Exercises ASCI course: Computer Vision by Learning

Antonio Garcia-Uceda Juarez
a.garciauceda@erasmusmc.nl

1 Exercise 1: Multi-layer Perceptrons (MLPs)

Objective: Implement a fully connected MLP model or arbitrary number of hidden layers, to learn to classify digits from MNIST data. Thus, this is a multi-class classification task.

Question 1.1: The implementation of the MLP network is in file './code/part1_MLP_CNN/mlp_pytorch.py', and the training procedure in './code/part1_MLP_CNN/train_mlp_pytorch.py'

In this exercise there are many hyperparameters / modifications of the MLP we can try to improve the performance: i) more hidden layers, ii) more units per hidden layer, iii) use of non-linear activation in hidden layers, iv) activation in output layer, v) batch size, vi) regularisation by means of dropout, batch normalization, and/or weight decay, among others. We focused only on studying the effect of i), ii) and iii), with iii) by means of non-linear RELU. All models tested are displayed in table 1.

The accuracy curves of the models tested are shown in the a in figures 2 and 2. We can observe how the performance in the training set increases with the complexity of the model. However, the performance in the validation set remains almost unchanged and with an accuracy approx. 0.40, close to the value indicated in the exercise. Thus, the deeper models suffer from a strong overfitting, which is not desirable. We can see that the models with 1000 units per hidden layer achieve a better performance in the training set than those with 100 units, but these suffer from stronger overfitting, due to the much larger number of trainable weights as seen in table 1. Finally, the use of non-linear RELU is definitely beneficial as the performance of all models improves, more clearly in the training set and only slightly in the validation set. With all said, we chose the model '100-100' as the best one, which performs very similar to the '100', but using two hidden layers can allow for much higher capacity than only one, with a small increase of trainable weights of 3%. The loss and accuracy curves of this model are shown in figure 3.

Table 1: All MLP models tested

Name	Num. Hidden Layers	Nu Units Hidden	Use RELU	Num. Weights
100	1	100	yes / no	308.310
1000	1	1000	yes / no	3.083.010
100-100	2	100	yes / no	318.410
1000-1000	2	1000	yes / no	4.084.010
100-100-100	3	100	yes / no	328.510
1000-1000-1000	3	1000	yes / no	5,085,010
100-100-100-100	4	100	yes / no	338,610

2 Exercise 2: Convolutional Neural Networks (CNNs)

Objective: Implement a deep CNN model, in particular the VGGnet, to learn to classify digits from MNIST data, and compare with performance of MLP model in Exercise 1. Thus, this is a multi-class classification task.

32nd Conference on Neural Information Processing Systems (NeurIPS 2018), Montréal, Canada.

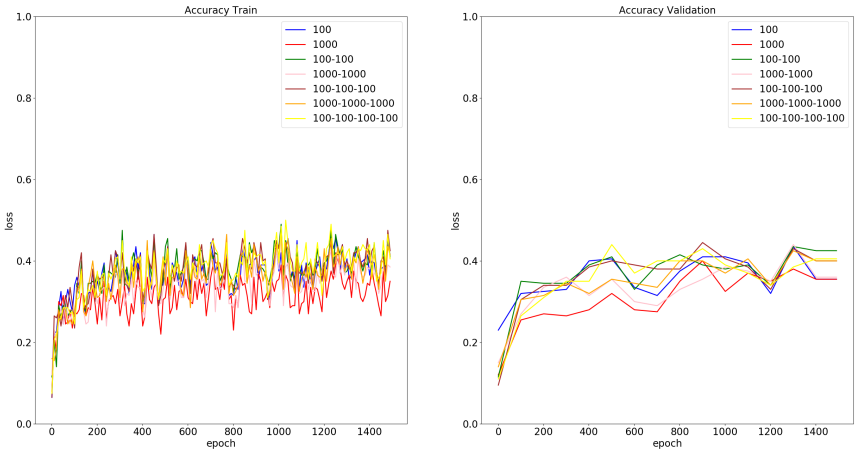

Figure 1: Training history of accuracy on i) training and ii) validation data, for the MLP models tested without non-linear RELU activation, displayed in table1

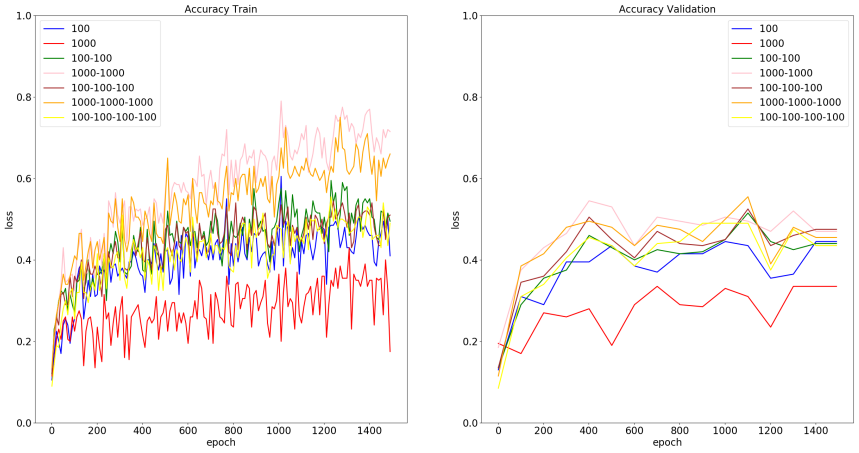

Figure 2: Training history of accuracy on i) training and ii) validation data, for the MLP models tested with non-linear RELU activation, displayed in table1

Question 2.1: The implementation of CNN-VGGnet network is in file ’./code/part1_MLP_CNN/convnet_pytorch.py’, and the training procedure in ’./code/part1_MLP_CNN/train_convnet_pytorch.py’

The loss and accuracy curves are shown in figure 4. The large oscillations are most likely due to the fact that the loss / accuracy is computed every 10 batches, and not after a full epoch, since otherwise the data points would be too few to be plotted. The accuracy level (in the validation set) at the end of training is approx. 0.72, close to the value indicated in the exercise. Most likely the accuracy can be further increase by letting train the model for longer, but this was not possible due to time constraints. It is also clear that the accuracy obtained with the CNN model is much higher that those reported for the MLP models in Exercise 1.

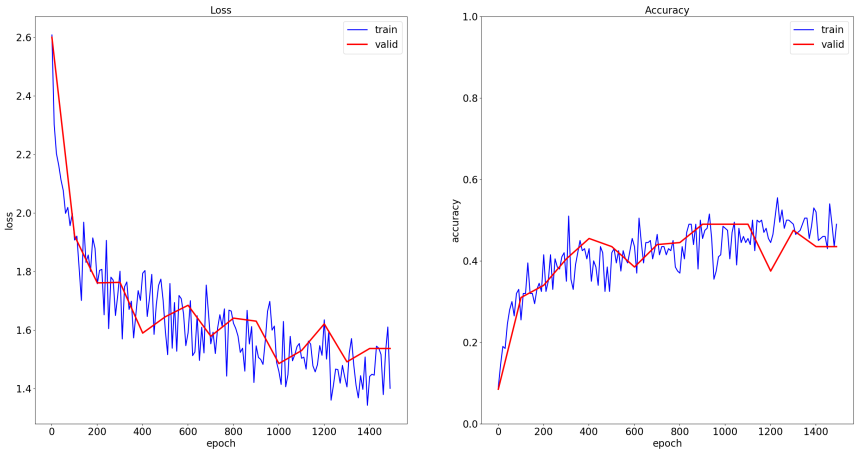

Figure 3: Training history of both i) loss and and ii) accuracy for the best MLP model: 100-100 hidden units, and with RELU activation.

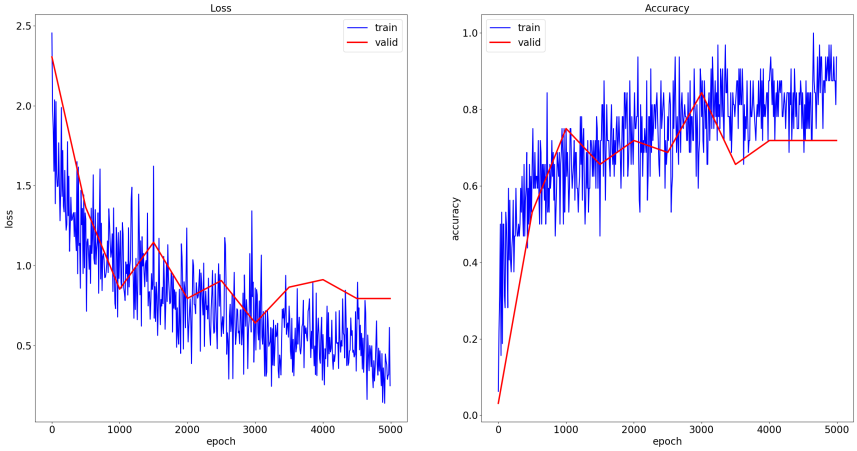

Figure 4: Training history of both i) loss and and ii) accuracy for the CNN-VGGnet model.

3 Exercise 3: Recurrent Neural Networks (RNNs)

Objective: Implement recurrent networks (a normal RNN and LSTM model) to learn to predict the last digit of a palindrome of length T. Thus, we do multi-class classification with data (long) sequences of numbers.

Question in code: The command is to limit the gradients of weights beyond a given value 'config.max_norm'. This is to avoid the problem of exploding gradients typical of very deep networks, which is specially severe for RNNs when analysing long sequences.

Question 3.1: The implementation of RNN network is in file './code/part2_RNN/vanilla_rnn.py', and the training procedure in '../code/part2_RNN/train.py'.

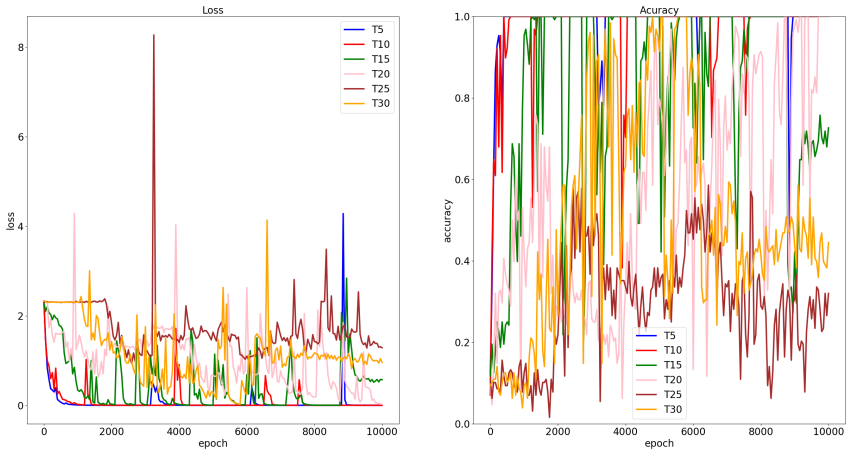

Figure 5: Training history of both i) loss and and ii) accuracy for the RNN models tested for increasing length of input sequences.

Question 3.2: Experiments with the vanilla RNNs have been performed for different lengths (T) of the input sequence. The training history of both loss and accuracy are shown in figure 5. We can see that the model can predict with perfect accuracy the last digit of the sequences, up to a length of approx. $T = 15$. For larger inputs, the models suffer from a very erratic convergence, and the accuracy drastically drops.

Question 3.3a: The explanation of parts of a LSTM network are below. Say that all states are learnt as a non-linear combination of the inputs available: i) x_t (or h_{n-1} of previous layer for deeper networks), and ii) h_{t-1} from previous time step.

1. Input gate i: This is the input to the learn state of the LSTM cell. This would correspond to the normal hidden state in a vanilla RNN. The sigmoid activation is chosen so that this input is condensed between $[0, 1]$, so as normalized input.

2. Modulation gate g: This layer serves to 'modulate' or modify the input layer, so that the information can be i) 'kept' ($g = 1$), ii) modified ($g = (0, 1)$), iii) forgotten ($g = 0$), or iv) even 'reverted' ($g = [-1, 0)$), or any combination of these in a high dimensional space. The $tanh$ activation is chosen so that to condensed the layer to $[-1,])$, to comprise the options just exposed.

3. Forget gate f: This layer serves to 'forget' or remove parts of the state of the LSTM cell from previous steps that either i) are not interesting anymore, or ii) can be replaced by other more relevant information from more recent steps. The sigmoid activation is chosen so that this input is condensed between $[0, 1]$ which represent states between 'forget' and 'remember'.

4. Output gate o: This layer serves a similar purpose as the forget gate 'f', so that in can remove some parts of the state of the LSTM cell. However this gate acts over the state c, which in turn is a combination of all the previous gates. The sigmoid activation is chosen for the same reason as for the forget gate 'f'.

Question 3.3b: For the formula of trainable parameters of a LSTM cell: all 4 gates 'i, g, f, o' consist of a matrix-matrix multiplication with three weights matrixes: i) 'W_x' multiplying the state x_t, of dimension nxd, ii) 'W_h' multiplying the hidden state from previous step h_{t-1}, of dimension nxn, and a bias vector 'b' of dimension n, in case this is chosen. Thus, the formula is: $4(mn+n^2+n)$, if bias is used, or $4(mn + n^2)$ otherwise.

Question 3.4: The implementation of LSTM network is in file '../code/part2_RNN/lstm.py', and the training procedure in '../code/part2_RNN/train.py'.

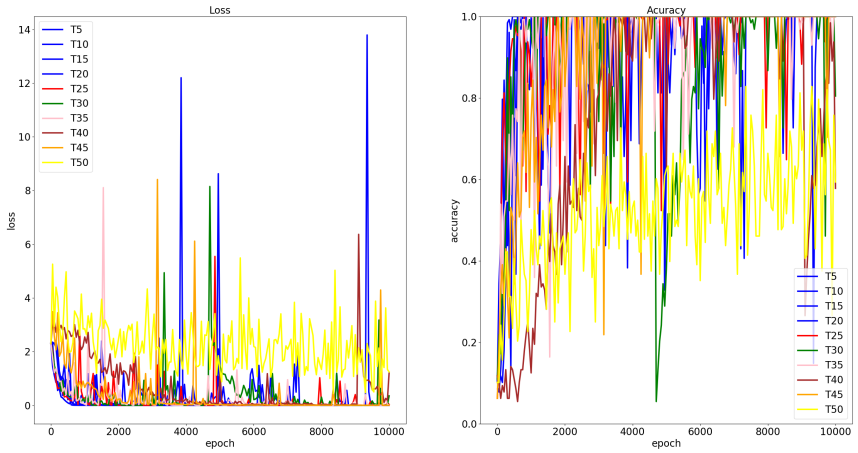

Figure 6: Training history of both i) loss and and ii) accuracy for the LSTM models tested for increasing length of input sequences.

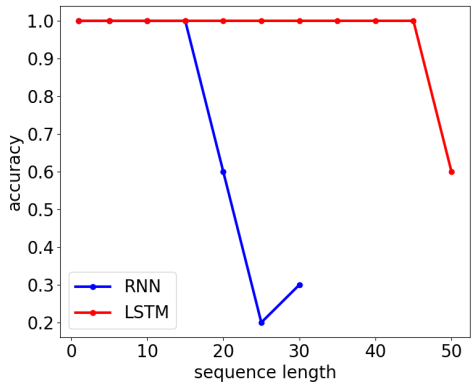

Figure 7: Difference in performance between RNNs and LSTMs tested, as the final accuracy for increasing length (T) of the input sequence

Experiments with the LSTMs have been performed for different lengths (T) of the input sequence. To make the models converge, the learning rate needs to be increased for larger inputs. A rate of $lr = 0.001$ was used for lengths up to $T = 15$, thereafter it was increased to $lr = 0.01$ for lengths up to $T = 20$, thereafter to $lr = 0.02$ for lengths up to $T = 40$, and thereafter it was carefully tuned up to a max. value of $lr = 0.1$. The training history of both loss and accuracy are shown in figure 6. We can see that the model can predict with perfect accuracy the last digit of the sequences up to a length of $T = 45$, much larger range than that observed for the vanilla RNNs. These models have a smoother convergence, and the accuracy quickly reaches the maximum value. This exposes the much greater capacity of LSTMs to remember sequence states very far away in time, thanks to the gating layout explained above. The comparison of the performance of both models is shown in figure 7.

Question 3.5: The implementation of LSTM for MNIST classification is in file './code/part2_RNN_bonus1/lstm_mnist.py', and the training procedure in './code/part2_RNN_bonus1/train_lstm_mnist.py'.

The implementation of the model / training procedure have been done successfully. Converged results have not been obtained yet. But in view of the results presented in Question 3.4 for the LSTM models,

we believe that we needed to increase the learning rate and tune up to obtain optimal convergence. We could not do further tests due to time constraints.

4 Exercise 4: Generative Adversarial Networks (GANs)

Objective: Implement a GAN network to learn to generate random images similar to the MNIST dataset so that they are indistinguishable from them.

Question 4.1: The implementation of GAN network and training procedure is in file '../code/part3_GNN/a3_gan_template.py'.

The GAN implemented is composed of i) a generator and ii) a discriminator. The generator creates images from a latent space, i.e. random multidimensional Gaussian noise. This is a fully connected network of 4 hidden layers that transforms the latent space to the higher dimensional space of 28x28 images. Each layer is composed of i) a fully connected layer, followed by ii) batch normalization, and iii) non-linear activation Leaky-RELU. On the other hand, the discriminator distinguishes between images created by the generator (or fake) from real images. This is a fully connected network of 3 hidden layers to perform classification. Each layer is has a similar definition as for the generator. The discriminator is trained with a binary-cross entropy loss function to learn to classify the images between 'fake' (0) and 'real' (1). The generator is trained to maximize the loss: $log(D(G(z))$, with D the discriminator and $G(z)$ the fake images. A separate optimizer is used for both subnetworks, being the Adam with default learning rate 0.0002.

Question 4.2: The samples of images generated are shown in figure 8. Additionally, the loss history of both discriminator and generator are shown in figure 9. We observe that the loss for the generator is very large at the start of training, logically, and then decreases as training goes on, indicating that the generator improves at the task. At the same time, the loss for the discriminator increases with time, which is expected since as the generator improves, the discriminator find more difficult to distinguish the fake from real images.

Question 4.3: The generated images from interpolated samples in the latent space are shown in figure 10. In this case, the levels are between the digits '4' and '9'. It is clear the intermediate states correspond to gradual blending between the two digits.

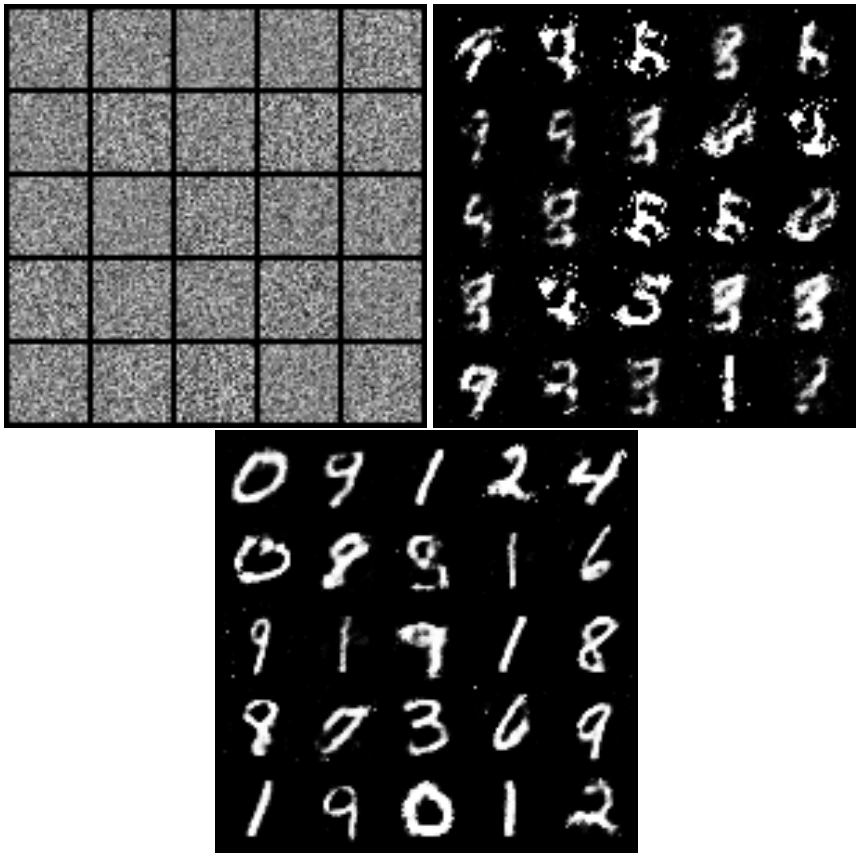

Figure 8: Batch of 25 images generated i) at the start of training, ii) halfway during training, and iii) at the end of training

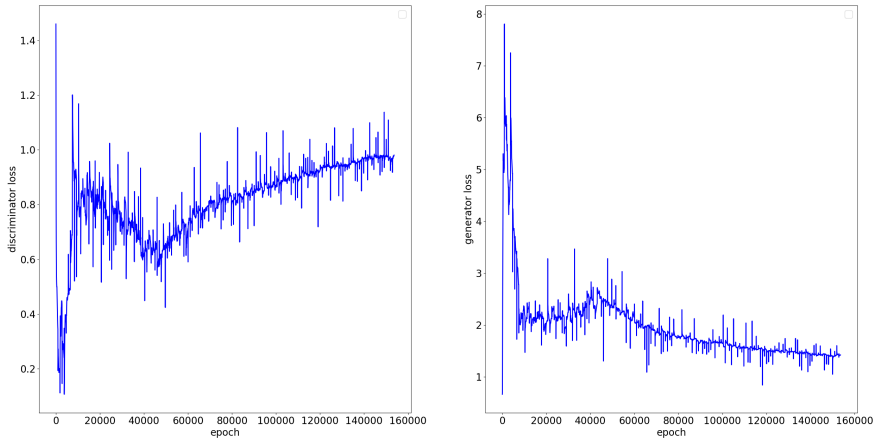

Figure 9: Loss history of both i) discriminator and ii) generator

Figure 10: 9 images generated from interpolated levels of two random samples of the latent space.