Machine Learning
Università della Svizzera italiana

Assignment 2

Stefano Taillefert

June 8, 2021

Intro

The submission archive includes everything you need to run the models (in `.pkl` format), but the code and the full `.h1` models are also available at:

`https://github.com/Steeven9/ML-Assignment-2`

The repo will be made public after the deadline to avoid surprises; also since it's versioned you can check that the models were not updated after the deadline.

To run the models, simply `cd` into `deliverable/` and run `run_task1.py` or `run_task2.py`. Make sure to have the `custom_utils.py` file in the same directory!

To compile and train the models, run `create_models.py [task_number]` from the `src` folder. Without argument, both tasks will be executed. The `custom_utils.py` file is required here as well (yes, two copies are already included in both folders for your convenience).

In both cases the script will automatically download the necessary datasets, so everything should work out of the box on first try. Hopefully.

Problem 1

1. - 4. The model is in the file `deliverable/nn_task1.h5`, while the code used to generate it is in the first section of `src/create_models.py`.
For this task I ran all the code on my Mac without encountering particular problems.

5. Here's the plot I obtained:

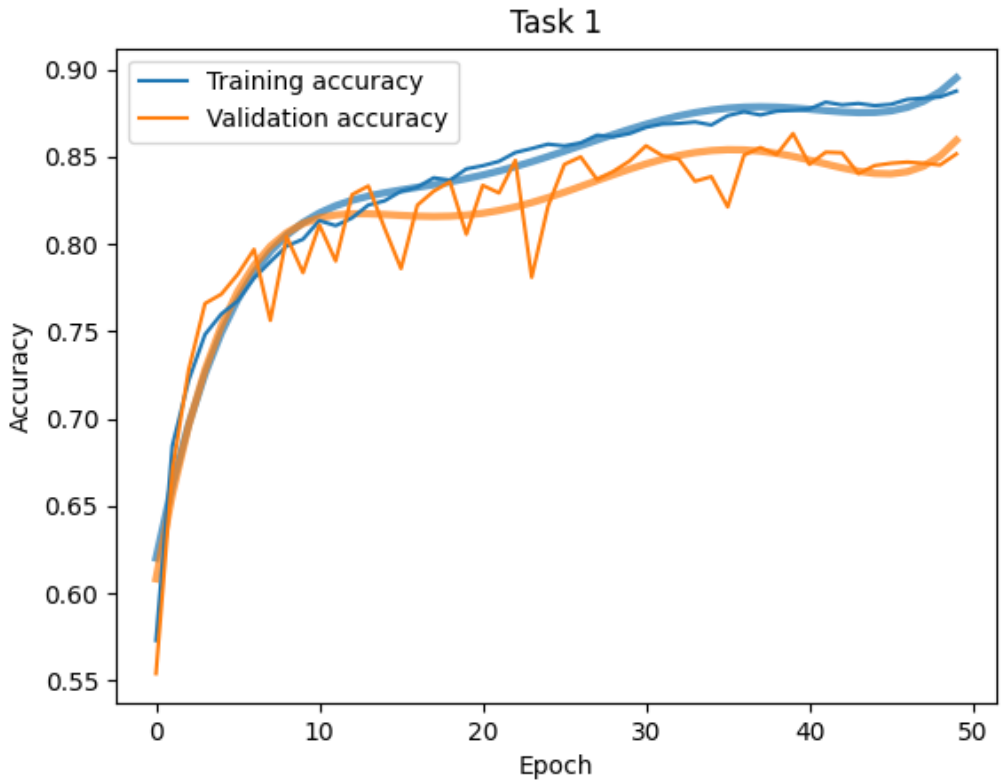

Figure 1: The plot for task 1

6. - 7. The results from the various models are summarized in the table below:

Model	Test loss	Accuracy	MSE
8 neurons - 0.003 LR	0.3564	0.8576	0.0674
16 neurons - 0.01 LR	0.4255	0.8363	0.0798
64 neurons - 0.01 LR	0.4881	0.8113	0.0920
16 neurons - 0.0001 LR	0.3692	0.8536	0.0703
64 neurons - 0.0001 LR	0.3952	0.8416	0.0756

All five models have been trained and evaluated in sequence on the same dataset. As we can see, the best one is the one with 8 neurons and 0.003 learning rate, the initial one. This is based on the lowest test loss and MSE, and the highest accuracy.

Problem 2

1. - 3. See the following sections for the details of the behavior of the two models. In both cases the hidden layer I added is a dense one with 128 neurons and ReLu activation function, while the output layer has three neurons since we have three classes (rock, paper, scissors).

4. For this task I tried to use my gaming rig at home (Intel i5-9600K, 16GB DDR3@3200, RTX 2060) instead of Colab. I spent a whole day struggling with CUDA and Python on Windows 10 (ugh) and after some fiddling I managed to make it work. The first part, without data augmentation, worked fine, but for the second part I ran out of VRAM after the first epoch. That was a bit underwhelming to say the least.
I included both the result from Colab and my machine for the first part, while the second one has been run only on the cloud.

I am unaware if it's a hard limitation or if I could tune the model to get around it (maybe less neurons for the hidden layer?); despite the fact that I would've loved to run everything locally, I didn't have much time left to investigate further. Here's the error I got:
`W tensorflow/core/common_runtime/bfc_allocator.cc:456] Allocator (GPU_0_bfc) ran out of memory trying to allocate 3.59GiB (rounded to 3853516800)requested by op sequential_6/vgg16/block1_conv1/Relu`

Without augmentation

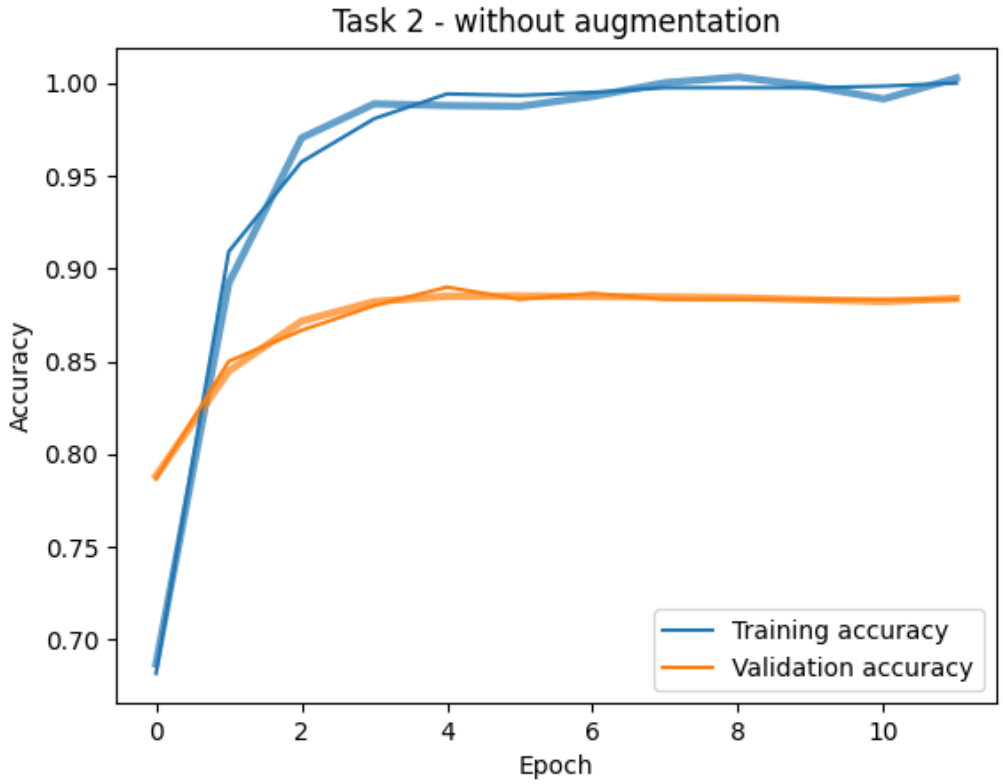

Figure 2: The plot for task 2 without data augmentation

The results from the model without augmentation are:

Test loss: 0.2259 - Accuracy: 0.9366 - MSE: 0.0319 (my pc)
Test loss: 0.3201 - Accuracy: 0.9100 - MSE: 0.0464 (Colab)

The model is in the files `deliverable/nn_task2_no_augmentation.h5` and `deliverable/nn_task2_no_augmentation.pkl` (you can choose which one to run, they should be the same), while the code used to generate it is in the second section of `src/create_models.py`

With augmentation

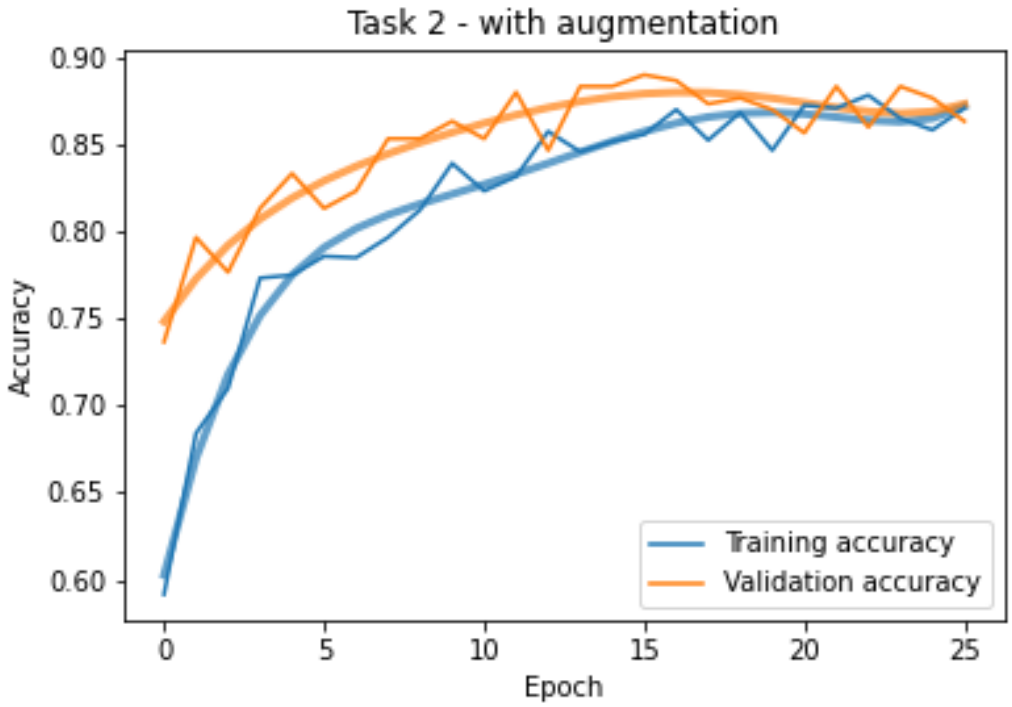

Figure 3: The plot for task 2 with data augmentation

The results from the model with augmentation are:

Test loss: 0.2214 - Accuracy: 0.9133 - MSE: 0.0421 (Colab)

The model is in the files `deliverable/nn_task2_augmentation.h5` and `deliverable/nn_task2_augmentation.pkl` (you can choose which one to run, they should be the same), while the code used to generate it is in the second section of `src/create_models.py`

Conclusion

Based on the experimental results, using data augmentation didn't improve much our model; instead we actually obtained worse scores than without it. Here's a summary of the values:

Model	Test loss	Accuracy	MSE
No augmentation (local)	0.2259	0.9366	0.0319
No augmentation (Colab)	0.3201	0.9100	0.0464
With augmentation (Colab)	0.2214	0.9133	0.0421

As we can see, the three values are not very far from eachother. Test loss and accuracy decreased - which are respectively a good and bad thing - while MSE went up, which is definitely not good. On the other hand, those are very little varainces and one could argue that an accuracy of 0.9366 is already very good, so there is not much room for improvement anyway.

The interesting part is that I obtained better results running the code on my machine rather than on Google's platform, despite having the exact same code, data and seeds for tensorflow and numpy (just in case).