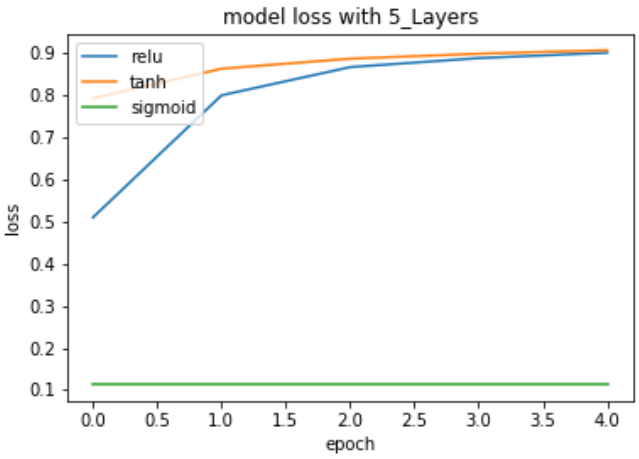
Machine Learning

Assignment 2

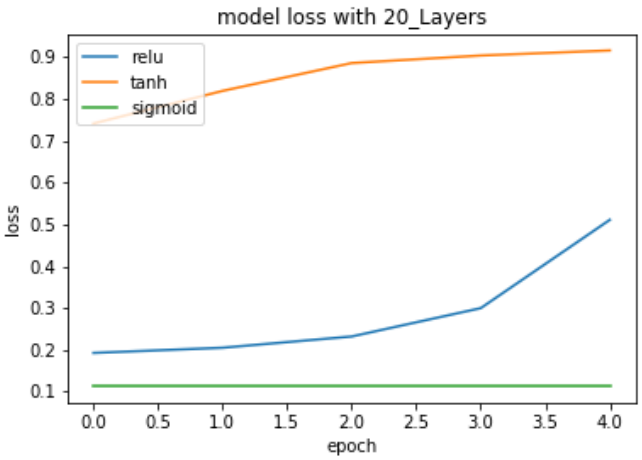
Σιατήρας Βαγγέλης

Exercise 1.B

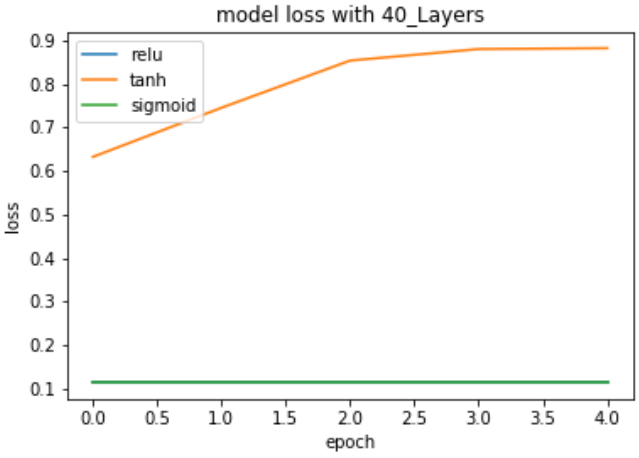
With 5 layers	
ReLU	Test loss = 0.323 Test accuracy = 0.908
Tanh	Test loss = 0.333 Test accuracy = 0.909
Sigmoid	Test loss = 2.3 Test accuracy = 0.114



With 20 layers	
ReLU	Test loss = 1.283 Test accuracy = 0.511
Tanh	Test loss = 0.347 Test accuracy = 0.916
Sigmoid	Test loss = 2.301 Test accuracy = 0.114



With 40 Layers	
ReLU	Test loss = 2.301 Test accuracy = 0.114
Tanh	Test loss = 0.633 Test accuracy = 0.838
Sigmoid	Test loss = 2.301 Test accuracy = 0.114



With 5 layers:

The models with the ReLU and the Tanh activation functions in the hidden layers have almost identical accuracy. On the other side with the Sigmoid activation function we see that the accuracy is almost 0, so the model is not learning. This insight about the Sigmoid function is because its gradient at very high or low values (values of the preactivation function $w^{Tx}+b$) is almost 0 and gradient update would hardly produce a change in the weight and the bias. Also, the very small value of learning rate forces the product *learning rate * gradient w.r.t weight* to be very small so the update of the weight will be very small.

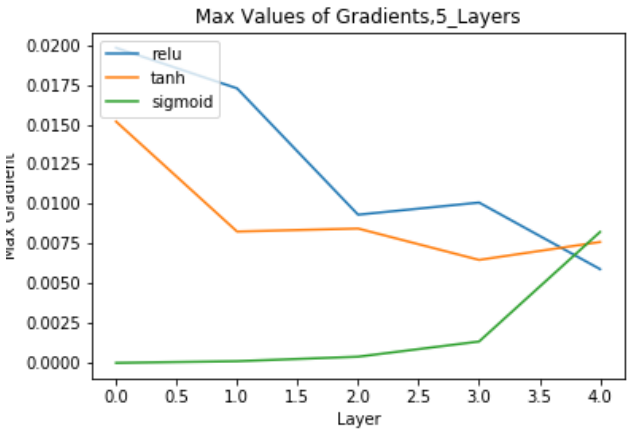
With 20 layers:

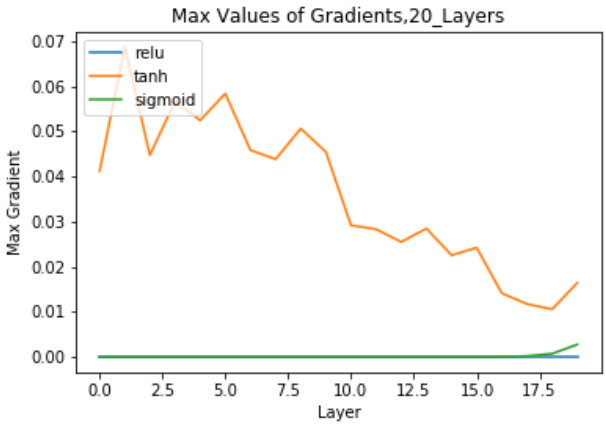
The Sigmoid activation function, except from the saturated neurons problem described above, suffers from the vanishing gradient problem wherein the gradients of the deeper neurons vanish. From the fact that the gradient of the Sigmoid function has a range of $(0, \frac{1}{4}]$ and that in the output layer neurons the expression of their gradients is a product of 19 sigmoid gradients belonging in the aforementioned range, we can conclude that their gradient values have been probably driven to zero. Regarding the Tanh activation function, considering that it is zero centered, it possibly makes it easier to model inputs that have strongly negative, neutral, and strongly positive values. Finally, we see a significant drop in the accuracy of the network with the ReLU function.

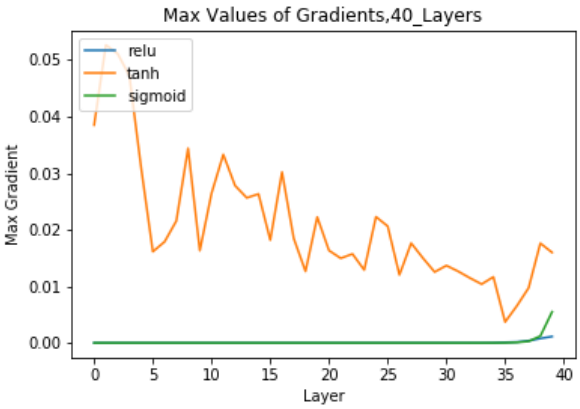
With 40 layers:

The network with the ReLU activation function in the hidden layer suffers from a dying ReLU problem. The bias term becomes too negative meaning that the gradient of the ReLU during backward pass is 0. Therefore, the weights and the bias causing the negative preactivations cannot be updated. If the weights and bias learn in such a way that the preactivation is negative for the entire domain of inputs, the neuron never learns, causing a sigmoid-like saturation. With the Tanh activation function and after some epochs the network is starting to perform in the same way as the network with fewer layers. This is probably because the Tanh is zero centered, so it allows the ideal gradient weight update where some weights increase while the other weights decrease, something that is not possible with ReLU. The network with the Sigmoid activation function is still unable to recover and has the same very bad accuracy.

Exercise 1.C







In the first case we see that in the network with the Sigmoid activation function the value of the max gradient is very close to 0, which confirms the insight mentioned above, meaning there is no change in weight and the bias. Referring to the gradients of the ReLU, along with the ones of the Tanh, we see a significant drop to the value of max gradient when moving deeper in the network. So, the deeper layers of the network learn very slowly or tend to not learn at all.

In the second and in the third plot we see that the gradients of the ReLU Function are 0 so the network does not update the weights and is not learning. With the ReLU function, when the gradient becomes 0, the network cannot perform back-propagation and cannot learn. Referring to the Tanh function we see that as we increase the number of layers we are getting closer to the vanishing gradient problem, as we see the same behavior as before, where the neurons in the deeper layers learn more slowly leading to the gradients to have values of zero (eventually after adding some more layers).

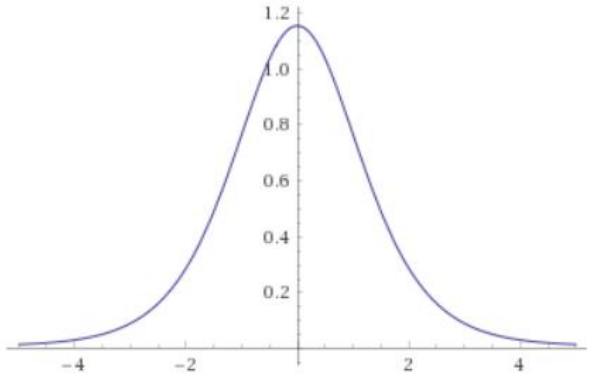
Exercise 1.D

LeCun:

$f(x) = 1.7159\tanh(\frac{2}{3}x) + 0.01x$

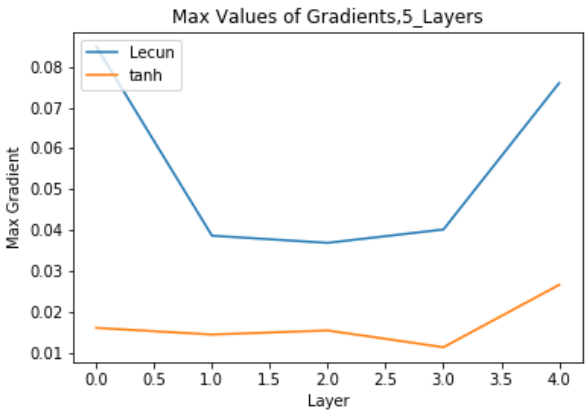
$f'(x) = 1.14393\,\mathrm{sech}^2(\frac{2}{3}x) + 0.01$

Gradient range is $\{y \in R: \frac{1}{100} < y \leq \frac{155393}{100000}\}$

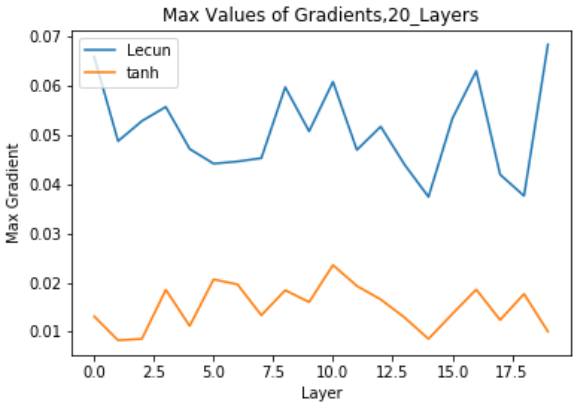


In the plots below we see the maximum values of the gradients of the weights for a network with the LeCun and Tanh activation functions in the hidden layers. Both networks are untrained, so the values of the weights and the biases come from the default Keras initializer glorot_uniform. If we compare the two activation functions, we see that LeCun is a modified Tanh where the constants have been chosen to keep the variance of the output close to 1, because the gain of the Sigmoid is roughly 1 over its useful range. Also, the derivative of the function has a similar behavior with the gradient of the Tanh, something that is shown in the plots. As we move backwards with back-propagation, we compare the max values of the gradients of the weighs (of the 2 functions) and see that the gradients of the weights with LeCun function are bigger in comparison with those of the Tanh in each layer. Moreover, as we increase the number of layers, we see that the Tanh gradients tend to get close to 0 as we move backwards to the hidden layer, in contrast with the LeCun function, where the accuracy of the network tends to be stable across all the different networks. This means that the weights and the biases get updated and therefore the error propagating backwards improves the performance of the network.

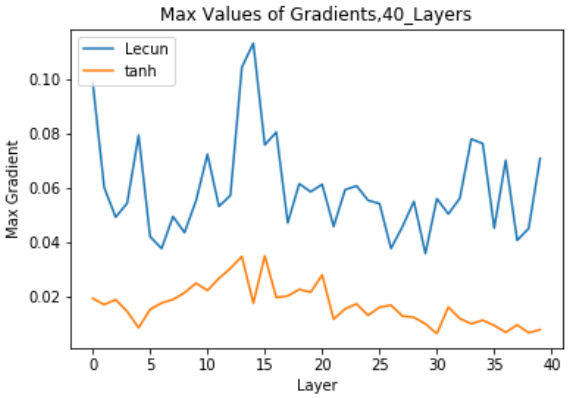
With 5 layers	
ReLU	Test loss = 2.664 Test accuracy = 0.129
Tanh	Test loss = 2.367 Test accuracy = 0.104



With 20 layers	
ReLU	Test loss = 2.654 Test accuracy = 0.118
Tanh	Test loss = 2.32 Test accuracy = 0.103



With 40 layers	
ReLU	Test loss = 2.699 Test accuracy = 0.104
Tanh	Test loss = 2.363 Test accuracy = 0.044

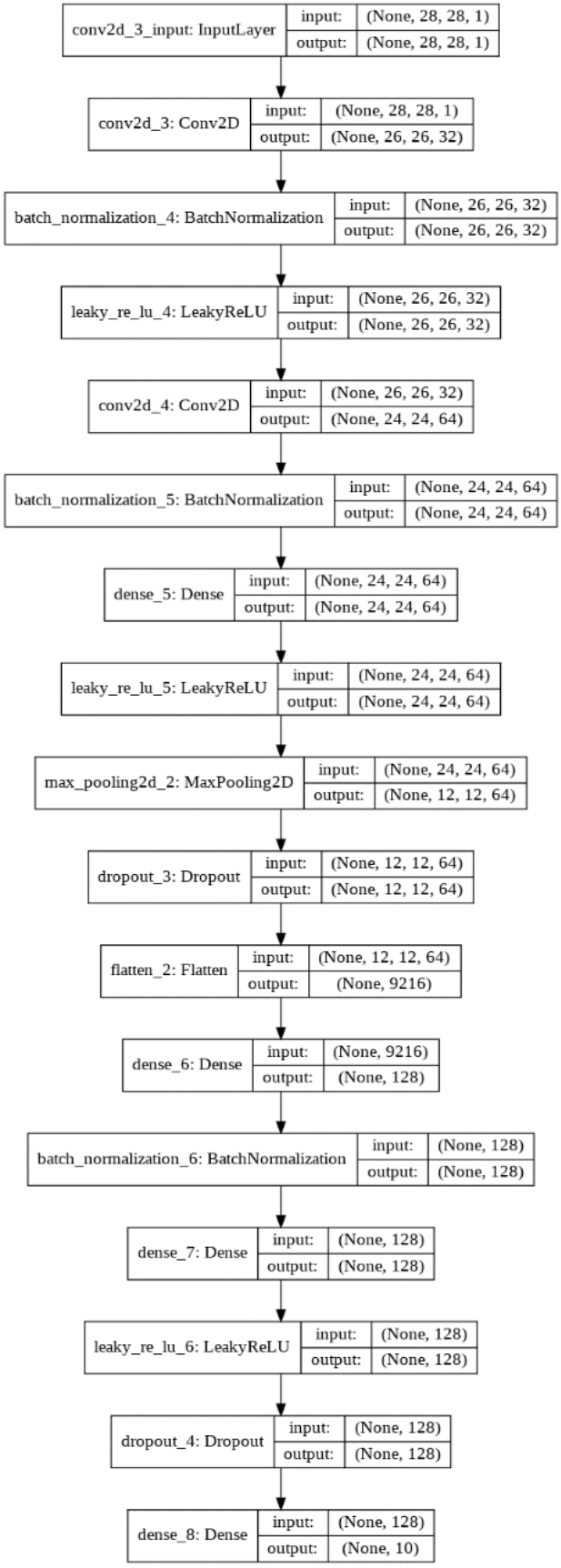


Exercise 3

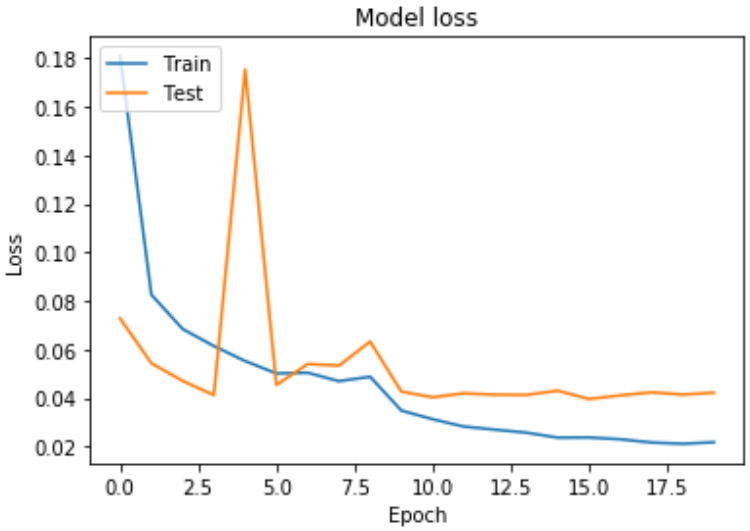
In the network below we try to adopt all the knowledge gained from the lectures and after many tries with many combinations of the parameters and type of layers, we get the following result:

In our network we use an l2 Regularizer instance representing the sum of the squared weights. We want to cut the weights by very small values by modifying the cost function (Weight Decay). We continue by preprocessing the MNIST dataset and we choose the following architecture for the network. We start with a convolutional layer with 32 filters and a 3x3 kernel with the l2 Regularizer applied. Then we apply batch normalization in order to normalize the data in each mini-batch (thus avoid getting too big or too small). We use leaky ReLU as an activation function in order to avoid the dying ReLU problem (We could have chosen the classic ReLU as we have our modified cost function thus we eliminated the useless weights and with the batch normalization the input of activation function is normalized by means of bias and variance so ReLU will converge but the network with leaky ReLU has a slightly better accuracy). Then we use a second convolutional layer, this time with 64 filters and a 3x3 kernel, which will create 64 different representations of our input of the previous layers. We use batch normalization again and then a fully connected layer of 64 nodes and a linear activation function. We want our neural network to learn directly from the data, since this is something that works for real world data, like our images. As we have an image to learn to our neural network and a simple linear regression model is working fine thus it has limited learning power. Then we use leaky ReLU in order to enable back propagation even for the negative input values and apply max pooling with 2x2 window. After that, we introduce dropout with a rate of 0.25 in order to cut the training at a point where the network starts to overfit the data and then we flatten the output. After several runs and experiments, we chose a fully connected network with 128 neurons and the abovementioned kernel Regularizer. This would be our final step until we realized that by adding another one linear fully connected layer along with a dropout with a rate of 0.5 give a little better accuracy. At last we have the output layer with 10 neurons (one hot encoding) each one representing one number [0,9]. The accuracy of the network in the test set 99.1% and the architecture is shown below along with the curves is shown below.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
batch_normalization_1 (Batch (None, 26, 26, 32)        128
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 26, 26, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 24, 24, 64)        18496
_________________________________________________________________
batch_normalization_2 (Batch (None, 24, 24, 64)        256
_________________________________________________________________
dense_1 (Dense)              (None, 24, 24, 64)        4160
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 24, 24, 64)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 12, 12, 64)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 12, 12, 64)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 9216)              0
_________________________________________________________________
dense_2 (Dense)              (None, 128)               1179776
_________________________________________________________________
batch_normalization_3 (Batch (None, 128)               512
_________________________________________________________________
dense_3 (Dense)              (None, 128)               16512
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 128)               0
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_4 (Dense)              (None, 10)                1290
=================================================================
Total params: 1,221,450
Trainable params: 1,221,002
Non-trainable params: 448
_________________________________________________________________
```


conv2d_3_input: InputLayer
input: (None, 28, 28, 1)
output: (None, 28, 28, 1)
conv2d_3: Conv2D
input: (None, 28, 28, 1)
output: (None, 26, 26, 32)
batch_normalization_4: BatchNormalization
input: (None, 26, 26, 32)
output: (None, 26, 26, 32)
leaky_re_lu_4: LeakyReLU
input: (None, 26, 26, 32)
output: (None, 26, 26, 32)
conv2d_4: Conv2D
input: (None, 26, 26, 32)
output: (None, 24, 24, 64)
batch_normalization_5: BatchNormalization
input: (None, 24, 24, 64)
output: (None, 24, 24, 64)
dense_5: Dense
input: (None, 24, 24, 64)
output: (None, 24, 24, 64)
leaky_re_lu_5: LeakyReLU
input: (None, 24, 24, 64)
output: (None, 24, 24, 64)
max_pooling2d_2: MaxPooling2D
input: (None, 24, 24, 64)
output: (None, 12, 12, 64)
dropout_3: Dropout
input: (None, 12, 12, 64)
output: (None, 12, 12, 64)
flatten_2: Flatten
input: (None, 12, 12, 64)
output: (None, 9216)
dense_6: Dense
input: (None, 9216)
output: (None, 128)
batch_normalization_6: BatchNormalization
input: (None, 128)
output: (None, 128)
dense_7: Dense
input: (None, 128)
output: (None, 128)
leaky_re_lu_6: LeakyReLU
input: (None, 128)
output: (None, 128)
dropout_4: Dropout
input: (None, 128)
output: (None, 128)
dense_8: Dense
input: (None, 128)
output: (None, 10)

Test loss = 0.04217260131984949



Test accuracy = 0.9916