Comparing Feed Forward Neural Network In Bare Numpy Vs PyTorch's Implementations in the case of Sentiment Analysis and Categorization

Basem Rizk
University of Southern California
brizk@usc.edu

Abstract

This paper overviews and compares an implementation of feed forward neural network using $Numpy$ package only with a high level identical network implementation using $PyTorch$. The paper examines both implementation on classification tasks of data-sets used in (Rizk, 2020). The experiments delves into time performance, and accuracy differences and derives insights based on weight initialization importance, data-set fit based on its size compared to complexity of the model, and provides observations over the generalization gap based on networks' hyper-parameters.

1 Introduction

As hardware becoming more capable and for the interest of utilizing, and analyzing the tremendous amount of big data generated all over the web particularly in the form of text, whether formally in news website or informal discussion over social media, neural networks became more achievable. Moreover, one can theoretically achieve similar results if not better due to previous short comings to former work (Rizk, 2020) on sentiment analysis and categorization with Naive-Bayes and Perceptron Classifier with big enough amount of data, with neural networks. Through this paper, I compare my basic implementation of feed forward neural networks from scratch utilizing $Numpy$ only in $Python$, with the mainstream methods of building them using $PyTorch$. Furthermore, experiments were performed to not only test differences between the mainstream and my implementation in terms of time performance and achievable accuracy by training over 4 different sets; but also, few experiments to measure generalization gap performance affected by changes of the different available hyper parameters to build such networks. The paper tests out one form of regularization which is Dropout (Dellinger, 2019) layers, and how it affects generalization.

2 Data source

My implementations are tested over Sentiment Analysis and Categorization tasks on 4 different data-sets, same used by (Rizk, 2022). Each of the data-sets comes from a different domain and their features vary based on but not limited to language used and data points size and corresponding encoding length.

2.1 Encoding

For the English data-sets 4dim and products of reviews, 50-dimensional GloVe English vectors word embedding is used, while for the Odiya language in the odiya data-set of headlines, a 300-dimensional fastText Odiya vectors word embedding is used, and for the questions data-set 100-dimensional 'ufvytar' vectors word embedding of uncertain origin is utilized. Moreover, unknown vocab embedding corresponds on the mean of all vectors in the employed word embedding.

2.2 Features

Each data point per each data-set in the form of string of text is tokenized on punctuation characters into a variable length array of tokens. The first appearing f tokens (f is a hyper-parameter of my networks) are encoded based on the relevant given embedding file to construct an ordered vector of float values of length $d * f$ where d corresponds to the embedding dimension per token, making up features vectors of fixed lengths.

3 Neural Network Structure

For each of the data-sets two neural networks classifiers were trained, one using PyTorch and another using an implementation from scratch employing Numpy.

3.1 Baseline Implementation

The baseline implementation for the classifiers consists of one layer (excluding output layer) which

is of size ($d * f$ x u), where d is embedding dimension, and f and u are hyper parameters, where f corresponds to max sequence size, meaning the number of tokens to taken into account to create the features of the data point, and u corresponds to the number of hidden neurons per layer. Hence we have a weight matrix w_A of shape ($d * f$ x u) and bias vector b_A of shape (1 x u). Moreover a $ReLU$ activation function is applied on this one hidden layer, so that the output of the layer is according to equations 1, and 2.

$$z = X^T * W_A + b_A \tag{1}$$

$$ReLU(z) = max(0, z) \tag{2}$$

Furthermore, the output layer of the neural network made of a weight matrix w_B of shape (u x c), where c corresponds to the number of classes (unique labels) and a bias vector b_B, and finally a $Softmax$ activation function is applied to the outcome to get a final output of the network in the format of c probability values that sum to 1, according to equation 3, where z calculated similar to equation 1.

$$Softmax(z) = (\frac{\exp z^T}{\Sigma_{j=1}^{c} \exp z_j})^T \tag{3}$$

3.2 Back-propagation

Back-propagating to update the network, and making the model learn is performed based on per layer based on its type and the type of activation function employee. Per every linear layer I calculate the change of weights, and biases per equation (**?**), and (**?**) respectively, where $\frac{dE}{dI_j}$ corresponds to gradient based on incoming gradient and activation function used, given the incoming gradient calculated for prior layer i at layer j updates as in equation 6. Per the $Softmax$ dense (fully connected) layer, which is employed as output layer in every model built here, the $\frac{dE}{dI_j}$ as calculated as $o_j - o_t$ where o_t corresponds to true labels. For $ReLU$ it is calculated as derivative of $ReLU$ as equation **??** multiplied by incoming gradient, and as such for $Sigmoid$, calculated as $(\sigma(o_j) * (1 - \sigma(o_j))$ multiplied by the incoming gradient.

$$\frac{dw_ij}{dE} = X^T \frac{dE}{dI_j} * \frac{1}{\text{batch size}} \tag{4}$$

$$\frac{db_ij}{dE} = \Sigma_{j=1}^{c} \frac{dE}{dI_j} * \frac{1}{\text{batch size}} \tag{5}$$

$$\frac{dE}{dI_i} = \frac{dE}{dI_j} w_i j^T \tag{6}$$

$$f(o_j) = \begin{cases} 0 & \text{if} o_j < 0, \\ 1 & \text{otherwise.} \end{cases} \tag{7}$$

3.3 Weights Initialization

Based on (Lin), $pytorch$ default weight and bias initialization per the linear layers (following every activation tested in this paper) is done as equation 8 per a normal distribution with mean of zero and standard deviation of $\frac{1}{k}$, where k corresponds to the layer input size.

$$\mathcal{U}(-\sqrt{\frac{1}{k}}, \sqrt{\frac{1}{k}}) \tag{8}$$

However, through my implementation in Numpy, I have tested few other weights initialization methodologies. They are all zeros, totally random, scaling by small but not too small value of 0.01, Xavier initialization (Dellinger, 2019) and a similar initialization to torch's linear default where I call it normal weight initialization implemented in $numpy$ using $np.random.normal$ with scale of $\frac{1}{k}$.

3.4 Variations

In addition to $Softmax$ and $ReLU$ activation functions, $Sigmoid$ was also implemented and tested out. The implementation supports multiple layers networks, which was tested out and included in my experiments aiming to get a best fit model. Dropout (Srivastava et al., 2014) layers as one mean of regularization is also implemented as it drops randomly neurons with a probability of p in the previous layer, meaning setting their outcome to zeros, and scaling up the leftovers during training by $\frac{1}{1-p}$, then let values pass normally during classification phase.

4 Experimental Results

I conduct a list of different experiments, where I draw network wise observations as well as general ones in terms of the data-set fitness with the tested network architectures.

4.1 Time Efficiency

Calculating the average time to train an epoch going through all batches compared shows that PyTorch's implementation has about 1.58 speed up given approximately the average training time per

dataset	bare-numpy	torch	speedup
4dim	1.25	0.7	1.78
products	18.1	11	1.64
questions	2.38	1.64	1.45
odiya	3.5	2.45	1.46
4dim-c	.0085	0.225	-26.2
products-c	.0645	0.118	-1.8
questions-c	.0044	0.0049	-1.1
odiya-c	.071	0.0686	1

Table 1: Comparing average epoch training time on the 4 different data-sets with configurations of (batch size, max-sequence, one layer hidden units) of (64, 300, 200) with 4dim, (256, 350, 250) with products, (8, 20, 100) with questions and (128, 10, 450) with odiya. Moreover the second part of the table shows comparison based on one execution of classification time on Vocareum on the test data

epoch on each of the data-sets. One more thing to note here is that PyTorch seems to exceed to outperform my model is given higher batch sizes and wider dimensions, as possibly they have fine tuned the memory allocation to fit the growing size of dimension better. Moreover, Based on the second part of the table, it shows that the speed up possibly comes from the back-propagation part of PyTorch. Furthermore, it is quite interesting to even see that my implementation is even faster at classification than PyTorch according to this trial on Vocareum, bearing in mind that PyTorch is given the test data in on single batch. I believe that these points support the idea that PyTorch has some way of organizing the memory probably that is biased toward training.

4.2 Wider Vs Deeper Layers

By training classifiers for products with few permutations and different ranges of hidden units in the network, the plot 1 of the learning curves is produced. Based on my observation, I notice at one layer level testing out with sizes 50, 80, and 100, they all observe similar updates but there is sweet spot at 80 where the model observes them earlier, with more stable validation curve. Moreover, reducing width of network by splitting 100 units into two consecutive layers of size 50 in this particular case seem to gain more stability for training accuracy performance albeit larger spikes with validation. Furthermore, Dropout helps with stabilizing the training curve, and with reducing the generalization gap, but more epochs might be needed to prove

W-Init	t-loss	t-acc	v-loss	v-acc
normal	0.32	87.62%	0.51	80.07%
random	2.24	78.55%	3.94	71.45%
scaling	0.32	87.38%	**0.5**	**80.72%**
xavier	**0.23**	**92.03%**	0.54	80.66%
zeros	1.06	45.54%	1.06	45.46%
torch	**0.70**	**86.74%**	**0.74**	**80.59%**

Table 2: Comparison between weight initialization techniques on bare-numpy implementation with torch-default-linear weight initialization after training for 20 epochs on odia data-set with one hidden layer of ReLU with 450 neurons, learning rate of 0.05 and batch size of 128.

validation accuracy improvement. $Sigmoid$ function is tested out here as well, and it does seem to be even further stable than all the former in this case.

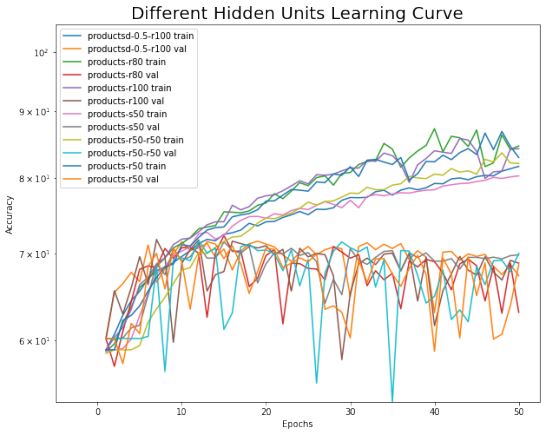

Figure 1: Comparison of the learning curve while training different neural networks trying different hidden units sizes on products data-set

4.3 Weights Initialization Effect

According to Table 2 which lists training and validation losses and accuracy accordingly for identical configuration but with exception of weight initialization techniques on my implementation and the torch default linear weight initialization. It seems that xavier did the best with my implementation. Moreover although clearly torch's default beat it, there is an interesting phenomena where the losses of my implementation of normal, xavier, as well as scaling is clearly smaller than torch's, which raises a question if there is some sort of an implicit regularization inherited in PyTorch's implementation.

4.4 Dropout Regularization Effect

Plot 2 shows a comparison including both my implementation's and torch's models while employing vs not employing a Dropout regularizing layer. Perceiving the questions-d-train representing with-dropout train graph, against questions-d-val representing with-dropout validation graph, as well as questions-train representing the no-regularization train graph against questions-val, the former (including dropout) observe a more steady rate of improvement and much smaller generalization gap.

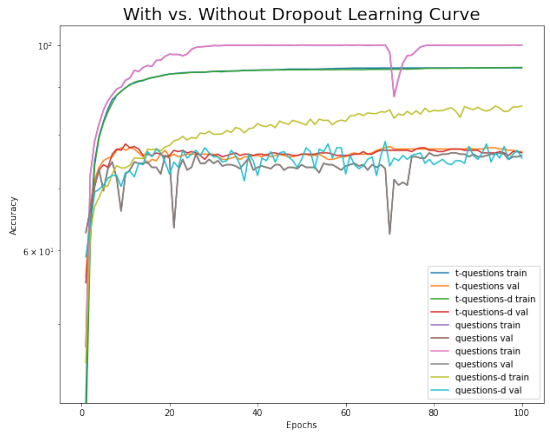

Figure 2: Comparison of the learning curve while employing vs not employing a Dropout (regularization) layer of training vs. validation accuracy through torch (t-) and my implementation of a classifier for *odia* dataset with normal scaling with $ReLU$ layer of 18 units, lr=0.05, max sequence length of 10 and batch size of 8 through 100 epochs, while plugging in the (-d) models a dropout layer of 0.5 zeroing probability.

4.5 Most Tuned Values Compared

Table 3 compares most fit classifiers with (Rizk, 2022) in Naive Bayes and Perceptron classifiers. Products, and 4dim based on validation and training accuracy seem to be either noisy data-sets as networks seem to be very prone to over-fitting more with than with questions and odia. Another justification for their related observations is that the long texts inputs needs more complex network, but as you extend the layer, they easily over-fit as the dataset is relatively small compared to the complexity of the model. However based on test-set results, it is very unclear why the model obtains such low scores especially with questions and odia despite having narrow generalization gap and high accuracy based on validation test, and how come bare-numpy is doing better almost always, and products' model performs relatively better than the rest.

Model	NB	PP	bare-numpy	torch
4dim	**0.93**	0.9	0.325	0.25
Products	**0.83**	0.82	0.51	0.503
odia	**0.93**	0.89	0.359	0.367
questions	0.71	**0.85**	0.216	0.204

Table 3: Comparison between best tuned models with Naive-Bayes (NB) and Perceptron (PP) results from (Rizk, 2022) on Test Data-sets

4.6 Max Sequence Size Vs. Layer Size

Initially, reducing max sequence size from 10 to 5 in Odia, for example showed an improvement in terms of the generalization gap between empirical and validation accuracy, however, it seems that putting such conclusion from such observation was not correct, as by reducing the hidden units numbers, and a larger sequence size showed much better accuracy. Applying the opposite over 4dim by increasing the max sequence size from 100 to 150, testing the latter with the same applied hidden units amount of 32 $relu$ gave worse performance, however, by making it wider to be 64, albeit fluctuations, the model achieved much higher accuracy with similar generalization gap with validation accuracy from 34% to 53%, supporting the former observation.

5 Conclusion

Testing out the various classifiers utilizing the 4 different data-sets based on two implementations of mine using $Numpy$ and $PyTorch$, it is deduced that $PyTorch$'s is indeed more time efficient particularly with the way it implements back-propagation. Moreover, there seem to be some form of regularization effect by PyTorch's, which my model was closer to get by replicating the weight initialization 'normal' technique, but not exact yet.

The papers' experiments mostly delve into many aspects of neural networks, while the common and shared interest is better generalization. A common problem where the model seems not able to learn, so I would add up more units to the hidden layer, but as I do, the model is over-fitting the training data, and the generalization gap becomes larger. Adding Dropouts seemed to help in some cases but further training is needed. Furthermore the observation shows a directly proportional relation between network size and max sequence size given to the network.

References

Linear — pytorch 1.12 documentation. https://pytorch.org/docs/stable/generated/torch.nn.Linear.html. (Accessed on 10/13/2022).

James Dellinger. 2019. Weight initialization in neural networks: A journey from the basics to kaiming.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958.