Mike Leske

R00183658

COMP9074 – Deep Learning

Assignment 1 – TensorFlow 2

1 Part A – TensorFlow and the Low Level API

1.1 Q1 (i)

Below the following conventions apply:

m = number of instances

n = number of features

c = number of classes

forward_pass function

```
@tf.function
def forward_pass(x, w, b):
    logits = tf.add(tf.matmul(tf.cast(x, dtype=tf.float32), tf.cast(w, dtype=tf.float32)), b)
    softmax = tf.exp(logits) / tf.reduce_sum(tf.exp(logits), axis=-1, keepdims=True)
    return softmax
```

The responsibility of the forward_pass function includes 2 major operations:

1. Calculate the logits
2. Perform Softmax operation on logits

The logits are represented by a matrix multiplication between the feature input vector x [m, n] and the weight w [n, c], plus the addition of the bias [1, 10]. For this operation the TensorFlow functions tf.matmul() and tf.add() are used. The resulting operation is a [m, c] vector, i.e. 10 class probabilities for each instance.

Softmax calculation:

$$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$$

The softmax operation is a commonly used final activation function for multi-class classification problems. It forces all predictions to sum to 1 and hence its output represents a distribution function across all possible classes. For each instance we normalize the class predictions by raising e to the power of the logits (`tf.exp(logits)`) and dividing by the sum of raising e to the power of the logits for this instance (`tf.reduce_sum(tf.exp(logits), axis=-1, keepdims=True)`). Here, tf.reduce_sum() represents the summation, joined by parameters to instruct the operation in which dimension to summarize.

The resulting matrix then provides class probabilities which sum to 1 per instance.

cross_entropy function

```
@tf.function
def cross_entropy(y_pred, y):
    y_pred = tf.transpose(y_pred)
    ce = -tf.reduce_sum(y * tf.math.log(y_pred), axis=0)
    loss = tf.reduce_mean(ce)
    return loss
```

The cross-entropy function is used to calculate the loss between the forward_pass softmax predictions and the true class labels. Cross-Entropy is the negative summarization of the multiplication of the true class labels times the log of the class predictions. Here, the * operation represents the tf.multiply() operation, i.e. element-wise multiplication.

Per instance cross-entropy:

$$\mathrm{L}(p^i, y^i) = -\sum_{\mathrm{j}=1}^{\mathrm{c}} y_j^i \log(p_j^i)$$

Because the true class label only has a single class set to 1 per instance, only the corresponding prediction for that class will be carried on. All other predictions per instance will be multiplied by 0. The tf.reduce_sum(…, axis=0) operation ensures we have a single loss per instance. Here, ce represent a [m,] row vector holding the per-instance loss.

Lastly, the total loss calculates the average loss across all instance losses.

calculate_accuracy function

```
@tf.function
def calculate_accuracy(y_pred, y):
    predictions = tf.transpose(y_pred)
    predictions_correct = tf.cast(tf.equal(tf.argmax(predictions, axis=0),
                                           tf.argmax(y, axis=0)), tf.float32)
    accuracy = tf.reduce_mean(predictions_correct)
    return accuracy
```

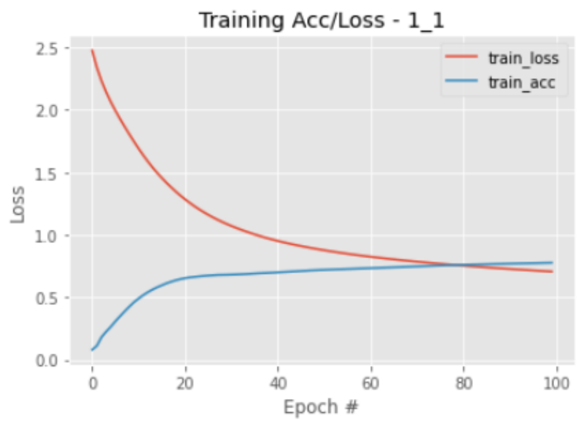
The accuracy calculation aims to report the percentage of instances correctly classified by the softmax forward_pass. For each instance the class with the highest probability is extracted using the tf.argmax() function. These values are compared by tf.equal(). This operation results in a True or False per instance. The tf.cast() function converts the Boolean statements to either 0.0 (False) or 1.0 (True). The overall result of this chain or operations is a [m,].

Lastly, the average of this vector is calculated resulting in the accuracy value across all instances.

Evaluation of the Softmax model

A single softmax layer alone is able to achieve an accuracy of ~77% on the 60000 training instances and ~ 76% of the 10000 test instances after 100 training epochs. For the size of this very small neural network this represents an impressive result.

```
The train accuracy is: 0.7762333
The train loss     is: 0.7061968
The test accuracy is: 0.7651
The test loss     is: 0.7227065
```



From this plot one sees that the accuracy quickly rises from ~10% to ~65% accuracy within the first 20 epochs. After that the accuracy curve starts to flatten out, although the network continues to learn. Similarly, the training loss curve start to flatten after approximately 40 epochs.

When comparing training loss and accuracy after 100 epochs with the test loss and accuracy, it becomes visible that no noticeable overfitting occurs within this small neural network. Both, loss and accuracy results for training and test data are relatively close together, indicating that more complex/larger models may be able to learn even more dependencies from the dataset.

1.2 Q1 (ii)

forward_pass function 1_2_1

```
@tf.function
def forward_pass(x, w1, b1, w2, b2):
    h1 = tf.add(tf.matmul(tf.cast(x, dtype=tf.float32), tf.cast(w1, dtype=tf.float32)), b1)
    a1 = tf.maximum(h1, 0)

    logits = tf.add(tf.matmul(tf.cast(a1, dtype=tf.float32), tf.cast(w2, dtype=tf.float32)), b2)
    softmax = tf.exp(logits) / tf.reduce_sum(tf.exp(logits), axis=-1, keepdims=True)

    return softmax
```

The forward_pass function has been modified such that the following operations occur:

1. Calculation of hidden layer activations
2. Softmax predictions

First, the hidden layer logits h1 are computed in exactly the same way as described in earlier for the Softmax layer. Next, a ReLU activation function is applied to the h1 logits. ReLU is a very simple activation function that only keeps positive logits. Negative logits will be set to zero. Hence, the ReLU function takes the maximum value of a zero or the logit.

Then the a1 activations are fed into the Softmax layer and processed as explained in Q1 (i).

One should note that with 2 layers (hidden layer, softmax layer) we have a set of 2 weight matrices and 2 bias vectors – whose sizes are related to the size of the hidden vector (see code).

forward_pass function 1_2_2

```
@tf.function
def forward_pass(x, w1, b1, w2, b2, w3, b3):
    h1 = tf.add(tf.matmul(tf.cast(x, dtype=tf.float32), tf.cast(w1, dtype=tf.float32)), b1)
    a1 = tf.maximum(h1, 0)

    h2 = tf.add(tf.matmul(tf.cast(a1, dtype=tf.float32), tf.cast(w2, dtype=tf.float32)), b2)
    a2 = tf.maximum(h2, 0)

    logits = tf.add(tf.matmul(tf.cast(a2, dtype=tf.float32), tf.cast(w3, dtype=tf.float32)), b3)
    softmax = tf.exp(logits) / tf.reduce_sum(tf.exp(logits), axis=-1, keepdims=True)

    return softmax
```

In this subtask the forward_pass functions compute softmax predictions after 2 hidden layers with ReLU activation. Hence, this function operates with 3 weight matrices and 3 bias vectors.

Network architecture evaluation:

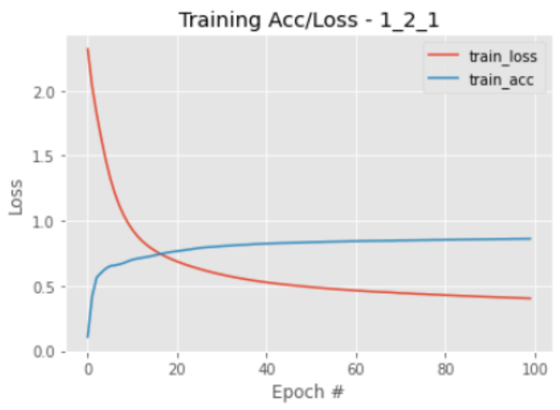
Training both neural network architectures it becomes immediately visible that the higher complexity allows the mode to learn more dependencies within the data and thus produce more accurate predictions. Compared to the Softmax-only network the accuracy after training the architectures for 100 epochs is ~9% higher on both training and test data.

Both network architectures show light forms of overfitting, as a gap between train and test accuracies and losses starts to appear. Especially, when a model performs better on training data than on the test data, overfitting is likely to occur.

It is also apparent, that both architectures – despite being different in number of layers and layer sizes – result in pretty comparable performance. This indicates that, although the additions of Dense layers helped the model to learn more relevant dependencies, the usage of Dense layers is not the best approach to tackle 2D image datasets. In contrast to Convolutional Neural Networks, networks with Dense layers only are not able to learn spatial dependencies between pixels due to the images' flattened representation. Hence, a neural network with Dense layers only will faster reach its maximum predictive capabilities.

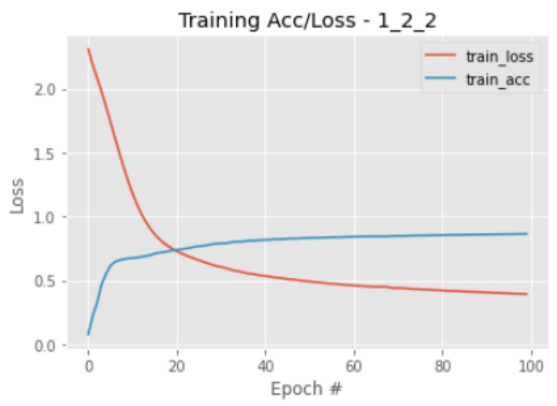
It also is interesting to notice that architectures A and B show an extremely steep improvement in the accuracy curve during the first approximately 5 epochs and then rapidly flatten out.

```
The train accuracy is: 0.8613667
The train loss     is: 0.40304086
The test accuracy is: 0.8469
The test loss     is: 0.43847206
```



Network architecture A – [300, 10]

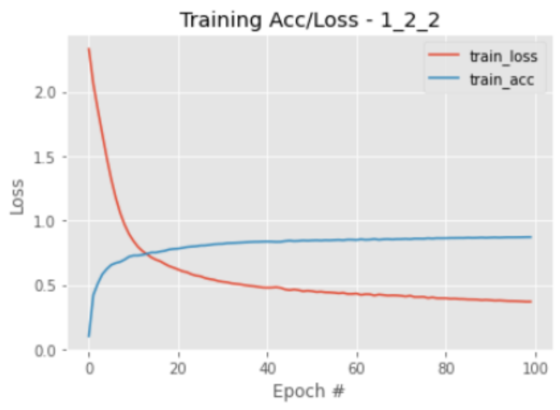
```
The train accuracy is: 0.86375
The train loss     is: 0.39269957
The test accuracy is: 0.85
The test loss     is: 0.42928022
```



Network architecture B – [300, 100, 10]

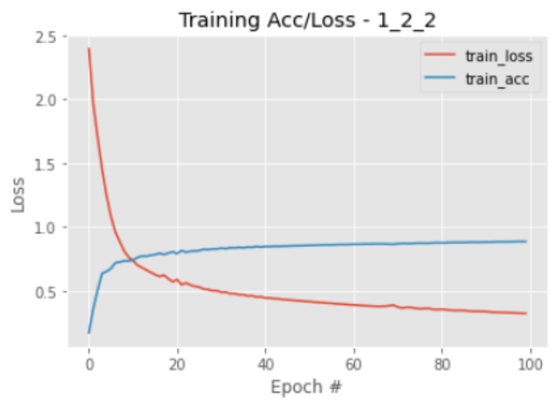
Further increasing any of the 2 layers only improves the networks' predictive capabilities by a small amount.

```
The train accuracy is: 0.87085
The train loss     is: 0.36930332
The test accuracy is: 0.8542
The test loss     is: 0.4106273
```



Network architecture C – [300, 300, 10]

```
The train accuracy is: 0.8857
The train loss     is: 0.32277155
The test accuracy is: 0.8671
The test loss     is: 0.37474087
```



Network architecture D – [600, 300, 10]

1.3 Q1 (iii)

According to common best practices the regularization term is calculated based on the weight matrices only and excludes the bias vectors. This implementation is in line with the TensorFlow L1 and L2 regularization implementation.

cross_entropy function 1_3 – L1 and L2 regularization

```
@tf.function
def cross_entropy(y_pred, y, weights=[], l1=None, l2=None):
    # Transpose y_pred to allow multiplication
    # Calculate Cross-Entropy Loss per instance
    # Calculate total loss
    y_pred = tf.transpose(y_pred)
    ce = -tf.reduce_sum(y * tf.math.log(y_pred), axis=0)
    loss = tf.reduce_mean(ce)

    # Implement regularization
    # Solution allows to have L1 and L2 regularization in parralel
    reg = 0
    if l1:
        # Implement L1 regularization
        #   1. Sum up the absolute weights of the weight matrices
        #   2. Multiply summed weight with L1 regularization scale parameter
        weights_sum = tf.reduce_sum([ tf.reduce_sum(tf.abs(w)) for w in weights ])
        reg += tf.multiply(tf.constant(l1, dtype=tf.float32), weights_sum)

    if l2:
        # Implement L2 regularization
        #   1. Sum up the squared weights of the weight matrices
        #   2. Multiply summed weight with L2 regularization scale parameter
        weights_sum = tf.reduce_sum([ tf.reduce_sum(tf.square(w)) for w in weights ])
        reg += tf.multiply(tf.constant(l2, dtype=tf.float32), weights_sum)

    # Create separate reg_loss term for later visualization
    reg_loss = loss + reg

    return loss, reg_loss
```

When L1 regularization is used, the scaled absolute sum of all weight matrix parameters is added to the training batch/iteration cross-entropy loss. The usage of L1 regularization pushes the weight values overall towards zero reducing complexity of the network. tf.abs() returns the specified matrix with absolute values, i.e. negative values are turned into positive values. Next, tf.reduce_sum() accumulates all values of a given matrix.

When L2 regularization is used, the scaled sum of all squared weight matrix parameters is added to the training batch/iteration cross-entropy loss. The usage of the square operation especially penalizes large weights and therefore incentivises the neural network to converge to smaller weight values in general. tf.square() returns the specified matrix with element-wise squared values. Next, tf.reduce_sum() accumulates all values of a given matrix.

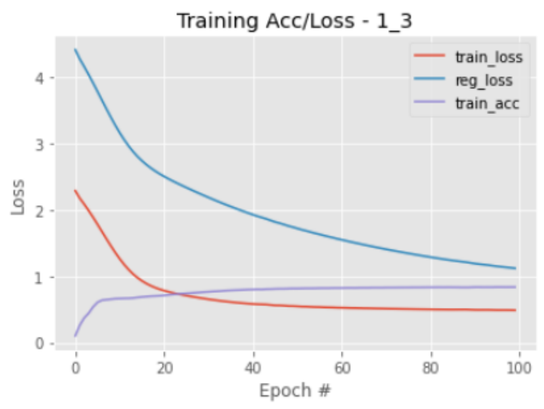
The above code uses list comprehension aggregate weights across matrices to avoid additive code clutter.

The original network architecture B was not overfitting too drastically. Hence, the impact of either L1 or L2 regularization is not visible immediately and rather modest.

The most obvious effect of the usage of either L1 or L2 regularization is the smaller gap between train and test accuracy and loss. Such a reduced gap indicates less overfitting during learning as the test results come much closer to the final train results.

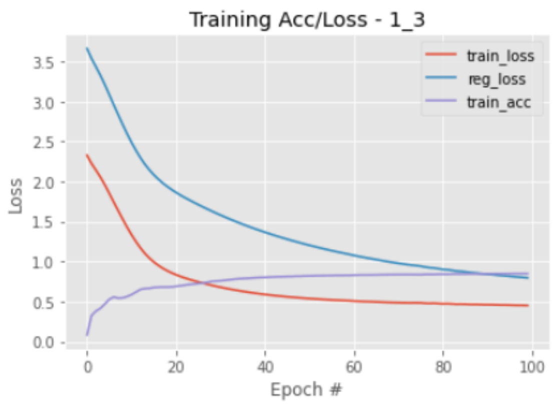
Due to the added regularization term to the overall cost, one notices higher loss values per epoch than without regularization. This not a bad thing, as it allows the neural network to learn for a much longer time span. For the diagram below in each epoch the cross-entropy loss and the cross-entropy + regularization loss was tracked. When the classical cross-entropy curve already converged to an almost flat line, the reg_loss curve still shows a significant slope indicating the model's ability to continue learning and reduce loss.

```
The train accuracy is: 0.8361667
The train loss     is: 1.1197656
The test accuracy is: 0.8267
The test loss     is: 1.1374371
```



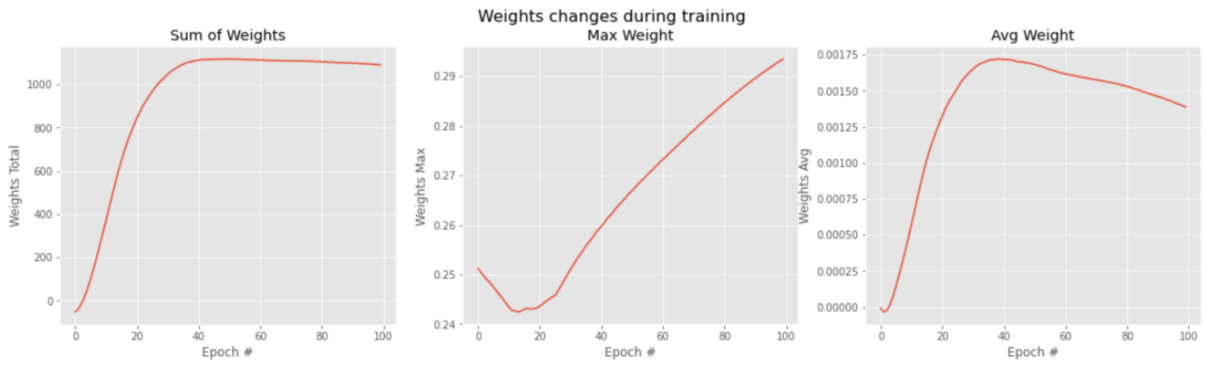
Network architecture B – L1 Reg (0.0002)

```
The train accuracy is: 0.84901667
The train loss     is: 0.79608595
The test accuracy is: 0.8368
The test loss     is: 0.8202187
```

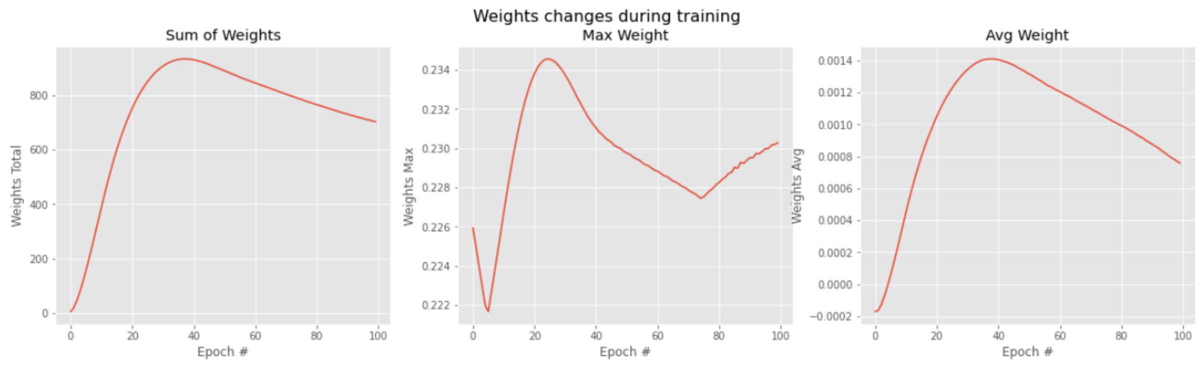


Network architecture B – L2 Reg (0.002)

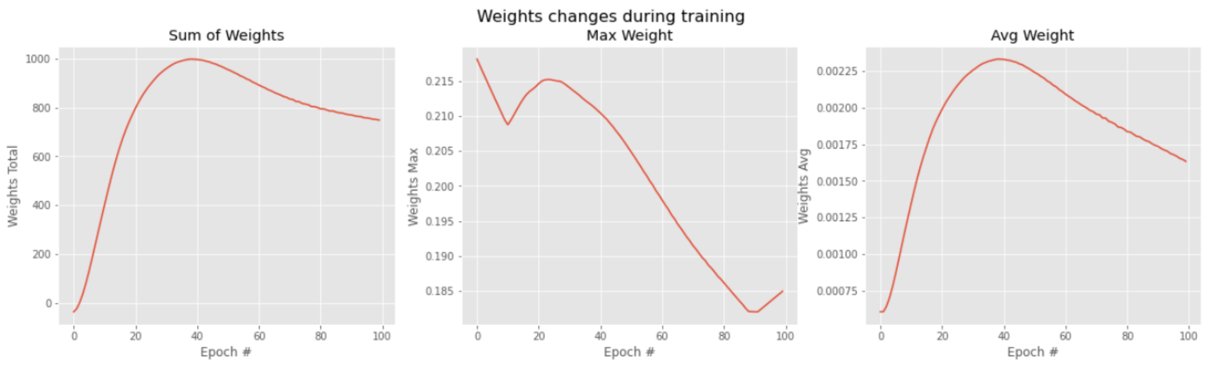
Finally, we can visualize the total sum of model weights, the maximum weight and the average weight over time (epochs). As the weight matrices were initialized using a normal distribution with a very small variance, it is expected that within the first few iterations the sum of weights increased as the neural network is learning. After 35-40 iteration once can nicely see the effect of regularization as in both the L1 and L2 regularization case the total sum of weights curve slope downwards again, while in the "no regularization" case the total sum of weights remains flat.



Weights metrics during training with no regularization



Weights metrics during training with L1 regularization



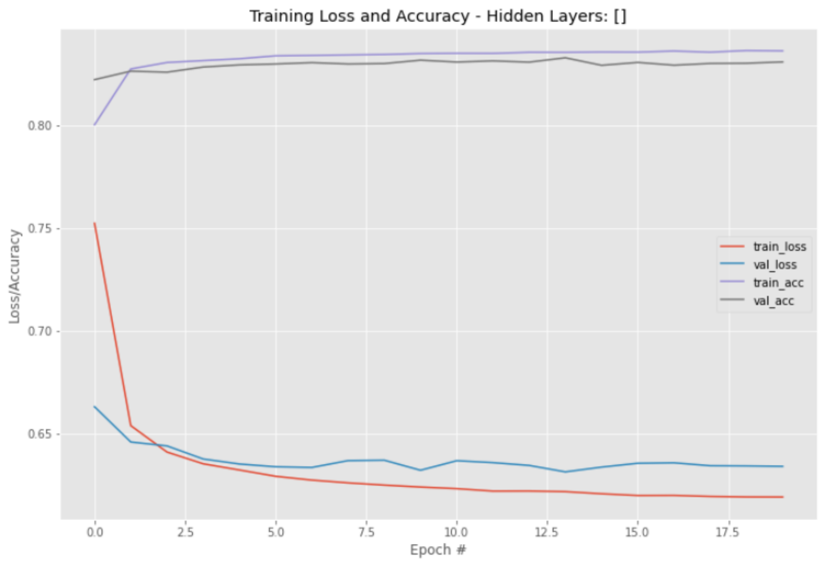
Weights metrics during training with L2 regularization

2 Part B – Keras - High Level API

2.1 Q2 (i)

The Softmax-only Neural Network achieves a test accuracy of ~85.6%, which again is a noticeable performance given the simplicity of the model of 7850 weights: 784x10 features weights + 10 bias weights. Both train_acc and val_acc run close to parallel. The gap between train_loss and val_loss is slightly bigger. Hence, this simple Softmax-only model can be considered to slightly overfit the data during the training stage.

```
Test accuracy: 0.8557646870613098
Test loss     : 0.5463404059410095
```

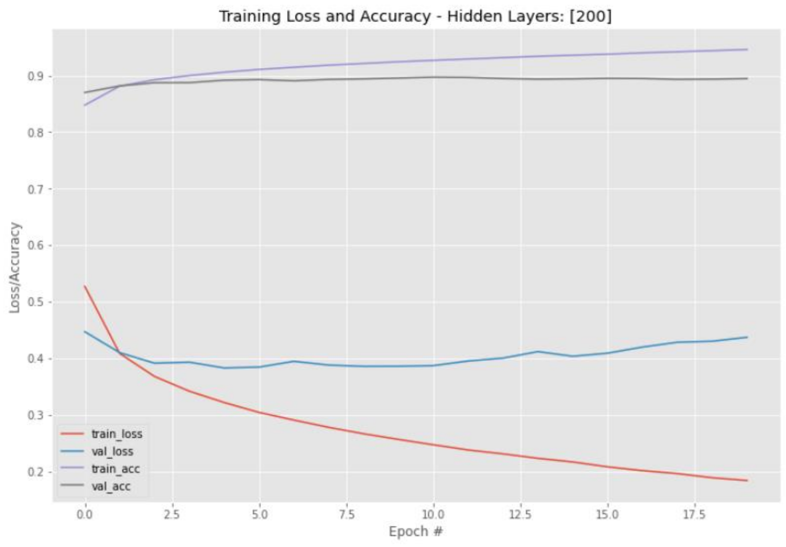


Next, larger neural network with the following architectures are trained:

- L1 200 + Softmax
- L1 400 + L2 200 + Softmax
- L1 600 + L2 400 + L3 200 + Softmax
- L1 800 + L2 600 + L3 400 + L4 200 + Softmax

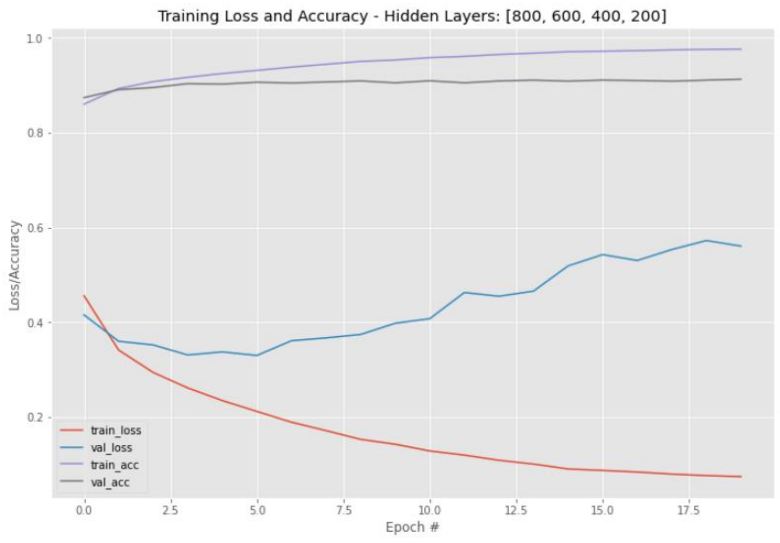
After adding a single Dense layer with 200 neurons, the test accuracy jumped up to ~91.6%, while the training set reached 94.6% accuracy after 20 epochs. This model however, already shows significant signs of overfitting. This behaviour is especially visible in the loss curves for the training and validation loss. After 2 epochs, these 2 curves start to diverge and continue to do so. While the loss of the training set continues to shirk, the validation loss remains flat from epoch 3 onwards and even starts to increase after 11 epochs.

```
Test accuracy: 0.915647029876709
Test loss    : 0.3435260057449341
```



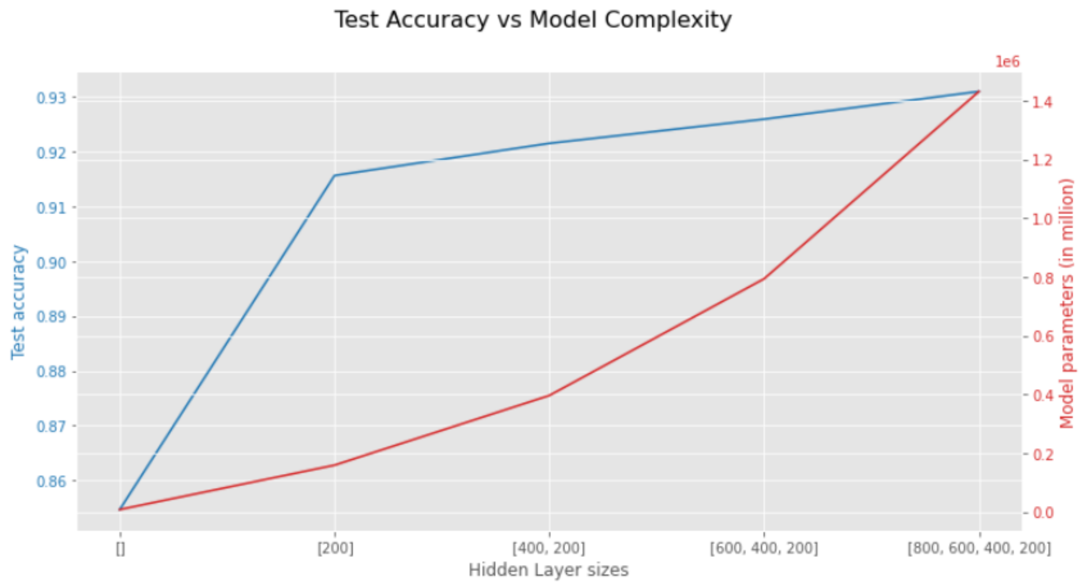
This overfitting behaviour is visible with all larger network models. For example, the largest neural network with hidden layer sizes 800, 600, 400 and 200 followed by a Softmax layer is capable of achieving 97.6% train accuracy and 93% test accuracy, but the divergence between train loss and validation loss is alarming.

```
Test accuracy: 0.9309999942779541
Test loss    : 0.43323734402656555
```



Other graphs from the IPython notebook are not repeated here, but the overall pattern remains the same.

Finally, one should also trade-off between a models' performance and its complexity/size. The different experiments showed, that the introduction of a single hidden layer gave a significant boost to the test accuracy compared to Softmax-only model. However, the largest model with 4 hidden layers (800, 600, 400, 200) "only" achieves ~1.5% higher accuracy, but comes with a 9x higher number of trainable parameters driving computational cost.



2.2 Q2 (ii)

Dropout is a regularization method for neural networks in the way that during every training batch / mini-batch a random number of neurons per layer is zeroed out. This forces the neurons on each layer to generalize their learning as neurons must not rely on the activity of any specific neuron in the previous layer. This learning behaviour in turn prevents neurons from giving very high weight to any neighbouring neuron, but rather weights are distributed across all incoming “connections”.

In this task Dropout will be applied to the 2 deepest networks from 2.1:

- L1 600 + L2 400 + L3 200 + Softmax
- L1 800 + L2 600 + L3 400 + L4 200 + Softmax

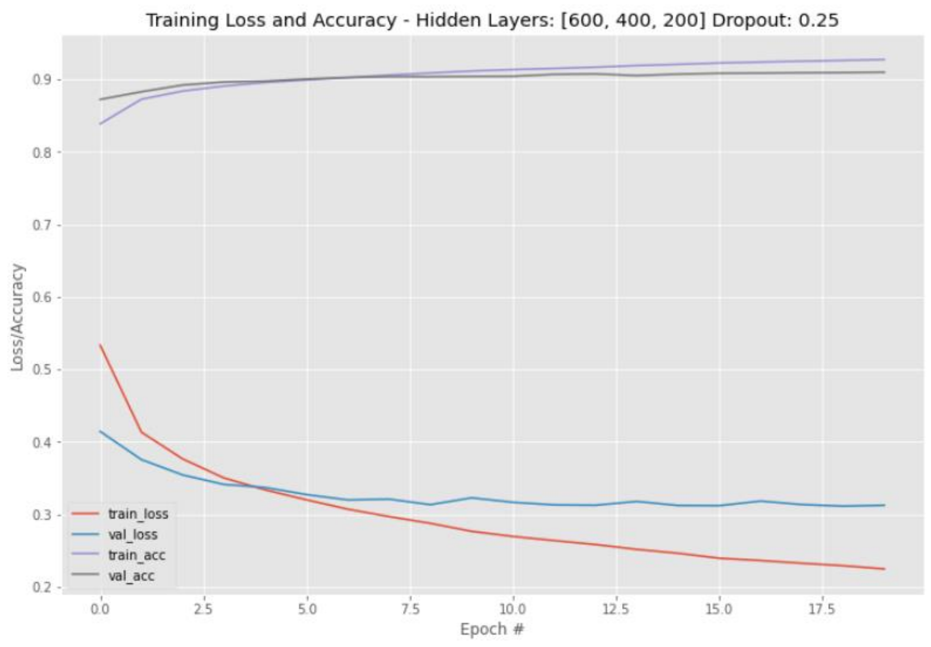
3 different dropout values will be evaluated:

- 0.1
- 0.25
- 0.5

For both networks the usage of 10% dropout nodes has no noticeable impact on the performance. Both, the accuracy and loss curve still diverged after a few epochs.

With the usage of 25% dropout nodes, overfitting still occurred, but for both networks the train and test accuracy curves ran close to parallel, while the loss curves were much closer than with no dropout or 10% only.

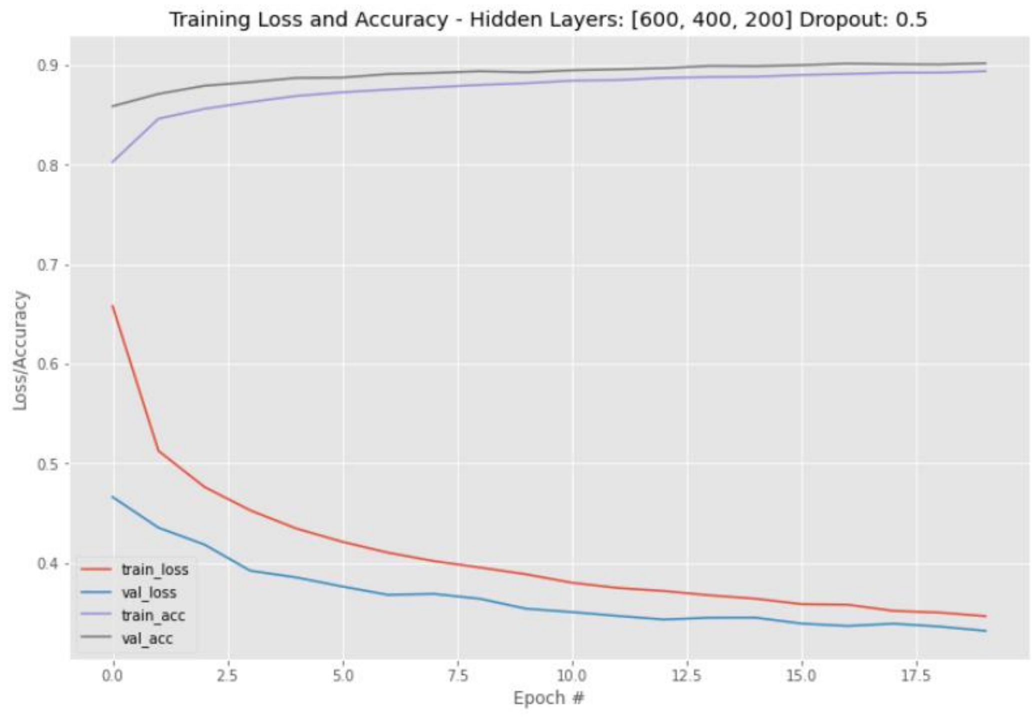
```
Test accuracy: 0.929058849811554
Test loss     : 0.24030464887619019
```



With a dropout rate of 50% overfitting could be prevented in both networks. Furthermore, now the networks were able to generalize better to the data so that throughout all 20 epochs the validation accuracy was slightly higher than the training accuracy, while the validation loss was lower than the training loss.

Ultimately, with 50% dropout rate the test accuracy resulted in a higher performance than seen on the training data set during any single epoch. This is expected to be caused by the fact, that dropout is used only during training stage, but the neural network can use all neurons when checking the performance of the validation and test data sets.

```
Test accuracy: 0.9224705696105957
Test loss     : 0.2642958164215088
```



Dropout adds complexity to each layer as an additional dropout matrix need to be randomly filled per mini-batch and element-wise multiplied with the input of the previous layer. For the given dataset and the rather small neural network the computational cost was not apparently visible.

Although Dropout can use distinct rates per layer, my code uses the same dropout rate across all layers as a hyper-parameter fine-tuning was out of scope for this assignment.

Graphs from [800, 600, 400, 200] network are excluded from this report, but can be found in IPython notebook PartB.ipynb.

3 Research

Batch Normalization

Normalizing the input data / features for classical Machine Learning algorithms like Linear Regression has shown to improve the time needed to learn the model's parameters. If the input data is not normalized (or in some cases standardized) elongated cost functions can exist, in which an optimization algorithm like Stochastic Gradient Descent may have difficulties in calculating the correct direction and strength of parameter change.

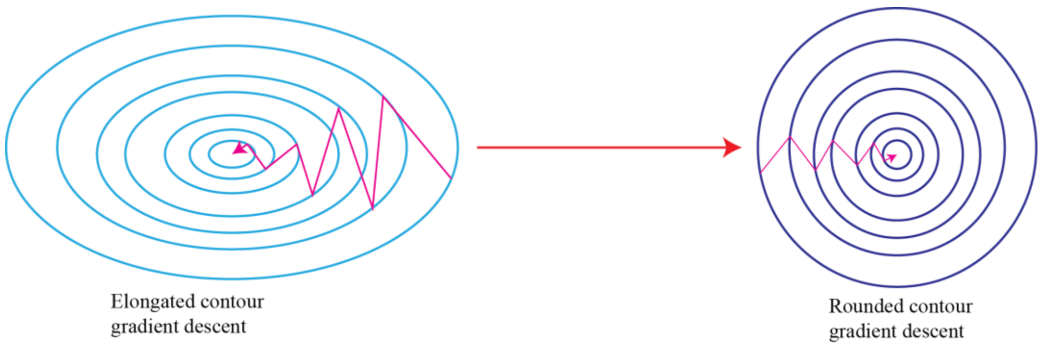


Figure 1 Picture taken from [1] – Elongated versus round cost function for normalized data

In 2015 Ioffe and Szegedy [2] presented a mechanism to normalize each hidden layer and hence improve a neural network's convergence time and performance by eliminating the so called covariate shift. While in deep neural networks only the first hidden layer can take advantage of a normalized input, subsequent hidden layers are presented with whatever is the outcome of the previous neural network layer. Batch Normalization allows each hidden layer of a neural network to be zero-centered and normalized. Such layer normalization also helps significantly reducing the occurrence of gradient vanishing and gradient explosion as a chain of very small or very large weights is very unlikely to occur.

[2] defines the following steps to implement Batch Normalization:

1. Calculate the batch mean $\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$
2. Calculate the batch variance $\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$
3. Normalize layer data $\widehat{x_i} \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \mathcal{E}}}$
4. Scale and shift layer data $y_i \leftarrow \gamma\widehat{x_i} + \beta$

While step 3 zero-centers and normalizes a layer's values, step 4 allows the neural network to learn 2 additional parameters per activation, β and γ, to scale and shift the normalization operations. By making β and γ learnable parameters, their gradients with respect to the loss function can be calculated, and appropriate parameter updates can be made by an optimizer during backpropagation.

It became common practice that Batch Normalization is implemented between a layer's logit computation (input * weights) and the layer's activation function.

$$\begin{tikzcd}
x \arrow[r, "{\omega^{[1]}, b^{[1]}}"] & z^{[1]} \arrow[r, "{\beta^{[1]}, \gamma^{[1]}}", "\text{Batch Normalization}"'] & z'^{[1]} \arrow[r, "\text{Activation function}"'] & a^{[1]}
\end{tikzcd}$$

Figure 2 Order of neural network operations with Batch Normalization (first layer)

By introducing Batch Normalization into each layer of the neural network, large changes in activations are limited to an individual layer only. The normalization ensures a "stable" activation level being fed forward to the next layer.

To work well with the commonly used practice of mini-batching during neural network training, the authors of [2] propose the usage of exponentially moving averages for μ and σ. The final moving averages after training the neural network can subsequently be used during the inference stage where usually individual samples are sent through the neural network instead of mini-batches. [3]

[1] https://udohsolomon.github.io/_posts/2017-06-21-understanding-batch-normalization/
[2] Sergey Ioffe, Christian Szegedy, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, https://arxiv.org/pdf/1502.03167.pdf
[3] Coursera Deep Learning by Andrew Ng, Course 2