Machine Learning for Embedded Systems

Home assignment 1 – Neural Network in C

Name: Ondřej Schejbal

UNI-ID: onsche **School code:** 214308IV

GitLab repository: gitlab.cs.ttu.ee/onsche/ias0360

In my work I have decided to focus on optimizing the **memory size** of the prepared model. I have considered parameters, which can affect the learning process the most and I tried to find the best optimization parameters which will keep approximately the same accuracy, while having the simplest model I can (simple model should implicate less trainable parameters leading to less total weight).

My target was to minimize the size of the model while also keeping the accuracy of the initial model as high as possible.

As the initial model for this project, I have taken the final model from Lab 6. The NN uses Adam optimizer and consists of these layers:

- Flatten
- Dense layer with output of size 80 with relu activation function
- Dense layer with output of size 60 with relu activation function
- Dropout layer
- Dense layer with output of size 10 (number of possible output classes)

In this project I have defined that the model memory size will be computed as the **total number of parameters of the model + it's memory allocation size in KB when saved as TensorFlow Lite model**.

I have decided to experiment with optimizing the model in following areas:

1. Activation function used in the dense hidden layers
2. Number of layers in the NN and their shapes
3. Dropout rate, optimizer algorithm, learning rate and epoch count
4. Pruning and Quantization

In the next sections I will describe my approach and experiments for each of the areas mentioned above.

One of the things I had to consider while evaluating the model was that for each model fitting, the accuracy always varied a little bit. This behavior shouldn't be surprising and can be quite expected based on the NN nature. I have discovered, that for the same model, the accuracy can vary between each run, but the value difference is never higher than 0.003. Based on that I have decided that I won't consider model with accuracy which differs in this threshold as a better or worse choice.

The total file size can also differ for each model compilation, so I have decided to ignore any difference below 0.09 KB.

1. Activation function optimization

First, I have tried to check if the selection of activation function has any effect on the weights. I have used the initial model and tried switching the activation function for the hidden layers with the other available functions which keras has implemented.

The initial model uses *elu* and I have tried switching it up with *relu, sigmoid, softmax, softplus, tanh* and *exponential* activation functions. I have left the rest of the initial model untouched so I could clearly observe the effect of just switching the activation function.

My expectations were that the activation function should not have any effect on the total model weights and I was able to verify this thought. The measured differences can be seen in the table below. Only for the *softplus* function I was getting slightly higher weight of the saved model, but the value got over the threshold only slightly. The accuracy of other functions proved to be either comparable with the initial one or slightly worse. Even though the activation function doesn't have any significant effect on the total weight of the model I have decided to use *relu* instead of *elu* in my final model. The reason was, that I am more familiar with the function and it got slightly better results in this first experiment

Activation function	Original (elu)	relu	sigmoid	softmax	softplus	tanh	exponential
Accuracy	0.9734	+0.0017	-0.0625	-0.1125	-0.0027	-0.0016	-0.8754
Weight	68538.703125	-0.1406	+0.0064	+0.082	+0.1969	+0.0703	+0.082

Figure 1 - Measured stats for selected activation functions. Orange color indicates that the result is within ignored threshold. Green means improvement and red indicates worsen results.

2. Number of layers and their shapes

Number of layers and their shapes is the category, which is crucial for model accuracy and straightly affects the model weight and therefore I have dedicated more focus to this part.

The initial model uses 3 dense layers in total. The basic assumption is that more layers should increase the total weight of the model, but my personal suggestion is that it more depends on the total number of neurons in the whole NN.

Because there can be endless possibilities of number of layers and their various shapes, I have decided to divide the experiments into 2 categories.

a) Adding and removing dense layers
b) Experiments with different shapes of the dense layers

Since I am not focusing on model accuracy optimization in this work, I focused on finding the model which is as light as possible while trying to keep the original accuracy as close to the initial model's accuracy as possible.

Because the options are infinite, in each category I am going to summarize my observations for each of them and present the overall performance effects they had in comparison to the initial model.

a) Adding and removing dense layers

This category was mostly to confirm the basic idea, that simply adding, or removing huge layers significantly affects the total model weights. I have experimented with adding/removing layers one by one and exploring the affects. I did not change or add any Dropout layers since I am focusing on those in different section. I have also not modified the shapes since that's the purpose of subsection b) in this section.

I have found out that adding layers did not increase the accuracy, with big number of layers added the accuracy actually got lower. Not significantly but considering the total weight of the model, which increased dramatically, then adding layers is not beneficial for our model at all.

I was surprised that removing the second dense layer of shape *(784, 80)* affected the decreased the total model weight the most while affecting the accuracy minimally (within tolerated threshold). I have decided to choose this version in my final model. Observed changes can be seen in Figure 3 below. Figure 2 shows the state of my model after this phase.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_2 (Flatten)          (None, 784)               0
_________________________________________________________________
dense_5 (Dense)              (None, 60)                47100
_________________________________________________________________
dropout_2 (Dropout)          (None, 60)                0
_________________________________________________________________
dense_6 (Dense)              (None, 10)                610
=================================================================
Total params: 47,710
Trainable params: 47,710
Non-trainable params: 0
```

Figure 2 - Chosen model which lead to best model weight decrease while almost preserving the initial models accuracy

Dense layers count	3 (initial)	1	2 (initial – 1st d. l.)	2 (initial – 2nd d. l.)	4	5
Accuracy	0.9734	-0.0522	-0.004	-0.004	-0.0033	-0.0053
Total param count	68 270	-60 420	-20 560	-4 660	+6 480	+12 960
Weight	68538.703125	-60657	-20640.875	-4678.7656	+6505.5781	+13011.2812

Figure 3 - Observed results when adding and removing layers. Green color marks the chosen model

b) Experiments with different shapes of the dense layers

In the point a) above I have decided to work with NN consisting of only 2 dense layers. One of them has shape *(None, 10)* which is used to convert the previous layer size to our desire output classification size, which is 10, so the only layer shape I was able to experiment with was the layer with shape (*None, 60)*. I have shown the most signific changes in values observed in the Figure 4. I have decided to take the output size of 50 for my final model.

Dense layer output shape	60 – state before observations	20	30	50	90	110
Accuracy	0.9698	-0.024	-0.012	-0.0051	+0.0055	+0.005
Total param count	47 710	-31 800	-23 850	-7950	+23 850	+39 750
Weight	47897.796875	-31924.1875	-23943.1328	-7981.0234	+23943.207	+39905.3164

Figure 4 - Observations from modifying the output shape of the dense layer. Green color marks the chosen model

3. Dropout rate, optimizer algorithm, learning rate and epoch count

In this section I briefly describe my experiments with multiple optimization parameters. These parameters are really important for optimizing the accuracy of the model but are not relevant for the total weight of the final model. I have decided that it would be a good idea to run some tests, in order to confirm this expectation.

Dropout rate helps to prevent overfitting of the model as it randomly sets some weights to zero. Initial model uses dropout rate 0.2. I have expected that the dropout rate will not have significant impact on the weight of the model as it doesn't change the output model and only helps in training phase. I experimented with values from the range [0.1; 0.8] and I was able to confirm my expectations. In the current models state it did not even improve the accuracy.

Optimizer algorithm in the initial model is Adam. I have investigated keras library and tried every other optimizer it has implemented (SGD, RMSprop, Adadelta, Adagrad, Adamax, Nadam and Ftrl). None of them proved to affect the model's total weight and none of them even proved to improve the accuracy in any significant way.

Different **Learning rate** and **Epoch count** parameter values also did not influence the weight of the model at all.

4. Pruning and Quantization

Pruning and Quantization are considered to be the most effective techniques when it comes to model size reduction. Because of that they are also the most used ones. I have followed the official keras guides and my notes from the lectures to apply these techniques to my model.

Pruning is based on trying to set as many weights to 0 as possible. I have applied the *prune_low_magnitude* method to my model. According to the documentation it is required to strip the model of pruning by using *strip_pruning* function before compressing it to the TF Lite model in order to reduce the model size.

I have started with applying the pruning to the whole model and to my surprise it did not bring any improvement in total model weight size. I have used different pruning parameters – I have modified the *initial_sparsity, final_sparsity, begin_step* and *end_step* params and none of them proved to show any improvements.

I have considered that maybe the pruning doesn't have any effect because my current model (resolution of previous steps) has not as many neurons for the pruning to take effect, so I decided to go back and try pruning on the initial model which has 3 dense layers, but even there I did not get any improvement.

I have then tried to prune only the dense layers and then I have tried to only prune one (the biggest) dense layer. None of that brought me improvement, which was unexpected, but I was not able to find suitable parameters which would bring expected improvements. I have tried many different values for the parameters and different approaches, but nothing produced any improvement. I suspect that the model is too small to see the improvement. My final code is shown in Figure 5.

```
prune_model = True

def apply_pruning_to_dense(layer):
    # other layers for pruning can be also filtered here
    if isinstance(layer, tf.keras.layers.Dense):
        return tfmot.sparsity.keras.prune_low_magnitude(layer)
    return layer

if prune_model:
    myModel = tf.keras.models.clone_model(myModel, clone_function=ap-
ply_pruning_to_dense)
```

Figure 5 - Code - pruning applied to specific layers

Quantization is a process of changing the weight value types to lower bit ones in order to decrease total model size. I have made the model aware of quantization by using the *quantize_model* keras function.

Experiments with quantization had the same resolution as applying the pruning to the model. I have experimented with quantizing all layers, only one type of layer or only one layer. None of the alternatives brought me any improvements in the total model weight. Quantization of the model can be seen in Figure 6.

```
quantize_model = True
def applyQuantizationToSomeLayers(layer):
    # here I experimented with only adding some type of layers
    # if isinstance(layer, tf.keras.layers.Dense):
    #     return tfmot.quantization.keras.quantize_annotate_layer(layer)
    # return layer
    return tfmot.quantization.keras.quantize_annotate_layer(layer)

def quantizeModel(model, print_summary = True):
    annotated_model = tf.keras.models.clone_model(model,
clone_function=applyQuantizationToSomeLayers)
    quantized_model = tfmot.quantization.keras.quantize_apply(annotated_model)
    if print_summary:
        quantized_model.summary()
    return quantized_model

if quantize_model:
    quantizeModel(myModel, False)
```

Figure 6 - Code snippet - Quantization

Running model on my own data

I have prepared self-written numbers from 0 to 9 in the same form as they are in the MNIST dataset. Then I have run my final trained model on these numbers. My model was able to recognize 2 of my 10 handwritten letters. Which is not a good result, when I compare it to the shown accuracy, but

considering the fact, that I have stripped the model a lot and focused only on the model weight and I did not focus on optimizing the prediction accuracy, I think it makes sense. The initial model was able to perform the same prediction accuracy on my handwritten numbers.

Experimenting with other architectures

I have decided to test performances of 2 other pretrained architectures and these were MobileNet_V1 and ShuffleNetV2. I have used the Basic_CNNs_TensorFlow2 GitHub repository, which offers an easy method to train your own data on TensorFlow implementations of some popular architectures. I have downloaded the MNIST dataset locally and trained the selected models on it, each for 5 epochs, same number of epochs as I used for my architecture. Then I have evaluated the trained models on my handwritten numbers. We can see, that the MobileNet_V1 was surprisingly inefficient, but the ShuffleNetV2 architecture was able to recognize the most numbers.

Summary of results

Model	Memory size (parameters + TF Lite model size in KB)	Validation accuracy	Accuracy on my handwritten numbers
Initial model	68341.125	0.9728	0.2
Final model	39802.078125	0.9696	0.2
MobileNet_V1	Not measured	0.09746	0.1
ShuffleNetV2	Not measured	0.9787	0.6

Discussion

I think I was able to do a lot of precise experiments with researching which optimization parameters have the biggest influence on the total model weight. Of course, the possible combinations of these parameters are endless, so it was not possible to try them all. But overall, I think I was able to discover the main effects of selected parameters on model weight.

I have also learned how to use pruning and quantization on the model. From these techniques I was sadly not able to get any significant result. I think that it's because of a very small size of my NN which did not allow these techniques to have great impact on the model weight.

I was also able to successfully compare my NN architecture with MobileNet_V1 and ShuffleNetV2 architectures.