Practical Deep Machine Learning

Assignment 3 Report

Dr. Mohamed Mostafa
Fall 2020

Mohamed A. Abdel Hamed
900163202

Table of Contents

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 50x50 (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_CACHED_DATASET to load the dataset from the NPY files provided instead of loading all images again.
- The USE_CACHED_MODEL, if set, loads the pretrained model; otherwise, a fresh one is created.
- The TRAIN_MODEL, if set, trains the model on the training dataset before predicting the test set; otherwise, it proceeds to the test set right away.
- 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 = 1998) for reproducibility.
- **Important:** despite the fixed random seed, due to the use of Cupy (which involves non-deterministic operations), results may vary from run to run, and from device to device (depending on the used Cupy version)

2. Data Preprocessing:

- 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 Composition:

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:

After a fair deal of experimenting with the architecture, a few candidates stood out as the best performers. They all rely on the following types of layers:

I. **Conv2D (Convolutional layer):** uses convolution filters to extract features from input images. Since the vanilla implementation of convolution in Python is quite inefficient, a faster approach was inspired by https://github.com/SkalskiP.

II. **MaxPool2D (Max pooling layer):** used to downscale the input to be able to detect more general features.

III. **Flatten:** used to flatten the multi-dimensional output of convolutional and pooling layers to fit as input to the dense layers.

IV. **Dense (Fully connected layer):** used to perform the final stage in the classification process by combining extracted features into neurons.

V. **Dropout:** used to diminish overfitting by partially training the network once at a time.

VI. **ReLU:** produces a ReLU activation.

VII. **Softmax:** produces a Softmax activation used at the output layer.

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.

It's worth noting, however, that the network uses a lower learning rate for dense layers than convolutional layers, so it's not straightforward to pick a favorite single value for all layers. Thus, across many trials, a value of 0.04 was chosen for convolutional layers, and half that (0.02) was chosen for dense layers, with both getting exponentially in decay with each batch (more on that later).

3.3.3. Activation Function

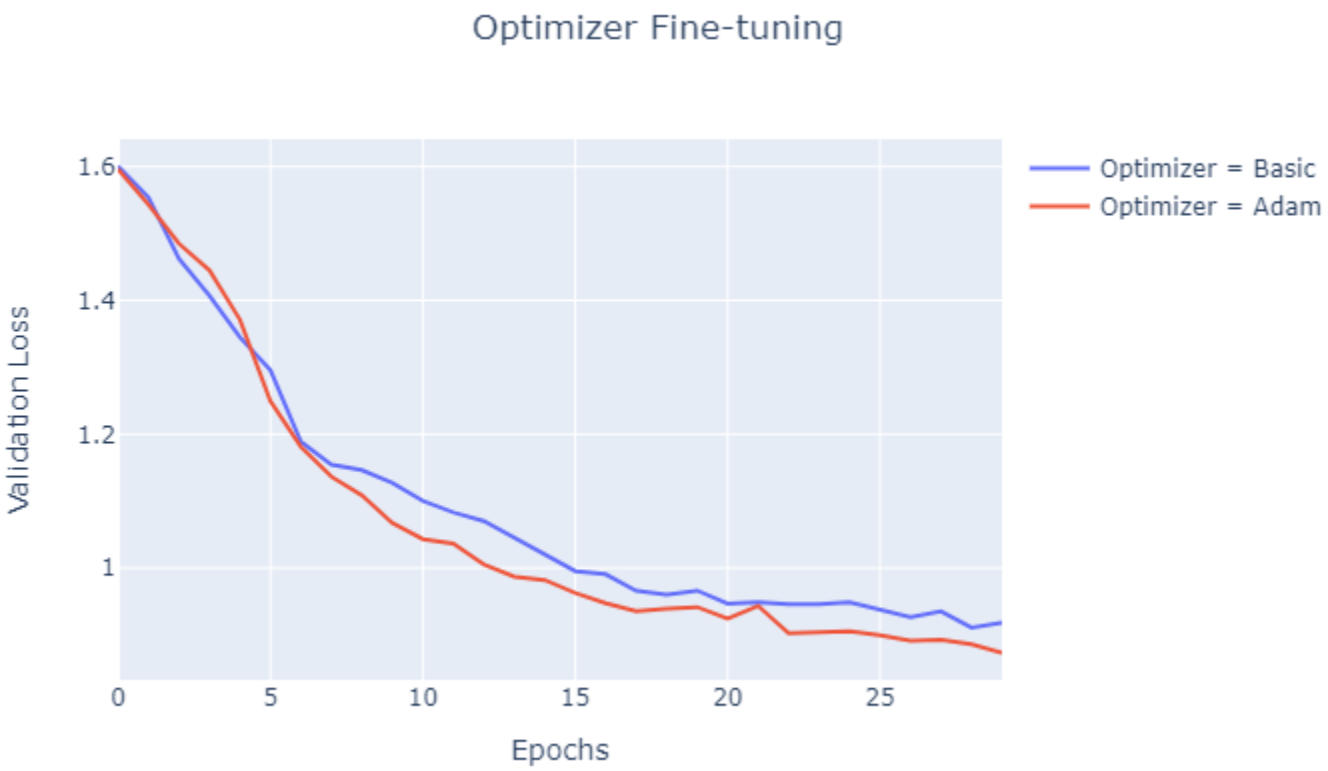
ReLU was chosen because of its analytical advantages (e.g. non-saturation, etc.) over sigmoid activation.

3.3.4. Loss Function:

Cross Entropy Loss was chosen as it is more commonly used in Neural Networks compared to Hinge loss, which is typically used with SVMs.

3.3.5. Optimizer Selection:

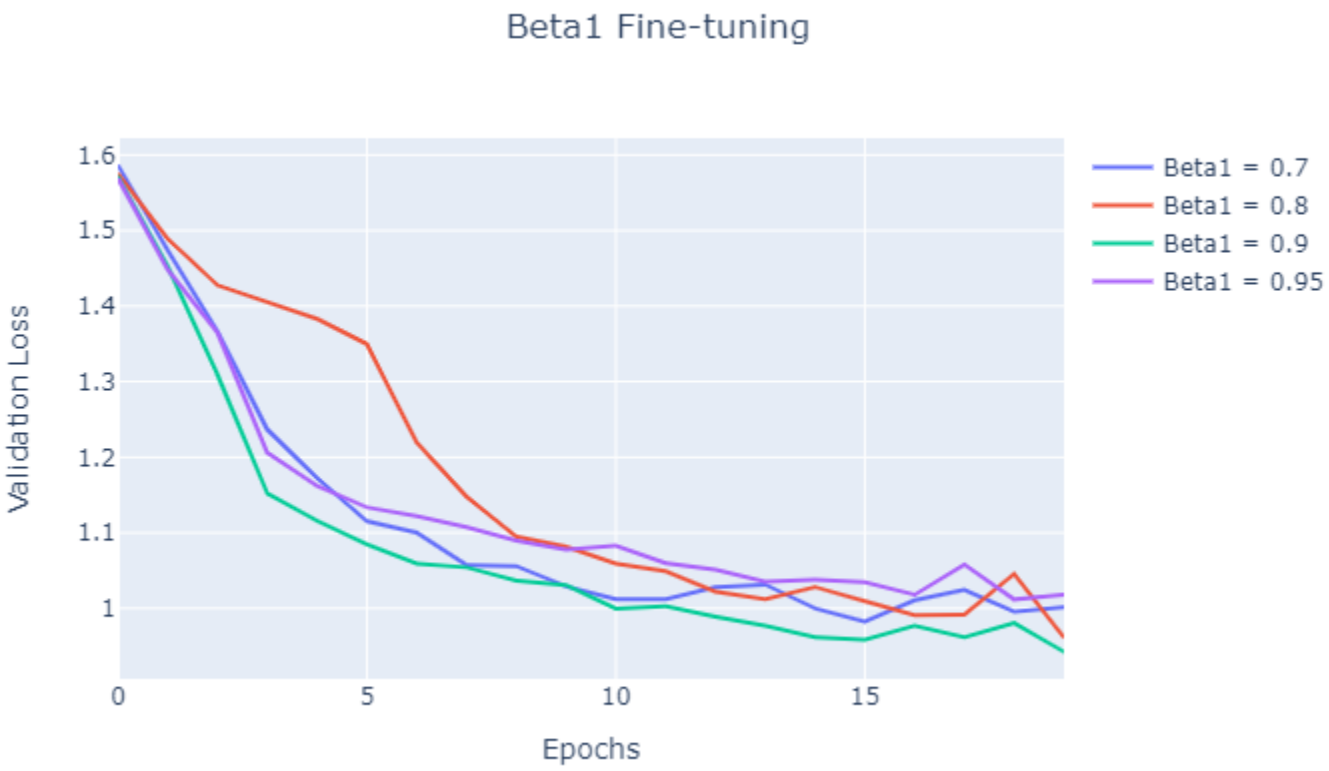
The following graph shows results for tuning the optimizer.



The results show a clear advantage for using the Adam optimizer over the basic one.

3.3.6. Beta1 Selection (for Adam):

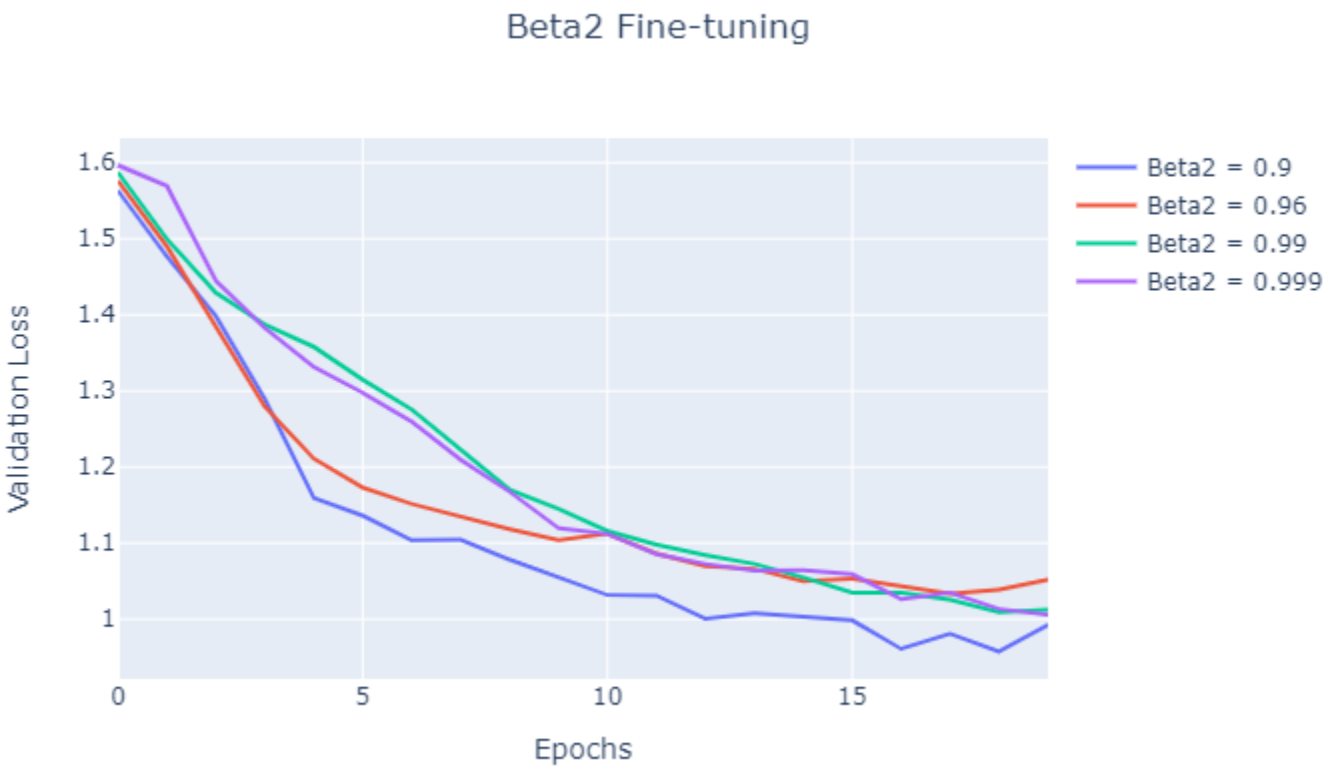
The following graph shows results for tuning the beta1 parameter.



Results show a clear edge for the most empirically trusted value of 0.9, which was chosen.

3.3.7. Beta2 Selection (for Adam):

The following graph shows results for tuning the beta2 parameter.



Results show an edge for the value of 0.9; however, 0.99 was observed to give better results when other hyperparameters are tuned.

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. Stopping condition:

The network controller stops the training when the validation loss does not improve for 30 (parametrized) consecutive epochs as a check for overfitting.

6. Learning rate decay:

An exponential decay factor was applied to the learning rate with each batch to force the gradients to make finer steps as the model matures.

7. Dynamic batch size upscaling:

Batch size is gradually increased as the validation accuracy exceeds certain limits to make sure next steps are more carefully taken.

8. Final Model:

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

```
# Final model hyperparameters
BATCH_SIZE = 32
MAX_EPOCHS = 300
LOSS_T = SoftmaxCrossEntropyLoss
OPTIMIZER = 'adam'
BEST_PER_EPOCH = True
L_RATE = 0.04
BETA1 = 0.9
BETA2 = 0.99
ALPHA = 0.01
```

Topology:

```
TOPOLOGY = [
    Conv2D(64, (3, 3), padding='same', learning_rate=L_RATE, lr_decay=0.99995),
    MaxPool2D((5, 5)),
    _ReLU_(ALPHA),
    Conv2D(64, (3, 3), padding='same', learning_rate=L_RATE, lr_decay=0.99995),
    MaxPool2D((5, 5)),
    _ReLU_(ALPHA),
    Conv2D(96, (3, 3), padding='same', learning_rate=L_RATE, lr_decay=0.99995),
    _ReLU_(ALPHA),
    Flatten(),
    Dropout(0.8),
    Dense(512, learning_rate=L_RATE/2, lr_decay=0.99995),
    _ReLU_(ALPHA),
    Dropout(0.7),
    Dense(512, learning_rate=L_RATE/2, lr_decay=0.99995),
    _ReLU_(ALPHA),
    Dropout(0.9),
    Dense(len(categories), learning_rate=L_RATE/2, lr_decay=0.99995),
    _SoftMax_()
]
```

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

9. CNN vs. FCN:

Below are the results of the optimized CNN vs. FCN image classifiers on the same dataset (RGB), tested on the 500 test images:

CNN	FCN
CRRn of class daisy = 78.00 % CRRn of class dandelion = 89.00 % CRRn of class roses = 48.00 % CRRn of class sunflowers = 87.00 % CRRn of class tulips = 67.00 % ACCR = 73.80 %	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 %

As expected, the CNN performs significantly better overall, but the FCN is slightly better at recognizing roses.