VE593: Problem Solving with AI Techniques
Project III

Simona Emilova Doneva (J11803099007)

December 1, 2018

1 Introduction

The general objectives of the project are to get familiar with the workings of Machine Learning realized with an Artificial Neural Network approach.

2 Learning with ANN

This part explains the main components of the implemented neural network training solution.

2.1 Training

The training process includes several important concepts.
One *epoch* corresponds to one iteration over the whole training data set. The *batch size* establishes the number of updates that will happen in one epoch. In other words the batch size corresponds to the number of samples that the network will train on before the weights and bias values are updated. As suggested by [1] when splitting the data into mini-batches for an epoch, the samples were first shuffled. This approach has been observed as beneficial for a faster convergence.
The most important function during training is *update weights*. This function loops over the samples in the received batch, aggregates the computed gradients from back-propagation, and finally updates the weights and biases of the network. The updates are averaged by dividing the learning rate by the number of samples in the mini batch. Furthermore, before the update *regularization* might be applied to the weights (not the case if the regularization parameter is set to zero).

```
if l1_regularization:
    regularized_weights = old_weights - alpha *
                          (lmbda/self.N) * np.sign(old_weights)
else:
    regularized_weights = old_weights - alpha *
                          (lmbda/self.N) * old_weights
```

2.2 Feed-forward and Inference

The *Feed-forward* and *Inference* functions follow a similar logic, but serve different purposes. Both functions take an input observations through the layers of the network, applying at each step the linear regression function and the corresponding for the layer activation function on top of it. The *Inference* function returns the final activation results. The *Feed-forward* returns all results from applying the linear function and the corresponding activation function for every layer.

2.3 Back-propagation

Back propagation is at the core of learning the weights and biases of the prediction model. It has been implemented using the matrix and array operations provided by the *Python numpy* library. An overview of the logic is provided in Figure 1.

```
Data: Input sample x, corresponding target y
Result: WeightsGradients, BiasesGradients
layersDeltas = [];
wigthtsGradients = [];
layersActivations, layersOutputs = forwardPass(x);
prediction = layersActivations[-1];
lastLayerOutput = layersOutputs[-1];
lastLayerOutputDeriv = activationFunctionPrime(lastLayerOutput);
lastLayerDelta = dSquaredLoss(prediction, y) * lastLayerOutputDeriv;
layersDeltas[-1] = lastLayerDelta;
foreach layer i from (Last Layer - 1) to layer 1 do
    layerErrorCotrubutions = dot(network.weights[i+1], layersDeltas[i+1]);
    layerDerivatives = activationFunctionPrime(layersOutputs[i]);
    layerDeltas = layerErrorCotrubutions * layerDerivatives ;
    layersDeltas[i] = layerDeltas
end
foreach layer i from 1 to Last Layer do
    activation = layersActivations[i];
    delta = layersDeltas[i];
    weightsGradientLayer = delta * activation;
    wigthtsGradients[i] = weightsGradientLayer;
end
biasesGradients = layerDeltas;
return wigthtsGradients, biasesGradients
```

Figure 1: Backpropagation Implementation Pseudocode.

Important to note is that *activationFunctionPrime* calls the derivative of the corresponding activation function that was applied to the layer outputs. As this function can vary, the program will look up the corresponding function to the layer in the network settings and use the appropriate derivative function.

3 Hyper-parameters search

3.1 Baseline architecture

All experiments were performed using the whole training, validation and test data sets. To select the initial layer architecture the hyper-parameters were fixed as follows:

- learning rate = 1.0
- epochs = 5
- batch size = 1, i.e. no mini-batches

- regularization parameter = 0
- activation = sigmoid

With this setting the learning performance was tested when having no hidden layer, and a single hidden layer with different number of neurons. The reported performance is the average over three epochs.

Model	Performance		
Hidden Layer Width	Train Acc	Valid Acc	Test Acc
0	57.67	57.37	58.37
5	73.99	74.84	74.60
10	86.90	86.79	87.14
15	90.16	90.22	89.98
20	91.21	91.20	**90.27**

Table 1: Initial Performance Overview

It was observed that having a hidden layer greatly improved the performance of the algorithm. Therefore further parameter search was based on a network with one hidden layer with 20 neurons. More experiments with the layer architecture are reported in part two of the report.

3.2 Learning Rate

For this parameter, the number of iterations over the whole data set was fixed to five and again no regularization or batching was used.
Starting with a learning rate $\alpha = 0.01$ it was observed that increasing the the number improves the speed at which a higher accuracy is achieved. Therefore this number was successively increased until a value was reached where the accuracy started to oscillate or decrease. The results are shown in Figure 2.

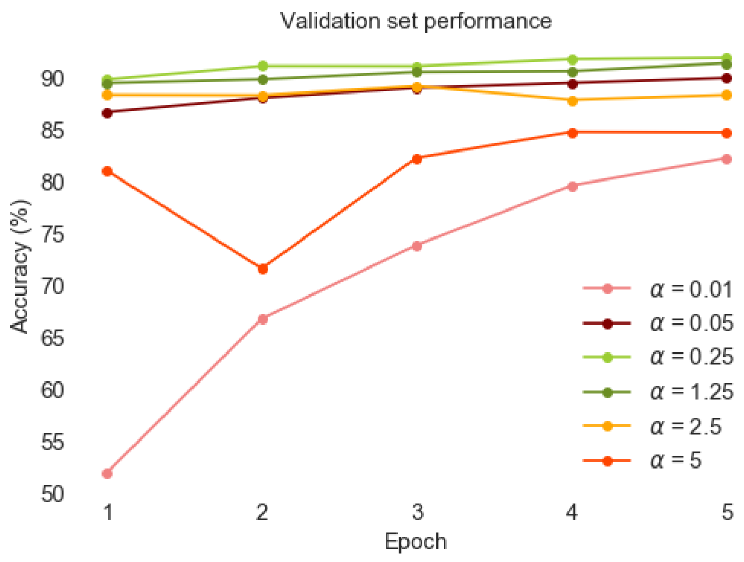

Figure 2: Learning Rate Impact on Validation Accuracy.

There are several insights that can be derived from the figure above. First of all it becomes evident that choosing a small order of magnitude for the learning rate slows down the gradient descent. Therefore a learning rate above 0.05 should be preferred. At the same time, if alpha was chosen too large, the performance becomes unstable and even worsens with increase in the number of epochs. This can be explained with the fact that with large α the steps that guide the algorithm towards the minimum of the cost function can *overshoot* the optimal solution.
Based on those observations a learning rate of 0.25 was selected as optimal given the other parameters fixed.

3.3 Number of Epochs

To determine the number of training epochs the technique of early stopping was utilized. That means that the training is terminated when the accuracy on the validation data set stops improving. To avoid stopping the learning too early this technique was implemented to interrupt the algorithm if didn't improve over ten subsequent epochs. Testing this approach using the currently established Network architecture reached training over 50 epochs, having the best classification accuracy of 93.73 % in epoch 40.

3.4 Mini-batch Size

In theory, the mini-batch size should above all impact the training time and not the performance in accuracy on test data. Therefore the search for the optimal value was based on comparing the validation error to the amount of training time [1]. To select an optimal batch size the validation accuracy is plotted against the time the network requires for training. The validation score is the average over five epochs for each batch size. Initial tests showed a dramatic drop in accuracy at first.

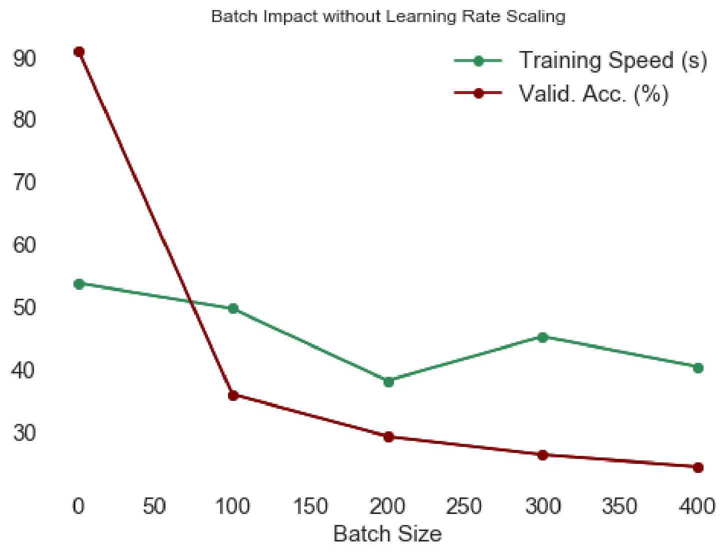

Figure 3: Batch Size Impact on Validation Accuracy and Training Speed.

This was explained with the fact that the learning rate had to be adjusted. As suggested in literature, the selected strategy for that was *"Linear Scaling Rule"*, describes as follows: *"When the minibatch size is multiplied by k, multiply the learning rate by k."* [2].

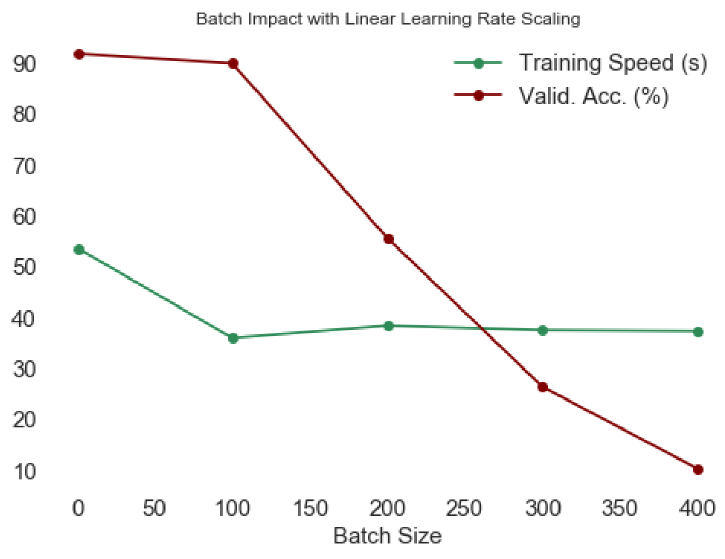

Figure 4: Batch Size Impact on Validation Accuracy and Training Speed.

Even following this strategy, the learning was worsened greatly with a batch size larger than 100. Therefore the search was continued for the range 1-100. The results are shown in Figure 5

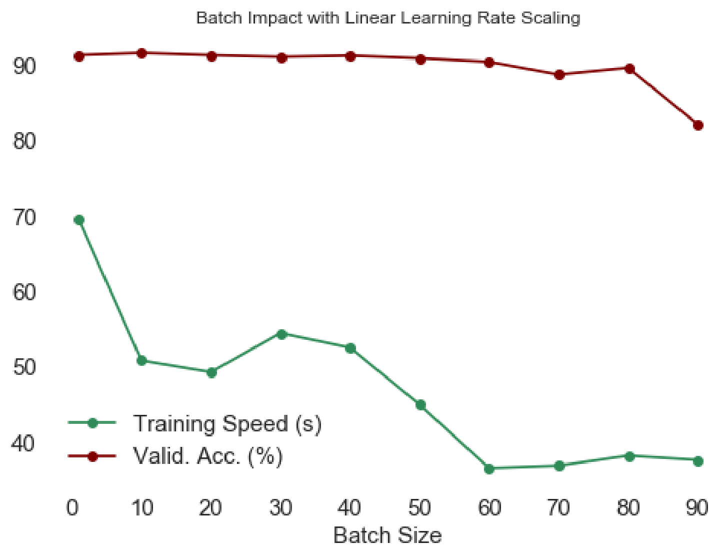

Figure 5: Batch Size Impact on Validation Accuracy and Training Speed.

While the best gain in speed was achieved with batch size of 60, there was also an observed decrease in accuracy. Therefore the batch size was set to 50 with a learning rate of 10.0. In this way a speed up of almost 50% with respect to the original 70 seconds was obtained without a sacrifice in performance.

3.5 Regularization

3.5.1 Over-fitting

Before proceeding with discussion on regularization, the need for this technique is motivated by showing the discrepancy in accuracy achieved by the model on training and test data. This difference is shown in Figure 5. The model is with the selected parameters until now (single hidden layer with 20 neurons, batch size of 50, learning rate 10.0, and no-improvement-in-ten rule for early stopping).

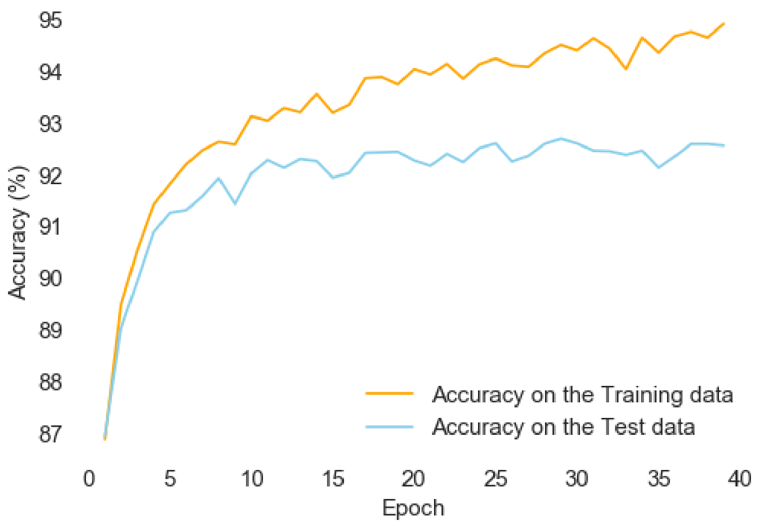

Figure 6: Accuracy Performance on Train vs. Test Data Set.

One can clearly observe that on the model behaves differently when evaluated against the training and the test data set. The accuracy associated to the training data gradually grows up to slightly below 95%. In contrast for the test data set the accuracy rises to just under 92%. After that the learning gradually slows down and the metric stops improving.

3.5.2 L2 Regularizaion

Using the previously chosen α the search for the regularization was started with λ = 1.0. Next this value was increased by factors of 10. The impact on the validation accuracy performance is plotted in Figure 7.

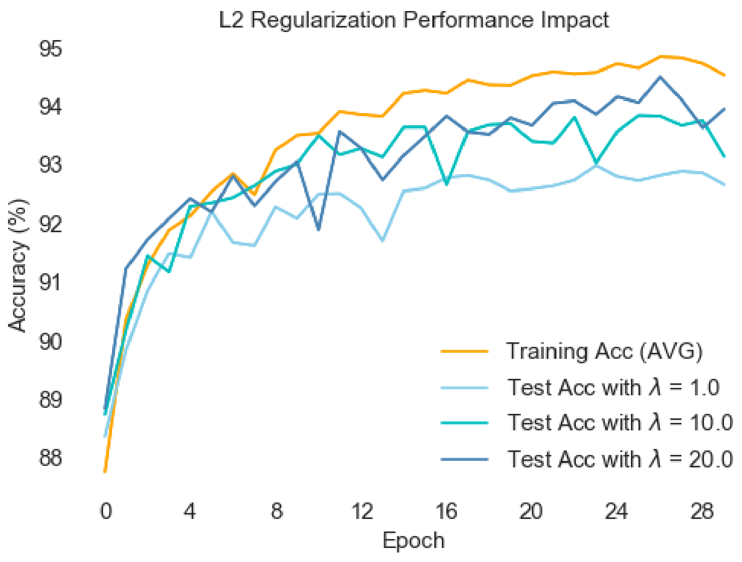

Figure 7: L2 Regularization impact with different lambda values.

One can observe that the accuracy on test and train data remain much closer together than when running unregularized.

3.5.3 L1 Regularization

As an alternative to L2 Regularizaion, L1 Regularizaion can be used to reduce over-fitting. While both techniques shrink the original weights, the latter approach reduces them by a constant amount toward zero (i.e. not proportional to the weights as it is for L2 norm) [3]. The effects of this approach applied to the original network architecture (fine-tuned for batch size and learning rate) are shown below.

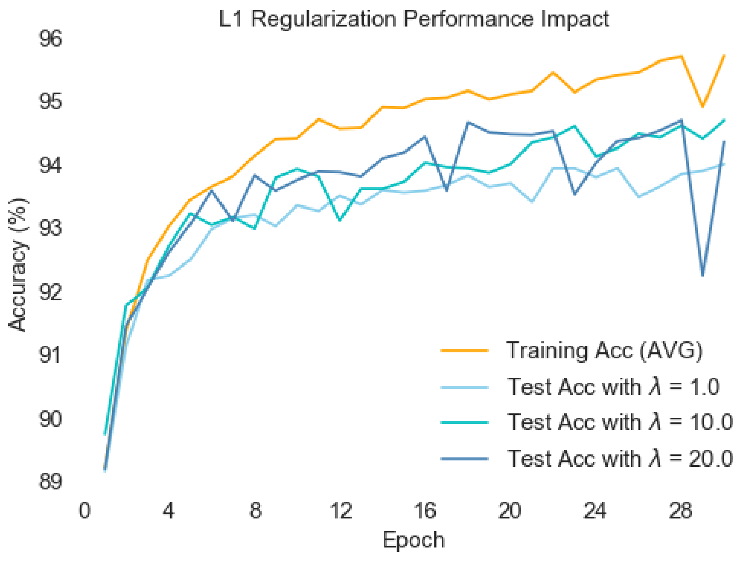

Figure 8: L1 Regularization impact with different lambda values.

3.6 Activation Functions

In this experiment different activation functions for the hidden layer were tested for their suitability to the task.

Model	Performance		
Activation Function	Train Acc	Valid Acc	Test Acc
sigmoid	95.57	94.66	**94.38**
tanh	92.09	91.64	91.30
relu	70.95	70.92	70.64
leaky relu	76.18	76.04	76.04

Table 2: Performance with different activation functions on the hidden layer over 30 epochs.

Figure 7 makes it evident that while after the 15th epoch all activation functions seem to yield similar performance, *relu* and *leaky relu* exhibit a much slower convergence rate.

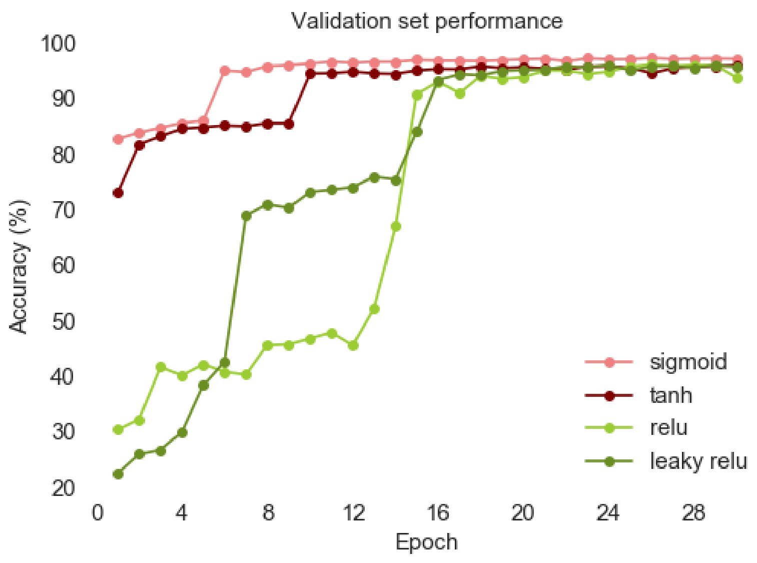

Figure 9: Performance over epochs using different activation functions.

3.7 Final Performance

Table 3 provides a comparison of the average performance achieved after fine-tuning the hyper-parameters to the initial performance shown in Section 3.1. Both models were trained over the whole data set over 50 epochs with early stopping turned on.

Model	Performance				
	Avg Train Acc	Avg Valid Acc	Avg Test Acc	Final Test Acc	Avg Training Time (s)
Basic	92.50	92.57	92.14	92.88	12.16
Fine-tuned	94.71	94.35	93.88	**94.77**	**9.17**

Table 3: Basic model vs. fine-tuned model performance comparison.

One can clearly observe the positive impact of fine-tuning the hyper-parameters. The new model not only achieves a higher test accuracy by almost than 2%, but also requires more less time to train.

4 Experiments with Different Architectures

4.1 Layer Architectures

The experiments shown in the first two following sub-chapters are executed using the hyper-parameter values established by the previous experiments, as follows:

- learning rate = 10.0
- epochs = determined by early stopping
- batch size = 50
- regularization parameter = 20.0
- activations = sigmoid

The idea of *multi-resolution search* [1] was followed, i.e. firs only a few values for the parameters in larger range were considered. After finding the best configuration, it was explored around this value in a more local interval.

4.1.1 Network Depth

Based on previous research, it has been established that using the same size for all layers generally shows a better or very similar performance compared to using a decreasing or increasing size [4]. Therefore the experiments were performed by keeping the number of hidden neurons the same over all layers.

Model		Performance			
Depth	Hidden Layer Width	Train Acc	Valid Acc	Test Acc	Avg Training Time
1	15	94.00	93.61	93.06	8.11 s.
2	15, 15	94.25	93.76	**93.55**	9.83 s.
3	15, 15, 15	94.01	93.60	93.13	12.03 s.
4	15, 15, 15, 15	92.65	92.57	92.15	14.25 s.
5	15, 15 , 15, 15, 15	91.78	91.80	91.37	16.70 s.

Table 4: Impact on performance of network depth.

One can observe that there is a slight improvement in performance when introducing one additional layer. However increasing the depth further shows to have no or negative effects. This might be due to the fact that the model has too many free parameters as compared to the complexity of the task.

4.1.2 Hidden Layer Width

This set of experiments was performed when keeping a single layer, but varying the number of hidden units it contains. At first it was established that increasing the number of hidden units to more than 100 led to a worsened performance (test accuracy around 75% with 200 neurons). Therefore the range below that number was explored.

Model	Performance			
Hidden Layer Width	Train Acc	Valid Acc	Test Acc	Avg Training Time
20	95.16	94.58	94.23	8.35 s.
30	96.63	95.83	**95.47**	10.48 s.
40	96.13	95.11	94.85	11.65 s.
50	96.64	94.75	95.44	12.75 s.
60	95.41	94.42	94.11	14.50 s.
70	93.39	92.42	92.06	15.50 s.
80	97.08	95.79	**95.64**	17.20 s.
90	95.01	93.91	93.69	19.98 s.

Table 5: Average performance with different layer width over 40 epochs.

It becomes evident from the table that increasing the number of neurons in the hidden layer has a positive impact on the model performance. At the same time the gap between the achieved accuracy on training and testing data increases, which indicates a higher degree of over-fitting for more complex models.
Investigating the outputs in more detail, it was further found that by having a larger number of neurons, the performance of the network was poor in the early stages on training, while it outperformed the simpler models when the number of epochs increased.

4.2 Hybrid Models

Finally experiments were performed to see whether having two layers of larger width would yield a positive impact on performance. The setting was having 50 neurons in two layers, and fine-tuning other hyper-parameters based on the gathered experience. However, no significant improvement was observed and therefore no further investigations were performed.

4.3 Conclusion

It has been observed that training a neural network can be challenging task and there are several other directions that could be considered for further research. The model showed to be most sensitive hyper parameters learning rate as well as the regularization parameter. Possible improvement strategies are applying a scheduling mechanism for the learning rate, and make it adapt as training evolves over the epochs. For regularization strategies such as *dropout* and *batch normalization* could further be implemented.

References

[1] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer, 2012.

[2] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[3] Michael A Nielsen. *Neural networks and deep learning*, volume 25. Determination press USA, 2015.

[4] Hugo Larochelle, Yoshua Bengio, Jérôme Louradour, and Pascal Lamblin. Exploring strategies for training deep neural networks. *Journal of machine learning research*, 10(Jan):1–40, 2009.