Cross Channel Normalization does not affect the dimensionality of the input, however max pooling does, being designed to down-sample our input representation. A formula we can use to determine output size when we – as is the case here – have an input of dimension n x n, is the following: $O = \frac{n-f+2p}{s} + 1$, where n is input dimension, f is filter dimension, p is padding and s is stride. This allows us to determine that the dimension of the input at the start of layer 6 is 27x27x96. This is calculated because after we apply the weights of the first convolutional layer, we have an input of 55x55x96, as the stride is 4x4 (therefore ((227-11+2(0)/4) + 1 = 55, then applying the hyperparameters of layer 5 (3x3 max pooling with stride 2x2), we get ((55-3+2(0))/2) + 1 = 27. 96 remains because it is the number of

convolutional filters, which does not change until a deeper layer which has different hyperparameters governing the number of filters. Meanwhile, the RBG number of 3 which appeared before disappears because when we apply a convolutional filter, each convolution produces a single value, thus a filter of 11x11x3 filter will produce only 1 value per convolution, so we end up with 55x55x96 as the outputs for layer 2.
Finally, using the same calculations as above and taking into account the hyperparameters and dimension reduction as a result of further convolutions and max pooling layers, the dimensions of the inputs prior to the final classification of the network becomes 6x6x256, or 9216 datapoints, representing a reduction of over 94% from the 227x227x3 image (154,587 datapoints) we started with. This dimensionality reduction is one of the great strengths and advantages of CNNs over more traditional neural networks that remain fully interconnected from input to output. These 9216 datapoints – now inputs for the fully connected part of our network – contain the most important information required for classifying our images, so a large amount of less informative important information has been filtered out along the way.



***7 Layers**:*

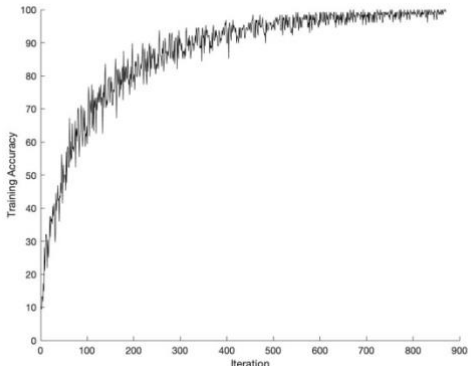
[[imageInputLayer([28 28 1]),
convolution2dLayer(5,22)
reluLayer
maxPooling2dLayer(2,'Stride',2)
fullyConnectedLayer(10)
softmaxLayer
classificationLayer()];

***Accuracy**:*

0.947

***Train Time**:*

86.33 seconds



***10 Layers**:*

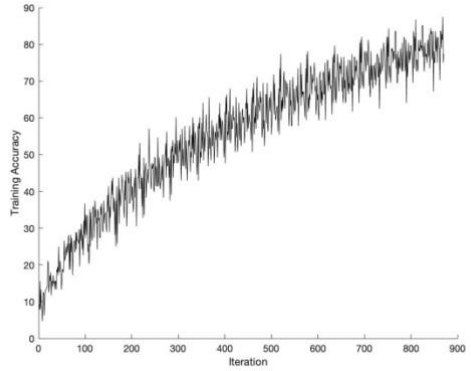
[[imageInputLayer([28 28 1]),
convolution2dLayer(3,14)
reluLayer
maxPooling2dLayer(2,'Stride',2)
convolution2dLayer(3,30)
reluLayer
convolution2dLayer(3,46)
fullyConnectedLayer(10)
softmaxLayer
classificationLayer()];

***Accuracy**:*

0.956

***Train time**:*

133.71 seconds



13 Layers:

[[imageInputLayer([28 28 1]),
convolution2dLayer(3,38)
reluLayer
maxPooling2dLayer(2,'Stride',2)
convolution2dLayer(3,23)
reluLayer
convolution2dLayer(3,38)
reluLayer
convolution2dLayer(3,10)
maxPooling2dLayer(2,'Stride',2)
fullyConnectedLayer(10)
softmaxLayer
classificationLayer()]

Accuracy:

0.739

***Train time**:*

249.62 seconds

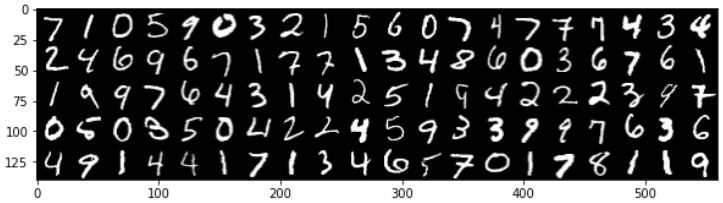
Training a CNN on the handwritten digits dataset, we saw the best accuracy with a 10 layer model, although at a sacrifice in training time compared to the 7 layer model. The more complex model of 13 layers saw a large drop in performance. We would suggest that smaller models work well here due to the simple nature of the handwritten digits, whereas for images of more complexity, deeper models may be required to map the more complex higher features.

Artificial neural networks - Exercise session 4

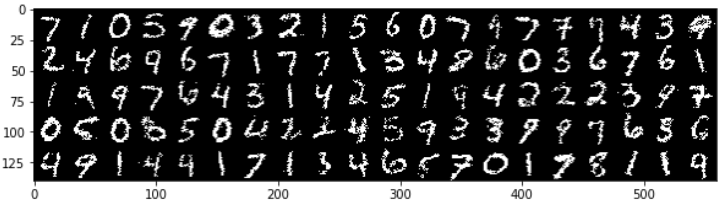
Generative models

1. Restricted Boltzmann Machines

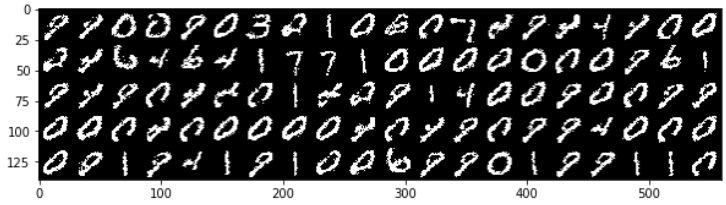
In training the RBM, the number of components is the number of units in the hidden layer. We found the default setting of 10 to be insufficient, in that identifiable 'real' images are not produced. Values of 50 or larger produce better results because the model is able to learn more patterns from the training set. As always there will be a trade-off in the training time, which will be longer the more parameters are added. Number of iterations refers to the number of passes over the whole train set. Although the below is as a result of 30 iterations, we found that after 10 iterations the pseudo-likelihood plateaus, so 10 iterations is perhaps enough to produce reliably good results. For smaller numbers of components, the gain in pseudo-likelihood is negligible beyond 20 iterations. We trained with **200 hidden units, a learning rate of 0.04 and 30 iterations.**



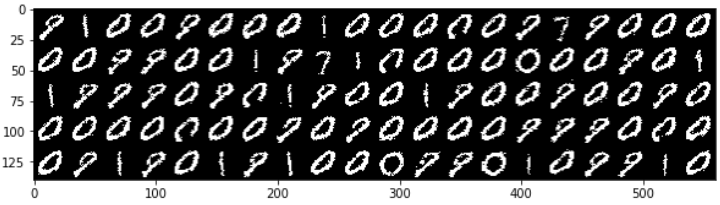
Training images



1 Gibbs step

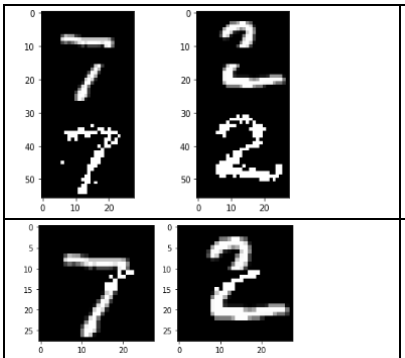


50 Gibbs steps



1000 Gibbs steps

As we can see above, the model creates realistic reproductions with 1 Gibbs step, which has been shown to work well in practice (Hinton, 2002). Increasing the Gibbs samples does not help us here, as 0s and 1s are favoured. The reason for this is because as we increase the number of Gibbs steps, we move further away from the initial distribution over the visible variables, and towards the equilibrium distribution. What this means is that the reproductions will be more likely to have homogeneous features of low variance as found in say 1, a simple vertical line, or 0, a simple round shape. Meanwhile, the one-step reconstructions remain close to the model's initial distribution, which is what we want.



Number of removed rows = 5 Reconstruction Gibbs steps = 10	Number of removed rows = 10 Reconstruction Gibbs steps = 10

Image reconstruction

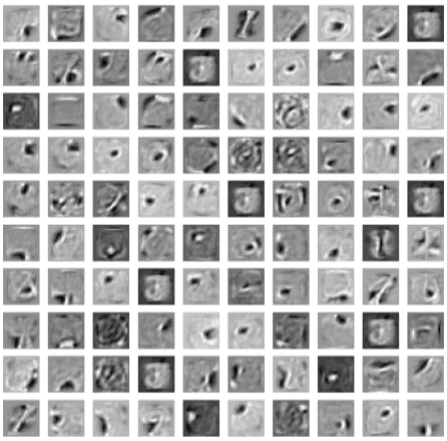
For image reconstruction, we need a higher number of Gibbs steps, possibly because the altered images are themselves further from the initial distribution of the model. The more rows we remove, the less successful the network is in reconstruction. Removing rows in the centre of the image also leads to more successful reconstructions rather than removing them from the top or bottom, as these tend to be where the most identifiable features of the images are erased.

2. Deep Boltzmann Machines

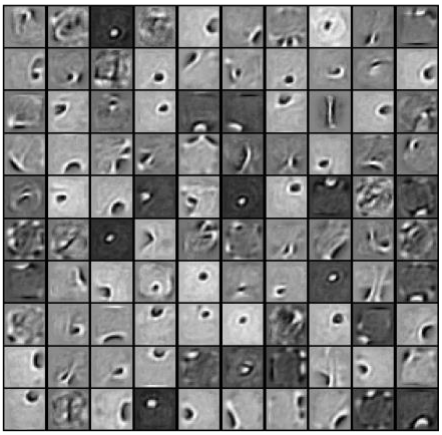
DBMs are similar to RBMs except that while RBMs comprise the input and one hidden layer only, DBMs will have more hidden layers added behind the initial one. All connections in a DBM are undirected, which among other advantages allows these models to use higher-level knowledge to resolve uncertainty about lower-level features, thereby creating better data-dependent representations (Salakhutdinov & Hinton, 2012).
These models also are comparable to Hopfield networks in that they are also defined with Hamiltonian Energy for the overall network, with the energy function for a 3 layer network being the following:
$E(v,h^1,h^2,h^3;\theta) = -v^T W^1 h^1 - {h^1}^T W^2 h^2 - {h^2}^T W^3 h^3$, with unknown model parameters $\theta = \{W^1, W^2, W^3\}$.

Below we can see a comparison between the components extracted from the only hidden layer of the RBM versus those from the first two hidden layers of the DBM. The first layer of the DBM shows features similar to those of the RBM, however the second layer evinces more specificity and detail, with each filter showing more delineated features. We would expect this, as each deeper layer should capture higher-level features.

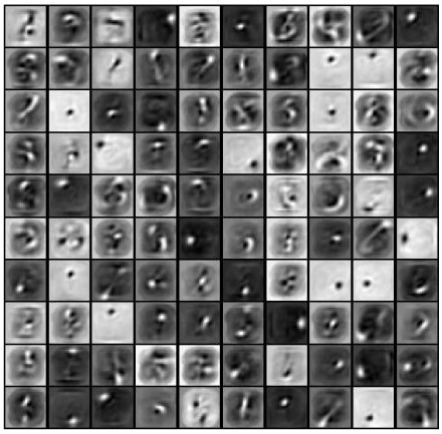
First 100 filters of the RBM



First 100 filters of the first layer of the DBM



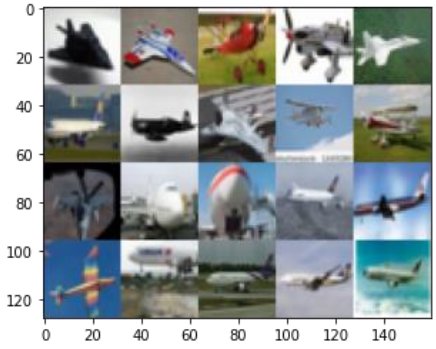
First 100 filters of the 2nd layer of the DBM



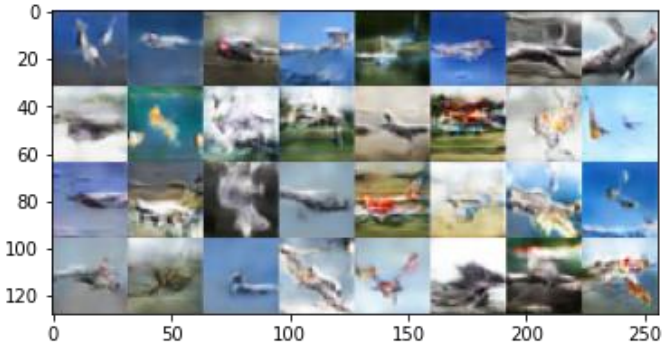
Samples generated by DBM after 100 Gibbs step

As we can see, although still imperfect – for example some images are not identifiable as numbers – the sampled results from the DBM are much cleaner that those of the RBM. Although the sampled images produced by the RBM were realistic and mostly legible, they were speckled and rough, while the DBM's output is clear. We expect this is a result of the DBM's ability to model more complex features. Deeper layers allow the model to learn higher-level features which are used for the construction of new images. We again see a bias here towards 0s and 1s for the same reasons as outlined above in part 1.

3. Generative Adversarial Networks



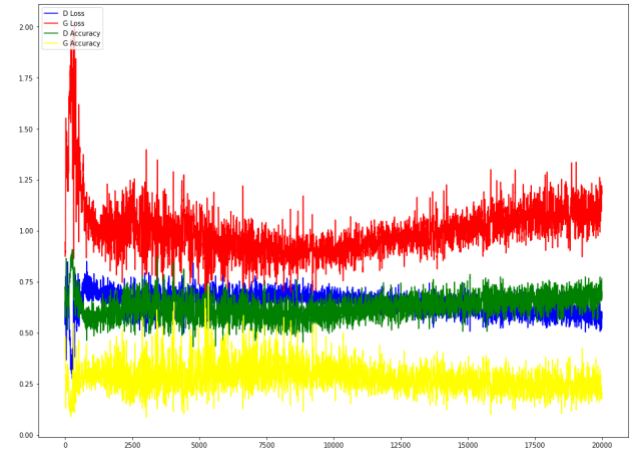
Selection of training images (class 0)



Generated images (class 0)

From the CIFAR database, we selected class 0 to train the Deep convolutional generative adversarial network. Class 0 comprises images of aeroplanes. We trained for 15,500 batches, with a batch size of 32 and a plot interval of 50. We can see the generated images do tenuously resemble the training images, though they are indistinct and would likely not be identified as aeroplanes (by a human) without the context of the training images. Over the training process, we did see very modest improvement in the sense that the images are becoming clearer, although this is perhaps not saying much when the goal is realism. Through training, we see the generator accuracy modestly dropping from around epoch 10,000 while the loss increases. Meanwhile we see an inverse relationship with the generator. This shows that the training was not stable.

Final accuracy and loss curves



The goal of training here is to have the generator reproduce the true data distribution, meaning that the discriminator can no longer distinguish between real and generated images. For this to happen, both players need to keep improving until an equilibrium deadlock situation is achieved. Clearly this is not happening here, as the discriminator appears to be performing better than the generator throughout. At the end of training, the generator had an accuracy of 0.1562 and a loss of 1.1213, while the discriminator had an accuracy of 0.9688 and a loss of 0.3284. Again, we would ideally see a situation where both networks are improving, to the point where the distribution of the generated images matches the true data distribution. This would mean the discriminator would predict 'real' or 'fake' with a probability of 0.5. The below formula illustrates this minimax 2-player game, where D(**x**) represents the probability that **x** is a 'real' training image rather than something coming from the generator, while D(G(**z**)) represents the probability that the image was created by the generator using the noise variables $p_z(z)$:

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$$

(Goodfellow et al., 2014)

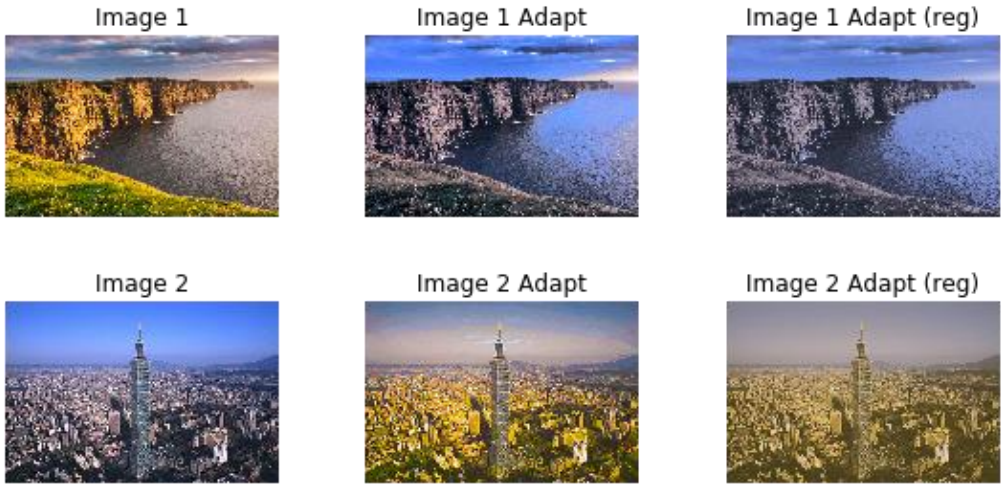
4. Optimal Transport

We used the Earth Mover's Distance (EMD) and Sinkhorn algorithms to transfer colour between two images. These algorithms map each image's colours as two probability distributions. In the case of EMD, the Wasserstein Metric is used to construct the optimal transport plan between the two distributions. It does this by calculating a cost matrix for each possible movement from one distribution to the other and tries to minimize the cost. The below formula is the Wasserstein Metric, where here, r and k represent

the two colour distributions, P_{ij} quantifies the number of colour pixels transported from the ith position of one distribution to the jth position of the other, while M is the cost matrix.

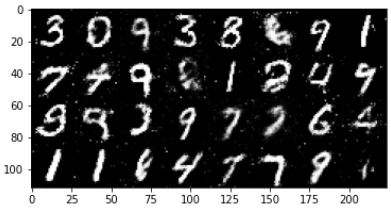
$$d_M(r,k) = \min_{P \in U(r,k)} \sum_{ij} P_{ij} M_{ij}$$

The Sinkhorn algorithm adds another term to the transport plan which subtracts the information entropy of P, which has the effect of encouraging more equal distributions, acting in a similar way to ridge regression regularization. Below we can the EMD outputs in the middle column and the Sinkhorn in the third column. The Sinkhorn images appear less blotchy and more smooth as we would expect, while the EMD images are arguably more charismatic in their relative lack of realism and larger contrasts. The EMD outputs seem to exaggerate slight differences which may not be desirable. Overall, this differs from simply palette swapping the images because it models and transports the distributions of the colours which takes into account the relationship of these distributions in a way that pixel swapping could not.

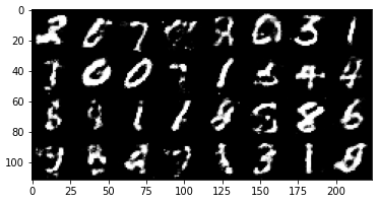


5. Wasserstein GANs

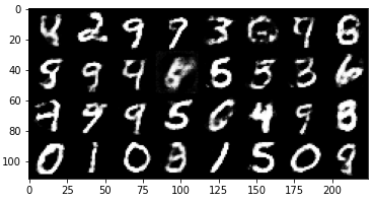
Wasserstein GANs work in this case by minimizing the Wasserstein distance between the real probability distribution (i.e. of the training images) and the generated distribution. Compared to standard GANs, this has been shown to increase stability during the optimization process and produce a more meaningful loss metric that correlates with the generators convergence and sample quality (Arjovsky et al, 2017).
Here the Kantorovich-Rubinstein duality is used, and a maximum (supremum) is taken over all 1-Lipschitz functions: $W(\mathbb{P}_r, \mathbb{P}_\theta) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{x \sim \mathbb{P}_r}[f(x)] - \mathbb{E}_{x \sim \mathbb{P}_\theta}[f(x)]$, where P_θ is the generated distribution.
In order to enforce the Lipschitz constraint, two methods are used here. One is weight clipping, where the weights are fixed at each gradient update in order to keep them in a compact space [-c, c]. The other is by using gradient penalty which adds a gradient penalty term which directly constrains the discriminator's output with respect to its input, ensuring that the learning function's gradient norm will not exceed 1.
What we see in the generated samples is indeed an improvement in stability from standard GAN to Wasserstein GAN, while the gradient penalty version narrowly achieves the least blurry output between the two WGANs. The standard GAN has a faster training time and appears to produce a more homogeneous distribution of digits.



Standard GAN



Wasserstein GAN with weight clipping



Wasserstein GAN with gradient penalty