Practical Deep Machine Learning

Assignment 2 Report

Dr. Mohamed Mostafa
Fall 2020

Mohamed A. Abdel Hamed
900163202

Part I

Image Classification Using Artificial Neural Networks

1. Code:

All source code for this part can be found the 'src' folder. The main.py file is the main runner for the code.

1.1 Dataset loading:

1. The main script looks for a directory called 'flower_photos' and assumes it contain the flowers dataset. If the folder is not present, it's downloaded (putting the dataset folder in the same directory as the code is highly recommended to avoid downloading it). Note that the attached 'flower_photos' is empty, you need to put the image folders there.
2. Images are read folder by folder according to their class.
3. Each image is resized to 64x64 (coarse-tuned).
4. The last (in alphabetical order) 100 images in each class are isolated into a separate dataset for testing

Note:

- You can set the flag 'USE_LOADED_DATASET' to load the dataset from the NPY files provided instead of loading all images again.
- You can set the flag RUN_EXP to run the fine-tuning experiments on the final model. Bear in mind that this will not necessarily give the same plots in this report as they were not conducted on the final model, but rather for getting a numerical sense for fine-tuning.
- **Final model parameters** are marked in all caps within the main.py script
- **Random seed** is also set in the main.py script (SEED = 73) for reproducibility.

2. Data Preprocessing:

The only preprocessing done on the images is image normalization (dividing by 255) to make it more convenient for training neural networks because high feature values lead to unreasonably magnified neuron activations. Instead, the channel values now represent a normalized degree of color.

3. Network Architecture:

3.1. Computational Structure:

The neural network is computationally formed as a computational graph composed of a chain of layers, each having its own set of neurons and weights with the exception of the input layer, which is modelled as a virtual layer (i.e. is not an object). We have the following main flows:

I. **Forward Pass**: starting from the first hidden layer, each layer hands calculates its activation taking in the activations of the previous layer (input features in case of input layer) as $A_i = Act(W_i * A_{i-1})$, where the Act function is supplied externally and could either be Leaky ReLU or Sigmoid. When we reach the last layer, its activations are passed to the loss function $L(A)$, and then $\frac{\partial L}{\partial A_{out}}$ is calculated and passed to the output layer to start the backwards pass. The loss function has two options: Hinge loss or NLL loss. Note that in case of NLL loss, a **Softmax** activation is used at the output layer instead of leaky ReLU.

II. **Backward Pass:** starting from the last layer, each layer i+1 hands over its $\frac{\partial L}{\partial A_i}$ to its predecessor i, which in turn uses that to calculate $\frac{\partial L}{\partial W_i} = \frac{\partial L}{\partial A_i} * \frac{\partial A_i}{\partial Z_i} * \frac{\partial Z_i}{\partial W_i}$ where $Z_i = W_i * A_{i-1}$

III. **Weight update:** after assembling the weight gradients of a layer, they are aggregated to the weights through an ***Adam SGD Optimizer*** (default, but user can use basic SGD instead).

3.2. Weights Initialization:

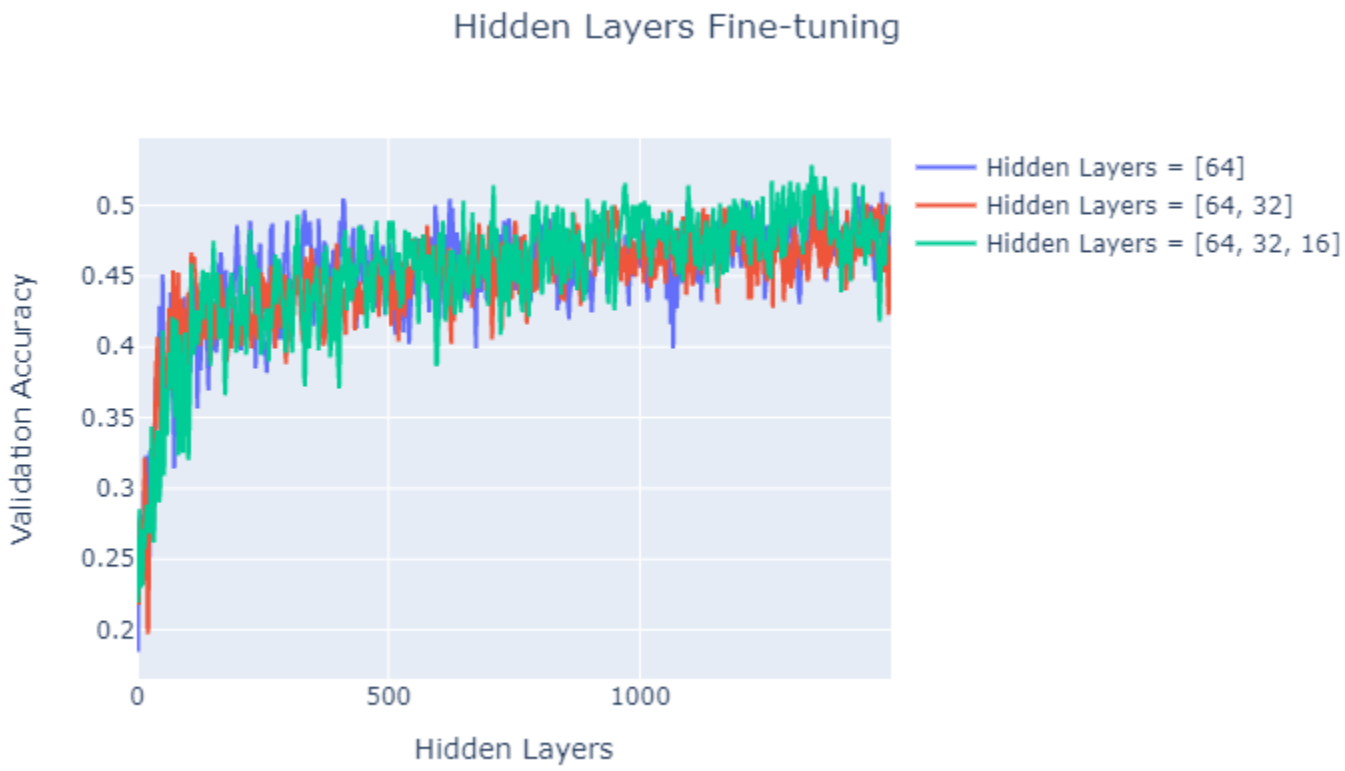
For that, ***Xavier*** initialization is used as it has been tested and verified to produce reasonable results compared to other methods of initialization.

3.3. Hyperparameters Fine-tuning:

Hyperparameters were generally selected by trying out a set of values and picking the one that achieves the highest validation accuracy. Note that hyperparameters are evaluated and fine-tuned independently from each other to mitigate the time cost of optimizing all of them in tandem. That, of course, assumes that variations in any of them bears insignificant changes on others, which is not the case in reality, but it still provides a helpful guide. It's also worth noting that the numerical experiments (i.e. the plots) are not necessarily performed on the final model parameters.

3.3.1. Network Architecture:

The following graph shows validation results for 3 different layer architectures.



Validation results don't show conclusive differences except for stability; thus, [64, 32] was selected for moderation.

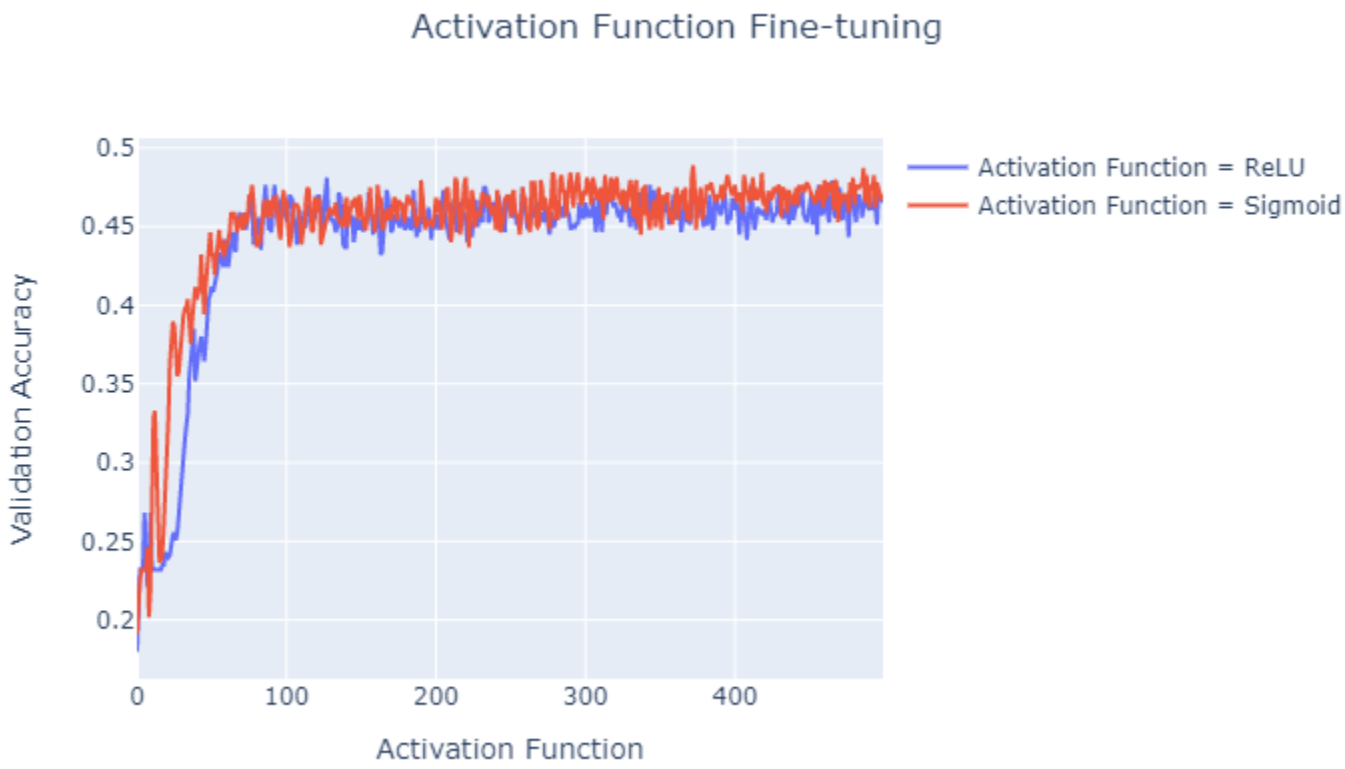
3.3.2. Learning Rate:

A numerical sense of the learning rate was obtained by fixing all other parameters and trying out a range of order of magnitudes. Then, the exact value was obtained by rerunning the experiments with a finer-grained step. The following graph shows results for tuning the learning rate.

Based on that, the value for learning rate was chosen to be 0.05 because it has an overall better performance in further epochs and is generally safer (less wiggly) than its nearest competitor 0.1

3.3.3. Activation Function

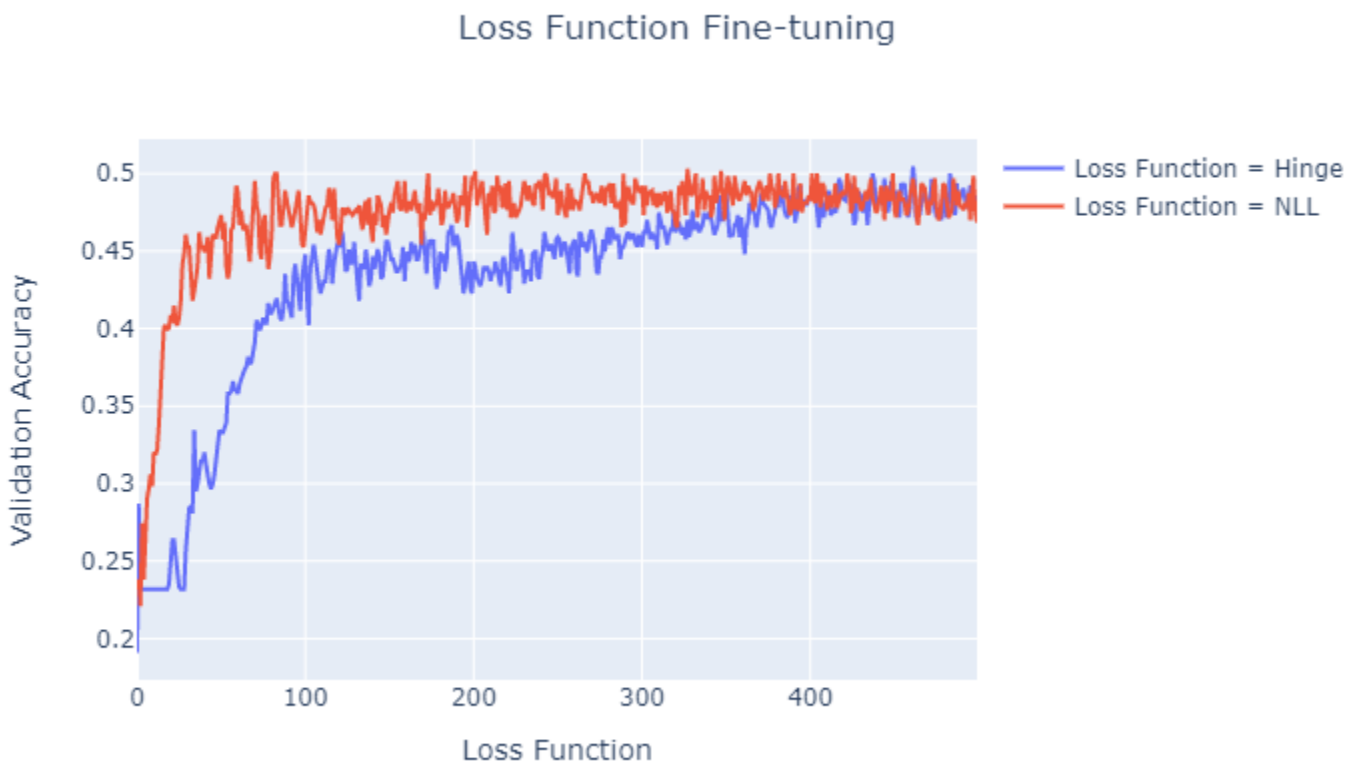
The following graph shows results for tuning the activation function.



We can see that both ReLU and Sigmoid have very close accuracies for large epoch counts with Sigmoid having a slight edge. Given that, ReLU was chosen, though, because of its analytical advantages (e.g. non-saturation, etc.) since the difference is marginal.

3.3.4. Loss Function:

The following graph shows results for tuning the loss function.



It can be seen that NLL is superior in accuracies, so it was picked for the final model. Moreover, it is more commonly used in Neural Networks compared to Hinge loss, which is typically used with SVMs.

3.3.5. Optimizer Selection:

The following graph shows results for tuning the optimizer.

We can see that Adam was more stable, hence more preferable for the reliability of results although they have a close average accuracy.

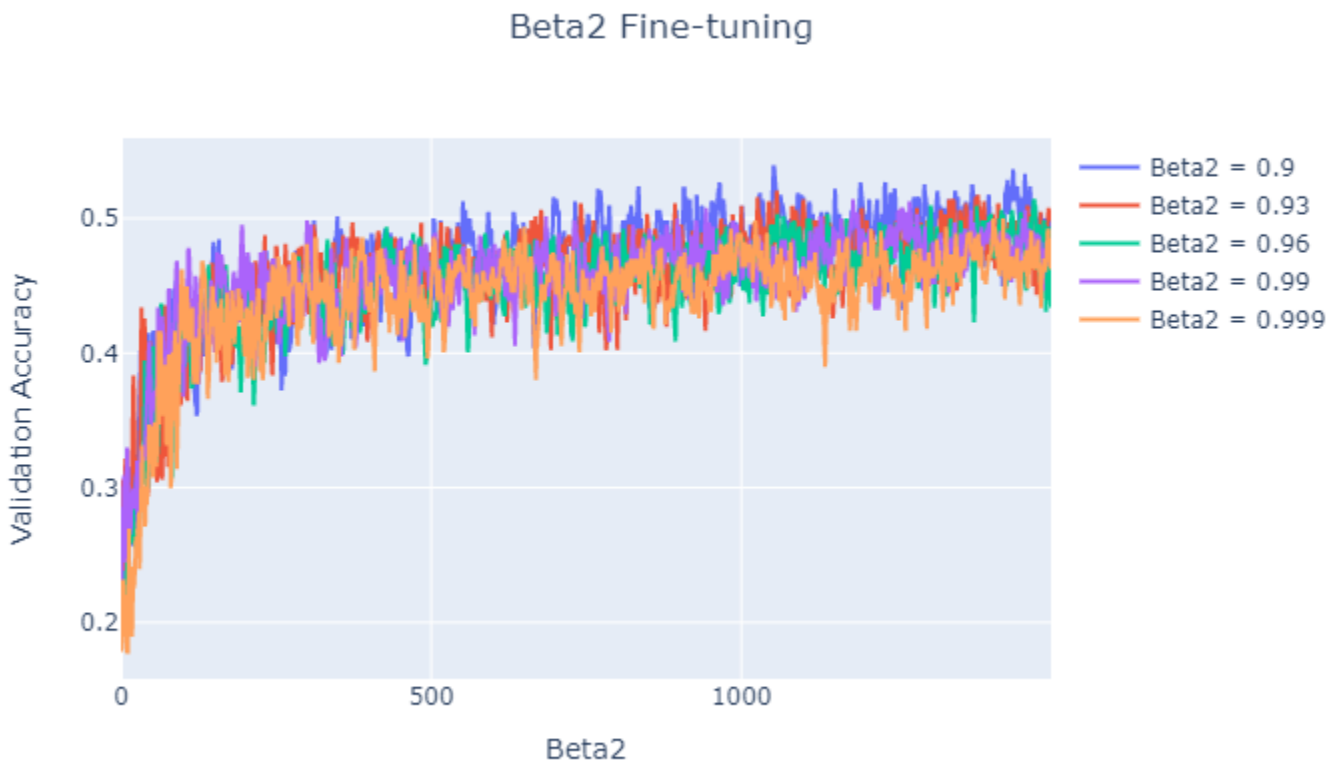
3.3.6. Beta1 Selection (for Adam):

The following graph shows results for tuning the beta1 parameter.

Results don't show a clear edge for any of the values tried (except for a slight edge to 0.95), so the most empirically trusted value of 0.9 was chosen.

3.3.7. Beta1 Selection (for Adam):

The following graph shows results for tuning the beta2 parameter.



Results don't show a clear edge for any of the values tried (except for a slight edge to 0.9), so the most empirically trusted value of 0.99 was chosen.

4. Epoch-based Selection:

In each epoch, the validation accuracy is evaluated and compared to the current best recorded validation accuracy. If it yields a better accuracy, the model weights associated with that epoch are saved replacing the previous best. After the last epoch finishes, the saved best weights are imposed to get the best model.

5. Final Model:

The following screenshot indicates the chosen parameters for the final model:

```
# Final model hyperparameters
TOPOLOGY = [X_train.shape[1], 64, 32, len(categories)]
BATCH_SIZE = 32
MAX_EPOCHS = 1500
ACTIVATION = 'relu'
LOSS_T = "nll"
OPTIMIZER = 'adam'
BEST_PER_EPOCH = True
L_RATE = 0.05
BETA1 = 0.9
BETA2 = 0.99
```

Below are the loss results of the final model:

We should take the model at the minimum validation loss (highlighted in the plot) as the model starts to overfit afterwards. Note that due to the relatively small batch size, the overfitting wouldn't be clear. To view it more closely, the following graph is the result of the same hyperparameters but for a batch size of 500:

The gap between the validation and training loss clearly shows the overfitting effect.

6. ANN vs. K-NN:

Below are the results of the optimized ANN vs. K-NN image classifiers on the same dataset (RGB), tested on the 500 test images:

ANN	K-NN
CRRn of class daisy = 45.00 % CRRn of class dandelion = 51.00 % CRRn of class roses = 54.00 % CRRn of class sunflowers = 76.00 % CRRn of class tulips = 34.00 % ACCR = 52.00 %	CRRn of class daisy = 15.00 % CRRn of class dandelion = 84.00 % CRRn of class roses = 20.00 % CRRn of class sunflowers = 56.00 % CRRn of class tulips = 17.00 % ACCR = 38.40 %

As expected, the ANN performs significantly better overall, but it's better at recognizing dandelions.