EE559 : Mini-projects

Students :
Lara Gervaise
Kenyu Kobayashi
Murat Topak

May 23, 2020

Contents

1 Project 1

1.1 Introduction

In this first project, we tested two main architectures of neural networks: **Fully connected neural networks** (FNN) and **Convolutional neural networks** (CNN). Both of the networks were designed to compare two digits appearing in a two-channeled image taken from the MNIST dataset. The goal of this project is to show in particular the impact of **weight sharing** and the use of an **auxiliary loss** to help the training of the main objective. The implementation is in Python, and utilizes the Pytorch library only.

1.2 Implementation

Firstly, it should be argued that CNN architectures cover more meaningful features than FNNs due to the nature of the convolution operation. Thus, CNNs are known to perform better than FNNs for classification tasks. In return for the improved accuracy on the task, CNNs take longer times to train compared to FNNs because even if they have fewer weights, the convolution operation is costly.

Secondly, for each of the CNN and FNN, we implemented four variations with and without **weight-sharing** and/or **auxiliary loss**. In the context of this project, **weight sharing** refers to the technique which consists in feeding both of our images into the same layers of the network, like in a **siamese network**. On the other hand, there is no weight sharing if within the same network, two images never go through the same layers. The **auxiliary loss** is meant to define two extra losses associated to the prediction of the digits whereas our main loss is always due to the final boolean value resulting from the prediction of the comparison of the two digits.

1.3 Expectations

Our expectations for these different variants were the following:

1. In addition to reducing the execution time, the weight sharing should improve the accuracy. The reason for that is that we were limited to only 1000 training samples. In that case, it made more sense for us to send two images to the same layer one by one instead of having them go through different layers and have more parameters to learn.
2. The auxiliary loss should increase the final accuracy since it allows the network to get more feedback during the training. It should also help to reduce the vanishing gradient problem by introducing additional losses in early layers.

1.4 Models

Each model received a two channeled-image in the forward pass, and fed each image to the same or different sequential sub-models depending on whether weight sharing was being used. While using the **ReLU** activation function for inner activations, the **Softmax** activation function at the output gave a probability distribution to predict the digit for each of the image.

Once a 10-dimensional distribution was generated for each image, they were concatenated to go through another sequential sub-model to compute the final prediction regarding the digit comparison problem. While the first set of submodels for digit prediction were either an FNN or a CNN, the final prediction was always made with an FNN. We used two hidden layers for digit prediction and another one for the final comparison.

We decided to implement **weight decay** and **batch normalization** (applied before each activation functions) as regularization methods for our models since they are the best suited for our not-so-deep models.

1.5 Hyperparameter Tuning

In order to find the optimal hyperparameters for our different models - that are CNNs and FNNs with and without weight-sharing and/or auxiliary loss - we adopted candidates for each hyper-parameter by revising previous works regarding the MNIST dataset. We fixed the number of epochs to 50, the batch size to 64 and considered the learning rate and weight decay as hyper-parameters. After this set-up we ran a grid-search with a 3-fold cross validation. Our learning rates varied between 0.001 and 0.015 with steps of 0.001. Concerning the weight decays, the candidates were the following: 0, 0.1, 0.01 and 0.001. We picked the best-performing learning rate and weight decay with respect to the cross-validation score on the training set.

1.6 Training & results

After fixing our models and hyperparameters, we chose the fast-converging **Adam algorithm** as our optimizer and trained our networks with the **Cross entropy loss** over 50 epochs. The training was done with **mini-batch gradient descent**, i.e. during one epoch we constantly consumed batches of 64 and update our weights after each mini-batch. Our results on the test set are presented in Table 1 to show the mean accuracy of each model after 10 run, along with the standard deviation and the execution time.

Architecture	Fully connected neural network				Convolutional neural network			
WS/AUX	-	WS	AUX	WS+AUX	-	WS	AUX	WS+AUX
Accuracy (%)	78.2±1.5	77.4±2.4	93.8±1.2	94.8±1.6	79.7±2.0	82.1±1.3	94.5±0.9	94.7±1.1
Execution time (s)	9.3±1.6	8.4±1.4	10.7±1.6	10.3±2.0	44.3±2.1	43.7±1.5	45.8±1.6	45.9±3.4

Table 1: Performance estimates through 10 rounds for each architecture

Our learning curves in Figure 1 and Figure 2 show convergence and non-overfitting. We found that further decreasing of the batch size provokes longer execution times while not giving significant improvement in terms of accuracy.

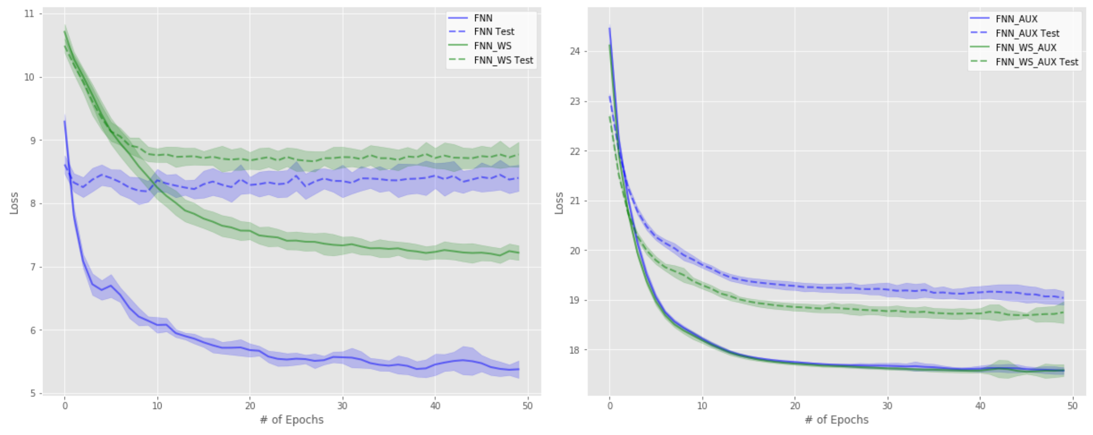

Figure 1: Training and testing loss for the different FNN models

Furthermore, we can observe that weight sharing and auxiliary loss **systematically improves** the performance of our models. However, we should note that the latter practice brought more significant improvement compared to the former one in terms of accuracy. This can be verified against the table by going from CNN to CNN_WS, or from FNN_AUX to FNN_WS_AUX.

Lastly, we show in Figure 3 the accuracy over the number of epochs for each of our models. Here we note that for simple models like FNN or CNN, the accuracy starts with a decent value and does not change much over the time. This is because at epoch 1, we already do many updates due to mini-batch gradient descent.

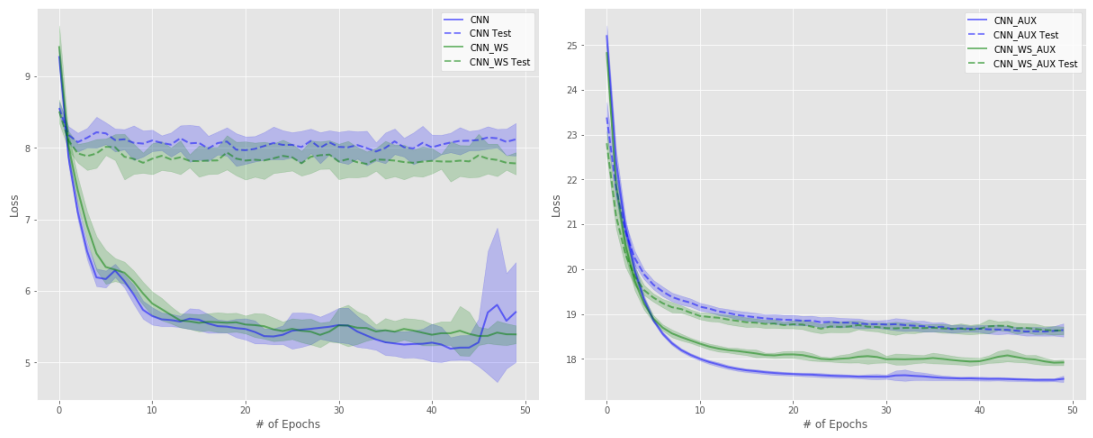

Figure 2: Training and testing loss for the different CNN models

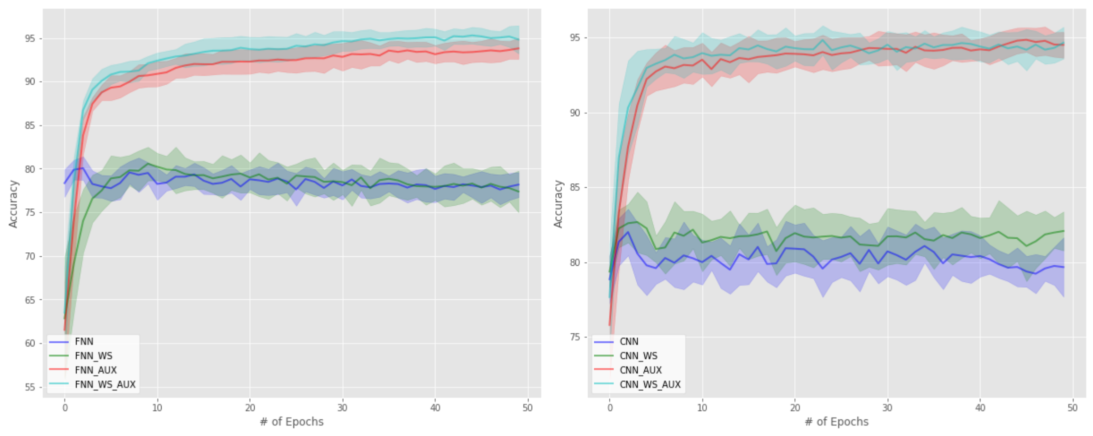

Figure 3: Training and testing loss for the different FNN models

1.7 Conclusion

In this project, we explored a variety of models for the digit comparison problem. Our starting point was to fix the models and investigate the effects of the weight sharing and auxiliary loss. Once we had our models, we fixed the number of epochs and the batch size for the gradient descent, and ran a grid search to tune the learning rate and weight decay with a 3-fold cross validation. We presented our estimates of the performances over 10 rounds of executions and noted the positive impact of the weight sharing and auxiliary loss. A possible extension of this project would be to investigate the effect of the number of hidden layers while fixing the number of neurons at each layer, or vice versa.

2 Project 2

2.1 Introduction

In this second project, we designed a mini deep-learning framework, using only pytorch's tensor operations and the standard math library. In particular, **no neural-network modules or autograd** was used.

2.2 Framework

The framework allows to build networks combining the following:

- **Fully connected layers**,
- ReLU, LeakyReLU, Tanh, Sigmoid and Softmax **activation functions**,
- **Regularization methods** such as batch normalization or the use of weight decay,
- Exponential, time-based and step-decay **learning rate schedules**,
- MSE or Cross entropy **loss functions** (the latter including the Softmax activation function),
- SGD, Adagrad, RMSProp or Momentum **optimizers**,
- Random, Zero, He or Xavier **initializations**.

2.3 Structure

Our implementation proposes several activation functions which can be combined with fully connected layers and batch normalization through the *Sequential* structure. The *Sequential*, *Linear*, activation functions - *ReLU*, *LeakyReLU*, *Tanh*, *Sigmoid* - and *BatchNorm* classes implement the *Module* interface with the following methods:

- *init*, the class's constructor method which initializes attributes when necessary,
- *forward*, used when doing the forward-pass,
- *backward*, used when doing the back-propagation,
- *step*, used to update the parameters following a back-propagation,
- *get_weights*, used as a getter for weights.

2.3.1 Sequential module

This module is the core of our framework. It allows to combine other modules to create a network. The *init* method takes as parameter a list of modules to set it as a class attribute. The *forward* method executes the forward-pass by feeding the input data to the first layer of the network, computing its output, feeding it to the next layer, and so on, until it computes the output of the last layer and returns it. The *backward* method works analogously as the *forward* method to execute the backward-propagation. It feeds the gradient of the loss to the last layer, and propagates it until it reaches the first layer. The *step* method updates all of the layers by calling the same method upon each module. Finally, the *get_weights* method returns the concatenation of all of the weights of the network. It is used inside the training procedure to compute the L2 penalty, which is then multiplied by the weight decay and added to the loss.

2.3.2 Linear module

This module creates fully connected layers. The *init* method takes as parameter the input dimension and output dimension of the layer, as well as an initialization method. The weights and biases of the layers are initialized according to this latter (by default, the random initialization is chosen). The *forward* method computes the output of the layer, with respect to the input, the weights and the biases. The *backward* method computes the gradient with respect to the gradient of the preceding layer in the backward-propagation. The *step* method updates the parameters of the layer with respect to the learning rate, the weight decay, and the optimizer (by default, the SGD optimizer is chosen). Finally, the *get_weights* returns the weights of the layer.

2.3.3 Activation function modules

These modules allows the use of the associated activation function in the network. The *init* method initializes the slope for the *LeakyReLU* activation function. The *forward* method computes the output of the activation function with respect to its input. The *backward* method computes the gradient of the activation function with respect to the gradient of the preceding layer in the backward-propagation. The other methods aren't used in these modules.

2.3.4 Batch normalization module

This module allows the use of batch normalization in the network. The *init* method initializes class attributes, such as the *gamma* and *beta* parameters (used respectively for scaling or shifting the normalization). The *forward* method takes the output dimension of the previous layer as its parameter. If the network is being trained, it uses the mean and the variance of the tensor to normalize the output. However,if the network is being tested, it uses the averaged mean and variance previously computed during the training. As in the other modules, the *backward* method computes the gradient of the batch normalization with respect to the gradient of the preceding layer in the backward-propagation. The other methods aren't used in this module.

2.4 Training

In order to train models, our framework provides a *train* functions, which takes as input the following: the train and test data targets, the number of epochs, the learning rate, the learning schedule type, the learning rate decay, the weight decay, the criterion, the batch size, and the optimizer. **All of these parameters can thus be tweaked to improve the performance of the given model**.

In addition to training the model, this function also takes as optional inputs *train_history* and *test_history* boolean values. If these are set to True, the function will return the training and testing history of the model over the epochs.

2.5 Evaluation

Last but not least, our framework provides functions in order to **evaluate** models easily. With the training and testing histories of a given model returned from the *train* function, complex plots can be generated through other provided functions.

By combining histories generated from several trainings, the *plot_one_loss* function plots the **mean loss** of the model over the number of epochs, with the associated standard deviation. In the same way, the *plot_one_acc* function plots the **mean accuracy** of the model over the number of epochs, with the associated standard deviation.

Finally, by combining several histories of several models, our *plot_all_loss* and *plot_all_acc* functions plot at once multiple subplots of all of the given models, in order to visualize and compare easily the performances of the different models. Examples of such plots are given in **Section 2.8 - Appendix**.

2.6 Results

With out framework, we built several networks with two input units, two output units and three hidden layers of 25 units. These networks were trained and tested on two different sets of 1, 000 points sampled uniformly in $[0, 1]^2$, each with a label of 0 if the point was outside the disk centered at (0.5, 0.5) of radius $1/\sqrt{(2\pi)}$, and 1 if the point was inside it.

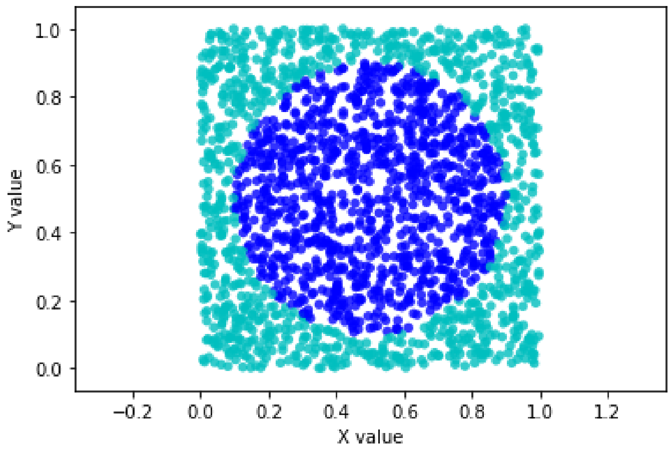

Figure 4: Distribution of the data on the coordinate system

We selected several models to test the performance of the different activation functions and ran a grid search to tune their following hyper parameters: the number of epochs, the weight decay, the learning rate, the learning schedule type and the loss criterion. Following this hyper parameter tuning, the the models were tested over their associated epochs, with the SGD optimizer and random weight initialization. In the appendix, plots are presented to show their training and testing loss / accuracies over the number of epochs.

2.7 Conclusion

The framework we built allows to create various networks, from very simple ones to more complex ones. The user has the ability to train his models with lots of optional parameters to choose from, and evaluate them through compact plots. With the dataset presented in **Section 2.6 - Results**, our model implementing ReLU activation functions for the early layers and the Sigmoid activation function for the prediction (combined with the cross entropy criterion) is the network that performs the best.

2.8 Appendix

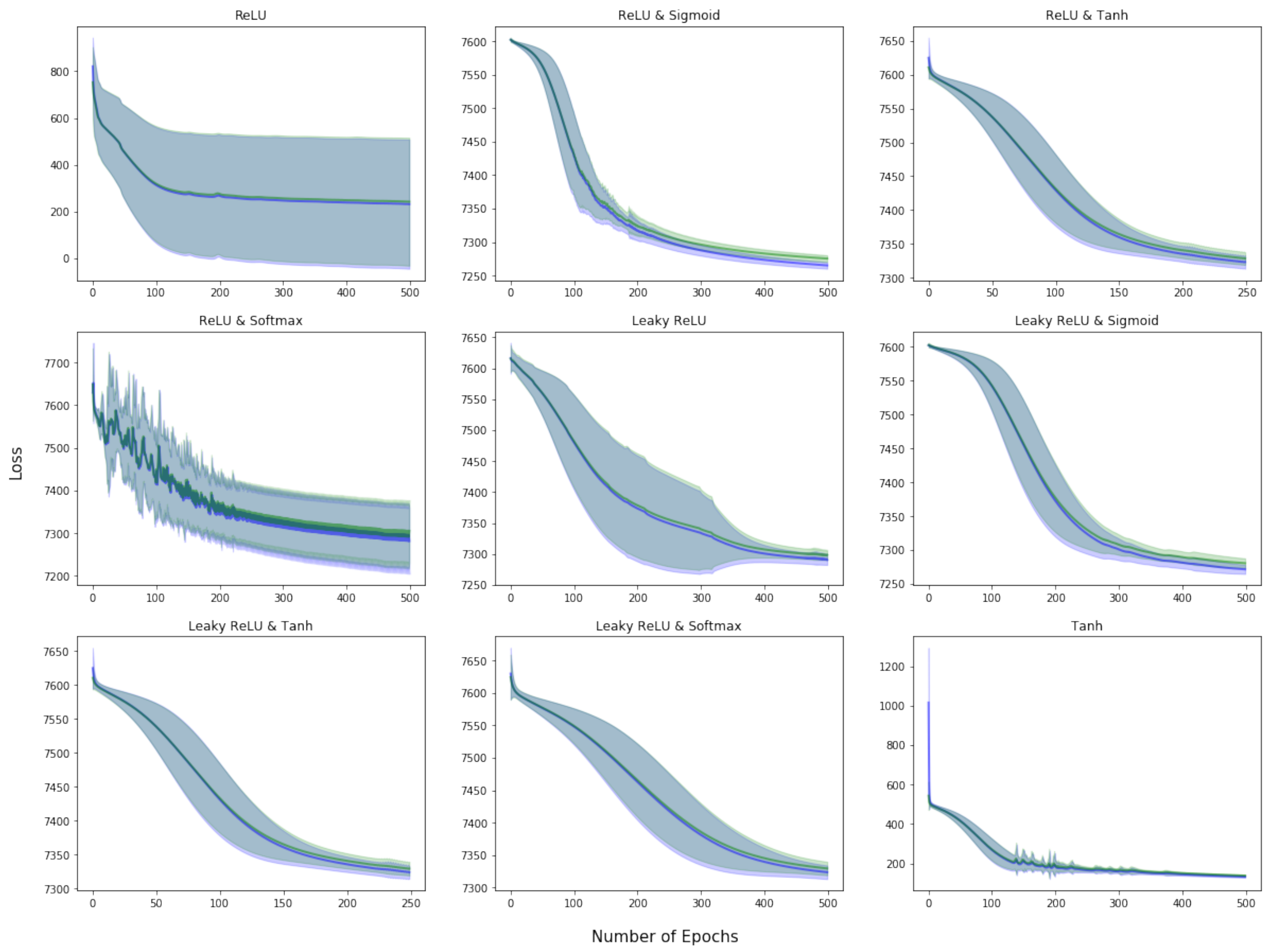

Figure 5: Training and testing loss for the different FNN models

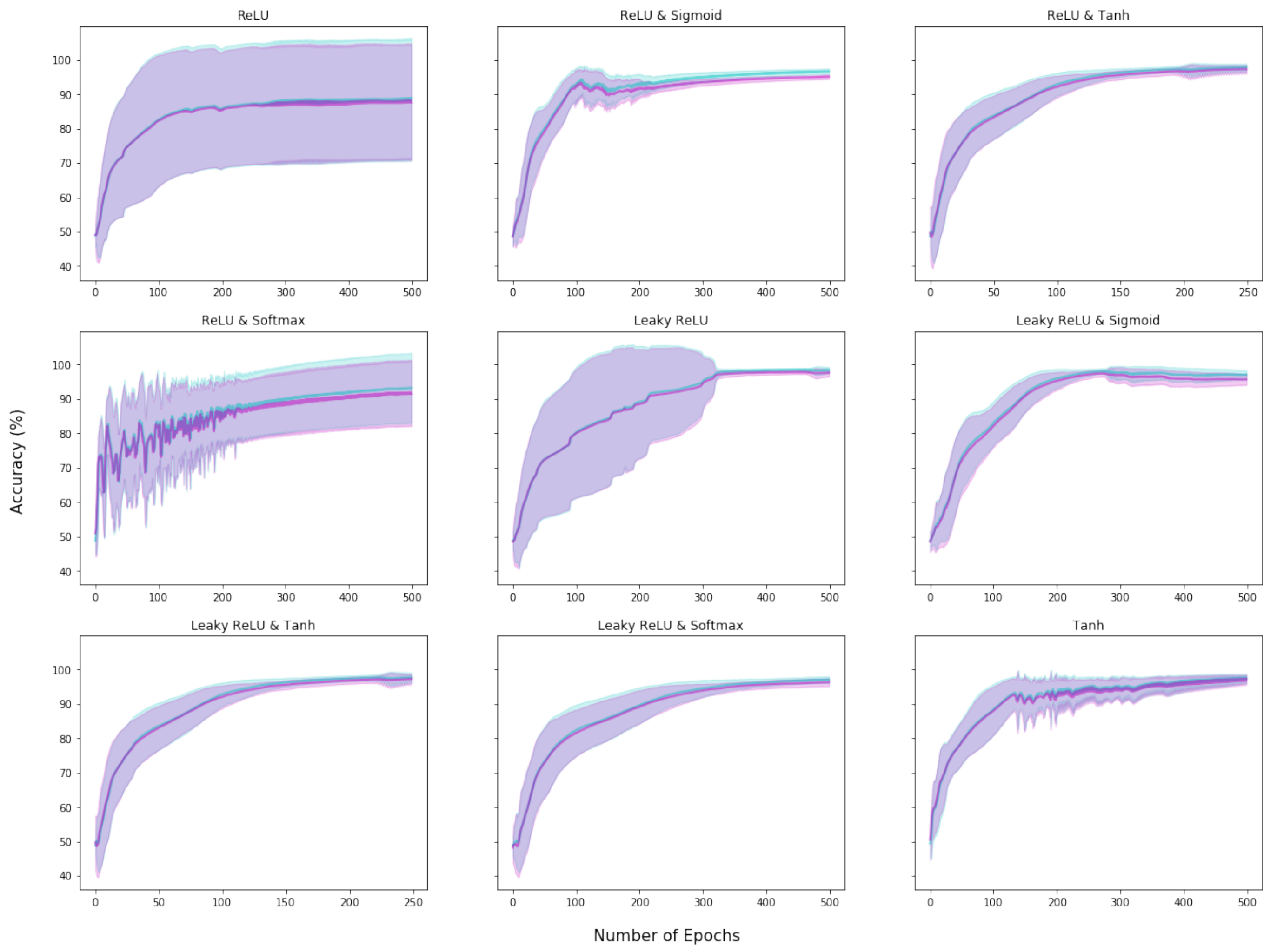

Figure 6: Training and testing loss for the different FNN models