Neural Networks

Elmar Hajizada

1 Task 1

The network given in the original task has 6 layers, with 28x28, 784, 125, 100, 50 and 10 neurons, respectively. The second layer just flattens the input so it does not have any parameters attached to it. As for the other layers, for each layer l with s_l neurons, the total amount of parameters is $(s_{l-1}+1)*s_l$. The $+1$ in the parenthesis comes from the number of biases and the rest are the weights going from layer $l-1$ to l. Using this, we get that there are $785*125+126*100+101*50+51*10 = 116285$ parameters in the network.

After we determined the parameters, we created a function to run a neural network with varied learning rates, models and epochs.

```
def neuralNetwork(model,rate,epochs):
      model.compile(loss=keras.losses.categorical_crossentropy,
                    optimizer=keras.optimizers.SGD(lr = rate),
                    metrics=['accuracy'])
      fit_info = model.fit(x_train, y_train,
                batch_size=batch_size,
                epochs=epochs,
                verbose=1,
                validation_data=(x_test, y_test))
      score = model.evaluate(x_test, y_test, verbose=0)
      print('Test loss: {}, Test accuracy {}'.format(score[0], score[1]))
      return score, fit_info
```

By running this for our given model with learning rate 0.1 for 10 epochs and extracting the training and validation accuracies with the code:

```
#Find the accuracy of the training and validation set
trainAccuracy = fit_info.history['accuracy']
validAccuracy = fit_info.history['val_accuracy']
```

we get the accuracies shown in figure 1.
From this, we can see that while the validation set starts higher than the training set, the training set eventually increases past the validation set. We also see that the training set always increases, the validation set starts to increase before being more stable and even decreases near the end. This could be because of the network eventually overfitting to suit the training set.

2 Task 2

In order to create a network with a single hidden layer, we essentially take our given network and remove all layers except the flattening network and the output network and add our own dense layer. By running this network on the MNIST Data set with 30 epochs and the learning rate 0.1, we get the training and validation accuracies shown in figure 2

Here we see a very similar result as before for both the training and validation sets, with the only difference being that the learning is slower than last time. This is likely due to the neural network not being as deep as before.

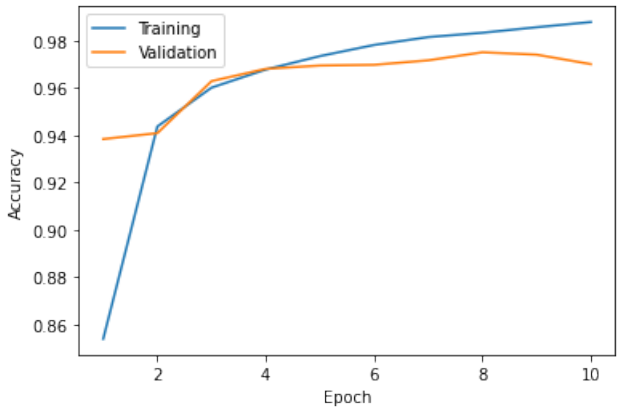

Figure 1: Plot with the accuracy of both the training and validation set over the epochs for the initially given network

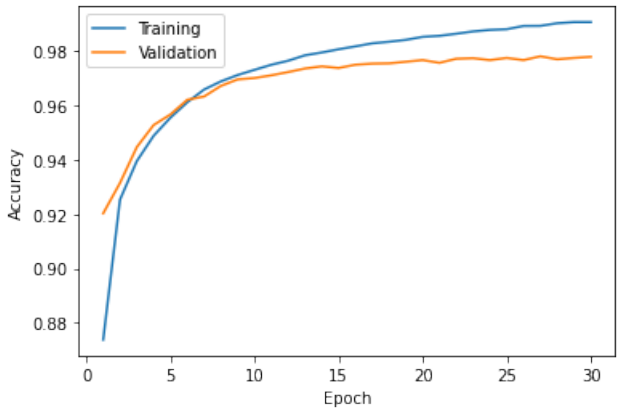

Figure 2: Plot with the accuracy of both the training and validation set over the epochs for the network with one hidden layer

In the next task, we tried our network with several learning rates, ranges from 0.001 to 1 and analyzed which learning rate was optimal for the network with 100 neurons in hidden layer. For each learning rate we repeated training process 3 times and calculated average test result for each.

```
def question_2b():
    lr_rates = [0.001, 0.01, 0.1, 1]
    avg_scores = []
    for lr_rate in lr_rates:
        score = 0
        for i in range(3):
            score += train_model(rate=lr_rate)[1]
        avg_scores.append(score/3)
    return avg_scores
```

Using this function we calculated average accuracy based on learning rates. The results that we got, has been represented in the following table:

Learning Rates and Accuracy			
lr = 0.001	lr = 0.01	lr = 0.1	lr = 1
0.881	0.936	0.973	0.967

As you can see from this table, optimal learning rate is 0.1, since the accuracy is the highest in that learning rate.

In task 2c, we were required to show the network performance based on the different learning rates, range from 0.001 to 0.1 and various number of neurons in hidden layer.
We first used different combinations of learning rates with number of neurons, to train the model for 10 epochs. Then for the optimal learning rate and number of neurons we trained model for 30 epochs and plotted train and validation accuracy over epochs.

```
def question_2c():
    dc = dict()
    nb_neurons = [10, 50, 100, 500, 1000]
    lr_rates = [0.001, 0.01, 0.1]

    for neurons in nb_neurons:
        for rate in lr_rates:
            dc[neurons, rate] = 0
    for neuron, lr_rate in dc.keys():
        dc[neuron, lr_rate] = train_model(nb_neurons=neuron, rate=lr_rate)[1]
    # sort dictionary according to the accuracy
    dc_sorted = {k: v for k, v in sorted(dc.items(), key=lambda item: item[1])}
    # take optimal nb_neurons and lr rate
    (opt_neurons, opt_rate), score = list(dc_sorted.items())[-1]
    plot_chart(dc_sorted)

    return (opt_neurons, opt_rate), score
```

After training phase we plotted a bar chart to show the network performance with respect to learning rates and number of neurons in hidden layer.

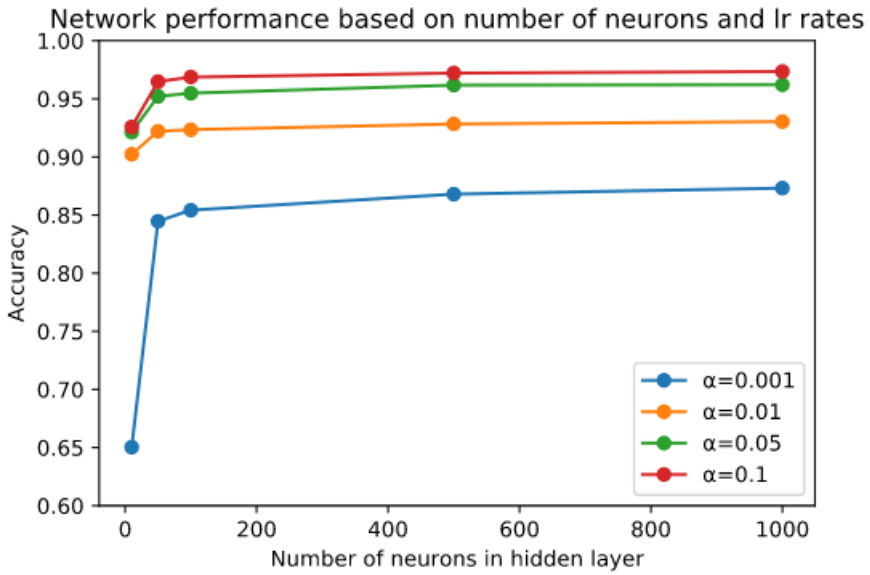

Figure 3: Network performance with respect to learning rates and number of neurons in hidden layer.

As you can see from Figure 3 above, optimal learning rate and number of neurons for this model is 1000 and 0.1, respectively. After training the model for 30 epochs with optimal parameters, we have reached about 0.9831 accuracy in validation set.

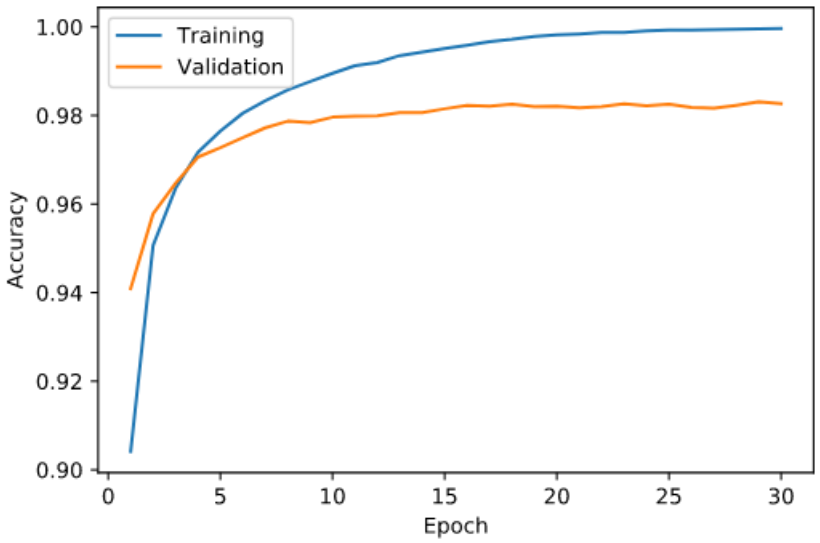

Figure 4: Training model for 30 epoch with optimal parameters.

3 Task 3

In this task we added Gaussian Noise to the training process in each batch. We tried various standard deviations such as 0.1, 1 and 10. As a sample example, we can show the following code where we added gaussian noise to our network.

```
model = Sequential()
model.add(Flatten())
model.add(GaussianNoise(stddev=0.1))
model.add(Dense(100, activation = 'relu'))
model.add(Dense(10, activation='softmax'))
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.SGD(lr = rate),
              metrics=['accuracy'])
```

After adding Gaussian Noise to our network we trained for 30 epochs with 3 different standard deviations.

Gaussian noise with standard deviations and Accuracy		
std = 0.1	std = 1	std = 10
0.9796	0.9566	0.0979

As you can see from table above, the best standard deviation for Gaussian Noise is 0.1. We got really poor result when standard deviation is 10. That happens because we add noise with high variation which actually changes distribution of the data a lot. That‘s why model does not learn much things during the training process.

Adding a noise could make vast improvements when we have over-fitting situation. Where the validation accuracy is very low compared to training one. Means model has learned training too hard but can not perform well in validation set. If we have small dataset, it can be usual to have over-fitting on our model. That‘s why Gaussian Noise could be added to improve randomness in the data and change the distribution of data in each batch so that model can not over-fit.

There is also another regularization technique which is L2 regularization. We wish to minimize cost function. So using L2 norm we add a component that will penalize large weights. By adding square norm of weight matrix and multiply it by regularization parameter, large weights will go down but small weights will be less affected. We trained our network with 3 different values for regularization parameter: 0.001, 0.01, 0.1. As a sample example we can show the following code where we added L2 regularization for our network.

```
model = Sequential()
model.add(Flatten())
model.add(Dense(nb_neurons, activation = 'relu', kernel_regularizer=regularizers.l2(0.001), ↩
    bias_regularizer=regularizers.l2(0.001)))
model.add(Dense(10, activation='softmax'))
model.compile(loss=keras.losses.categorical_crossentropy,
            optimizer=keras.optimizers.SGD(lr = rate),
            metrics=['accuracy'])
```

L2 regularization with penalty parameter and Accuracy		
alpha = 0.001	alpha = 0.01	alpha = 0.1
0.9763	0.9641	0.6707

After adding L2 regularization to our network, the highest result that we got was 0.9763 when penalty parameter is 0.001. We got poor result using penalty 0.1, because we penalize large weights too much, preventing them to be reach near optimal weights. However we did not get vast improvements on the validation set but we got much generalized result using L2 regularization.

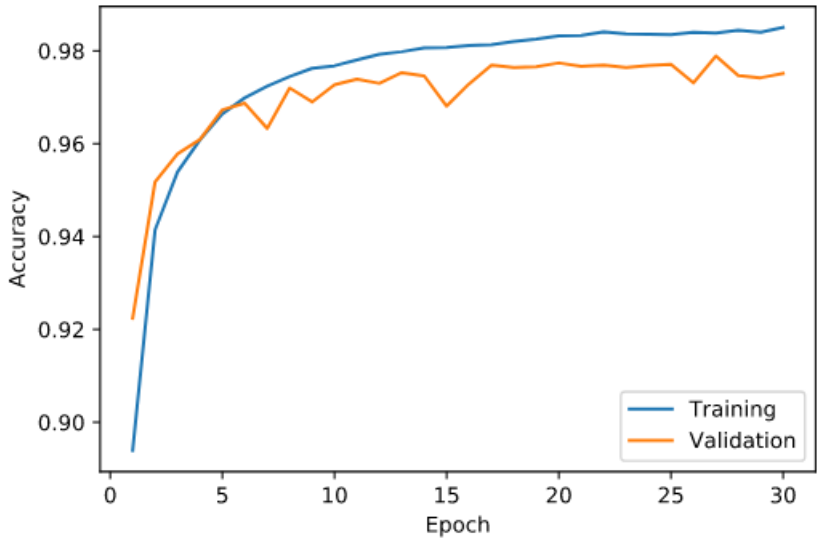

Figure 5: Network performance after adding L2 regularization with penalty parameter 0.001

As you can see now the gap between training and validation accuracy has been diminished and we have more generalized network.

4 Task 4

Now, we will try to create a good neural network using convolutional layers. To do this, we will add a convolutional layer before the flattening layer with 20 filters and a receptive field of 3x3, with padding to keep the image 28x28 and with a RELU activation function. This means that we eventually get 20 28x28

images. After this, we will use a Max-Pooling layer. The Max-Pooling layer goes through the images from the convolutional layer in a 2x2 grid and only picks the maximum in each grid. It's stride is 2x2 as well, so in the end it turns each of the convolutional layers into a 14x14 image. Then, we use another convolutional layer and MaxPooling layer to get 400 7x7 images. This input is then flattened and put through the same layers as in task 2.

```
model = Sequential()
model.add(Conv2D(20,3,padding='same',activation= 'relu'))
model.add(MaxPooling2D())
model.add(Conv2D(20,3,padding='same',activation= 'relu'))
model.add(MaxPooling2D())
model.add(Flatten())
model.add(Dense(100, activation = 'relu'))
model.add(Dense(num_classes, activation='softmax'))
```

By running this model for 30 epochs with the learning rate 0.1, we get the validation accuracy seen in figure 6, where we see that the accuracy eventually exceeds 99%.

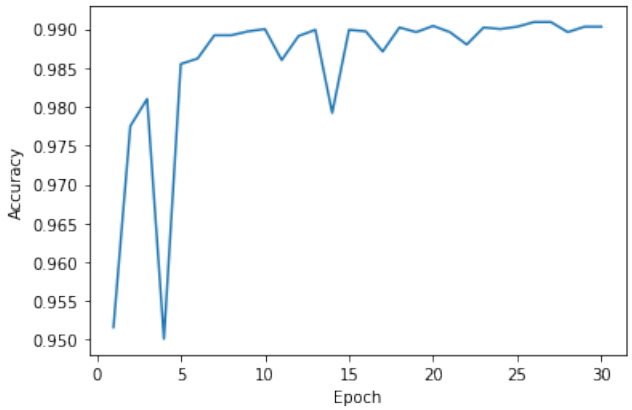

Figure 6: Plot of the validation accuracy over each epoch for our convolutional network

There are two reasons why we would want to use convolutional layers over dense ones. First, it uses fewer weights than the dense layers since it only uses as many weights as the receptive field and number of filters instead of as many weights as the previous layer times the number of weights of the layer, meaning that it is easier to update and improves performance. Secondly, it picks up some local effects in a translation invariant way. You can use it essentially to detect where a corner or the side of an object is in an image regardless of where the object is in the image and thus know where to start recognising the object. For a dense layer, it would not see this and have a much harder time seeing two translated images as the same image.