Homework 2: Neural Networks with TensorFlow

Large-Scale Data Analysis 2020

Stefano Pellegrini (mlq211)

May 18, 2020

1 Standard Neural Network

1.1 Dense layer

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(64, activation="sigmoid",
                          input_shape=(1,),
                          name="hidden1"),
    tf.keras.layers.Dense(32, activation="sigmoid",
                          name="hidden2"),
    tf.keras.layers.Dense(1, activation="linear",
                         name="output")])
```

1.2 Functional model

```
inputs = tf.keras.Input(shape=(1,))
l1 = tf.keras.layers.Dense(64, activation="sigmoid",
                           name="hidden1")(inputs)
l2 = tf.keras.layers.Dense(32, activation="sigmoid",
                           name="hidden2")(l1)
output = tf.keras.layers.Dense(1, activation="linear",
                               name="output")(l2)

model = tf.keras.Model(inputs=inputs, outputs=output)
```

1.3 Shortcut connections

```
inputs = tf.keras.Input(shape=(1,))
l1 = tf.keras.layers.Dense(64, activation="sigmoid",
                           name="hidden1")(inputs)
l2 = tf.keras.layers.Dense(32, activation="sigmoid",
                           name="hidden2")(l1)
l3 = tf.keras.layers.concatenate([inputs, l2], name="shortcut")
output = tf.keras.layers.Dense(1, activation="linear",
                               name="output")(l3)

model = tf.keras.Model(inputs=inputs, outputs=output)
```

The total number of parameters is 2242. There are 128 trainable parameters in the first hidden layer, 2080 in the second hidden layer and 34 in the output layer. There are no non-trainable parameters.

2 Convolutional neural network for traffic sign recognition

2.1 Adding batch normalization

As shown in the code below, I added the batch normalizations before each exponential linear unit activation function. I used "\" to indicate that the code continue in the next line.

```
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, (5, 5), activation=None,
                           input_shape=(img_width_crop, img_height_crop,
                           no_channels),
                           bias_initializer=tf.initializers. \
                           TruncatedNormal(mean=sd_init, stddev=sd_init)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.ELU(),
    tf.keras.layers.MaxPooling2D(pool_size=(2,2)),
    tf.keras.layers.Conv2D(64, (5, 5), activation=None,
                           bias_initializer=tf.initializers. \
                           TruncatedNormal(mean=sd_init, stddev=sd_init)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.ELU(),
    tf.keras.layers.MaxPooling2D(pool_size=(2,2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(no_classes, activation='softmax')])
```

2.2 Trainable parameters

The 1-th layer is a *Convolutional* layer. The input to each neuron is a 4D array or tensor of shape (n, 28, 28, 3), where n is the batch size (number of images fed at each iteration), 28 is the width and height of the image, 3 is the number of channels. Each neuron in the layer performs the convolution operation of a small region of the input layer neurons (local receptive field), altogether they apply 32 convolution matrices, or masks, to the input image, generating a feature map for each filter. Each filter mask of shape $(5, 5)$ has 25 trainable weights for each of the 3 channel, plus 1 bias. The weights and biases are shared among the neurons. Thus, the overall number of trainable parameters is 2432, which is given by $32 \cdot 25 \cdot 3 + 32$. The output of the layer is a 4D array of shape (n, 24, 24, 32) where the dimensions respectively corresponds to: batch size, height of the feature map, width of the feature map, number of feature maps for each image).

The 2-th layer is a *Batch Normalization* layer. The input is a 4D array (n, 24, 24, 32), that corresponds to 32 feature maps of size 24x24, for each of the n images fed into the network. This layer normalize to zero mean and unit variance the input values (by mini-batch mean and variance non trainable parameters). Also, it scale and shift the normalized values by using 2 trainable parameters (γ, β) for each feature map received from the previous layer. Thus, the overall number of trainable parameters are 64, which is given by $2 \cdot 32$.

The 3-th layer simply performs the *Exponential Linear Unit* activation function. Each neuron apply the non linear function

$$\mathrm{ELU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(e^x - 1) & \text{if } x < 0 \end{cases} \tag{1}$$

to the input values, different to the Rectified Linear Unit, ELU has an additional alpha constant , but there are no trainable parameters.

The 4-th layer is a *Maximum Pooling* layer. The input of the layer is a 4D array of shape (n, 24, 24, 32), 32 24x24 feature maps for each of the n input images. The neurons in this layer perform down-sampling by dividing the input into rectangular pooling regions of size 2x2, and computing the average of each region. This operation reduce the dimensionality of the input and make the input representation invariant to translations. There are no trainable parameters.

The 5-th layer is a second *Convolutional* layer. The input is a 4D array of (n, 12, 12, 32), representing 32 pooled 12x12 feature maps for each of the n images. The neurons of this layer apply 64 masks of shape $(5, 5)$ to the 32 pooled feature maps, each filter mask has 25 trainable parameters (for each pooled feature map), plus 1 bias shared among the pooled feature maps. Thus, the overall number of trainable parameters is 51264, which is given by $64 \cdot 32 \cdot 25 + 64$.

The 6-th is a second *Batch Normalization* layer. The input is a 4D array of shape (n, 8, 8, 64), and the output shape is unaltered. The neurons of this layer normalize, scale and shift the input values. There are 2 trainable parameters (γ, β) and 2 non trainable parameter (mini-batch mean and variance), for each of the 64 feature maps received from the previous layer. Therefore, the total number of trainable parameters is 128.

The 7-th layer performs the *Rectified Linear Unit* activation function. The input is a 4D array of shape (n, 8, 8, 64), there are no trainable parameters and the output shape is unaltered.

The 8-th layer is a second *Maximum Pooling* layer. It takes as input a 4D array of shape (n, 8, 8, 64) and output a 4D array of shape (n, 4, 4, 64). As in the 4-th layer, there are no trainable parameters.

The 9-th layer is a *Flatten* layer. It takes as input a 4D array of shape (n, 4, 4, 64) and it simply flatten it into a 2D array of shape (n, 1024). There are no trainable parameters.

The 10-th or *output* layer is a *Dense* or *Fully Connected* layer. It takes as input a 2D array of shape (n, 1024) and output a 2D array of shape (n, 43) corresponding to n vectors of length 43, whose elements are the probabilities, for each image, of being a certain class. Each neuron receives input from all the neurons in the previous layer and computes the weighted sum

$$a_i = \sum_{j=1}^{d} w_{ij} x_j + w_{i0} \tag{2}$$

where w is the weight vector, x is the input vector of length d, and w_0 is the bias. Then, the *Softmax* activation function is applied to the result of the linear operation. There are 43 output neurons (one for each class), 1024 weights for each output neuron, and 43 biases. Thus, the overall number of parameters is 44075, which is given by $1024 \cdot 43 + 43$.

Batch normalization parameters: batch normalization is used to ensure that, for any parameters values, the network always produces activation with the desired distribution [2]. To obtain this result, each scalar feature is normalized independently to zero mean and one unit variance. To make sure that the transformation represent an identity transform, independently for each dimension, the trainable parameters γ and β are used to scale and shift the normalized values [2]. In the batch setting, for each dimension k, and omitting k for clarity, the batch normalization for m values is performed by the following:

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i \tag{3}$$

computes the mini-batch mean,

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \tag{4}$$

computes the mini-batch variance,

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \tag{5}$$

normalize the input (ϵ is a constant),

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \tag{6}$$

scale and shift the normalized values [2].

2.3 Data augmentation

Data augmentation is a powerful method that use invariance property to improve generalization performance of our model. Basically, we artifially increase the dataset by making new training samples based on the existing one.

In the template notebook are applied the following transformation: First the image is randomly cropped, so only a random sub-image of 28x28 pixels is selected from the original 32x32 image. Then, the image is flipped left to right, this transformation is conditioned on the label because it must be applied only to certain classes, in fact it will not be reasonable to flip left to right any traffic sign that contain numbers or digits, because the resulting image will not correspond anymore to the original label and it will be meaningful for the training process. An other transformation is the change in the brightness of the images, which makes them randomly lighter or darker. Finally, any value is clipped to 0 and 1, so all negative values are set to 0 and are values larger than 1 are set to 1.

Added transformation: first, I randomly changed the color saturation, then, I randomly changed the contrast of the image. I think that these are two reasonable augmentations, since the resulting images will be different from the original ones but, at the same time, they will continue to correspond to the original label.

```
def augment(image, label):
    image = tf.image.random_crop(image, [img_height_crop,
                                         img_width_crop, no_channels])
    image = tf.case([(tf.equal(label, 11), lambda: tf.image. \
                 random_flip_left_right(image)),
                 (tf.equal(label, 12), lambda: tf.image. \
                 random_flip_left_right(image)),
                 (tf.equal(label, 13), lambda: tf.image. \
                 random_flip_left_right(image)),
                 (tf.equal(label, 17), lambda: tf.image. \
                 random_flip_left_right(image)),
                 (tf.equal(label, 18), lambda: tf.image. \
                 random_flip_left_right(image)),
                 (tf.equal(label, 26), lambda: tf.image. \
                 random_flip_left_right(image)),
                 (tf.equal(label, 30), lambda: tf.image. \
                 random_flip_left_right(image)),
                 (tf.equal(label, 35), lambda: tf.image. \
                 random_flip_left_right(image))],
                 default = lambda: image)
    image = tf.image.random_brightness(image, max_delta=0.1)
    # Added colors saturation transformation
    image = tf.image.random_saturation(image, 0.5,2)
    # Added contrast transformation
    image = tf.image.random_contrast(image, 0.7, 1.3)
    image = tf.clip_by_value(image, 0., 1.)
    return image, label
```

3 Experimental architecture comparison

Dropout is a technique that aim to prevents overfitting by randomly removing units from the neural network during the training process, in this way it is capable to approximately combines different neural network architectures [3]. For this exercise I tested if adding dropout to a simple CNN model improves its performance on an image classification task.

The simple model that is used as baseline in this exercise is the one provided in the notebook template, which I will call $original - model$. This model has two convolutional layers, each followed by an exponential linear unit activation function and a maximum pooling layer. It follows a flatten layer and an output dense layer with a softmax activation function. I compared the performance of this model architecture with three other architectures, where one or more dropout layers are added at different positions. The addition of the dropout layers is the only difference between the models, they all use 800 epochs for the training process and the Adam optimizer with a learning rate of 0.001. The dataset used for evaluating their performance is the data from the German Traffic Sign Recognition Benchmark [4]. Due to the random nature of a neural network training process, a statistical evaluation should be required to establish if a modification improved a model's performance. But, as requested, for this exercise I only performed six independent trials for each architecture and I reported the mean. Also, for each architecture I show the training progress of the first trial, using both accuracy and loss metrics, on training and validation set (called test in the plots). The validation or test set, is only used to show how the model performance change during the training process, it wasn't used for any hyperparameter tuning.

I started by testing the performance of the $original - model$. In Figure-1 is possible to visualize the progress of the training process, the plot on the right shows clear signs of overfitting, in fact, after 50 epochs the validation error start to increase. The mean of the accuracy and the loss on the test set was respectively 0.9409 and 0.6107, while the mean of the accuracy and the loss on the training set was 0.9999 and 0.0015.

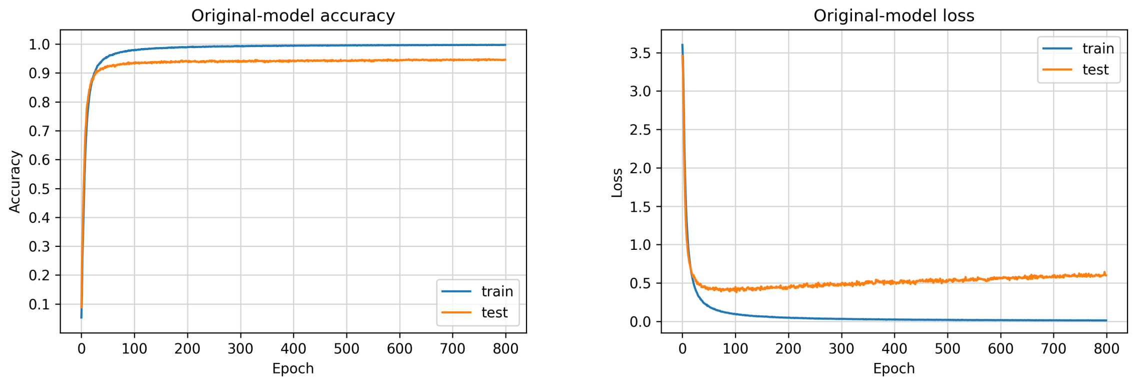

Figure 1: Original-model accuracy and loss during training.

In order to choose the position where I added the dropout layers, I simply tried different configurations and I chose the three architectures that shown the best performance. All dropout layers added use a drop rate of 0.5.

$Dropout1-model$: in the first modified model architecture I added two dropout layers, one after each maximum pooling layer. The code below shows the layer before and after each dropout layer, added to the original architecture. The "..." indicates that there are omitted lines of code.

```
...
tf.keras.layers.MaxPooling2D(pool_size=(2,2)),
tf.keras.layers.Dropout(0.5),
tf.keras.layers.Conv2D(64, (5, 5), activation=None,
                       bias_initializer=tf.initializers. \
                       TruncatedNormal(mean=sd_init, stddev=sd_init)),
...
tf.keras.layers.MaxPooling2D(pool_size=(2,2)),
tf.keras.layers.Dropout(0.5),
tf.keras.layers.Flatten(),
...
```

In Figure-2 is possible to see that this model does not show any sign of overfitting. Also, among all evaluated architectures, this model had the best performance on the test set, with an accuracy of 0.9764 and a loss of 0.0989. The mean of the accuracy and the loss on the training set was respectively 0.9995 and 0.0040.

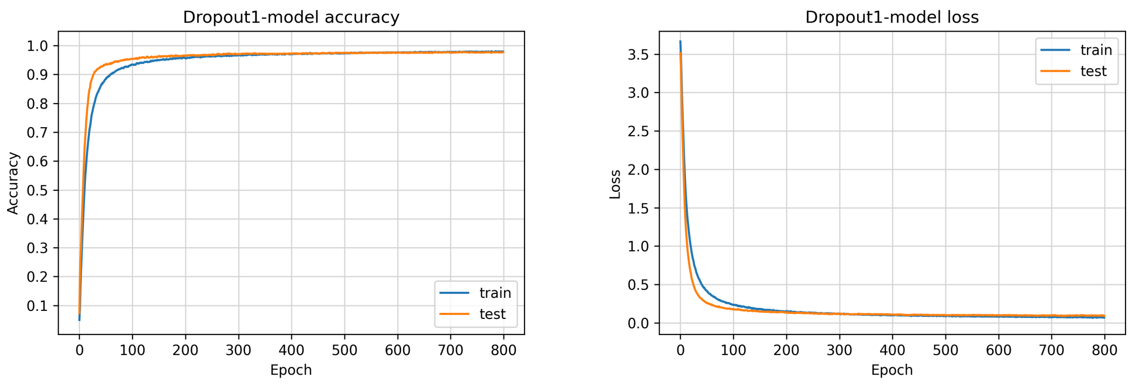

Figure 2: Dropout1-model accuracy and loss during training.

*Dropout*2 − *model* : the second architecture I evaluated has one dropout layer before the flatten layer. As in the previous model, I show the layer before and after the added dropout layer.

```
...
tf.keras.layers.Flatten(),
tf.keras.layers.Dropout(0.5),
tf.keras.layers.Dense(no_classes, activation='softmax')])
```

As we can see in Figure-3, again the dropout reduced overfitting and the performance is quite improved compared to the original-model. The test set mean accuracy and loss was respectively 0.9760 and 0.1144, so the performance was close to the dropout-1 model. While the training set mean accuracy and loss was 0.9998 and 0.0018.

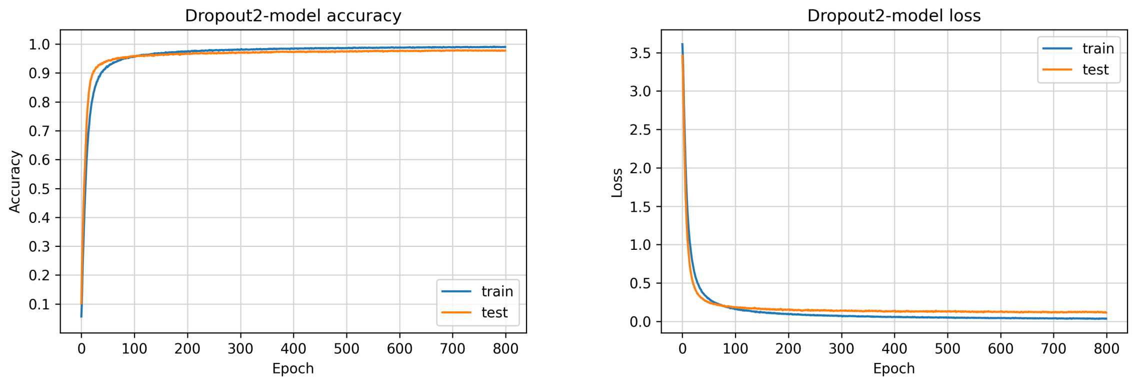

Figure 3: Dropout2-model accuracy and loss during training.

$Dropout3 - model$: in the third and last architecture, as shown in the code below, I added two dropout layers after each exponential linear unit activation function of each convolutional layer.

```
...
tf.keras.layers.ELU(),
tf.keras.layers.Dropout(0.5),
tf.keras.layers.MaxPooling2D(pool_size=(2,2)),
...
tf.keras.layers.ELU(),
tf.keras.layers.Dropout(0.5),
tf.keras.layers.MaxPooling2D(pool_size=(2,2)),
...
```

In Figure-4 it is possible to observe that the dropout in this configuration did not achieve the same generalization performance of the other two modified models. It is interesting to note that, on the training data, it had the worst performance among all models. Also, even if it is not as evident as in the training progress of the original-model, it seems that there are still some small signs of overfitting. In fact, while the error on the training set continues to decrease, the validation error stops decreasing after 150 epochs. The mean of the accuracy and the loss, on the test data, was respectively 0.9438 and 0.2646. While the mean of the accuracy and the loss, on the training data, was 0.9933 and 0.0297.

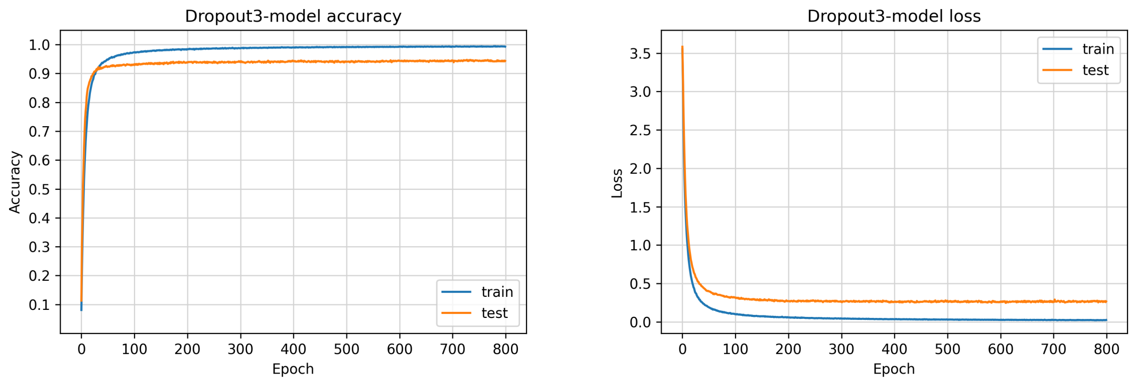

Figure 4: Dropout3-model accuracy and loss during training.

Figure-5 shows the performance, using accuracy and loss metrics, of the 4 different model architectures on the test data. As previously stated, the best performing models are dropout1 and dropout2, the former being slightly better than the latter. These two models, compared to the original one, shown a clear improved generalization ability. While I can't state that the dropout-3 model performed better than the original one, since the performance are only slightly improved.

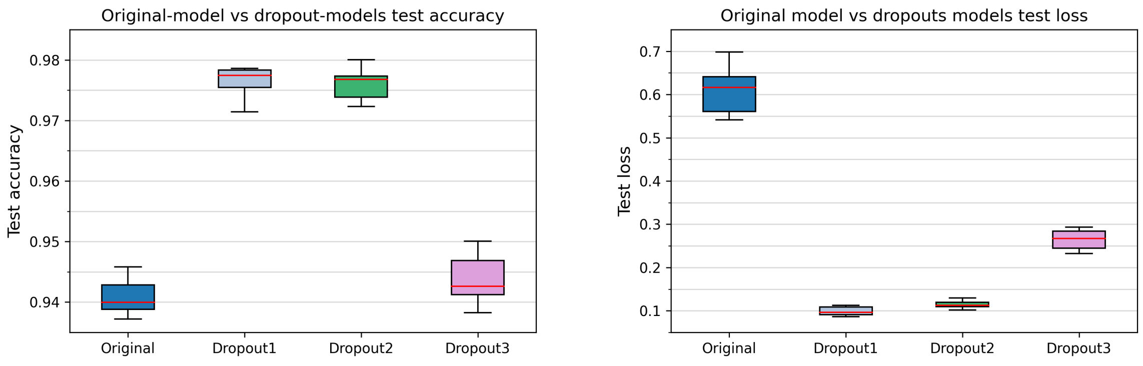

Figure 5: Different architectures accuracy and loss on test dataset.

Table-1 shows the mean of the accuracy and the loss, obtained on the training and test set, in the six independent trials.

Table 1: **Mean of accuracy and loss on training and test data.**

Model	Train acc mean	Train loss mean	Test acc mean	Test loss mean
Original	0.9998512	0.0014878	0.9408683	0.61068712
Dropout1	0.9994857	0.0040141	0.97637899	0.09893451
Dropout2	0.9997535	0.0018357	0.97604909	0.11444334
Dropout3	0.9932924	0.0296686	0.94375824	0.26455179

Even if a statistical test should be required to draw scientific conclusions, I can cautiously state that my results show that the dropout, if added in certain positions, seems to reduce overfitting, and improve a CNN image classification performance.

4 Keras training and testing mode

Models and layers in Keras have a Boolean flag that can put them in training and testing mode by specifying `training=True` and `training=False`, respectively.

The dropout technique is used to prevents overfitting, it can generates sub networks by randomly removing some units during the training process, and it can greatly improves the generalization performance [3]. A dropout layer only applies when training is set to True, in this way, during training the model can learn with some neurons randomly switched off at each batch iteration. While, during inference, the training is automatically set to False and the model uses all neurons for the making predictions. This makes sense because applying dropout during inference will only through away useful information, worsening the model's performance.

The batch normalization is a technique that address to stabilize the training process by stabilizing the distribution of nonlinearity inputs across the network [2]. In other words, it aims to reduce the Internal Covariate Shift by normalizing, scaling and shifting each input of a layer at each batch iteration [2]. The two parameters, γ and β, are used to scale and shift the normalized values, and are learned during the training process [2]. When training is set to True, batch normalization layer will normalize its inputs using the mean and variance of the current batch. While when training is set to False, it will normalize its inputs by using its internally stored average of mean and variance obtained during training [1]. This makes sense because, during inference, the layer takes as input only one test data point at a time, and, since it can't compute the mean or variance of the batch, it normalize the input by using the mean and the variance estimated during training.

5 Challenge (optional)

I did the following modification to the model and the training process:

- Added a third convolutional layer with 128 masks of size 3x3, followed by an exponential linear unit activation function and a maximum pooling layer with a max filter of size 2x2.
- Added batch normalization between each convolutional layer and its exponential linear unit acivation function.
- Added dropout after each maximum pooling layer, I set the firs drop rate to 0.25 and the other two to 0.5.
- Set the mean of the truncated normal distribution for the bias initialization of the convolutional layers to 0.05.
- Increased the learning rate to 0.05.
- Increased the epochs of the training process to 1200.
- Added contrast and colors saturation data augmentation transformation.

On average, the training process tooks approximately 1 hour and 55 minutes. I performed 7 independent trials and, on average, I obtained a classification performance on the test set of 99.1%. The results are shown in Table 2.

Table 2: **Classification accuracy on the test data.**

	Test accuracy	Training time
Trial 1	0.9915281	01:45
Trial 2	0.9901821	01:46
Trial 3	0.9908155	01:54
Trial 4	0.9921615	01:53
Trial 5	0.9891448	02:00
Trial 6	0.9925574	02:05
Trial 7	0.9911322	02:03
Mean	0.9910745	01:55
Median	0.9911322	01:54

Please find the code attached in the file `HW2_exercise5.ipynb`.

References

[1] M. A. et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[3] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[4] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel. Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition. *Neural networks*, 32:323–332, 2012.