IN3063/INM702 Programming and Mathematics for Artificial Intelligence Coursework

Deep Learning from Scratch on SVHN

By

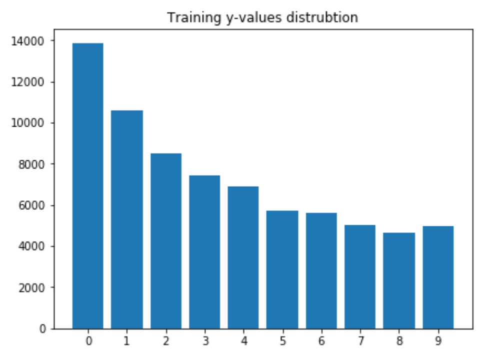
Douglas Landvik

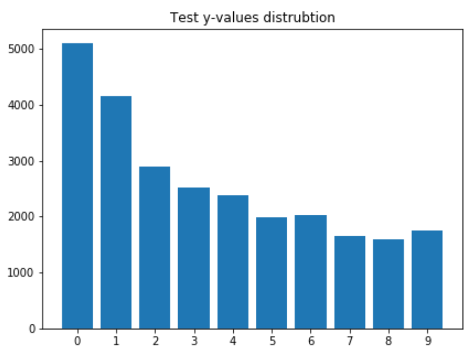
Introduction

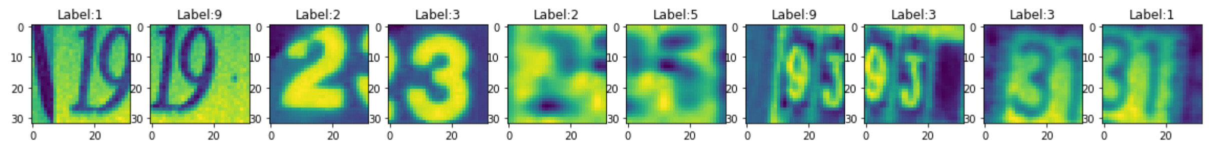
The 'SVHN' (Street View House Numbers) dataset is a real-world dataset that can be seen in a similar flavor to MNIST but with a greater magnitude of data and harder examples. The images are obtained from hose numbers in 'Google Street View'. As with MNIST, there are 10 classes but 'SVHN' consist of labels with a range between 1 and 10. This report is outlying an attempt to implement a Neural Network from scratch, with the aim to generalize well when predicting the label of an image of a number like the one in the dataset.

Preprocessing

The X_training was having 73257 samples of images with 32x32x3 pixels. The X_test was having 26032 samples of images with 32x32x3 pixels. The y_training was of the shape (73257,) with values of 1-10 and the y_test was having the shape (26032,) with the same range of values. The preprocessing was made in 7 steps: Removing the colors column with shape (3,), converting the y-values to a range of [0,9] instead of [1,10], reshaping the training data to (73257, 1024), hot-encode the y-sets, visualize the y-distribution and scale the X and y values to values between [0.1,0.99]. From the preprocessing phase, it could be noted that the dataset was rather unbalanced and noisy as seen with the figures below.







Evaluation metrics for multiclass problem

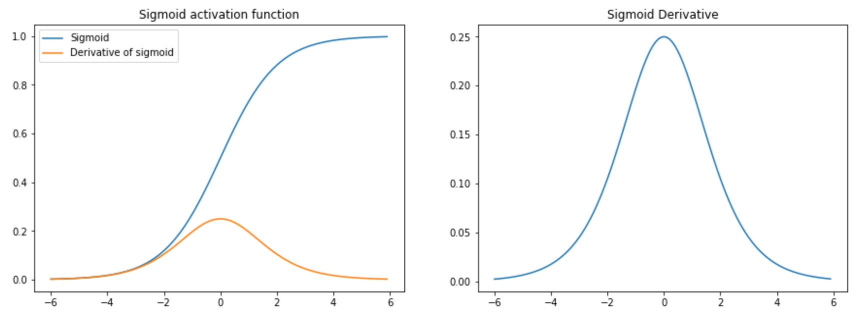
When picking performance metrics it is important to contemplate over the exact problem we are aiming to solve. Different performance metrics could be relatively important. Accuracy, precision, recall was used as performance metrics. Confusion matrix was also used to visualize the relative predictions/actual distribution.

The accuracy is the total correct predictions / all predictions made. It tells one the overall performance of the model. However by adding precision and recall we can get more information on how the model is performing. In multiclass problems the precision can be derived for each label. The precision and recall was implemented by following notations.

$$precision_i = \frac{M_{ii}}{\sum_j M_{ji}} \qquad recall_i = \frac{M_{ii}}{\sum_j M_{ij}}$$

Basic Neural Network

The initial network implemented was a 3 layer network with the input layer having 1024 nodes (for the pixels), the hidden layer having 16 nodes and the output layer having 10 nodes (one for each label). In the first setup a sigmoid activation function was used for both the hidden layers and the output layer. The image below is the output of theses functions on arbitrary numbers.



The initial weight-initialization used in the forward pass was random values between 0 and 1. However, the model outputted close to 0.9 for each label, which is not good when implementing the backpropagation, especially due to the vanishing gradient descent when input goes towards 0 or 1 of the sigmoid function that can be seen in the graph above. The weights were re-initialize to values within the range $\frac{-1}{\sqrt{n}}, \frac{1}{\sqrt{n}}$ in a truncated normal distribution. An untrained model outputted predictions of around 0.5 for each label.

Backpropagation

The cost function ‘Mean Squared Error’ was chosen as it one of the most intuitive ones. It is a function of y hat (output of the network)

$$C(\hat{y}) = \left(\hat{y} - y\right)^2$$

y hat is a function of the z2 layer (fed into sigmoid in our case)

$$\hat{y}(z2) = \frac{1}{1 + e^{-}z2}$$

z2 is a function of w (it is the weighted sum with the activation of the hidden layer a2)

$$z2(W2) = \left(a2 * W2\right)$$

When applying the chain rule one can derive:
$$\frac{\partial C}{\partial W2} = \frac{\partial z2}{\partial W2} \frac{\partial \hat{y}}{\partial z2} \frac{\partial C}{\partial \hat{y}}$$

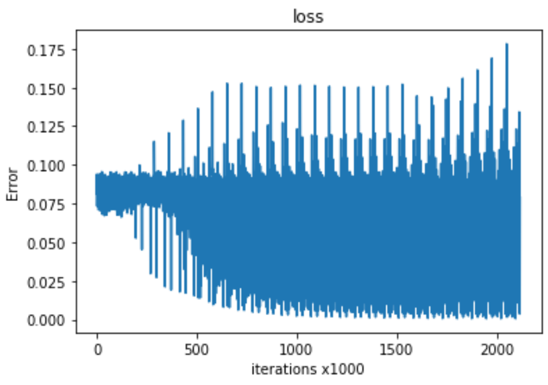
The first phase for the backpropagation was implemented from the definition above. To get the $\frac{\partial C}{\partial W1}$ we can reuse the beginning of the function defined above. The $\frac{\partial \hat{y}}{\partial z2} * \frac{\partial C}{\partial \hat{y}}$ usually is denoted as $\delta 2$. To derive $\frac{\partial C}{\partial W1}$ we get the following definition: $\delta 2 * \frac{\partial z_2}{\partial a_2} \frac{\partial a_2}{\partial z_1} \frac{\partial z_1}{\partial W_1}$. (Labs, 2014)

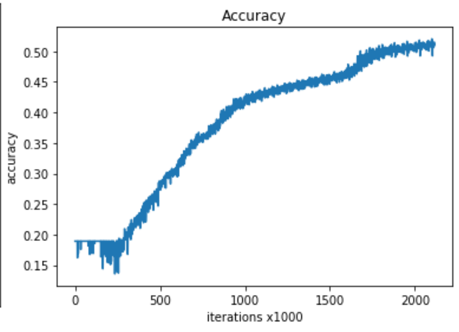
Setting learning rate

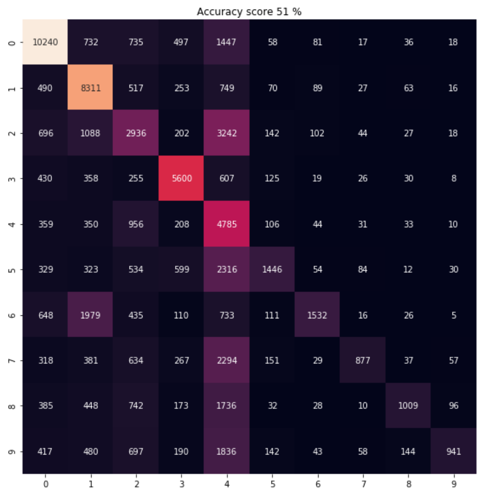
When deriving the partial derivative of the cost function to the different weight matrices we know in what direction every specific weight needs to go in order to increase the cost. As we want to decrease the cost we update the weights by subtracting the $\frac{\partial C}{\partial W} * scalar$. The scalar is called the learning rate as it determines how big steps to take towards the minimum of the function. If the steps are to big we will always overshoot the minimum and if the steps are too small we will never reach the minimum in a reasonable amount of computation time. The learning rate in the first run was set to 0.01.

Training the model

Since samples (73275) * trainable parameters(16544) = 1.212.201.600, one single epoch needed a lot of computations. We did not choose to feed in the whole network in case of stack overflow, as we need to save a lot of data, i.e. in the weight matrices. Online learning was used and the loss and accuracy was saved every 1000th sample. The model was run for 29 epochs and the following was the result:





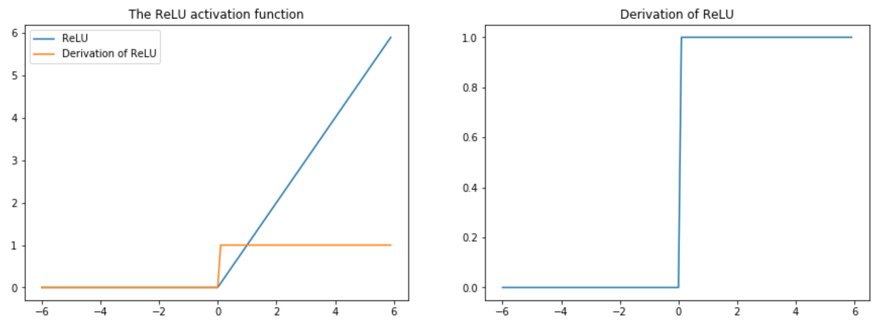

Accuracy score 51 %

	0	1	2	3	4	5	6	7	8	9
0	10240	732	735	497	1447	58	81	17	36	18
1	490	8311	517	253	749	70	89	27	63	16
2	696	1088	2936	202	3242	142	102	44	27	18
3	430	358	255	5600	607	125	19	26	30	8
4	359	350	956	208	4785	106	44	31	33	10
5	329	323	534	599	2316	1446	54	84	12	30
6	648	1979	435	110	733	111	1532	16	26	5
7	318	381	634	267	2294	151	29	877	37	57
8	385	448	742	173	1736	32	28	10	1009	96
9	417	480	697	190	1836	142	43	58	144	941

A total accuracy of 51% for each sample. The loss was rather blurry and should be saved less in the future runs. The accuracy had a steady upgoing trend and the models seemed to be run for longer as it had not converged. The confusion matrix tells us that the predictions for label 0 (actually 1) was the best, and decreasing how larger the target label. The x axis is the predictions and the y axis the actual. We also see that the precision for label 4 was rather good, with a corresponding low recall as many guesses were made on that specific label.

ReLU implementation

Rectified linear unit was used in the hidden layer in the next step. ReLU has the advantage of not having vanishing gradient as could be seen in the graph for the derivative of the ReLU.

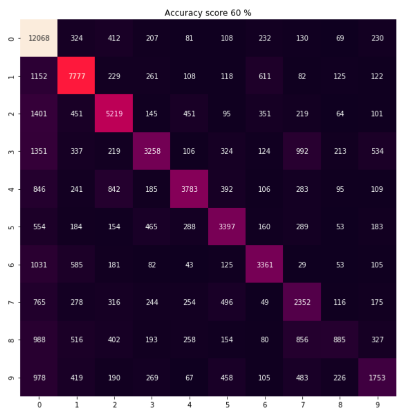


The weight initialization for the hidden layer was changed, as it is suggested, to: $\frac{-2}{\sqrt{n}}, \frac{2}{\sqrt{n}}$.

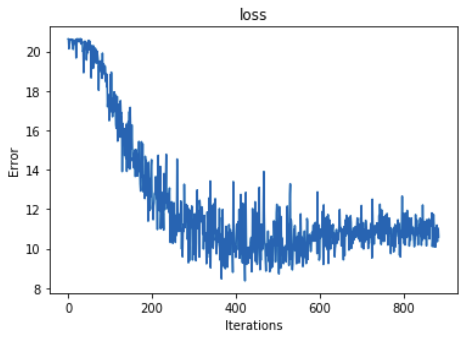
Batch size, epochs, and iterations

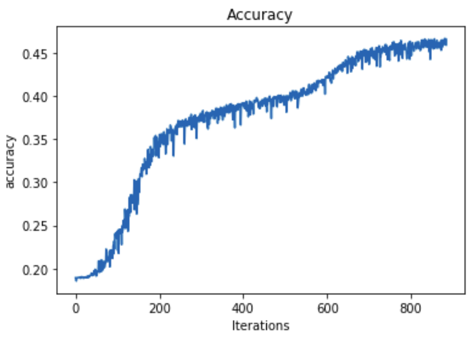
When training the first model an online training technique was used for 29 epochs (all training samples have been trained 29 times). The model then had a total iteration of 73275(samples) * 29(epochs) = 2.124.975 iterations. Every iteration was training a total of 16570 parameters. So if we assume one parameter training is denoted as one computation there are astonishing 2.124.975 * 16570 = 35.210.835.750 compuations: which also was noticed with the long training time. Therefore a mini-batch size was introduced that would make the iteration amount smaller and therefore the total computation smaller. The size chosen was picked by trial and error and finally picked of the value 20. The same network structure was used as the run for with sigmoid for all activation functions. The result was getting better with a total accuracy of 60 % after 29 epochs. The precision was also better throughout the set and the difficulties to classify label 4 was not as apparent as in the former model.

Evaluation

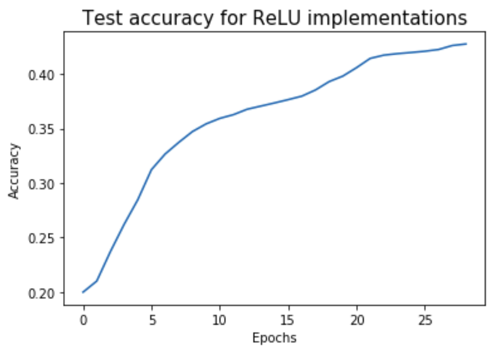

Accuracy score 60 %

	0	1	2	3	4	5	6	7	8	9
0	12068	324	412	207	81	108	232	130	69	230
1	1152	7777	229	261	108	118	611	82	125	122
2	1401	451	5219	145	451	95	351	219	64	101
3	1351	337	219	3258	106	324	124	992	213	534
4	846	241	842	185	3783	392	106	283	95	109
5	554	184	154	465	288	3397	160	289	53	183
6	1031	585	181	82	43	125	3361	29	53	105
7	765	278	316	244	254	496	49	2352	116	175
8	988	516	402	193	258	154	80	856	885	327
9	978	419	190	269	67	458	105	483	226	1753





Intermediate weights were saved, and when running over the test set we saw that the test predictions were following the training data, suggesting the model was not overfitting. The graphs tells us that the model could be run for longer as it has not converged completely.

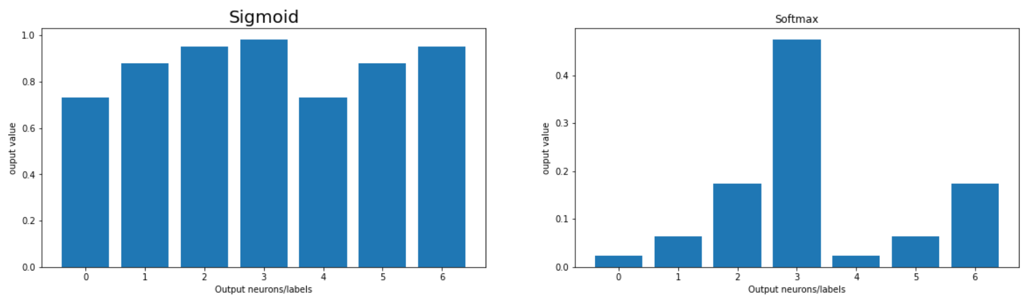


Softmax classifier

The sigmoid function, that we have used as the activation function so far has not been assigning a probability for each label. The output for our ten classes' sum has not added up to 1. However, the softmax activation function is assigning each output a probability, which is what we are looking for when classifying. The definition of the function is the following:

$$f_i(\overrightarrow{a}) = \frac{e^{a_i}}{\sum_k e^{a_k}}$$

Where a is the input vector (weighted sum from last layer). As the formula suggest : for each label the output will be $e^{weightedsum}$ divided by the sum of all the output neurons $e^{weightedsum}$. The distribution will therefore be dependent on each other, and the sum will be 1.



Derivative of softmax

In order to take the derivative of the softmax function the quotient rule of calculus is appropriate. It says if:

$$f(x) = \frac{g(x)}{h(x)} \quad f(x)' = \frac{g'(x)h(x) - h'(x)g(x)}{h(x)}$$

Categorical cross entropy cost function

Cross entropy was chosen as cost function. The entropy is based on information theory that tells how many questions you have to ask to know what symbol, e.g. a machine, will produce, if you already know the probability distribution of the symbols. As we will get a probability distribution from softmax this is applicable. So if the entropy drops we can ask fewer questions, which means we will need to minimize the entropy $H(y, a) = -\sum y_i log a_i$. When using the cross entropy with Softmax, as suggested, (Dahal, 2019) the partial derivative of the Loss w.r.t. 'Z' has an elegant solution: $\frac{\partial C}{\partial z_i} = \hat{y} - y$.

With MSE as cost function the partial derivative of the cost function was taking w.r.t. to the activation and by the chain rule we took the multiplication with the partial derivative of the activation to the weighted sum to get the partial derivative of the cost function to the weighted sum (delta2). Now those were omitted by implementing the first phase of the backpropagation from the above definition. An elegant and simple solution. When using the cross entropy the target was rescaled back to 0 or 1, instead of being 0.01 and 0.99. It is because the 0's is canceling out the cross entropy for the wrong classes, which is what is desired. (Bendersky, 2016)

Underflow and overflow of softmax

The first run of the softmax function was producing 'nan' values after some training. That can happen when the softmax function is experiencing underflow or underflow. The implemented function was unstable as the oversize of the floating point, outputted 'NaN' errors. It is suggested to multiply both the denominator and the numerator with a constant C, which was done by using max as the constant C. (Zhou, 2019)

Training

By trial and error we found out the parameters of 29 epochs and a batch_size of 5 was preferable. However, The softmax implementation only got a 30% training accuracy, the implementation was checked to converge to other datasets, such as the MNIST and the iris which got good results. Next step was to implement a fully connected Neural Network, where we could increase the complexity of the model.

Fully connected layers

A new Neural Network class was implemented with the ability to initialize with an arbitrary number of hidden layers. The output was still not good, and the model had a hard time converging and one can notice when experimenting with different learning rates, batch sizes and network topology, small changes made big changes in how the loss moved. A conclusion to draw is that how more complex a model gets how harder it is to tune. Dropout is usually implemented to prevent overfitting, as our model has not overfitted yet it might not be needed. However, it is a good idea to implement it. The inverted dropout was used, which is the most common one. It is based upon scaling up the remaining nodes. As the final step we will scale the remaining nodes by dividing it with the keepProb scalar we have chosen. We do so because we do not want to change the expected value of the activation.

$$a_i = \frac{a_i}{keepProb}$$

The dropout was successfully implemented by testing it on smaller datasets. But as predicted it did not help us as our model is underfitting and not overfitting.

Conclusion

Hyperparameter optimization

When working with a large dataset as the SVHN it could be a good idea to have smaller datasets to test the implementations on before running. For this project three datasets were used in the implementation in order to debug, but also find the best initial hyperparameters. The iris dataset from sklearn was used as the first dataset to feed into our model in development, merely to see if the implementation worked. The second dataset to use was the MNIST, which is similar but as mentioned less computational demanding and less noisy. The mnist dataset was used to start running with different batch sizes, learning rates and network topologies to faster see in what direction a certain combination of hyperparameters would take the loss and accuracy. A version of mini batch gradient descent was used throughout the training, due to the heavy computational power it would take to use classic batch gradient, but still use the vectorization of our training implementation on smaller batches.

Lesson learned

Deep learning networks benefits from large set of training data, but as the datasets grows the computational load increases. There is of big importance to use efficient network topologies, and when implementing the models from scratch that is hard to achieve. If one desire to have full control of the implementation of the network, using PyTorch would be a good idea. That is because PyTorch is optimized for computation by having most of the back end in C++. Also it has an ability to run on GPU, by CRUDL, which is much more efficient.

Further work

As described a version of mini-batch SGD was used throughout the experiments due to the large computational cost of using classic gradient descent. However, what needs to be done is to optimize the SGD or change it some other variant of gradient descent. For example, Adam, that uses both momentum and adapted learning rate.

References

Bendersky, E. (2016). *The Softmax function and its derivative - Eli Bendersky's website*. [online] Eli.thegreenplace.net. Available at: https://eli.thegreenplace.net/2016/the-softmax-function-and-its-derivative/ [Accessed 5 Jan. 2020].

Dahal, P. (2019). *Classification and Loss Evaluation - Softmax and Cross Entropy Loss*. [online] DeepNotes. Available at: https://deepnotes.io/softmax-crossentropy [Accessed 5 Jan. 2020].

Labs, W. (2014). *Neural Networks Demystified [Part 4: Backpropagation]*. [online] YouTube. Available at: https://www.youtube.com/watch?v=GlcnxUlrtek [Accessed 5 Jan. 2020].

Zhou, V. (2019). [online] Victorzhou.com. Available at: https://victorzhou.com/blog/softmax/ [Accessed 5 Jan. 2020].