Report for the Deep Learning Course Assignment 1

George Chouliaras
g.chouliaras@student.vu.nl

Abstract

In this study, is desribed the procedure of familiarizing with feed forward neural networks for image classification. For this task we use the CIFAR10 dataset which contains 60000 32×32 color images, equally divided into 10 classes. In the first task, we obtain experience with the theory behind the fundamentals of neural networks and then we create from scratch a Multilayer Perceptron (MLP) using only Numpy routines. In the second task we create again the same MLP but this time in Tensorflow. This implementation is both easier and faster. Additionally we tune the MLP in order to achieve the highest possible accuracy on the test set. In the third task, we become acquainted with Convolutional Neural Networks (CNNs) which are slightly more complicated than MLPs. The CNN achieves much higher test accuracy than the MLP something that justifies its popularity in image classification tasks.

1 Task 1

In this task, we create from scratch a multilayer perceptron using only Numpy routines. In the pen and paper exercises we study the theoretical aspects of the MLP, while in the programming part, we apply the theory in practice in order to create a working MLP for image classification. The main challenges of this task are the following. Firstly, the feed forward neural network must be created in a dynamic way, meaning that it should allow for any number of hidden layers and any number of nodes. Secondly, special attention should be given in the computation of the gradients, since a small mistake can be propagated through the network as it is trained. This is what makes neural networks difficult in debugging.

After successfully implementing the MLP in Numpy, we plot the total loss (cross-entropy plus regularization) every 100 iterations of training. We mention that in the computation of the cross entropy we divide by the batch size. The total loss can be seen in Figure 1.

Figure 1: Total loss of MLP in Numpy for 1500 iterations.

We observe that the total loss is in general decreasing, however there are some fluctuations. Note that these results, refer to the implementation using the default parameters, e.g. learning rate 0.002, batch size 200, 1500 iteration steps, weight decay 0, weight scale 0.0001 and 1 hidden layer with 100 nodes. Except from the loss, we also keep track of the accuracy in the whole test set, every 100 iterations. These results can be seen in Figure 2. In this figure, we see that the accuracy on the test set increases with the number of iterations, however there are also here some fluctuations. With the default settings we achieve an accuracy around 46% in the last iterations.

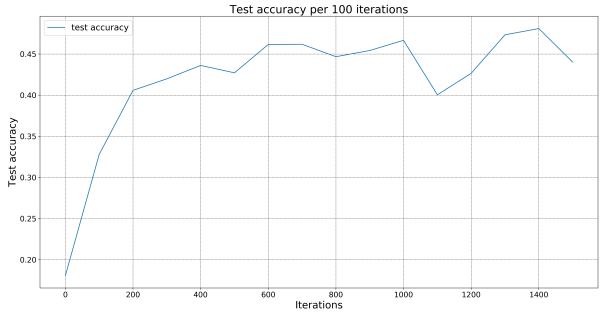

Figure 2: Accuracy of MLP in Numpy in test set every 100 iterations.

2 Task 2

In this task we create again an MLP as in task 1, but this time using Tensorflow. The main challenge of this task is getting familiar with Tensorflow as well as Tensorboard. Since Tensorflow provides all the necessary functions to create the necessary components of the network, building an MLP in Tensorflow is easier than in Numpy. Moreover, after the successful implementation of the network, it was found that the required running time is much lower when we use Tensorflow, compared to Numpy. This is mostly due to the fact that Tensorflow contains optimized functions for computing the gradients, something that makes the implementation much faster.

After implementing the model, we can visualize our graph in Tensorboard. This functionality provides a visual representation of the flow of the tensors within the network. The graph of the MLP implementation with 1 hidden layer in Tensorboard is depicted in Figure 3.

In this graph, we can see the flow of the input tensors, the weights for each layer, the layers themselves and the functions which compute the loss, the accuracy in train and test set, the confusion matrix and also the function for the training of the network. After successfully implementing the MLP, we tune its hyper-parameters so as to improve its predictive performance. To achieve that, we apply the following procedure.

Firstly, we tried different initialization methods, namely random normal, xavier and uniform initialization while keeping the rest of the hyper-parameters fixed. The one that seems to work best in this case is the random normal initialization with mean 0 and standard deviation 10^{-4}, since this is the one that achieved the highest accuracy in the test set.

Furthermore, we tried different activation functions, namely rectified linear unit (ReLU), exponential linear unit (elu), tanh and sigmoid. The best results were observed using the ReLU activation, something that justifies the fact that this is nowadays the default activation since it performs better than the other in most cases. With activations such as the sigmoid and the tanh, we observed that after some iterations, the accuracy remains the same and does not improve further. This is due to the fact that these activations saturate at extreme values, having as a consequence the gradients to be zero. However, saturation must be avoided at all costs, since zero gradients mean no learning and hence, no improvement in the accuracy.

Moreover, we tried different weight regularization types, such as ℓ_1 regularization, ℓ_2 regularization and no regularization. We observed similar results with ℓ_1 and ℓ_2 regularizations, with slightly better accuracy using ℓ_2 regularization. We then tried many different values for the weight regularizer

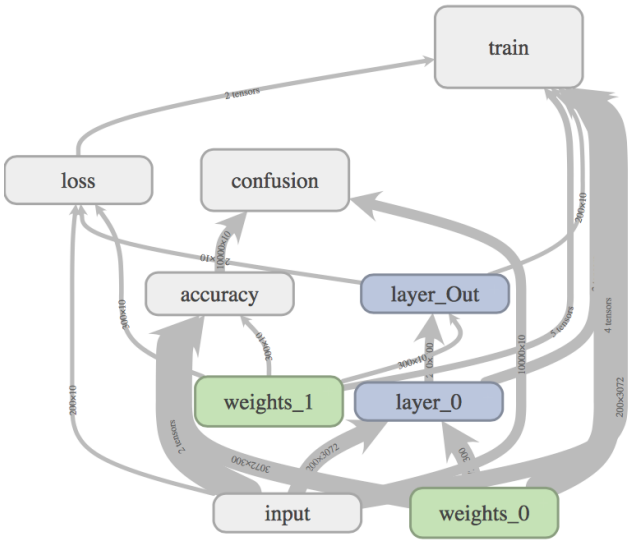

Figure 3: Graph of MLP in Tensorboard.

strength. We started by using the default value of 10^{-4} and then we increased it until the accuracy does not improve further. The best results were produced by using the value 2.3 for the weight regularizer strength. When we tried to use no regularization (and weight regularizer strength 0), we observed that the test accuracy at some point plateaus and does not make any improvement. This is probably due to overfitting and indicates the significance of adding regularization in the models.

Next, we tried different optimizers in order to compare them. More specifically, we used Stochastic Gradient Descent, Adadelta, Adagrad, Adam and RMSProp. After experimenting with the optimizers, the first observation was that the optimizers have probably the largest impact in the accuracy, among the other hyper-parameters. The worst results were produced by the SGD optimizer, while the highest accuracy was observed using Adam. Good results were also observed with Adagrad and RMSprop. For the Adam optimizer, we tried different values for its parameters β_1 and β_2 and ϵ. The best results were produced using $\beta_1 = 0.91$, $\beta_2 = 0.999$ and $\epsilon = 10^{-4}$. Note that the selection of ϵ is largely connected to the selection of the learning rate. Using $\epsilon = 10^{-4}$ and learning rate $2 \cdot 10^{-5}$ worked well in our case.

After tuning the aforementioned hyper-parameters we started increasing the number of iterations and the number of nodes in the hidden layer (one parameter at a time) until there are no further improvements in the accuracy. For the hidden units, 300 hidden nodes produced the highest accuracy, while more nodes did not improve more. When it comes to the max steps, 7000 iterations produced the best results, although the improvements in the accuracy are getting smaller as the number of iterations gets larger.

Finally, we also tried different batch sizes and more hidden layers. It was found that larger batch sizes did not bring significant improvements and hence we choose to use the default value (200), in order to have a faster implementation. Furthermore it was observed that by adding more hidden layers, the accuracy drops significantly. This means that when we change the architecture of the MLP, we have to re-tune all the parameters, as it strongly affects the behaviour of the network. In Table 1 we summarize the values for the hyper-parameters that produced the best results.

Using the parameters in Table 1 we achieve an accuracy around 0.54 in the last iterations. We should note that due to time limitations, we could not check for every possible value for all the hyper-parameters and hence these parameters are not optimal. Furthermore, better results can be achieved by introducing dropout, because it significantly decreases overfitting and also makes the implementation faster, since not all the neurons are used at the time of training. In Figure 4 we see

Table 1: Hyper-parameter values for the MLP with the highest test accuracy.

Hyper-parameter	**Value**
learning rate	$2 \cdot 10^{-5}$
optimizer	Adam ($\beta_1 = 0.91$, $\beta_2 = 0.999$, $\epsilon = 10^{-4}$)
max steps	7000
batch size	200
hidden layers	1
hidden units	300
weight regul. stregth	2.3
dropout rate	0
weight initialization	$\mathcal{N}(0, 10^{-4})$
weight regularization	ℓ_2
activation	ReLU

the values of the loss, every 100 iterations of training. It can be seen that the loss has a downward trend, however it also fluctuates as in the Numpy MLP. Moreover, we see that the decrease is larger in the first iterations and gets smaller as the number of iterations increases.

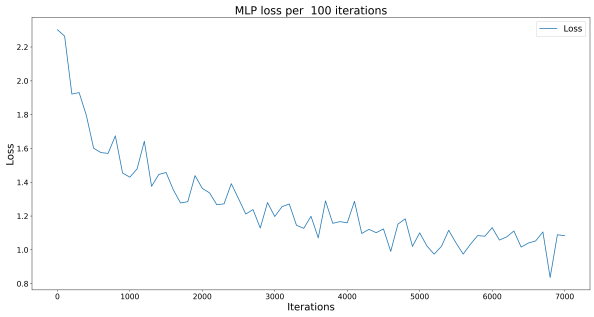

Figure 4: Total loss of MLP in Tensorflow for 7000 iterations.

Figure 5 depicts the accuracy in the whole test set, every 100 iterations of training. The increase in accuracy is sharper in the first iterations, and gets smaller as we do more training steps. In the last iterations the network achieves around 0.54 accuracy in the test set.

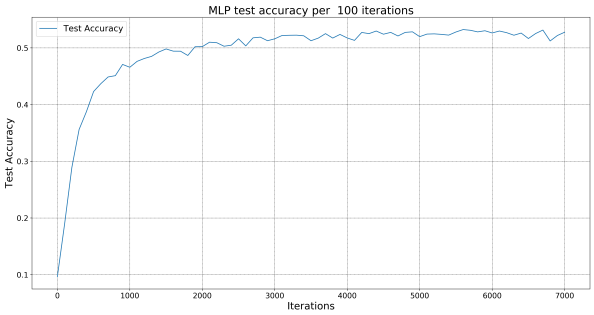

Figure 5: Test accuracy of MLP in Tensorflow for 7000 iterations.

Using the model with the best parameters, we also plot a confusion matrix which can be seen in Table 2. The columns represent the predicted labels, while the rows indicate the true labels. In the diagonal we can see the number of correct predictions for each class. We can observe that class 5 is the one with the least correct predictions. The main reason for this, is that 251 images of class 5,

were wrongly predicted as class 3. On the other hand, the class with the most correct predictions is class 1.

Table 2: Confusion matrix for the MLP with the best parameters

		Predicted class									
		0	1	2	3	4	5	6	7	8	9
True class	0	**583**	47	84	29	28	21	19	32	103	54
	1	40	**679**	15	25	16	7	26	15	51	126
	2	73	24	**416**	109	108	82	93	57	16	22
	3	26	24	84	**413**	57	175	109	48	17	47
	4	38	17	119	89	**441**	53	113	80	21	29
	5	20	12	84	251	76	**403**	58	59	14	23
	6	14	35	55	118	78	51	**596**	15	13	25
	7	31	26	45	85	88	73	20	**592**	10	30
	8	92	101	21	33	22	17	5	10	**644**	55
	9	31	219	21	42	16	17	22	32	58	**542**

In order to have a better idea about our predictions, we also visualize some examples for classes 0, 1 and 2 for which our model made confident wrong predictions. To achieve that, we do the following. Firstly, we find the indices of the wrong predictions per class. Then, for each class, we sort the predictions in order to find the 8 images with the highest probabilities per class. Finally, we find the indices for these images and we plot them, as can be seen in Figures 6, 7 and 8.

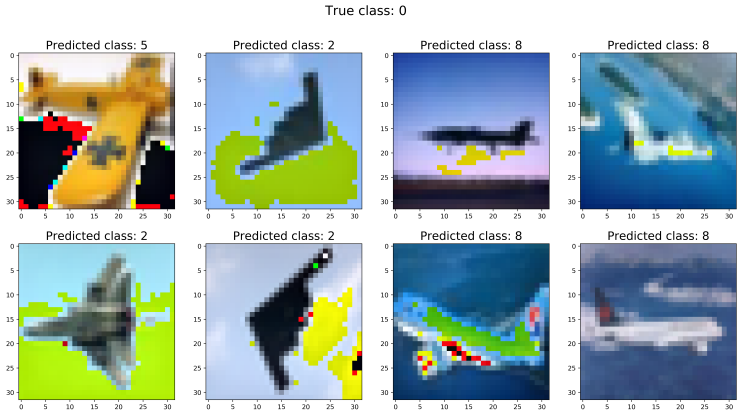

Figure 6: Confident wrong predictions for objects of class 0.

3 Task 3

In this task, we create a convolutional neural network in Tensorflow, in order to achieve better results in the image classification task at hand. After implementing the given architecture in Tensorflow, we can visualize our ConvNet in Tensorboard. The graph of the model is depicted in Figure 9. The reason why we see two sets of convolutional layers and fully connected layers, is because we used the same variables for training and test set (we reuse them).

After successfully implementing the ConvNet, using the default parameters (learning rate 10^{-4},batch size 128, max steps 15000, Adam optimizer with default parameters, random normal weight initialization, weight decay 10^{-3}) we achieve an accuracy around 0.7 after 4000 iterations.

Figure 10 depicts the loss of the ConvNet every 100 iterations of training. We observe that the loss reaches lower values than in the case of MLP. We can also see that the loss fluctuates greatly as it decreases.

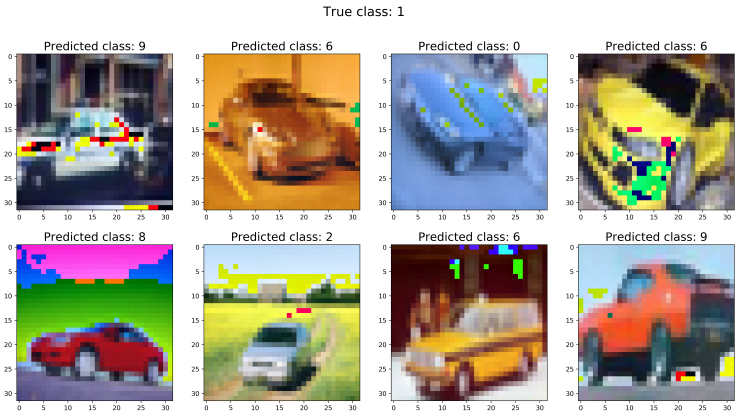

Figure 7: Confident wrong predictions for objects of class 1.

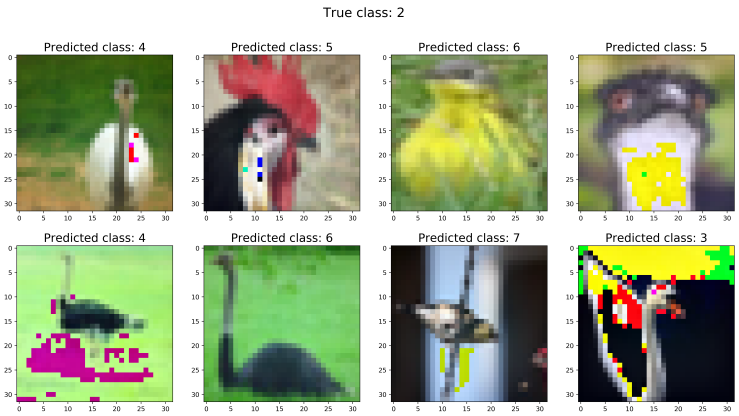

Figure 8: Confident wrong predictions for objects of class 2.

During the training process, we also keep track of the accuracy on the train set as well as the accuracy on the whole test set. The results can be found in Figure 11. In this figure, we can see that the test accuracy has a smoother increase than the train accuracy, which fluctuates largely. Moreover, we can observe that after 2000 train steps, the training accuracy gets larger than the test accuracy. At iteration 4000 we see that while the test accuracy is around 0.7, the train accuracy is around 0.8. This indicates that the model probably starts to overfit. To prevent that, we should apply stronger regularization.

Due to time limitations, we could not optimize the hyper-parameters to increase the accuracy. However, following a similar procedure as in task 2, we can find which parameters work best for the model and can achieve the highest predictive performance. Note that the tuning of the ConvNet is more tedious than in the case of MLP, since the training of the ConvNet requires more computational resources and hence more time. Despite that, we see that even with default parameters, the ConvNet achieves much better results than the MLP, something that explains why CNNs are widely used for image classification.

Further improvements can be achieved by applying batch normalization in the inputs of each layer. With BN we first normalize the input with zero mean and variance 1 and then we rescale and shift. This technique results to stronger gradients and hence in faster training since we can use higher learning rates. Batch normalization can be applied in Tensorflow using the function `tf.nn.batch_normalization`.

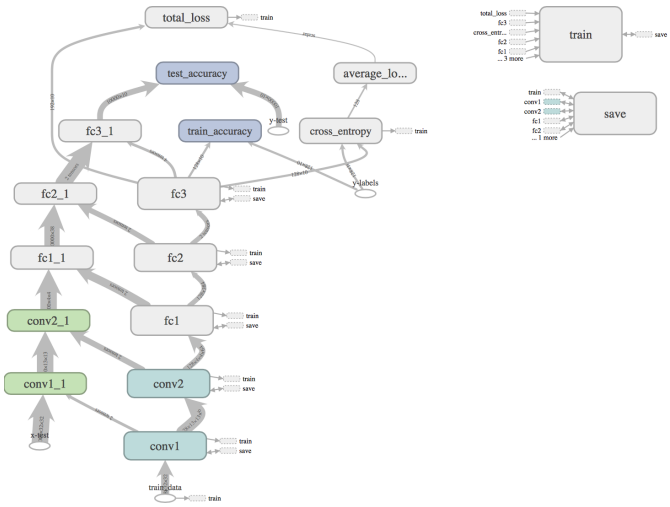

Figure 9: Graph of the ConvNet in Tensorboard.

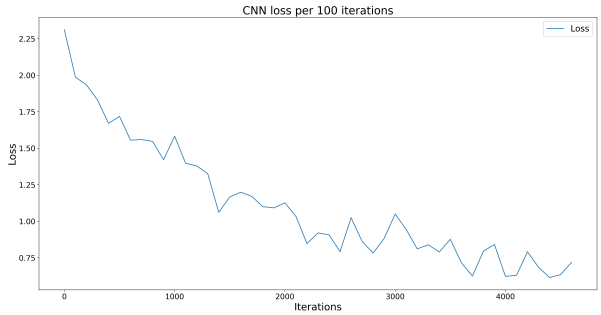

Figure 10: Loss in ConvNet every 100 iterations.

4 Conclusion

In this study, we got acquainted with feed forward Neural Network architectures namely Multilayer Perceptron and Convolutional Neural Networks and we applied them to image classification using the CIFAR10 dataset. Solving the pen and paper exercises provided a good understanding of the theoretical foundation of NNs and fundamental techniques such as backpropagation. The implementation of the MLP in Numpy was tedious, mostly due to the difficult debugging, but it resulted to a solid understanding of the MLP functionality.

In the second task we used Tensoflow in order to create an MLP. After the implementation of the model in both Numpy and Tensorflow, it is quite clear that Tensorflow makes things much easier, as it includes routines for almost every possible aspect of the Neural Networks. Getting familiar with Tensorflow was quite cumbersome. However, after getting used to it, one can realize that it is pretty straightforward. Moreover, in the second task we gain experience of the network tuning process. The main difficulty here is the large number of hyper-parameters that can be tuned. After the tuning process, we realize that regularization plays a very important role, different optimizers affect highly the predictive performance and ReLU is probably the best activation so far. Additionally, when changing the architecture of the network (adding more layers) all the parameters need to be re-tuned.

In the third task we get familiar with the more advanced CNNs. These are slightly more complicated than the MLP, however using Tensorflow their implementation is quite straightforward. Moreover,

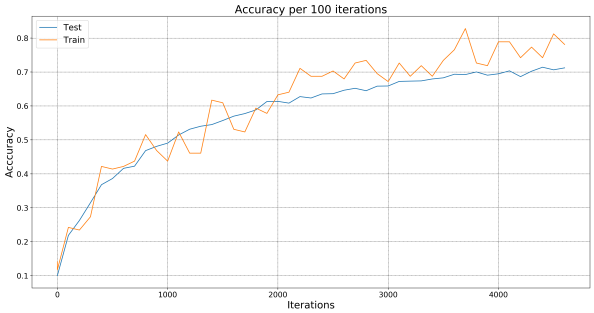

Figure 11: Train and Test accuracy in ConvNet every 100 iterations.

using Tensorboard, we can visualize our models and also keep track of various parameters and variables during training, something that makes the debugging much easier. Due to the more complex architecture, CNNs need more computational resources in order to train and hence are more difficult to be tuned. However, their predictive performance is much better than the MLP.