Turkish Lira Project report

Federico Cioschi

Università degli Studi di Milano

February 2021

1 Introduction

This report describes the work I did in order to solve the challenge proposed by the Turkish Lira project [Mal], assigned during the "Algorithms for Massive Datasets" course, attended at the "University of Milan". The task of the project is to implement from scratch an image classifier based on neural networks using Tensorflow. In particular, the classifier must take into account the images contained in the "Turkish Lira Banknote" dataset [Bal20] published on Kaggle. The trained classifier must take an image as input and provide as output the corresponding predictions w.r.t. the 5, 10, 20, 500, 100, and 200 lira classes.

2 Dataset description and preprocessing

The dataset contains 6000 PNG images of Turkish Lira banknotes and occupies 3.5 GB of memory. Images are equally divided into 6 subfolders, one for each denomination: 5, 10, 20, 50, 100, 200. They have a 1280x720 resolution with 3 color channels. Banknotes are depicted from different perspectives, rotated, with different backgrounds, sometimes folded or covered, sometimes partially out of sight (see Figure 1).

Figure 1: Turkish Lira banknotes from different perspectives, rotated, with different backgrounds, sometimes folded or covered, sometimes partially out of sight.

Various data augmentation techniques were already applied by the author to enlarge the dataset [Bal20]: brightness increasing and decreasing, image flipping, salt-and-pepper noise addition (see Figure 2).

I used the `tf.keras.preprocessing.image_dataset_from_directory` function to create a `tf.data.Dataset` object for both the training dataset and the validation dataset. Moreover, this function preprocessed the images resizing

Figure 2: Turkish Lira banknotes with brightness decreased or increased, flipped, salt-and-peppered.

them to the resolution of 64x64 and then converting them into `(64,64,3)` tensors. Training dataset was composed by the 80% of the images in the original dataset, while the remaining 20% constituted the validation dataset. During training, images from the training dataset were provided to the model in batch of 100 items at a time. Finally, the first layer of each model which I defined in this project re-scales the RGB values from the $[0, 255]$ range to the range $[0, 1]$, which is better for neural networks processing.

3 Network Architecture and Implementation

I addressed the project task using Convolutional Neural Networks (CNN). CNNs are feed-forward deep neural networks which have revealed to be very good and efficient for image classification. In order to implement CNNs, I employed the open source library TensorFlow, as requested by the project task.

As stated in [Sta], the most common architectures for CNNs follows this pattern:

```
INPUT -> [[CONV -> RELU]*N -> POOL?]*M -> [FC -> RELU]*K -> FC
```

CONV, followed by RELU, represents a convolutional layer which neurons use the ReLU activation function. POOL instead, represents a pooling layer. Finally FC, followed by RELU, represents a fully connected layer which neurons use the ReLU activation function.

The main idea of this architecture is to alternate convolutional layers with pooling layers: the former are responsible for collecting and filtering information from the input, while the latter are responsible for aggregating this information and therefore reducing the spatial dimension of the input. Finally, everything is flattened and processed by one or more fully connected layers, which leads the network to calculate the final classification.

3.1 Input size

Typically in CNNs the width and the height of the input are equal and corresponds to an integer divisible by 2 several times. Indeed pooling layers usually halves the width and the height of the input, de facto reducing by a factor 4 its the spatial dimension. As suggested in [Sta], CNNs should repeatedly reduce the spatial dimension through pooling layers until the image has been merged spatially to a small size.

After a bit of tweaking, I found that the best trade-off, in order to have powerful CNNs on one side and to limit their complexity on the other side, was to set the initial input size to 64x64: in this way the input size, after `M=3` pooling operations, is shrinked to a size of 8x8. This is the reason why, during preprocessing, images are resized to the 64x64 resolution.

3.2 Layers Depth

Layers in CNNs do not expand only in height and width, but also in depth. Obviously the depth of the first layer is only 3, because it reflects the size of the images given as input to the network; indeed images are encoded as `(64, 64, 3)` tensors, because each pixel is described by three RGB values. However, in the following layers, the depth corresponds to the number of neurons which act as filters on the same receptive field. A typical pattern is to double the number of filters after each pooling layer. The increasing number of filters in deeper layers of the network is justified by the fact that neurons in these layers are responsible for bigger regions of the original input image: so they have to activate themselves in presence of a larger number of shapes that could appear in that region. In this project, I set to 32 the depth of the layers in the first "`[CONV -> RELU]*N -> POOL]`" block. Since `M=3`, then I doubled the depth two times: thus, the second block has layers with 64 filters and the third one has layers with 128 filters.

3.3 Convolutional Layers

Commonly, before a pooling layer, more convolutional layers may be stacked together. As explained in [Sta], it is better to stack together some convolutional layers with a small filter size, than having a singular convolutional layer with a big filter size. This allows to the network to capture more powerful features of the input with fewer parameters.

For these reasons, I set the filter size (`kernel_size` in TensorFlow) of the convolutional layers to 3x3. Moreover, since convolutional layers are not meant to reduce the spatial dimension of the input, I set stride to 1 and I enabled padding.

I have not fixed the number `N` of convolutional layers to be stacked together. Indeed, I evaluated the performance of the network for different values of `N`, see Section 5.

3.4 Pooling Layers

I used TensorFlow `MaxPooling` to implement pooling layers. Since pooling layers have the purpose to reduce the spatial dimension of the input, I set their filter size (`pool_size` in TensorFlow) to 2x2; moreover I set their stride to 2 and I disabled padding. This setting, suggested in [Sta], allows to the pooling layers to reduce the spatial dimension of the input of 75%.

3.5 Final layers, loss function and optimizer

I set parameter `K` to `1`, so CNNs implemented in this project have only one fully connected layer before the output layer. This layer has 90 neurons with a ReLU activation function. This setting has revealed to be able to reach optimal results, so I did not take into account other solutions.

The output layer uses the `softmax` activation function, which is useful to highlight the winning class and also to interpret the network output as a probability distribution.

Moreover, as loss function I used `categorical_crossentropy`, which is the most effective for classification problems [McC13].

Finally, I used `Adam` as optimizer, which is an extension of the classical Stochastic Gradient Descent method that uses moments to avoid to get stuck in saddle points during training. As `learning_rate` parameter in `Adam`, I used `0.01`.

4 Scalability

The dataset used for this project takes up only 3.5 GB of memory. For this reason I saved it entirely in local storage. Then, I called the `tf.keras.preprocessing.image_dataset_from_directory` function to create a `tf.data.Dataset` object from image files in the dataset directory. This object was useful to load batches of images from storage to main memory during training and validation. If the dataset was larger, I could save it on a distributed file system. At that point, the dataset path in the distributed file system could be past similarly to `image_dataset_from_directory`, as if it were a local path. For this reason, the solution I used would have no problem dealing with larger datasets.

In order to efficiently load dataset images from disk to memory, I called two important functions on the `tf.data.Dataset` object [Tena]:

- `cache`, which keeps images in memory after they have been loaded from disk; if the dataset is too large to fit into memory this method can also create a performant cache-file on disk;
- `prefetch`, which allows to overlap current training step with data loading for the next training step.

Finally, I defined all the CNNs in this project under the `MultiWorkerMirroredStrategy`, which allows to synchronously distribute training across

multiple GPUs, hosted on multiple computers [Tenc], [Ten20], [Tenb]. This means that each training batch is potentially divided in slices distributed to the available GPUs, each of which in sync processes the assigned input slice and then aggregate the obtained results with those of the other GPUs. Implementing this solution in TensorFlow was really simple. The only things I had to do were to declare the strategy and then to define model under that strategy. For example:

```
strategy = tf.distribute.MultiWorkerMirroredStrategy()
with strategy.scope():
    model = tf.keras.Sequential()
    model.add( ... )
    ...
```

This allowed to TensorFlow to automatically detect all the GPUs available on the local machine. Moreover, it is possible to inform TensorFlow about the availability of other remote devices configuring the `TF_CONFIG` environment variable. This variable is a JSON string declaring the IP addresses of the other devices and their tasks. An example of configuration, from [Tenc], is:

```
os.environ["TF_CONFIG"] = json.dumps({
    "cluster": {
        "worker": ["host1:port", "host2:port", "host3:port"],
        "ps": ["host4:port", "host5:port"]
    },
   "task": {"type": "worker", "index": 1}
})
```

All the experiments done for this project were run using only one GPU, the one offered by the Google Colab environment. However, the code implementation is completely compatible with distributed training.

5 Experiments and Results

5.1 Experiments

As discussed in Section 3, I took into account the following architecture in order to build the requested CNN:

`INPUT -> [[CONV -> RELU]*N -> POOL]*M -> [FC -> RELU]*K -> FC`

As already justified in Section 3, I fixed `M=3` and `K=1`. Contrary, I decided to experiment CNN performances for different values of `N`, in particular `1,2,3`. Moreover, doing preliminary attempts, I found that optimizations like the addition of Batch Normalization layers, or the scheduling of an epoch-dependent upper limit for the Learning Rate Decay, could greatly improve network performances. For these reasons I decided to execute 9 experiments, summarized here:

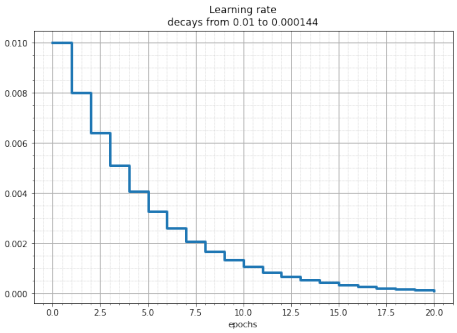

Figure 3: Learning Rate Decay

- testing of a basic architecture for each value of `N=1,2,3`
- testing of an architecture with Batch Normalization, for each value of `N=1,2,3`
- testing of an architecture with Batch Normalization and scheduling of Learning Rate Decay, for each value of `N=1,2,3`

In particular, in order to reduce the Learning Rate at each epoch, I used the following lambda function (also plotted in Figure 3):

```
lambda epoch: 0.01 * math.pow(0.8, epoch)
```

which is applied to the model through the `tf.keras.callbacks.LearningRateScheduler` callback.
Basically models are trained for 20 epochs, but I defined two callbacks for early-stopping:

- `no_learning_early_stopping_callback`, which stops training if training accuracy is under 30% for 5 consecutive epochs, as a symptom that model is not learning (continuing training would be pointless);
- `max_accuracy_early_stopping_callback`, which stops training if both training accuracy and validation accuracy are 100% for 2 consecutive epochs, in order to avoid overfitting once a so good result has been achieved.

A possible optimization which I did not take into account is the addition of a Dropout layer. However, as shown in Subsection 5.2, various models achieved optimal results even without Dropout, so I decided to not take it into account.

The experiments were conducted in the Google Colab environment, which provides:

- CPU: Intel(R) Xeon(R) CPU @ 2.20GHz
- GPU: Tesla T4
- RAM: about 13 GB

5.2 Results

Appendix A shows aggregated results for training and validation accuracy, while Appendix B, shows aggregated results for training and validation loss. Notice that graphs in Figures 5 and 7 show the same results of graphs in Figures 6 and 8, but they are zoomed when possible.

As shown in Figure 5, among the models which do not use batch Normalization and Learning Rate Decay, only the one with parameter `N=1` succeeded in learning significantly within the first 5 epochs. Indeed, the other two models were stopped by the `no_learning_early_stopping_callback`. Despite the model with `N=1` reached the maximum value of training accuracy, it did not happen the same for validation accuracy, indicating a bit of overfitting.

The introduction of Batch Normalization certainly introduced a big improvement. All the models which use Batch Normalization succeeded in learning and they also achieved maximum or almost-maximum values for accuracy, both in training and in validation. Among them, the best is certainly the one with parameter `N=1`, which indeed was stopped by `max_accuracy_early_stopping_callback` at epoch 14, after having maintained the maximum training and validation accuracy for two consecutive epochs. Also the model with parameter `N=2` obtained the same result, but only at epoch 20 and after wide fluctuations in both training and validation accuracy. Probably, those fluctuations were caused by too large `Adam`'s learning rate parameters which prevented accuracy and loss to converge towards their best values. Finally, the model with parameter `N=3` has a very strong gap between training and validation accuracy, signaling overfitting.

The introduction of Learning Rate Decay provided big additional improvements to models performance. The fluctuations in accuracy and loss are almost absent, except a bit in model with parameter `N=3`. Both the models with parameter `N=1` and `N=2` were early stopped by `max_accuracy_early_stopping_callback`, the second one even at epoch 11.

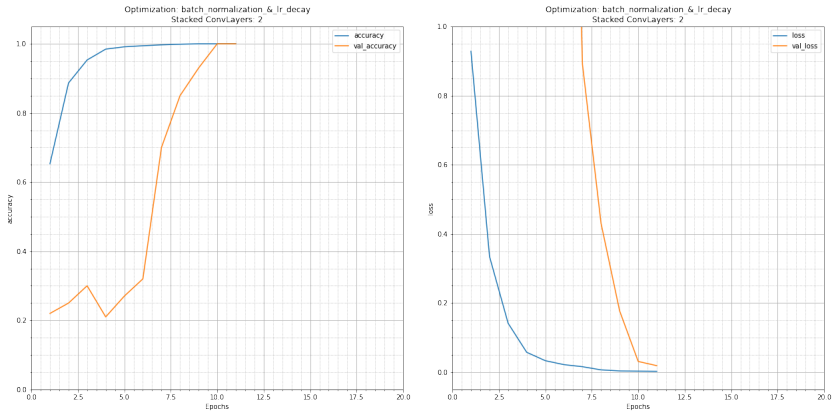

Figure 4: Best model, accuracy and loss graphs.

The model with Batch Normalization, Learning Rate Decay and parameter `N=2` was the quicker to converge and to be early stopped. According to its graphs, accuracy and loss converged to their best values very quickly and also without fluctuations; this happened both in training and in validation. Its training required only 17 minutes and 32.01 seconds. For these reasons, I think that this model is the best between all. Its architecture is shown in Figure 9, and its accuracy and loss graphs are singularly shown in Figure 4.

It should be noted that also two models with parameter `N=1`, the one with Learning Rate Decay and the one without it, achieved excellent results.

A Accuracy Results

A.1 Accuracy: General View

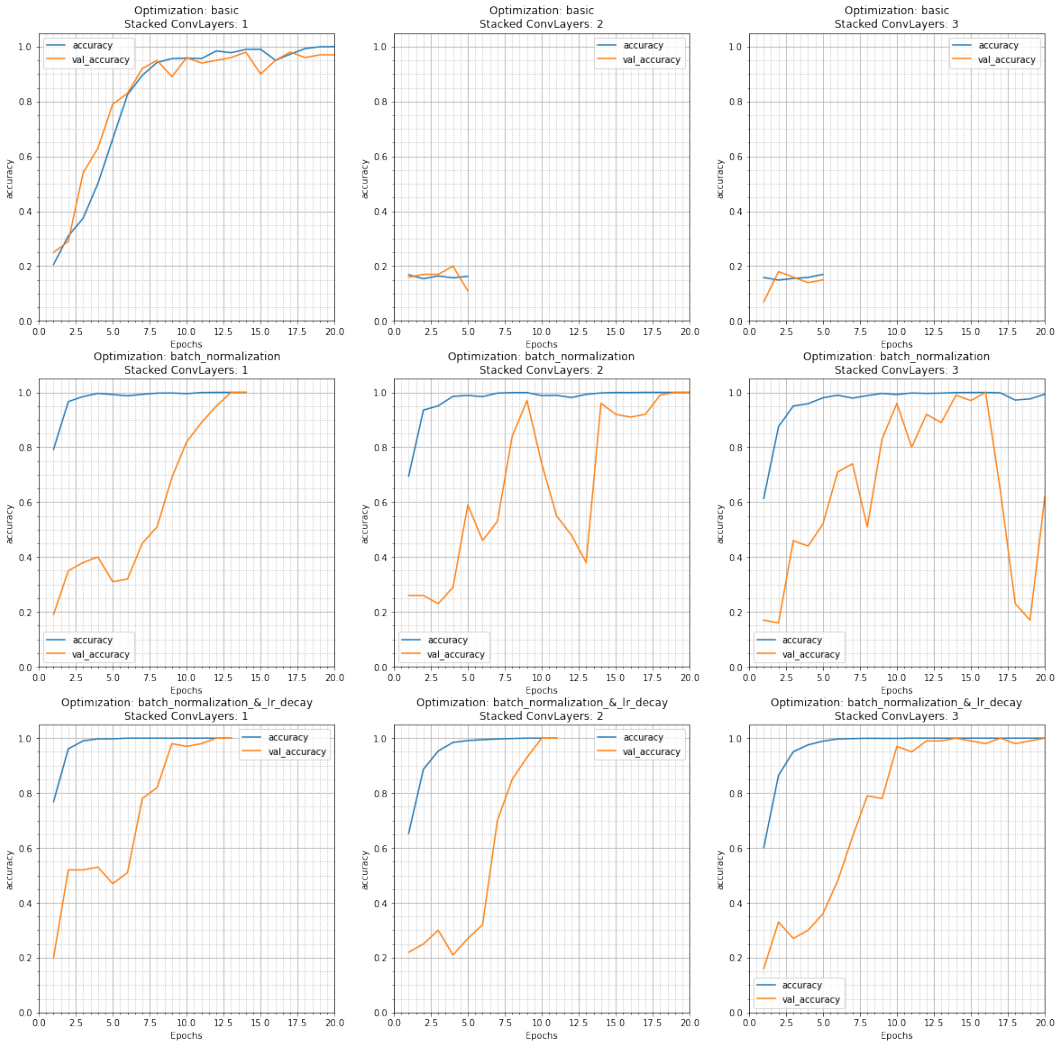

Figure 5: General view of accuracy results; all graphs have the same scale.

A.2 Accuracy: Zoomed View

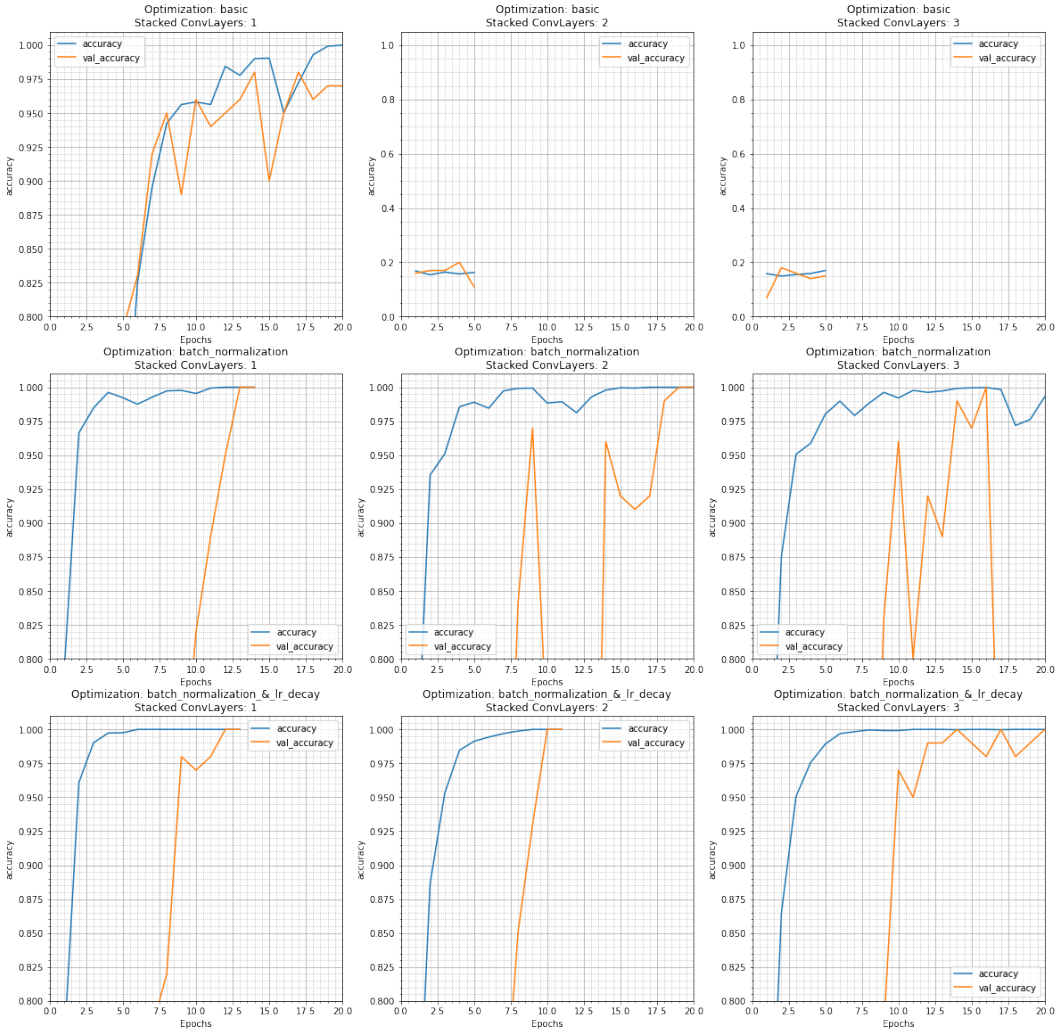

Figure 6: Zoomed view of accuracy results; all, except the two graphs in the upper right corner, are zoomed in by a factor of 5.

B Loss Results

B.1 Loss: General View

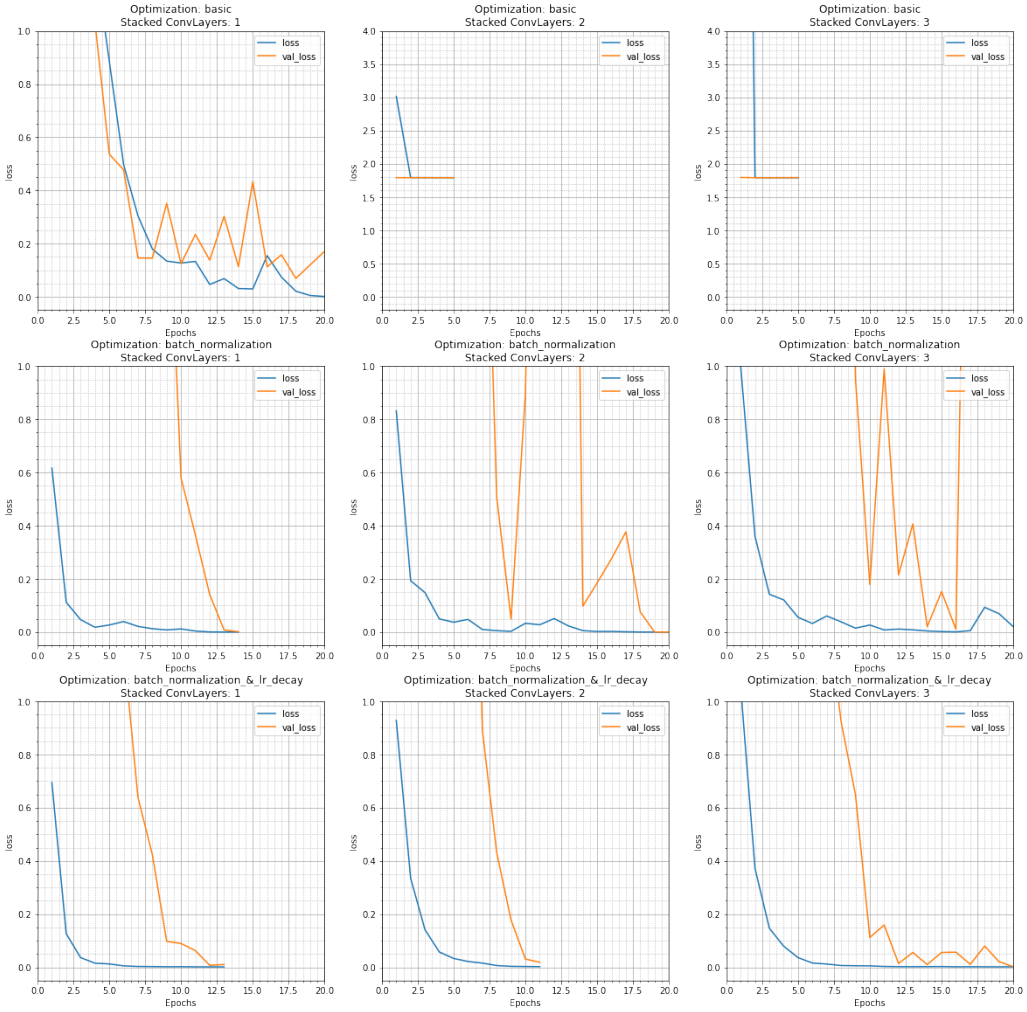

Figure 7: General view of loss results; all graphs, except the two graphs in the upper right corner, have the same scale.

B.2 Loss: Zoomed View

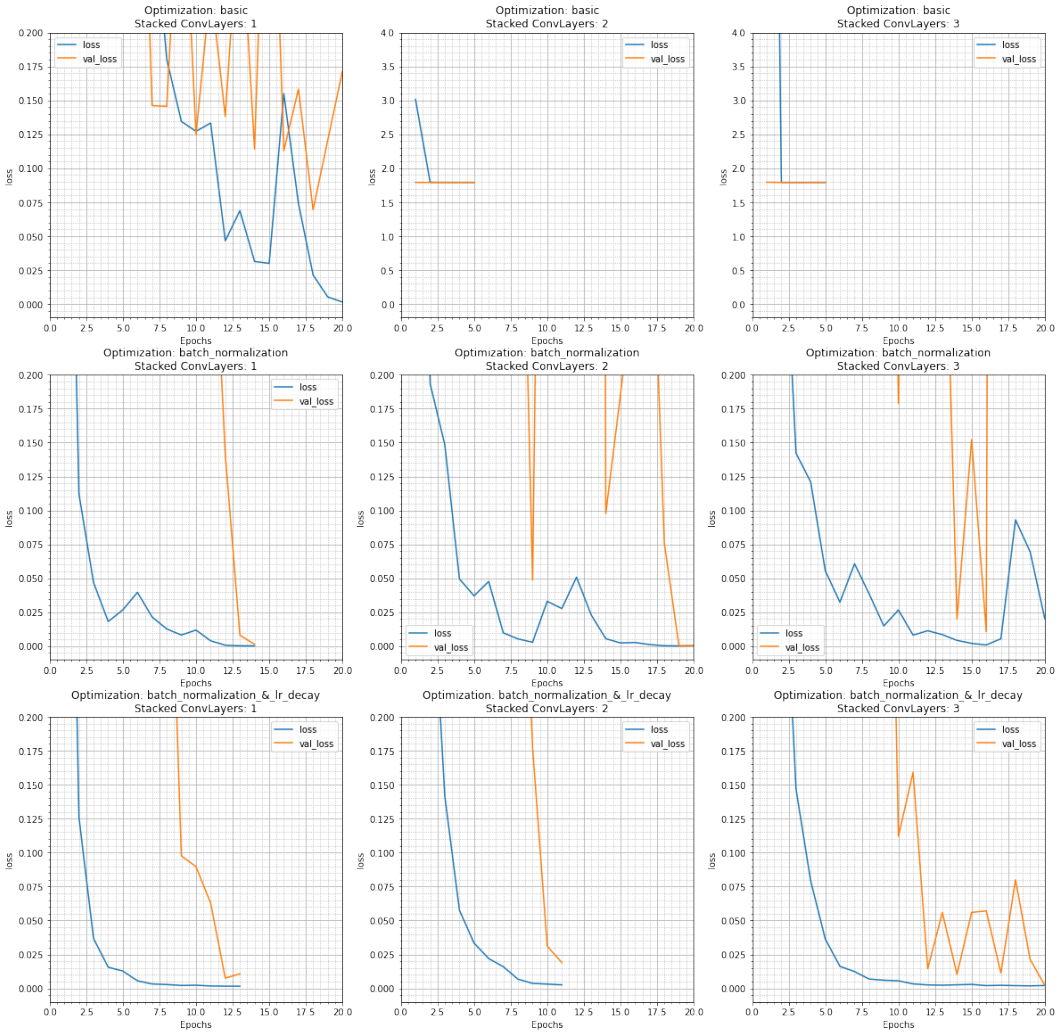

Figure 8: Zoomed view of loss results; all, except the two graphs in the upper right corner, are zoomed in by a factor of 5.

C Best Model

Layer (type)	Output Shape	Param #
Rescaling	(None, 64, 64, 3)	0
Conv2D	(None, 64, 64, 32)	864
Batch Normalization	(None, 64, 64, 32)	128
Activation	(None, 64, 64, 32)	0
Conv2D	(None, 64, 64, 32)	9216
Batch Normalization	(None, 64, 64, 32)	128
Activation	(None, 64, 64, 32)	0
MaxPooling	(None, 32, 32, 32)	0
Conv2D	(None, 32, 32, 64)	18432
Batch Normalization	(None, 32, 32, 64)	256
Activation	(None, 32, 32, 64)	0
Conv2D	(None, 32, 32, 64)	36864
Batch Normalization	(None, 32, 32, 64)	256
Activation	(None, 32, 32, 64)	0
MaxPooling	(None, 16, 16, 64)	0
Conv2D	(None, 16, 16, 128)	73728
Batch Normalization	(None, 16, 16, 128)	512
Activation	(None, 16, 16, 128)	0
Conv2D	(None, 16, 16, 128)	147456
Batch Normalization	(None, 16, 16, 128)	512
Activation	(None, 16, 16, 128)	0
MaxPooling	(None, 8, 8, 128)	0
Flatten	(None, 8192)	0
Dense	(None, 90)	737280
Batch Normalization	(None, 90)	360
Activation	(None, 90)	0
Dense	(None, 6)	546

Total params: 1,026,538
Trainable params: 1,025,462
Non-trainable params: 1,076

Figure 9: Best model architecture: `INPUT -> [[CONV -> RELU]*2 -> POOL]*3 -> FC -> RELU -> FC`

References

[McC13] James D. McCaffrey. *Why You Should Use Cross-Entropy Error Instead Of Classification Error Or Mean Squared Error For Neural Network Classifier Training*. Nov. 5, 2013. URL: https://jamesmccaffrey.wordpress.com/2013/11/05/why-you-should-use-cross-entropy-error-instead-of-classification-error-or-mean-squared-error-for-neural-network-classifier-training/ (visited on 02/11/2021).

[Bal20] Fatih Baltacı. *Turkish Lira Banknote Dataset*. Apr. 19, 2020. URL: https://www.kaggle.com/baltacifatih/turkish-lira-banknote-dataset (visited on 02/12/2021).

[Ten20] TensorFlow. *Getting Started with Distributed TensorFlow on GCP*. Dec. 7, 2020. URL: https://blog.tensorflow.org/2020/12/getting-started-with-distributed-tensorflow-on-gcp.html (visited on 02/11/2021).

[Mal] Dario Malchiodi. *AMD-2019-20-projects*. URL: https://docs.google.com/document/d/16sEIlAI882WaL5TZOXwlqp6w5BBSeYfvltTtEbEOn1M/edit (visited on 02/12/2021).

[Sta] Stanford. *Convolutional Neural Networks (CNNs / ConvNets)*. URL: https://cs231n.github.io/convolutional-networks/ (visited on 02/08/2021).

[Tena] TensorFlow. *Better performance with the tf.data API*. URL: https://www.tensorflow.org/guide/data_performance (visited on 02/11/2021).

[Tenb] TensorFlow. *Distributed training with Keras*. URL: https://www.tensorflow.org/tutorials/distribute/keras (visited on 02/11/2021).

[Tenc] TensorFlow. *Distributed training with TensorFlow*. URL: https://www.tensorflow.org/guide/distributed_training (visited on 02/11/2021).

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.