Artificial neural networks - Exercise session 1

Prepared by Ömer Yılmaz (r0779149)
omer.yilmaz@student.kuleuven.be

1. The perceptron

- A perceptron is a very simple model. It defines a linear decision boundary. Therefore, it is not expected to separate datasets which are not separable with only a line. However, a layer of perceptrons can define a convex region or multilayers of perceptrons can separate several convex or regions with holes etc.

I wasn't always able to train the models successfully, because the models were not always linearly separable. When I used 4 points of dimension 2, Cover's Theorem says that 7/8 of the whole dataset can be linearly separated. For example, 2 of the 16 different combinations for the input space of {-1,1} for x and y failed in the below code.

```
allTargetCombinationsFor4Points = transpose(dec2bin(0:2^4-1)-'0');
for T = allTargetCombinationsFor4Points
    P = [-1 -1 1 1;-1 1 -1 1];
    net = newp(P,T','hardlim','learnp');
    net = train(net,P,T');
    pause
end
```

One of the failing configurations can be seen in Fig. 1.

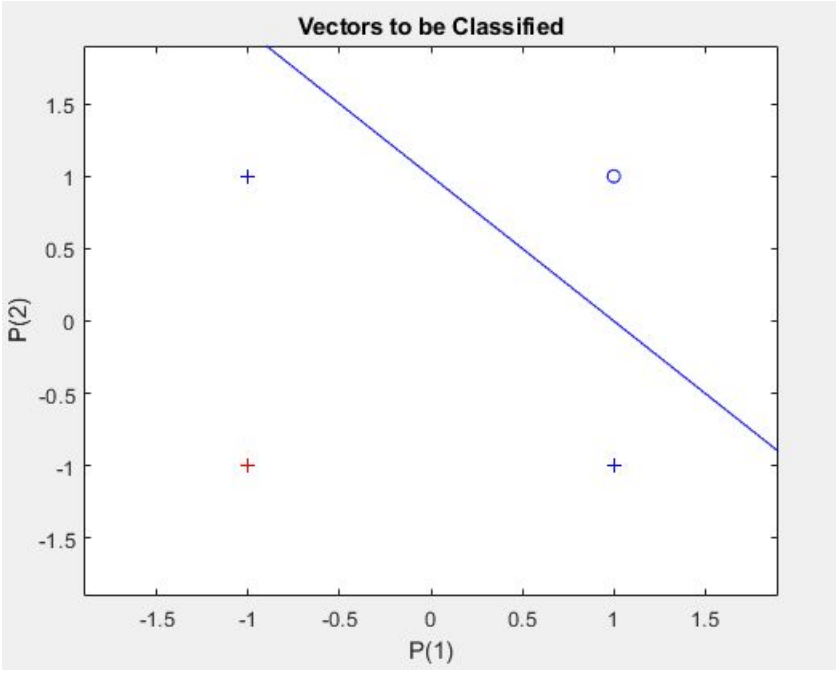


Fig. 1: Point at [-1;-1] is misclassified as this case is not linearly separable

When some of the input points are comparatively distant, normalized perceptron weight learning method has given better results.

2. Backpropagation in feedforward multilayer networks

- Gradient descent is a first order method. It is expected to train more to fit the model as observed by the experiment. Levenberg-Marquardt method is based on the Newton's method with a constraint of size for the step. When this constraint is very big compared to the other terms, its effect is similar to gradient descent. When it is negligible, it is similar to the Newton's method. Newton's method is an iterative method that starts from a point and uses a tangential hyperplane from that point. The hyperplane crosses the 0 value hyperplane for a function. Eventually it finds the 0 value point of the function. We use it here to find a 0-crossing value of the gradient graph of weight-loss graph. During the procedure it requires us to calculate the inverse of the Hessian matrix of the original function. Inversion is costly and numerically unstable. However, there is a linear mapping between changes in the position and changes in the gradient. We use this relationship to estimate the Hessian which brings us the Quasi-Newton methods. There are several approaches. DFP formula allows us to avoid the direct inversion of the Hessian. Newton's method may diverge or the Hessian matrix can be singular.

In the exercise, Levenberg–Marquardt algorithm was far superior against the gradient descent. Gradient descent not only trained late but also did not successfully fit the data. Using an adaptive learning rate for the gradient descent helped only a bit.

Fletcher-Reeves conjugate gradient algorithm, which is a second order algorithm, performed much better than gradient descent. The situation was similar with Polak-Ribiere conjugate gradient algorithm and BFGS quasi Newton algorithm (quasi Newton).

Levenberg–Marquardt algorithm was more performant overall. It learned faster. Also, quasi Newton algorithm has been slightly better than the conjugate gradient algorithms for this dataset.

When I increased the dataset 10 times more, none of them was able to train with 1000 epochs successfully.

- The comparison was similar when we add noise to the data although all algorithms found it hard to train well especially after applying more than 0.2 std noise (0 mean). Because of the increased uncertainty, having more samples clearly helped. Gradient descent trained much more slowly and performed worse as can be seen from Fig. 2.

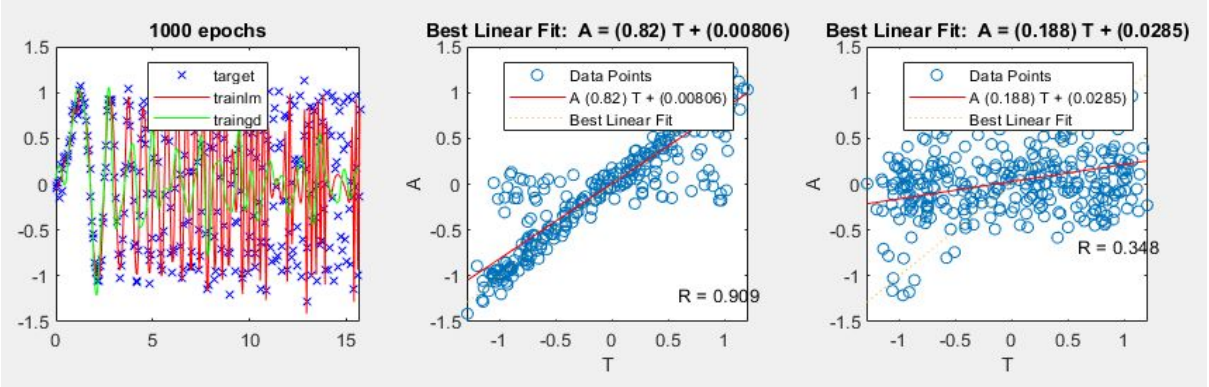


Fig 2: Comparison of Levenberg-Marquardt (middle) and gradient descent (right) algorithms (1000 epochs)

- First of all, I created the dataset thinking all three of the training, validation and test

sets should have a homogen sampling distribution from the 13600 samples. They also shouldn't have same values. I managed this with the following code:

```
Tnew = (d1*T1 + d2*T2 + d3*T3 + d4*T4 + d5*T5)/(d1+d2+d3+d4+d5);

index = randperm(13600,3000);
trainIndex = sort(index(1:1000));
valIndex = sort(index(1001:2000));
testIndex = sort(index(2001:3000));

trainX1 = X1(trainIndex);
trainX2 = X2(trainIndex);
trainX = [trainX1';trainX2'];
trainT = Tnew(trainIndex)';

valX1 = X1(valIndex);
valX2 = X2(valIndex);
valX = [valX1';valX2'];
valT = Tnew(valIndex)';

testX1 = X1(testIndex);
testX2 = X2(testIndex);
testX = [testX1';testX2'];
testT = Tnew(testIndex)';

Ftrain = scatteredInterpolant(trainX1,trainX2,trainT');
[xx,yy] = meshgrid(trainX1,trainX2);
plot3(xx,yy,Ftrain(xx,yy));
```

All three sets are shown at Fig. 3.

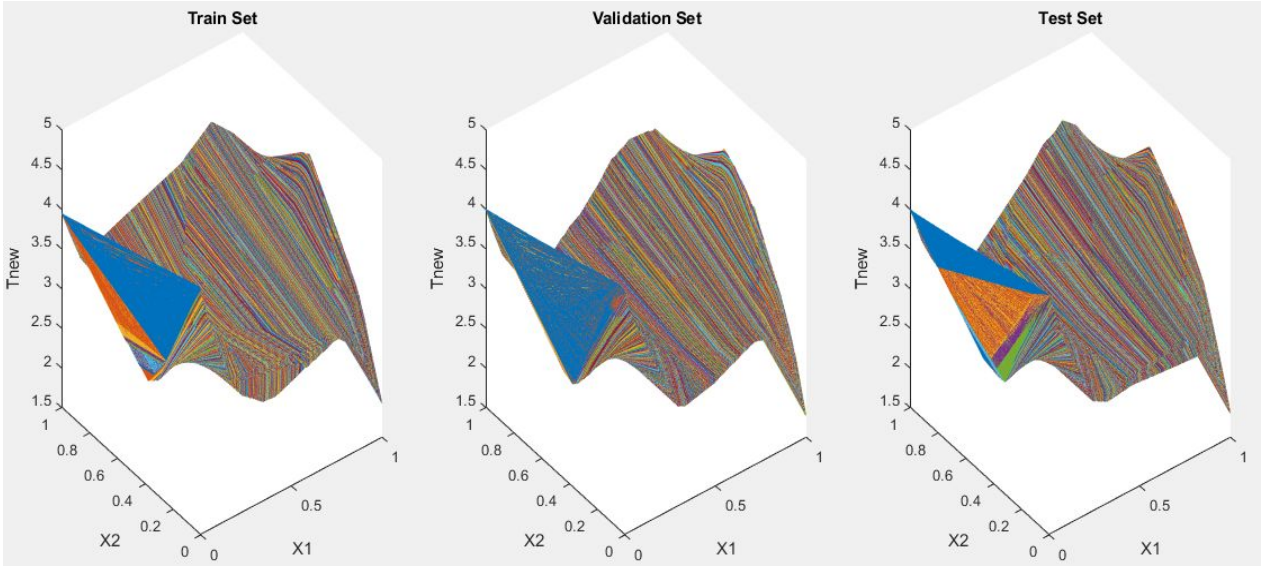


Fig 3: Three homogenically and randomly sampled data sets

I used log-sigmoid transfer function for training, because I wanted to have a nonlinear

activation. I first tried 200 number of neurons. At this point, I expected Levenberg–Marquardt algorithm to perform better. Then, I run the for loop using code sketch below (not in a one-shot way though possible). I observed Levenberg–Marquardt algorithm is indeed the best choice.

```
% I run the for loops playing with the fields of these vectors with multiple converging
% steps as an exhaustive search would be computationaly inefficient.
trainAlgorithms = ["traingd", "trainlm", "traingda", "traincgf", "traincgp", "trainbfg"];
numberOfNeurons = [100 200 300 400 500];

minLoss = {inf 0 0};
for trainAlg = trainAlgorithms
    for numN = numberOfNeurons
        net = feedforwardnet(numN,trainAlg);
        net.layers{1}.transferFcn = 'logsig';
        net.trainParam.showWindow = 0;
        net = train(net,trainX,trainT);
        valPred = sim(net,valX);

        loss = mean((valT - valPred).^2);
        if loss < minLoss{1}
            minLoss{1} = loss;
            minLoss{2} = trainAlg;
            minLoss{3} = numN;
        end
    end
end
```

Prediction values was quite similar to test set as shown at Fig. 4.

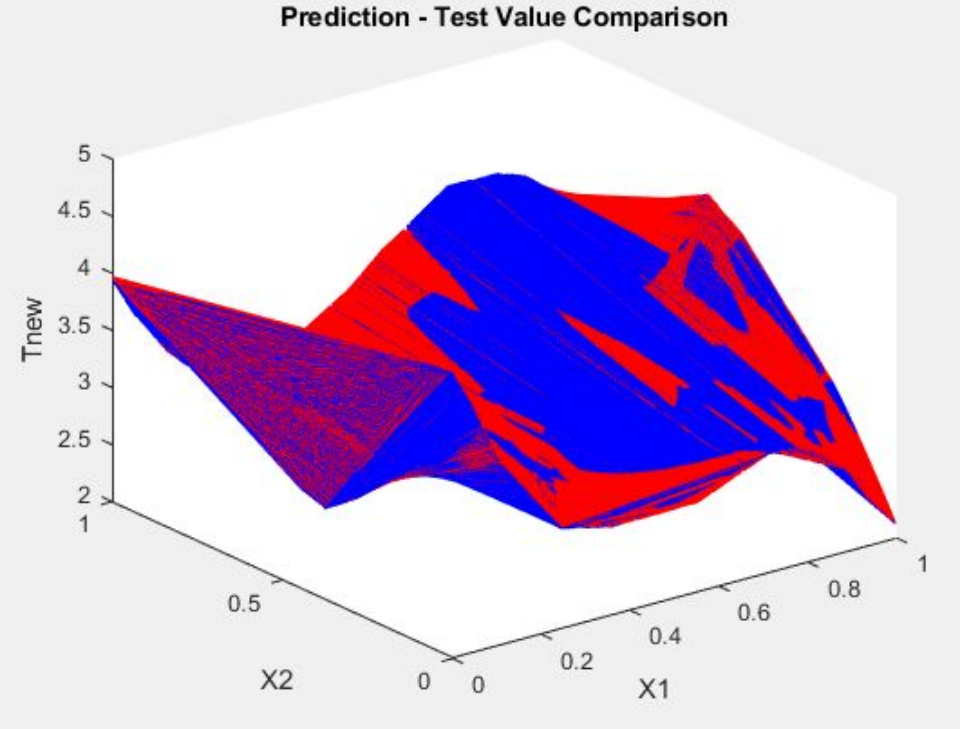


Fig 4: Test set values are closely predicted

Mean squared error was: 0.00023556.

3. Bayesian Inference of Network hyperparameters

- Bayesian inference algorithm of Matlab, *trainbr*, trains according to Levenberg-Marquardt algorithm with an. Levenberg-Marquardt algorithm itself is called with *trainlm* and tries to minimize mean squared error. *trainbr* on the other hand minimizes weighted sum of squared errors and squared weights. This regularization mechanism allows us to handle overfit networks possibly with large number of neurons. In particular, with small data sets Bayesian inference can let us include validation data set to training and train directly.

As expected, Levenberg-Marquardt algorithm and Bayesian inference algorithm outperformed all others. These two are also very similar when there is no noise on the data set. However, when I introduced noise of 1 std, 0 mean, Bayesian inference algorithm have performed a bit worse (R values of 0.826 against 0.908) as can be seen in Fig. 5.

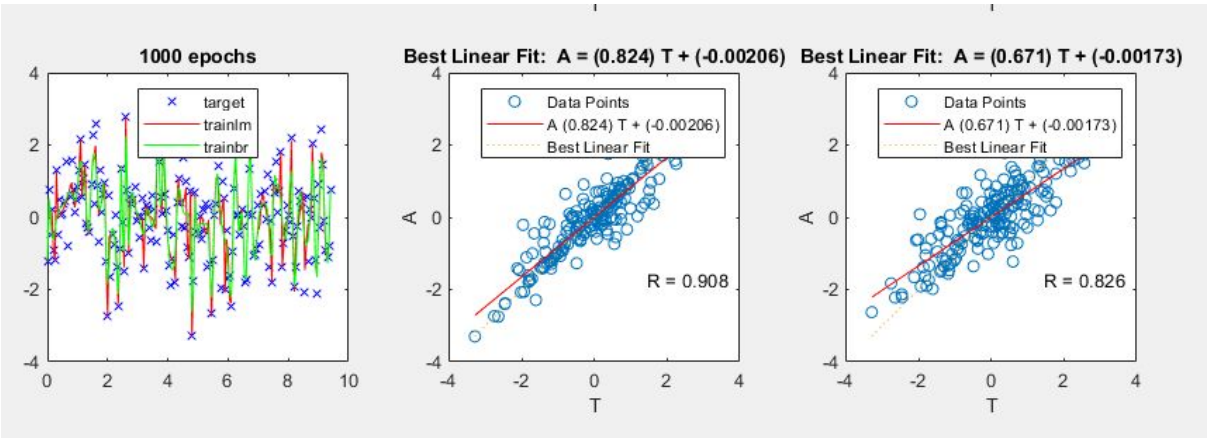


Fig 5: Comparison of Levenberg-Marquardt (middle) and Bayesian inference (right) algorithms (1000 epochs)

The difference decreased with the noise of smaller std and also with bigger data sets. Here [1], it says adding noise is equivalent to ridge regression which cancels (a bit) the purpose of *trainbr* usage as is the case here.

Using higher number of neurons helped all algorithms but more to the Bayesian inference as in Fig. 6. This is because of its generalization capability with weight regularization (it can regularize extra neurons).

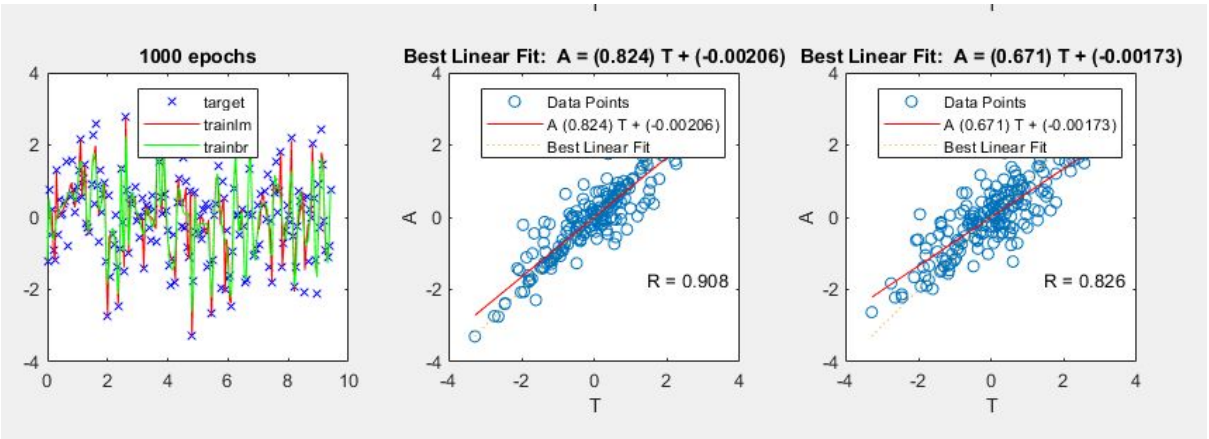


Fig 6: With twice as much neurons, Bayesian inference (right) algorithm catches up even with noise

1: Bishop, Chris M. "Training with noise is equivalent to Tikhonov regularization." *Neural computation* 7.1 (1995): 108-116.

Artificial neural networks - Exercise session 2

Prepared by Ömer Yılmaz (r0779149)
omer.yilmaz@student.kuleuven.be

1. Hopfield Network

- The number of attractors is bigger than 3. We also have [1; 1]. As shown in Fig 1.

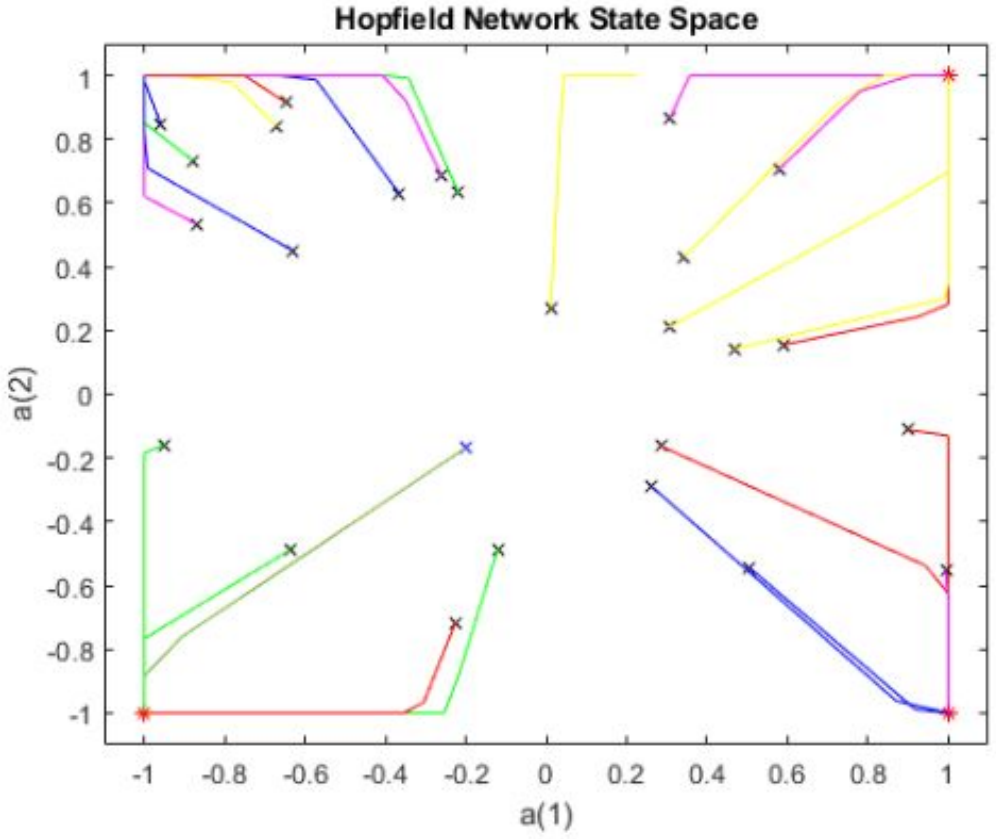


Fig. 1: 25 initial points reach to 4 attractors

A point usually converges to an attractor in less than 20 iterations. However, [0; 0] point never converges. And the point closer to it requires more iteration steps. For example [0.01; 0.01] needs 41 iterations.

- When I try all 3 combinations of [-1;-1], [-1;1], [1;-1], [1;1] points as initial attractors, the remaining point is also an attractor and the initial attractors stayed. [0;0] never converges as mentioned above and [0;-0.001] or [0,-0.5] doesn't converge although halves the nearest attractors ([-1;-1], [-1;1], similar logic is valid for other pairs of attractors as well).
- I tried with different number of inputs and the 3 attractors remained. When I tried the [0;0;0] point, it converged to the closest point of the plane created by the three attractor points. It followed the normal path as in Fig. 2.

Similar analysis is valid for other 3 corner choices to be attractors.

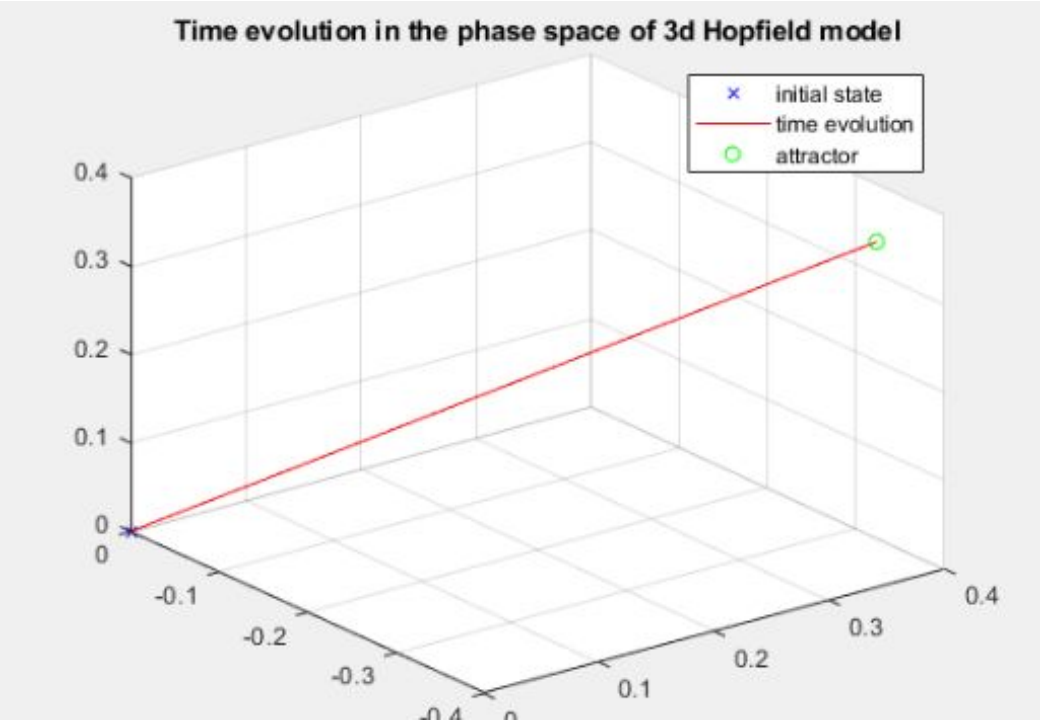


Fig. 2: Path of [0;0;0] is the normal to the plane created by [1; 1; 1], [-1; -1; 1], [1; -1; -1] attractors

- When the noise is increased, the number of iterations required to restore the noiseseless image increased and the accuracy of restoration decreased. Because the increased variance moves the original image's position to the attraction region of another attractor.

2. Neural network

- I understood what *getTimeSeriesTrainData* does and created a validation data set using the last 100 samples of the training set. Then, I ran a for loop on the training and validation check with different number of neurons and different lag values as shown in the below code piece.

Inside the box, you can see I made sure that new predictions are based on the previous predictions coming from the system. It doesn't use the values coming from *getTimeSeriesTrainData.* At every time step, there is a prediction and I used this prediction to be the last value of next input column vector meanwhile sliding the current vector and making the all other (earlier) fields of the next inputs the same with last part of current input vector.

For training algorithm, I used *trainlm* which was a clear choice for me after *Exercise Session 1.* I could have tried *trainbr* though. I had divided the training set into validation and training as can be seen from the below code piece. *trainlm* could have been used to prevent overfitting. I tried different lag values {1, 2, 5, 10, 20, 40, 50} and hidden neuron values {50, 100, 200, 300}. I have found 200 number of neurons and 50 number of features (lag) gave the least error.

However, the final result wasn't a success (as I expected) which can be seen in Fig.3. Because, the system just memorizes some input sets that are given sequentially. For example, shuffling the input set after running *getTimeSeriesTrainData* wouldn't result in a

serious performance drop apart from some correlations among the inputs given. There is no cell concept at this neural network.

```
maxPerf = [0 0 0];
for p = lags
    [santaFeTrainX, santaFeTrainY] = getTimeSeriesTrainData(santaFeTrainRaw, p);
    [santaFeValX, santaFeValY] = getTimeSeriesTrainData(santaFeValRaw, p);
    len = size(santaFeValX); len = len(2);

    for numN = numberOfNeurons
        net = feedforwardnet([numN],'trainlm');
        net = train(net,santaFeTrainX,santaFeTrainY);

        perf = 0;
        tempValX = santaFeValX;
        for i = 1:len-1
            pred = sim(net, tempValX(:,i));
            tempValX(end,i+1) = pred;
            tempValX(1:end-1,i+1) = tempValX(2:end,i);
            % perform uses mse for these nets
            perf = perf + perform(net,pred,santaFeValY(i));
        end
        lastPred = sim(net, tempValX(:,len));
        perf = perf + perform(net,lastPred,santaFeValY(len));
        perf = perf / len;
        if perf > maxPerf
            maxPerf(1) = perf;
            maxPerf(2:3) = [p numN];
        end
    end
end
```

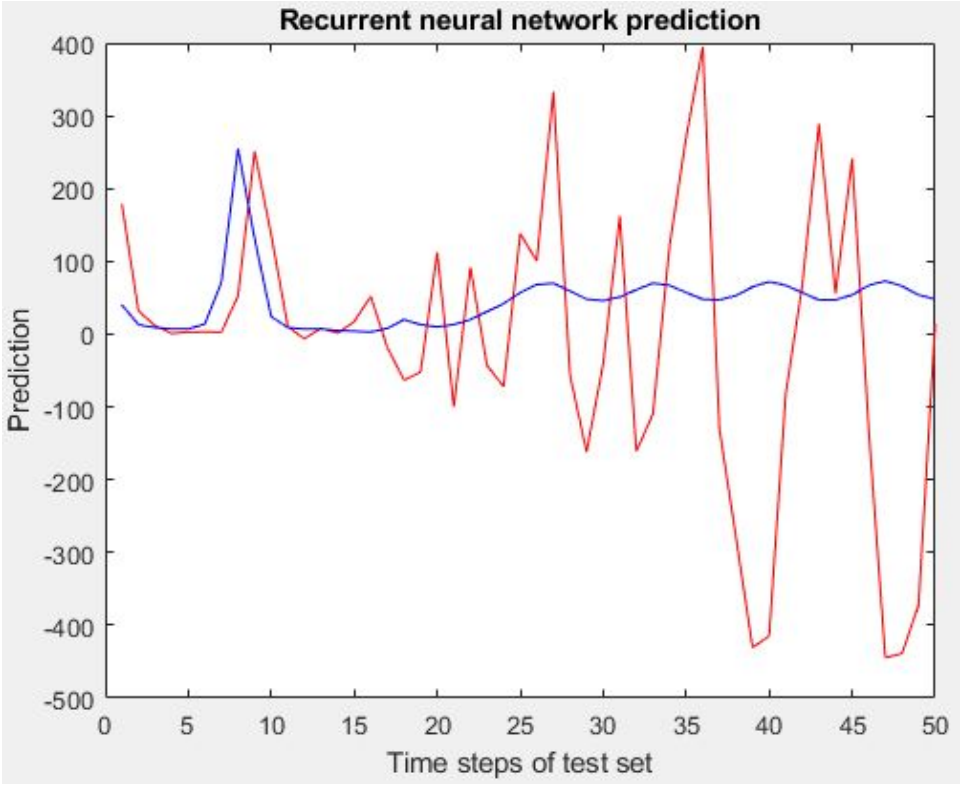


Fig. 3: Prediction with recurrent neural network (lag = 50, number of neurons = 200)

3. Long Short-Term Memory Networks

- LSTM networks are a complex type of recurrent neural networks. They compose of a cell and input, forget and output gates. The cell is the state of the network. It has the cumulative information of past. For a time step, the input of that time step and the output generated from the one time step back creates the new updated input set. However, the information coming from the past are equally important just like the word we used at a time matters but the earlier words do also matter. So, we need to find the best mixture of this updated input set and the past information which is the cell state. In order to do that, we use input gate and forget gate. They are created with the input of the time step and the output generated from the one time step back. They typically use sigmoid activations for the fields of the corresponding cell and updated input. Sigmoid functions work like a switch here and generally works in a complementary manner. Forget gate determines which cell units to suppress and similarly the input gate determines for the updated input set. Both are gathered and the new *cell state* of that time is created. Using this we can form the output of that time. The input of the time step and the output generated from the one time step back creates one more gate called output gate and they of course use sigmoid switches to determine the updated output together with cell states.

I expected this network to be much more successful than the feedforward one though from my past experiences, I know using the predicted output as the input of the next time step cumulates the errors without any feedback from the original value of that time. It should still give a much better output. I prepared the network with the following code.

```
minRMSE = [inf 0 0];
for nF = numFeatures

    mu = mean(dataTrain);
    sig = std(dataTrain);
    dataTrainStandardized = (dataTrain - mu) / sig;
    [XTrain, YTrain] = getTimeSeriesTrainData(dataTrainStandardized, nF);
    dataTestStandardized = (dataTest - mu) / sig;
    [XTest, YTest] = getTimeSeriesTrainData(dataTestStandardized, nF);
    len = size(XTest); len = len(2);

    for nHU = numHiddenUnits

        layers = [sequenceInputLayer(nF) lstmLayer(nHU) fullyConnectedLayer(1) regressionLayer];

        options = trainingOptions('adam','MaxEpochs',250,'GradientThreshold',1, ...
            'InitialLearnRate',0.005,'LearnRateSchedule','piecewise','LearnRateDropPeriod',125, ...
            'LearnRateDropFactor',0.2,'Verbose',0,'Plots','training-progress');

        net = trainNetwork(XTrain,YTrain,layers,options);

        rmse = 0;
        tempXTest = XTest;
        for i = 1:len-1
            [tempNet, tempPred] = predictAndUpdateState(net, tempXTest(:,i));
            net = tempNet;
            tempXTest(end,i+1) = tempPred;
            tempXTest(1:end-1,i+1) = tempXTest(2:end,i);
            rmse = rmse + sqrt(mean((tempPred - YTest).^2));
        end
        [tempNet, tempPred] = predictAndUpdateState(net, tempXTest(:,len));
        rmse = rmse + sqrt(mean((tempPred - YTest).^2));
        rmse = rmse / len;
        if rmse < minRMSE(1)
            minRMSE(1) = rmse;
            minRMSE(2:3) = [nF nHU];
        end
    end
end
```

At this section I didn't create a validation set simply because there was none on the MATLAB script either. Although it would be fair to use a validation set to tune the hyperparameters of the network in a real world application especially for comparison reasons, as you will see LSTM performed far better anyways. I chose the maximum epoch number as 250. I have tried different number of hidden neurons {1, 2, 5, 10, 20, 50, 75, 100, 120, 150, 200, 240, 300} and features (lag) {1, 2, 5, 10, 20, 30, 50}. I didn't want to use a lag value more than half of the test set size. I have found 150 number of hidden units and 50 number of features (lag) gave the least error. After finding these, I was going to train with increasing maximum number of epochs, but the loss was already sufficiently small.

I learned that *InitialLearnRate* decreases with the *LearnRateSchedule* after *LearnRateDropPeriod* number of steps. I played with them after deciding the *numFeatures* and *numHiddenUnits* though I didn't observe substantial change. *GradientThreshold* is there to limit the gradient which happens to be very high at the earlier layers of very deep networks, also known as "exploding gradients".

- I have used the predictions for the upcoming input vector similar to the section 2. It is shown above inside the red box.
- LSTM network performed better by a margin as you can see from Fig. 4 below. As I already discussed, feedforward set doesn't have the memory concept. Feedforward network is just an artificial neural network which takes N number of inputs, which happened to be time series here, and make a prediction. It has no idea about the inputs earlier than N steps. On the other hand, LSTM keeps the information from whole history in its cells. It is very important to train it in order.

However, after some time steps, LSTM couldn't predict the correct values too simply because the cumulated errors was to high. If the predictions were based on the history of actual data and not the predictions, which is many times the case in reality, LSTM model would be working quite well.

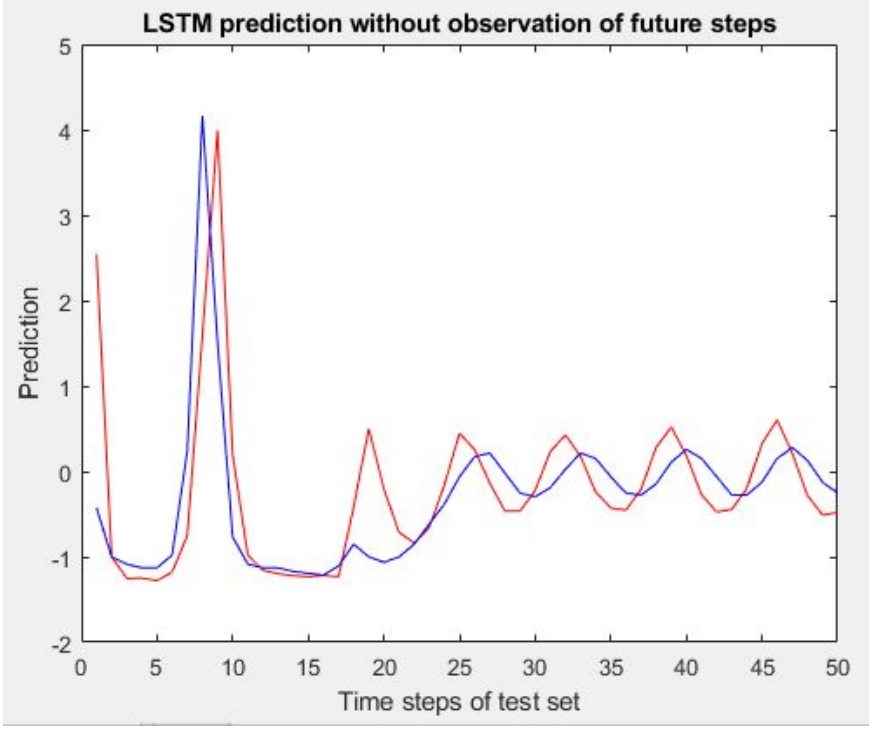


Fig. 4: Prediction without updating states with observed feature values, (lag = 50, number of hidden unit neurons = 150)

Artificial neural networks - Exercise session 3

Prepared by Ömer Yılmaz (r0779149)
omer.yilmaz@student.kuleuven.be

1. Principal Component Analysis

- I have followed the instructions of the assignment with the following code:

```
for q = [50,40,30,21]
    [V,D] = eigs(covX,q);
    reducedX = V'*X;
    Xhat = V*reducedX;
    sqrt(mean(mean((X-Xhat).^2)))
end
```

The amount of components are 50. Therefore, when I used 50 components, the error was almost zero except a numerical difference. When I used less and less component, the error increased more and more without any dramatic change.

I have done the same procedure for the Choless-All dataset with 21 principal components. However, the great majority of the information was kept with only few principal components. The biggest variance of the data was along the first few eigenvectors. So, the change in the reconstruction success with different number of principal components was quite sudden as can be seen from Fig. 1.

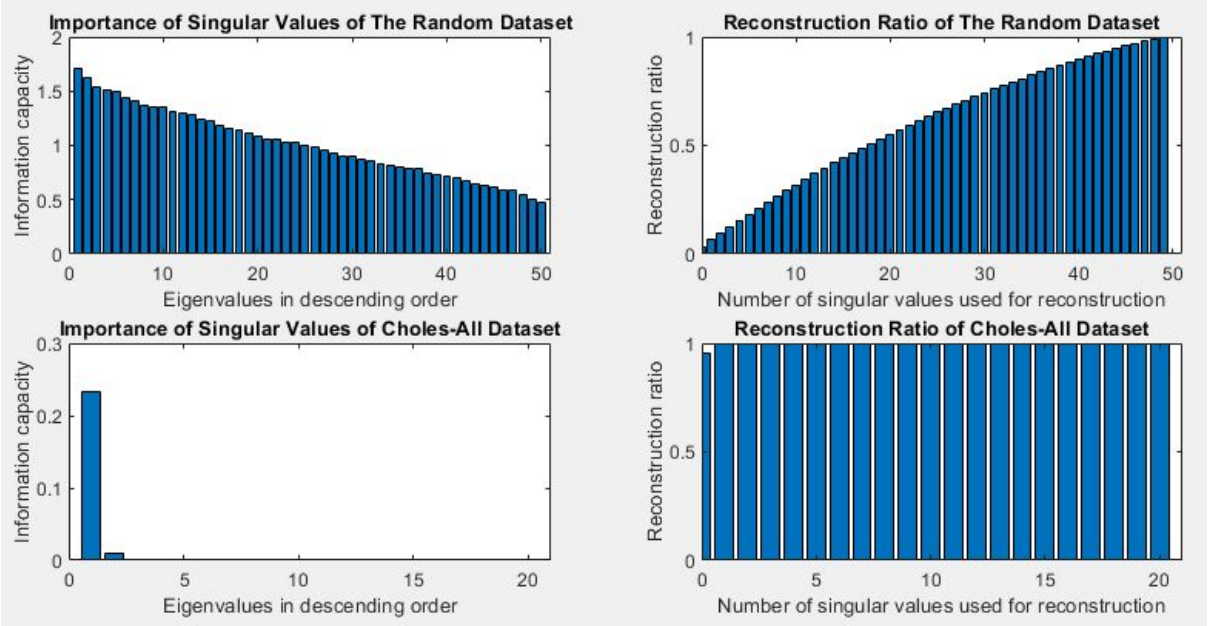


Fig. 1: Reconstructions of a random data (above) and a highly correlated data (below)

- mean function returns the average through the first axis whose length is more than 1.

The image of the mean value can be seen at Fig. 2.

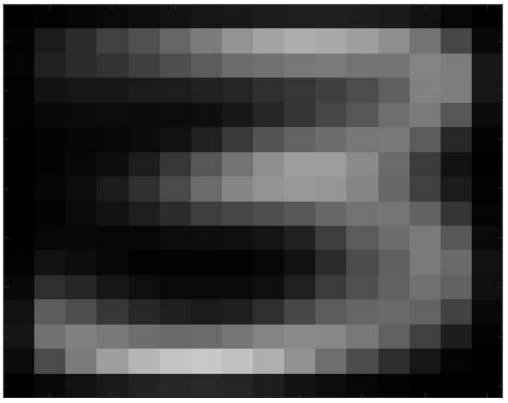


Fig. 2: Mean of all three values

- Eigenvalues of the covariance matrix of all threes are shown at Fig. 3 below.

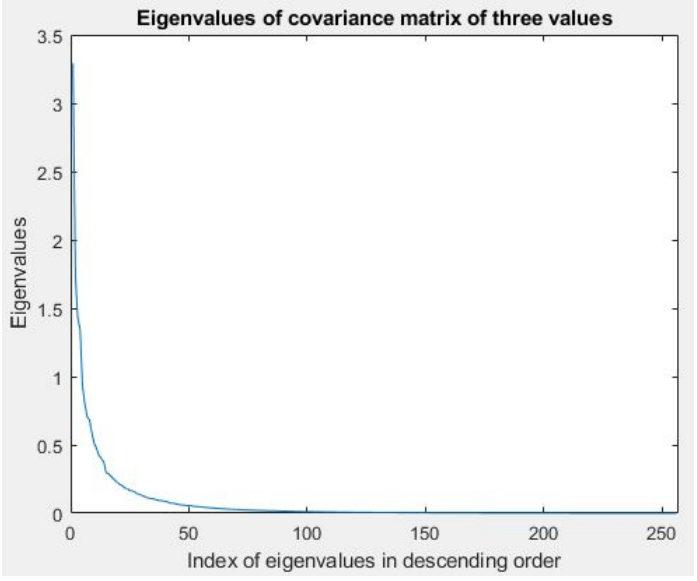


Fig. 3: Eigenvalues of the covariance matrix of all threes

- Three sample images are reconstructed with 1 to 4 components, Fig. 4.

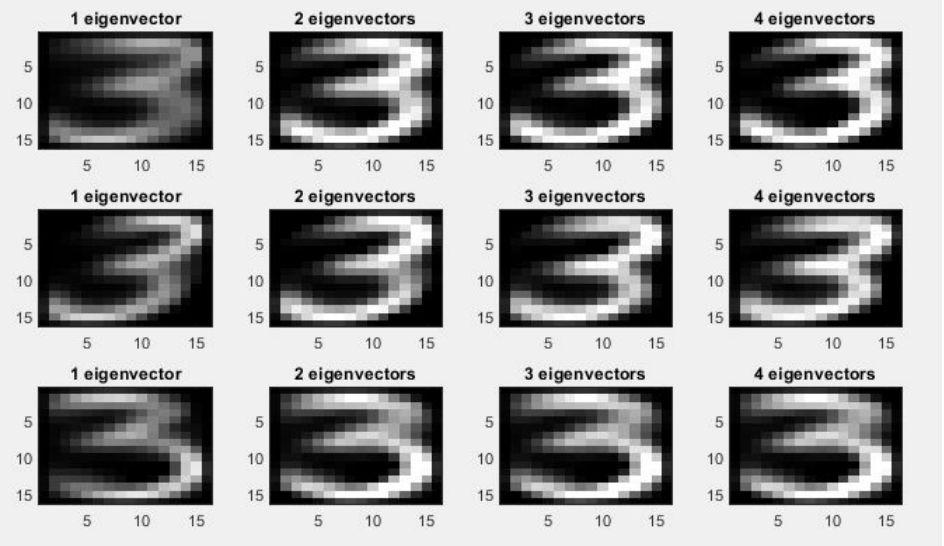


Fig. 4: Three sample images reconstructed with 1, 2, 3 and 4 components

- I have created a recovery error function as can be seen in the below code piece:

```
function mse = reconstruction_error(X,q)
    covX = cov(X);
    [V,D] = eigs(covX,q);
    Xhat = transpose(V*V'*X');
    mse = sqrt(mean(mean((X-Xhat).^2)));
end
```

Using this function I have found the reconstruction errors for each number of eigenvectors used. At Fig. 5, you can see the mean squared error after reconstruction.

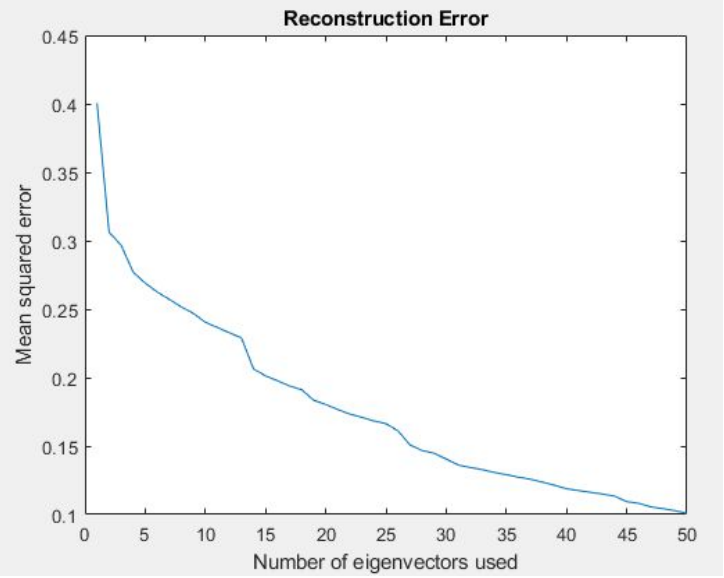


Fig. 5: mean squared error with the increasing number of eigenvectors used for reconstruction

- If we use all 256 eigenvectors to reconstruct the images, the mean squared error \ would ideally be 0 except some numerical error. The eigenvectors are orthogonal to each other and the transpose is the inverse. The error is found is 5.7107e-16. The reconstruction amount is increasing in Fig. 6 as opposed to mse in Fig. 5 with more eigenvectors used.

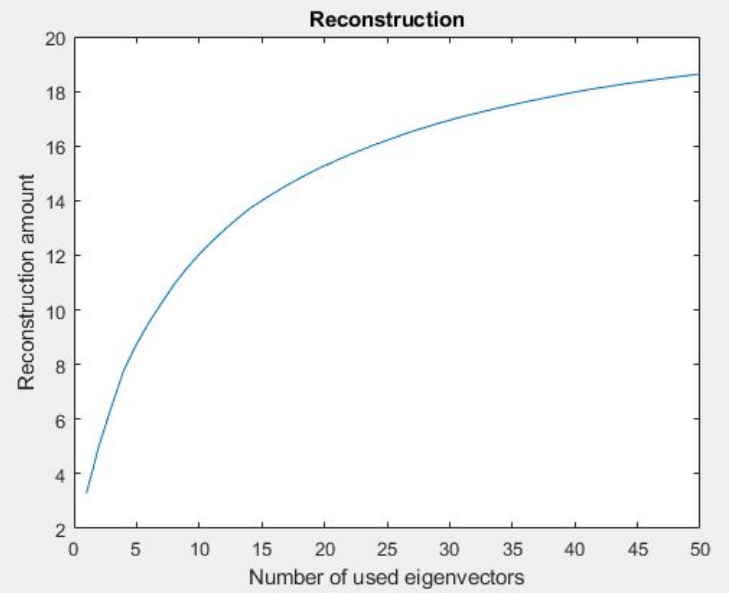


Fig. 6: The reconstruction amount with the increasing number of eigenvectors used for reconstruction

2. Stacked Autoencoders

- I used as 'MaxEpochs' a big number, 1000 and waited for the minimum gradient to be reached. Because, I didn't want to miss a better architecture because of less epochs. I had a very clear idea about the last layer. If I used a bigger number of neurons, obviously the encoder would perform better. But, our purpose is to encode the information using a small number of hidden layers. I also knew that I should use decreasing number of neurons for upper layers to stack more and more. Although one may speculate, extracting rich features of the input at first may be more beneficial for the upper layers and this feature vector can be bigger than the input vector, I didn't experimented and followed the given exercise's procedure.

Layers	[100 50]:	[200 50]:	[500 50]	[200 100 50]	[400 200 50]	[300 200 100 50]
Error (%)	1.4	1.8	2.0	1.3	1.7	1.4

After some time I figured out, autoencoders with sudden changes between the numbers of the neurons used in the input layer and the hidden layer increases the likelihood of the error. Encoding should be gradual.

I have achieved a better result from the default one with three layers of 200, 100 and 50 neurons. I also tried different multilayer networks with the variations of code piece below for 2, 3 and 4 hidden layers and none of them was even close to the performance of the stacked autoencoders.

```
max3HL = [0 0 0 0];
numbersOfNeurons = combnk([300 200 100],2)';
numberOfEpochs = [1000];
for nN = numbersOfNeurons
    for nE = numberOfEpochs
        net = patternnet([nN(1) nN(2) 50]);
        net.trainParam.epochs = nE;
        net=train(net,xTrain,tTrain);
        y=net(xTest);
        plotconfusion(tTest,y);
        if 100*(1-confusion(tTest,y)) > max3HL(1)
            max3HL(1) = 100*(1-confusion(tTest,y));
            max3HL(2:4) = [nN(1) nN(2) nE];
        end
    end
end
```

Multilayer network performed substantially worse than the stacked autoencoder which is actually the expected result. Because, when we are building the upper layers of autoencoders we are transferring the weights for the earlier layers. This is similar to transfer learning. If we didn't train the earlier layers before and train everything in one-shot like at the fine tuning phase, the performance would degrade considerably. In addition to this, for longer and longer stacks the gradients at the earlier layers may have vanished and learning a better architecture would be much harder. Also, one may speculate using more

parameters in one training would result in a better prevention from the local minimums of the whole loss function, the effect is much less important compared to the vanishing gradients. This greedy, one by one learning scheme can also be likened to the training of Deep Boltzmann Machines where layers handled separately. Though the gradients are found quite differently.
Here are the checking for d code pieces

Fine tuning is very important. Because, the trained autoencoders have errors and these errors are accumulated to the upper layers. The ground truth of the upper layers are wrong and the errors coming from earlier layers are no way handled at the upper layers. So, this is an integrated error. Fine tuning handles this problem very well.

rng('default') is a seed for the random number generator of MATLAB. This is a quite common procedure in other languages as well and is used for code result reproducibility. The random generators of the computers are not truly random. Their random number generators produce the same results when we give them these seeds. This is also crucial for scientific findings. Because we want to back our results with the same experiments and here we want to initialize the weights of our network the same way everywhere the code runs.

3. Convolutional Neural Networks

- This layer checks for the basic shapes of the images. The upper convolutional layers catches more and more complex shapes building on top of earlier layers.
- [27 27 96] is the answer. Because, (227 - 11) / 4 + 1 = 55. (55,55,96) is the input to the Max pooling layer whose stride is 2.
- Before the fully connected layers (just after the last max pooling layer), the dimension is [13 13 256]. The filtered images are many times smaller than the [227 227 3] and there are much more filters. We convoluted the image and extracted features from it. The spatial differences between the pixels are extremely important in image handling. Therefore we need to consider input pixels together. We can manage this by filters and we can move the filters through the image to catch the actual searched objects inside the image file. We can build complex features on top of more basic features of the images the same way. When we flatten the image pixels into a vector, we no longer use this filtering mechanism. The architecture of the feedforward networks is not as good fit as convolutional neural networks for these tasks.

I have tried different architectures. First of all I increased the *'MaxEpochs'* option 200 instead of 15. I ran the initial network given. The accuracy was:0.9824. Then, I added Batch Normalization layers after convolutional layers. The accuracy became: 0.9848. I added one more [convolution2dLayer(5,48), batchNormalizationLayer, reluLayer] block increasing the depth and with more filters. The accuracy became: 0.9960. When I replaced *ReluLayer*s with *leakyReluLayer*s. I haven't observed considerable change: 0.9964. I used a deeper network each time increasing the filters to extract more and more features as given in code piece below. In 441.38 seconds, the training accuracy became 0.9952. This is again very good.

```
layers = [imageInputLayer([28 28 1])
  convolution2dLayer(5,12)
  batchNormalizationLayer
  reluLayer

  convolution2dLayer(5,24)
  batchNormalizationLayer
  reluLayer

  maxPooling2dLayer(2,'Stride',2)

  convolution2dLayer(3,48)
  batchNormalizationLayer
  reluLayer

  convolution2dLayer(3,96)
  batchNormalizationLayer
  reluLayer

  maxPooling2dLayer(2,'Stride',2)

  fullyConnectedLayer(10)
  softmaxLayer
  classificationLayer()];
```

I implemented a “depthwise separable convolutional block” like in MobileNetV2[1].

```
% Convolutional block
convolution2dLayer(5,24)
batchNormalizationLayer
reluLayer

% Depthwise separable convolution block
% Expansion layer
convolution2dLayer(1,36)
batchNormalizationLayer
reluLayer

% Depthwise separable convolution layer
groupedConvolution2dLayer(5,1,'channel-wise')
batchNormalizationLayer
reluLayer

% Projection layer
convolution2dLayer(1,24)
batchNormalizationLayer

% Pooling layer
maxPooling2dLayer(2,'Stride',2)
```

Both 20 and 200 epochs of above two models were successful. Depthwise separable convolution block increases the computational speed by decreasing the number of parameters of a classical full channel convolutional block dramatically. For example, blocks of models like the one above or Vgg16[2] etc...

---

[1] Sandler, Mark, et al. "Mobilenetv2: Inverted residuals and linear bottlenecks." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018.

[2] Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." *arXiv preprint arXiv:1409.1556* (2014).

Artificial neural networks - Exercise session 4

Prepared by Ömer Yılmaz (r0779149)
omer.yilmaz@student.kuleuven.be

1. Restricted Boltzmann Machines

- More components tend to increase prediction performance as shown in Fig. 1.

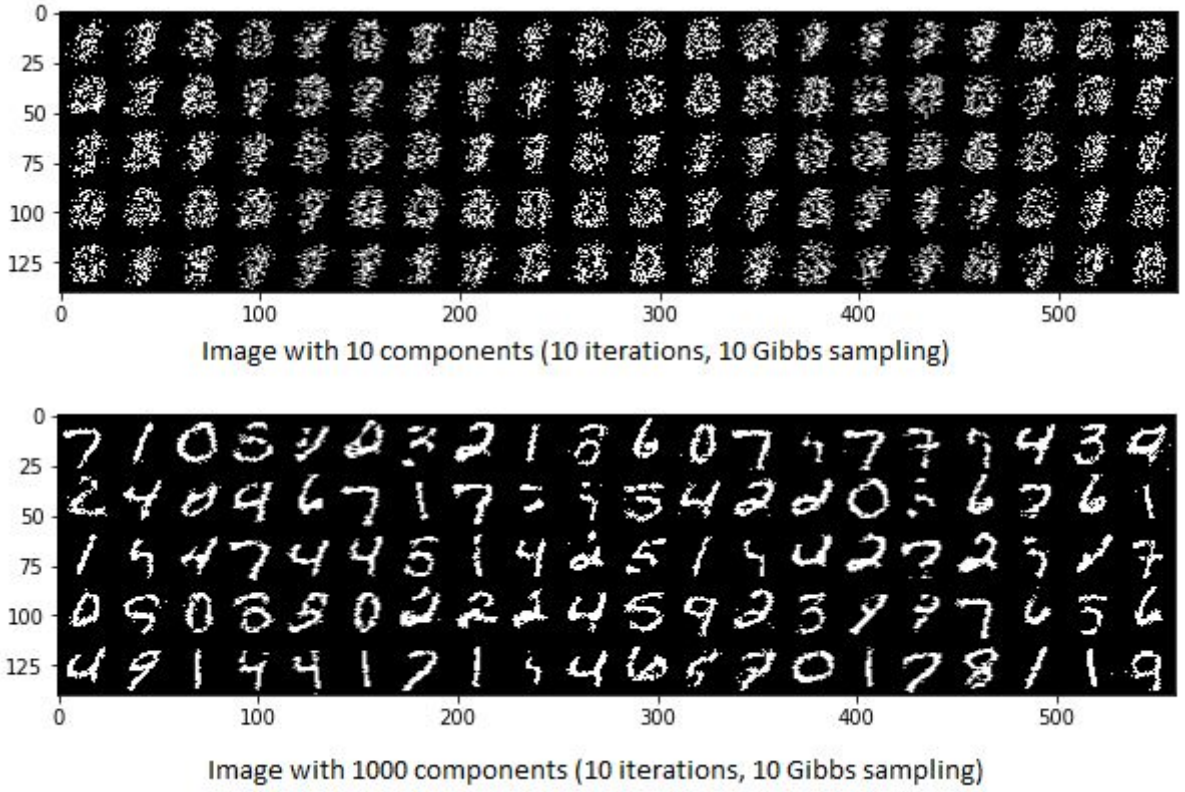


Fig. 1: Sampled images trained with 10 components (above) and 1000 components (below)

However, it requires more training time to achieve it.The time required for training increases with the number of components as shown in Tab. 1.

	10 components	100 components	1000 components
Avg. time per iteration	3 s	13 s	80 s

Tab. 1: Average time required for an iteration (lr=1, random states = 0)

The probability model P(v,h;θ) for the network is defined purely based on parameters (θ = {W, b, a}). Using this model we define a conditional probability model P(h|v;θ). When we are given data, we are actually given an empirical marginal probability $P_{data}(v)$ distribution for the visible units. Using both we can have an empirical, data-based joint probability distribution. We also have the general model-based joint probability. We want to increase the likelihood of our data given the parameter set. To do that we use gradient descent and the

gradient is the difference of the expectations (towards data) based on both models we have above. However, the mathematics is intractable for the exact model basel probability distribution. Instead of that, we use Gibbs sampling (Contrastive Divergence algorithm). Increasing number of Gibbs samples converges to the expectation found from closed-form model-based joint probability at infinity. So, increasing Gibbs samples gives us better gradients. Though, in practice a finite number is fine.

Having more epochs, increases the likelihood of our data. You can see the increased likelihood in Fig.2 below. However, there is a capacity of the network and to have better predictions much more iterations are required. When the iterations doesn't help, the capacity should be increased.

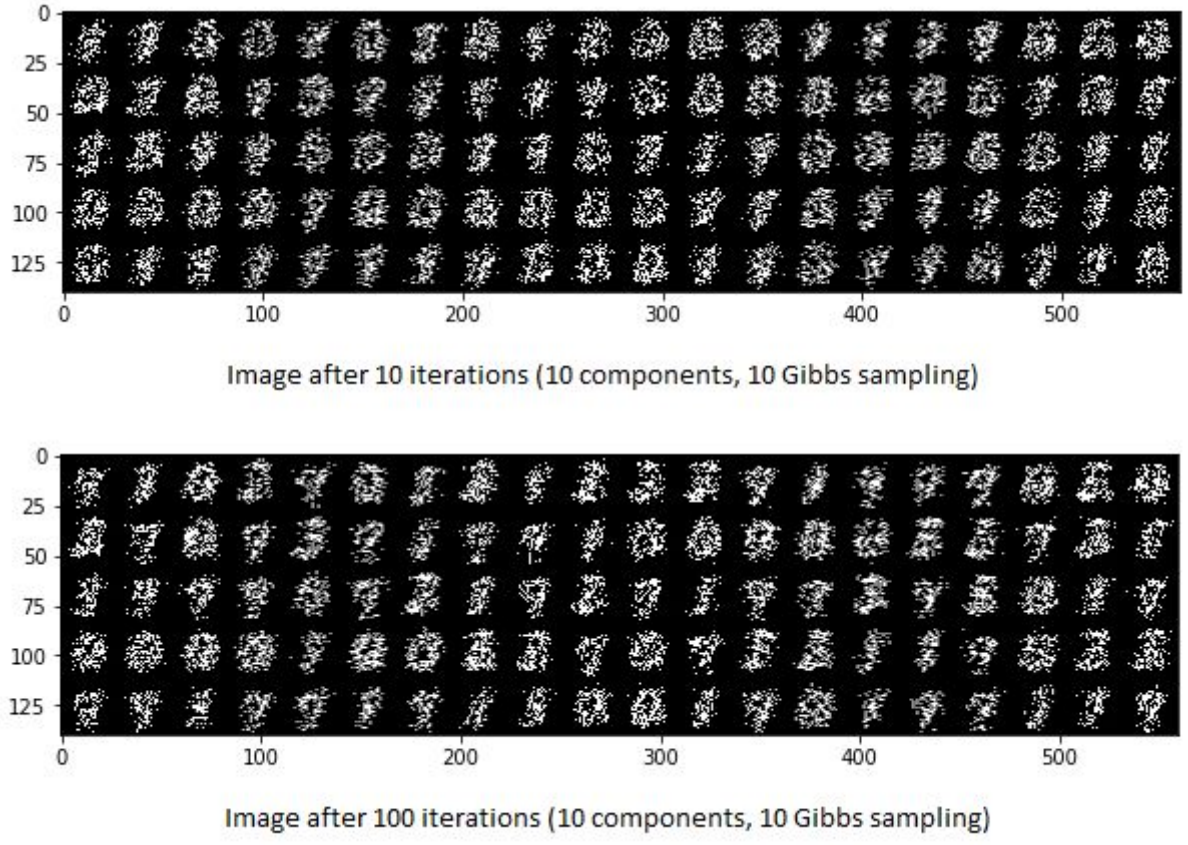


Fig 2: Sampled images trained with 10 iterations (above) and 100 iterations (below)

- More hidden units increased the representation power of the model and resulted in a better reconstruction. Learning rate increased the effectiveness of the iteration as it is multiplied with the gradients. However, increasing learning rate too much destabilized the network as I expected.

Removing more data resulted in a worse reconstruction. However, when the same number of pixels removed from different locations, the network managed to reconstruct the image a bit more successfully. Because in the first case, it is not clear what the figure was visually. In the second case although the numbers kept the same, conditional probability is increased.

2. Deep Boltzmann Machines

- Weights look different as shown in Fig. 3.

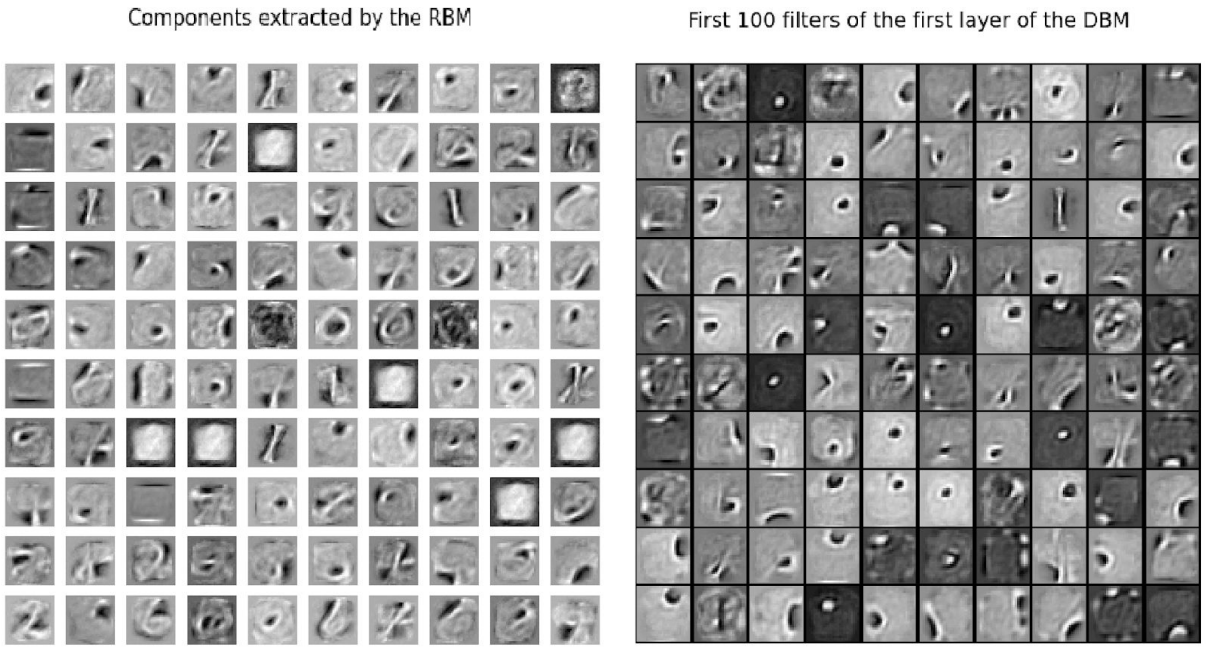


Fig 3: Learned features: first layer (RBM) and second layer (second RBM of DBM)

Second layer is able to learn more complex features of image similar to what happens at convolutional neural network layers. Its input is not the visible unit but the first order features. Then, another RBM is trained using these hidden units of the first layer and the final stack is called as DBM.

- The quality is a bit better, as shown in Fig.4, not because of the architecture but the training. The weights are fine tuned very well compared to mine. It has nothing to do with being deep. The layers of the DBM are trained separately anyways. Using sample_v method of the API, the first RBM's visible layer sampled.

Fig 4: Samples generated by Deep Boltzmann Machine

3. Generative Adversarial Networks

- It is not very clear when we should stop training a standart GAN network. Because the discriminator and the generator are competing with each other. Therefore, there is no clear optimum of a general optimization function to work on. One typically generates some image samples and evaluate how similar they look to the real ones.

Parameters of GANs may never converge because it is a zero sum game. Discriminator and generator may possibly always countermeasure each other. This can result in oscillations between the parameters possibly exploding. The discriminator can be too successful at its predictions so that the generator can learn nothing with vanishing gradients. Furthermore, GAN may generate quality images however it may suffer the problem called “mode collapse”. This means generating only a subset of the dataset we want to generate. For example we may have trained a generator with high accuracy for MNIST dataset. However, it may generate only ‘6’ values or ‘2’ values. All the other ‘modes’ are diminished. Decisions on the number of neurons, layers etc. have very high impact.

We, however, may diagnose when something goes wrong and take measures. For example, when the discriminator is having a comparatively small loss, this indicates our generator is not able to deceive the discriminator. It is not generating. The very small loss is equivalent to an accuracy much higher than 50%. On the other hand, if the accuracy is around 50% for many batches, this may indicate the generator is doing great or maybe the discriminator is just a very poor one. We can look at to the samples generated by the generator and tell of course. Also, we can increase the number of training cycles and help the poorer model. If the discriminator is predicting with great accuracy, maybe it is time to train the generator heavily to make it catch up. The accuracy of the disciminator -in the long run- is not expected to be smaller than 50%. Because even a blind guess would have 1 accurate prediction among 2 over longer trials. However for batches, especially for the smaller ones, anything can be observed very easily. So, the loss and the accuracy will deviate a lot and numbers don’t mean too much. When the losses converge to some number, we can guess that the network is no longer learning. It doesn’t mean the generator performed well though. Also, when it doesn’t converge the outputs could be nice as well.

DCGAN is trained with 32000 batches of the bird class of the data set. The losses of the generator and the discriminator are given in Fig. 5 a) the accuracies are given in Fig. 5 b).

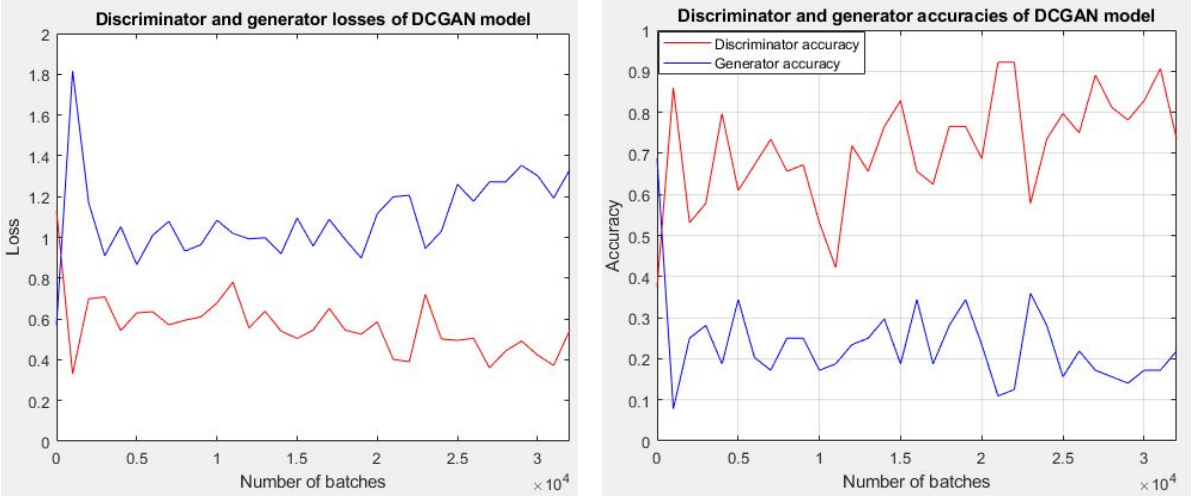


Fig 5: a) Generator loss and discriminator loss **b)** generator accuracy and discriminator accuracy

From the Fig. 5, it seems that the network continues to learn. It didn't converge to some accuracy and deviates. However, it is quite stable. Some generated images with 4000 batch intervals are given in Fig 6.

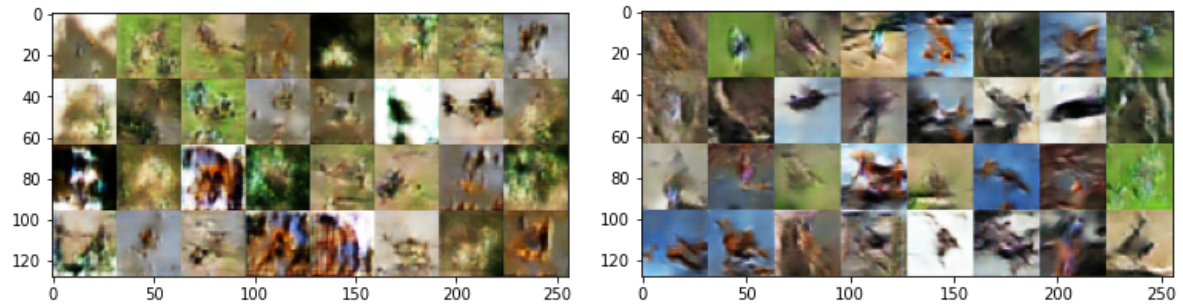
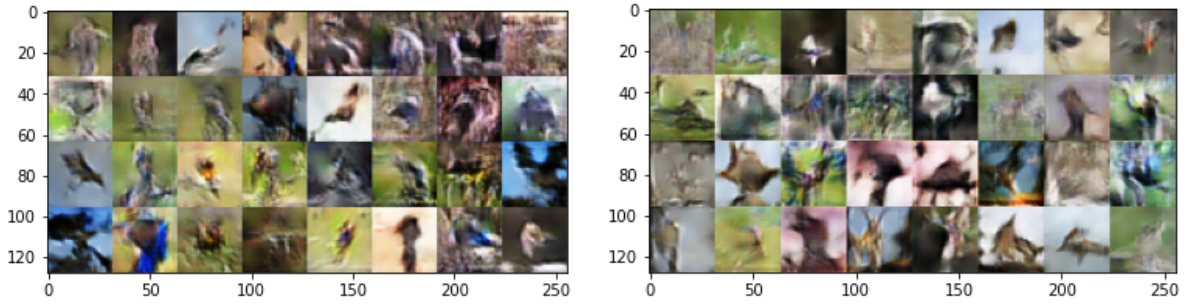
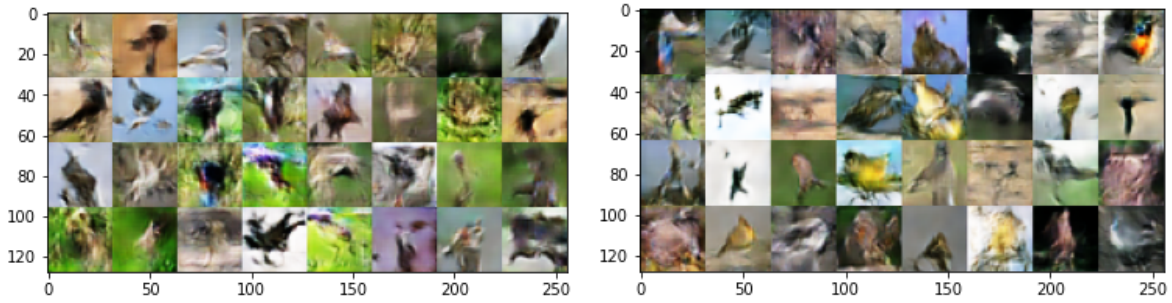


Fig 6: Generated images after **a)** 4000 batches **b)** 8000 batches



c) 12000 batches **d)** 16000 batches



e) 20000 batches **f)** 32000 batches

The network generated quite successful images. Although the loss and the accuracy continues to change, the network didn't improve much after 16000 batches . After 32000 batches I stopped training the network. DCGAN already generates some great bird images anyways.

4. Optimal transport

- The color of the target domain is to be transferred with a map. However, this is not a direct map of corresponding pixels of two arrays which would result in dramatic value changes. It finds the least total pixel changes with transferring also between distinct pixels. Non optimal color swapping wouldn't transfer the styles but try to show the pixels of the target locally (no spatial transports, completely swapping or averaging pixels with a weight).
- The quality of images improved with Wasserstein GAN models as shown in Fig. 8 because the standart GAN's discriminator doesn't give meaningful data when the generator

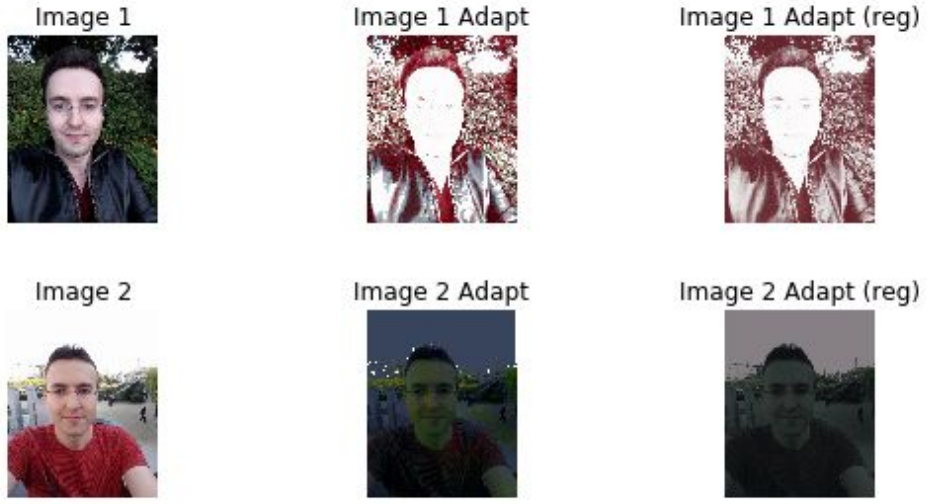


Fig 7: Optimal transport with **a)** Earth Mover's distance (middle) **b)** Sinkhorn transport (right)

is not performing well. Wasserstein distance smoothes the penalties for probability distribution differences. Each mode can contribute to the generator parameters preventing mode collapse because, we no longer use a simple binary cross entropy metric of accuracy. A mathematical result for Wasserstein GANs is that its Wasserstein distance function should obey Lipschitz constraint which can be achieved directly clipping the weights or using a regularization method for weights (gradient penalty).

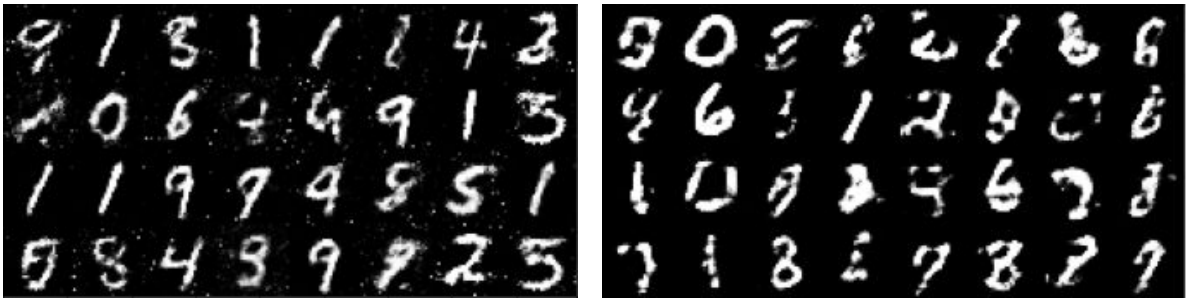


Fig 8: Generated MNIST images with **a)** GAN **b)** Wasserstein GAN with weight clipping

c) Wasserstein GAN with gradient penalty

Fine tuning the clipping parameter of Wasserstein GAN's, however, is hard. The gradients can vanish or explode very easily. In order to address this issue, gradient penalty is used similar to ridge regression. The best quality of images is generated as in Fig. 8 c).