Homework 2

Amirhesam Abedsoltan
University of Southern California
`abedsolt@usc.edu`

Abstract

This document contains our implementation details and results on text classification task. we implement one-layer feed-forward network. we implement the model once in bare bone python using numpy library and once using PyTorch deep learning framework. Additionally, we compare our results with the linear case that was implemented previously.

1 General framework

The general format for any classification task is $(X_1, Y_1), (X_2, Y_3), \ldots$. Where X_i stands for feature of the i^{th} sample and Y_i stands for i^{th} sample's label. Basically, we want to predict the label of an instance based on the features we have for that instance. In our task we have a text and we want labelize it by judging it's contents. So, the first step is to extract meaningful features from each text as our Xs. This has already been provided using glove and fast-text. Then, the second step is to use the generated features to predict the text's label through a statistical model.

2 Features

The features has been already given using a dictionary. It's based on glove (Pennington et al., 2014), or fast-text (Joulin et al., 2016) in case of Odia. Therefore, for each sentence of length l, we limited the length of the sentences to 300 in our analysis, we get an input with size of $l * 50 (l * 300$ in case of Odia and using fast-text). Also for unknown word we used the provided mean vector.

3 Numpy implementation

This section is about how we implemented the one layer neural-network using only Numpy. This implementation is based on forward-backward interface architecture that is implemented in both PyTorch and more recently in Tensorflow 2.0 (Abadi et al., 2015). In this architecture each layer object must implement two methods: 1) forward method, that gets the network inputs and generates the output of forward path which is identical to the datapath that is used during inference, and 2) backward method, that gets the gradient of the next layer and input of current layer as input and after setting the gradients of the layer's parameters compute the gradient with respect to its input and returns it to be used in the backward method of previous layer. Using these two methods, we can simplify the design of a neural network with arbitrary architecture as far as we call the forward and backward methods in the correct order (for this we usually keep the topologically sorted network). In more detail, for the inference phase, we simply need to start from the beginning of the network and run the forward method of the layers of network in order they appear; for our case of FNN we just need to start from the input layer, then hidden layers in order and finally, the output layer. On other hand, during training and back-propagation, we just need to start from running backward method of the loss layer, then output layer, hidden layers and finally the input layer in case it is needed.

3.1 Feed Forward

The training parameters are ω_1, ω_2, b_1. The blocks that I needed Relu and sigmoid function in order to use them as non-linear functions. Below is the overview of feed forward network:

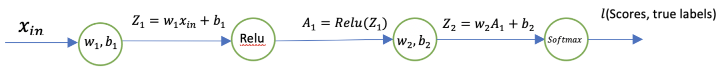

Figure 1: One layer neural network and feed forward network.

so we define $\omega_1 \in R^{input \times hidden}$,

$\omega_2 \in R^{hidden \times Classes}$, $b_1 \in R^{hidden}$. By using the architecture above we can find the feed forward values. I also saved the out put of each block of feed forward in a list named cache in my code. This is for later use in back-propagation.

3.2 Back-propagation

Basically, the idea of Back-propagation is based on chain rule and calculate the dervitives for each block once. More precisely if we are using several Relu units we only need to calculate the derivatives of parameters based on inputs and outputs of that Relu function and use those equations for all other Relu units too.
Here, we need to calculate all of $\frac{\partial l}{\partial \omega_1, \omega_2, b_1, b_2}$. However, to do it directly it would be very difficult. Instead we use chain-rule to make it more convenient. So, what we do here is to calculate the $\frac{\partial l}{\partial Z_2}$ directly and then use chain rule for other parameters:
First notice that the loss functions is cross entropy loss:

$$l(Z_2, y_i; \omega_1, \omega_2, b_1, b_2) = ln(1 + \sum_{j \neq y} e^{Z_{2i}[y_i] - Z_{2i}[j]}) \tag{1}$$

$$\nabla_{Z_{2i}} l(A_{1i}, y_i; \omega_1, \omega_2, b_1, b_2) = \begin{cases} (\frac{e^{Z_{2i}[k]}}{\sum_{j=1}^{C} e^{Z_{2i}[j]}} - 1), & k = y_i \\ \\ \frac{e^{Z_{2i}[k]}}{\sum_{j=1}^{C} e^{Z_{2i}[j]}}, & k \neq y_i \end{cases} \tag{2}$$

Where Z_{2i} is the output of the second linear block for input i.
Now we find a general back propagation for any linear block using chain-rule. Assume A_{prev} is the input of the linear block with parameters ω and b. Also Z is the output of the linear block. Now, because of chain rule we can write:

$$\nabla_{\omega} l = A_{prev}^T \cdot \frac{\partial l}{\partial Z} \tag{3}$$

$$\nabla_{A_{prev}} l = \frac{\partial l}{\partial Z} \cdot \omega^T \tag{4}$$

$$\nabla_b l = \frac{\partial l}{\partial Z} \tag{5}$$

Note that since we start calculating the derivatives from the end to start we have already calculated the $\frac{\partial l}{\partial Z}$. Also, we have A_{prev} and ω current values from feed forward calculation.

It only remains to calculate back-propagation for Relu. The input is Z and the output is A. Now we can write:

$$\nabla_Z l = \max\{\frac{\partial l}{\partial A}, 1\} \tag{6}$$

Now we can use these rules to find all derivatives we need for learning parameters. Then we can use general gradient descent to update them:

$$\omega_{t+1} = \omega_t - lr \cdot \nabla_{\omega_t} l \tag{7}$$

Where lr is learning rate. Also for **"extra mile"** we studied the role of regularizer on over-fitting. To do this we used L_2-norm to prevent wights to get very large numbers. Accordingly the update rule for ω parameters changed to this:

$$\omega_{t+1} = \omega_t - lr \cdot \nabla_{\omega_t} l - lr \cdot \omega_t \tag{8}$$

4 Numpy versus PyTorch

The main difference is basically by using pyTorch you don't have to take care of details and most importantly back-propagation(Adam Paszke and Lerer., 2017). The most time consuming part of implementing with Numpy was calculating back-propagation for each available blocks, like linear block, Relu and softmax in our case. However, the idea behind PyTorch is that they have done this once and we can simply use them. Another difference is when we want to feed in the inputs.Thanks to "dataloader" pre-existing class, it's really easy to do the batching automatically and fast. However, it can still be done manually as long as the inputs are in tensor format. Another benefit of pytorch is for the feedforward implementation. There are all sort of pre-existing blocks like linear, Relu, Drop-out, CNN, etc. Therefore you don't have to do it on your own.
Structural differences aside, the final result are pretty much the same. In fact we expected this to happen because Pytorch is implementing the exact thing that we implemented from the scratch. The only difference is about the initialization. I used normal distribution to initialize the parameter while PyTorch uses Xaviar initialization which since this is a shallow neural network the effect is not that significant.

5 Result and discussion

In this section we talk about how the hyper-parameters have been chosen for each data set and summarize the performance of both Numpy and PyTorch models. For each data set we disccus the role of number of the hidden units, learning rate, role of regulizer(**extra mile**) and accuracy.

5.1 4dim data set

For this data set, we tried learning rates range from 0.001 to 0.9. the learning for above 0.1 start diverging and for rates less than 0.01 it was slow. The best choice was 0.01.
Number of hidden units didn't have any significant effect.We tried for 10, 20,50 and 100 hidden units. Regulizer had a great effect on over-fitting. The plots below show this:

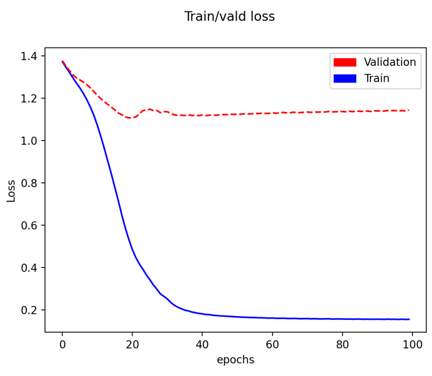

Figure 2: Loss plot without regulizer for 4dim.

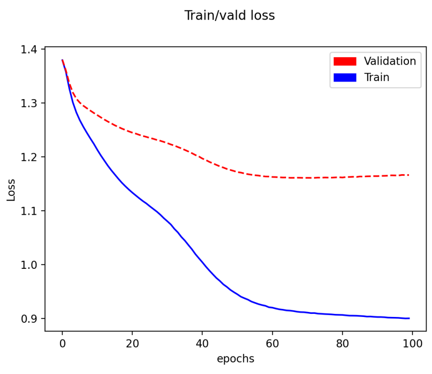

Figure 3: Loss plot with regulizer for 4dim.

As you can see the gap significantly reduced because of the L_2-norm regularize. Although this didn't improve the validation accuracy significantly adding regulizer helps better generalization for unseen data. Also, below you can see the accuracy result for this data set:

Model(4dim data set)	Accuracy
Numpy implementation(Train)	98%
Numpy implementation(validation)	53%
PyTorch implementation(Train)	98%
PyTorch implementa(Validation)	50%

Table 1: Accuracy for 4dim data set.

As you can see both implementation have almost the same performance.

5.2 Authors data set

For this data set, we tried learning rates range from 0.001 to 0.9. the learning for above 0.1 start diverging and for rates less than 0.001 it was slow. The best choice was 0.01.
Number of hidden units didn't have any significant effect.We tried for 10, 20,50 and 100 hidden units. Regulizer had a great effect on over-fitting. The plots below show this:

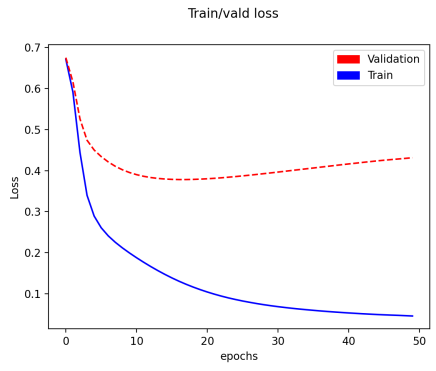

Figure 4: Loss plot without regulizer for Authors.

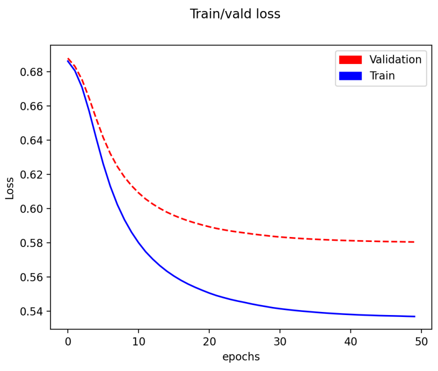

Figure 5: Loss plot with regulizer for Authors.

As you can see the gap significantly reduced because of the L_2-norm regularize. Although this

Model(authors data set)	Accuracy
Numpy implementation(Train)	88%
Numpy implementation(Validation)	80%
PyTorch implementation(Train)	87%
PyTorch implementation(Validation)	78%

Table 2: Accuracy for authors data set.

didn't improve the validation accuracy significantly adding regulizer helps better generalization for unseen data. Also, below you can see the accuracy result for this data set:

5.3 Products data set

For this data set finding a proper learning rate was more challenging. We tried the range from 0.001 to 0.9. Learning rate above 0.01 start diverging. The best choice was between 0.002 and 0.005.
Number of hidden units didn't have any significant effect. We tried for 10, 20,50 and 100 hidden units. For this data set although we tried different coefficient for adjusting regulizer, from 0.01 to 0.5, none of them worked properly. Big coefficient made the gradient stuck and small numbers didn't have any significant effect.

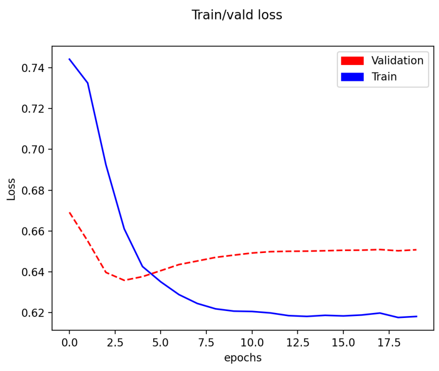

Figure 6: Loss plot without regulizer for products.

Also, below you can see the accuracy result for this data set:

Model(products data set)	Accuracy
Numpy implementation(Train)	82%
Numpy implementation(Validation)	63%
PyTorch implementation(Train)	82%
PyTorch implementation(Validation)	61%

Table 3: Accuracy for Products data set.

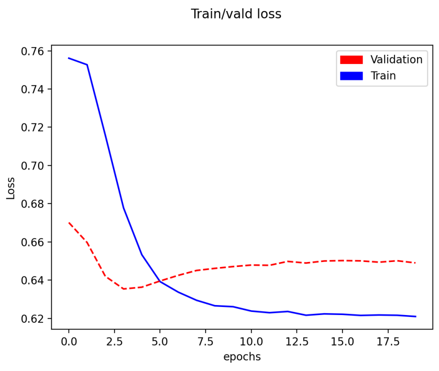

Figure 7: Loss plot with regulizer for products(coefficient 0.02).

5.4 Odia data set

For this data learning rate 0.01 did a great job. Number of hidden units didn't have any significant effect. We tried for 10, 20,50 and 100 hidden units. Regulizer was not really necessary in this data set since there was not a considerable over-fitting even without regulizer.

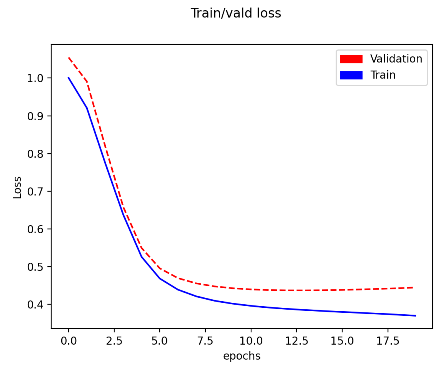

Figure 8: Loss plot without regulizer for Odia.

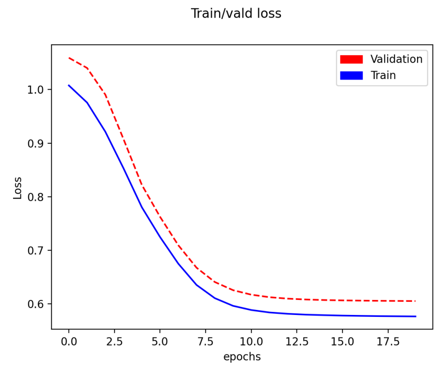

Figure 9: Loss plot with regulizer for Odia.

Also, below you can see the accuracy result for this data set:

Model(Odia data set)	Accuracy
Numpy implementation(Train)	80%
Numpy implementation(Validation)	80%
PyTorch implementation(Train)	78%
PyTorch implementation(Validation)	77%

Table 4: Accuracy for Odia data set.

6 Comparison with linear models

Let's first repeat the performance of last homework and also the performance of our non-linear model on test data set:

Model	4dim	authors	odiya	products
Naive Bayes	0.875	0.79	0.928	0.817
Perceptron	0.75	0.64	0.725	0.715
LogisticReg	0.85	0.61	0.92	0.846
Non-linear(this work)	0.67	0.8	0.77	0.677

Table 5: performance of linear models(test).

As you can see non-linear models have not outperformed the linear models in most of the data sets. In fact, Naive Bayes still has the best performance in all data sets. We should also becareful that this might partially be a result of input sentence trimming.

7 Extra mile

For this part we already studied the role of regulizer on over-fitting and also tried different number of hidden units. In addition here we study the role of activation function on convergence. We try Sigmoid, Relu and tangh as our activation functions for Authors data set.
below you can see the results in figures 10,11 and 12.

First notice that we have used the exact same settings for all of these activation functions. Meaning the exact same number of hidden units and the exact same learning rate. You can notice that Sigmoid has the slowest decrease in the cost function. There are several reason for this:

1. Comparing to Relu, Sigmoid's derivative is very small for large numbers and this makes the updates ineffective and slow. It takes time for Sigmoid to update the parameters with small and large values. That's also a reason why the initialization is more important when dealing with Sigmoid.

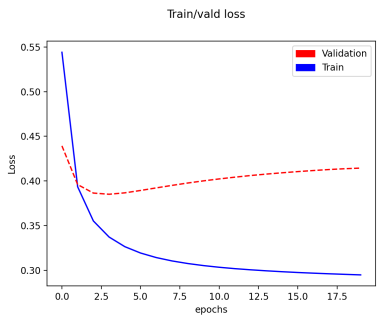

Figure 10: Loss plot with Relu for Authors.

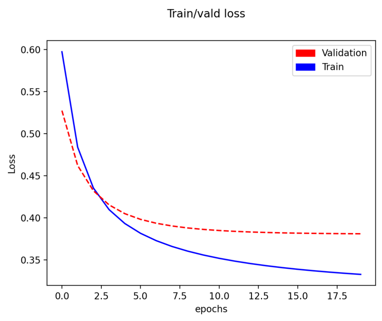

Figure 11: Loss plot with Sigmoid for Authors.

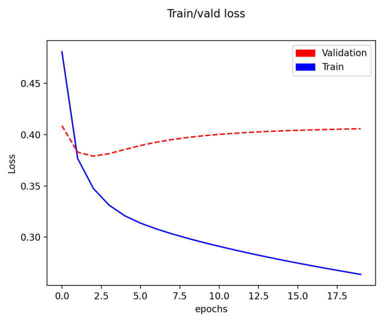

Figure 12: Loss plot with Tanh for Authors.

2. Comparing to Tanh: Tanh takes values between [0,1] while Sigmoid takes values between [-1,1]. If you look at the shape of these two function, see figure 13, you will notice that tanh is much steeper comparing to Sigmoid which results in faster updates for parameters.

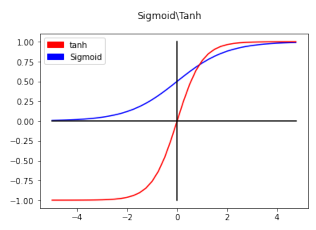

Figure 13: Sigmoid Vs. Tanh

8 Conclusion

We implemented non-linear models once using Pytorch and once from the scratch by using only Numpy. Our results show that all the models, despite of being simple can get good precision results one most of the datasets. However they are not able to beat the linear baselines. We showed that in our setup the Naive Bayes algorithm can still yield better. We also studied the role of regulizer on overfitting and also the role of activation function on speed of leaning for extra mile section.

References

Martín Abadi, Ashish Agarwal, Paul Barham, and et all. Eugene Brevdo. 2015. Tensorflow: Large-scale machine learning on heterogeneous distributed systems.

Soumith Chintala Gregory Chanan Edward Yang Zachary DeVito Zeming Lin Alban Desmaison Luca Antiga Adam Paszke, Sam Gross and Adam Lerer. 2017. Automatic differentiation in pytorch.

Armand Joulin, Edouard Grave, Piotr Bojanowski, Matthijs Douze, Hérve Jégou, and Tomas Mikolov. 2016. Fasttext.zip: Compressing text classification models. *arXiv preprint arXiv:1612.03651*.

Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. Glove: Global vectors for word representation.