HW2: Nonlinear Classifiers

Zhivar Sourati Hassan Zadeh
University of Southern California
souratih@usc.edu

Abstract

Text classification with its applications in different domains and contexts has experienced a surge due to the large amount of textual information getting produced everyday. Neural Networks (NN) as well have undergone through the same process with variants of models, architectures, and more detailed nuances. Although they show promising results on every task and dataset, their usage can be difficult sometimes. With all the strength and potential they offer comes specific limitations as well as required tweaking of their parameters and hyper-parameters to unveil their true power. In this study, we present our insights into their overall architecture as well as their specific components and their substantial effect on the overall performance.

1 Introduction

Considering the abundant resources of online information in social media as well as the other resources like scientific papers, organizing and analyzing all these documents in terms of various dimensions can be useful for variety of purposes such as sentiment analysis (Medhat et al., 2014) and news classification (Rana et al., 2014) with lower cost and faster pace.

Having Neural Networks (NN) as a tool that can pick up more complicated patterns than its linear counterparts, and also the introduced potentials and limitations via its inner components, we analyzed their performance on different datasets while exploring the effect of their inner components and hyper-parameters. The outline of our work is as follows: In Section 2, we discuss the datasets used briefly. Section 3 will discuss the methods we used for training the model followed by a short remark on the details on the experiments we did in Section 4. Finally, the results will be presented and further insights will be given for future works respectively in Section 5 and Section 6.

2 Dataset

Four datasets have been used, namely, "products", "questions", "4dim", and "odiya" each having their distinct characteristics. The summary of the datasets used is demonstrated in Table 1.

dataset	language	# records	# labels
products	English	32592	2
questions	English (encrypted)	4089	6
4dim	English	1560	4
odiya	Oriya	15200	3

Table 1: Summary of the datasets

3 Methods

3.1 Neural Network from Scratch

While there are so many frameworks and tools offered to us that can facilitate the exploitation of neural networks for various tasks, we opted to implement a feed forward neural network with dynamic number of layers from scratch to allow more in-depth analysis. The implementation and detailed discussion of forward pass and backward pass of our neural network are followed in Section 3.1.2 and 3.1.3 respectively.

3.1.1 Overall Architecture

The neural network that we discuss here consists of one embedding layer, multiple hidden layers in the middle of the network and at the end, another layer that maps the extracted features from data to logits associated with each label.

3.1.2 Forward pass

For every batch of input that wants to be fed to the model, the first layer that processes the data is the embedding layer through which every word will be mapped to its embedding vector and gets appended together with the limit of *max sequence length*. From this point on, every hidden layer receives the input from previous layer and applies a linear transformation on it followed by a activation layer. The last layer of the neural network is also

responsible for transforming the last hidden layer features to logit values associated with belonging to each class.

Having the logits at the last layer, we can use Cross Entropy Loss to somehow compute the misalignment of true label distribution probabilities and the predicted probability distribution. The equation for the computed loss value is presented in Eq 1 (note that a_k being the output of softmax at the last layer of the model associated with the k label is used interchangeably with $q(\hat{y}_k)$ being the predicted probability distribution over labels). Transformations done in the forward pass of the model (two hidden layers for the sake of brevity) can be summarized in Eq 2.

$$\begin{aligned} Loss &= -\sum_{k=1}^{c} p(y_k) log(q(\hat{y}_k)) \\ &= -\sum_{k=1}^{c} y_i log(a_k) \end{aligned} \tag{1}$$

$$\begin{aligned} z_1 &= xW_1 + b_1 \qquad a_1 = activ(z_1) \\ z_2 &= a_1 W_2 + b_2 \qquad a_2 = softmax(z_2) \end{aligned} \tag{2}$$

3.1.3 Backward pass

After computing the loss term, we should slightly change the parameters of the model based on how off is the model from true probability distribution and how each parameter is contributing towards changes in loss term. The second definition we provided here is basically the intuitive definition of gradient which consists of two parts, incoming gradient, being the gradient computed so far and the local gradient for the current unit. All the equations for the backward pass of the model (with the assumption of having two hidden layers) are presented in Eq 3 (note that N_B is the number of samples in one batch).

$$\frac{dL}{dZ} = dz_2 = A - Y \tag{3}$$

$$\begin{aligned} dW_2 &= a_1^T dz_2 / N_B \qquad db_2 = \frac{1}{N_B} \sum dz_2 \\ da_1 &= dz_2 W_2^T \qquad dz_1 = d\_activ * da_1 \\ dW_1 &= x^T dz_1 / N_B \qquad db_1 = \frac{1}{N_B} \sum dz_1 \end{aligned}$$

3.2 Pytorch Implementation

Pytorch (Paszke et al., 2019), being an Imperative Style, High-Performance Deep Learning Library, we compared our implementation with it to check if there is any room for improvements in terms of design and implementation details. Since the overall design of our framework was largely influenced by Pytorch in the first place, our frameworks only differ in the model implementation part. One of the differences that can be seen initializing the models is Pytorch model using the same initialization technique (He (He et al., 2015)) for both the weights and bias term while we initialize the bias term as a vector of full zeros. The most noticeable difference exists in the backward pass of Pytorch implementation. They basically form a backward graph that tracks every operation that has been done on values in the network to calculate the gradients using something called a dynamic computation graph. The Tensors which are the Pytorch-defined units of data contain all the information that is needed for backward propagation such as the gradient function and the history of operations if their gradient is needed to be computed as it is depicted in Fig 1. On the other hand, in our implementation we store all of the variables that we want to keep track of in a variable called *params* which is not as Object-Oriented as Tensors used by Pytorch, while the logic and functionality, is basically identical.

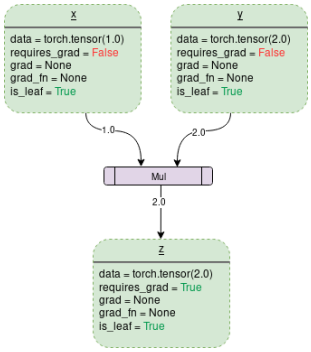

Figure 1: Simple demonstration of properties of Tensor in Pytorch framework

4 Experiments

To gain better insights into different parts of the neural network, we opted to devote part of our discussion to the effect of different components of neural networks. Particularly, we have explored initialization techniques, activation functions, as well as the effect of regularization. Plus, we also compared our results on our curated test split with another blind test split on Vocareum[1] as a sanity check, presented in Table 2. Please note that experiencing similar performance with both test splits and

[1] https://www.vocareum.com/

for the sake of doing more and better experiments delving deeper into the subject we did all our experiments and also compared them with each other using our curated test set. Moreover, at first we tried to have uniformity of used hyper-parameters across datasets but observed that different datasets require different hyper-parameters to perform well which we exploited Weights & Biases(Biewald, 2020) for exploration and sweeps.[2] For the learning rate and batch size used in our experiments, we tried different parameters and based on learning curves adjusted our learning rates and regularization term. Also for the number of layers and number of neurons in each layer, based on practical experience, we started from a simple model with one layer and 32 neurons and based on experiments opted to use a two layer neural network with 64 and 32 neurons in first and second layer respectively. Based on our observations especially for 4dim dataset, adding more layers would have made the neural network overcomplicated and made it deal with overfitting issues.

model (split)	products	4dim	questions	odiya
Pytorch (bt)	0.70	0.75	0.78	0.79
Pytorch (ct)	0.70	0.59	0.72	0.80
NN from scratch (bt)	0.70	0.70	0.79	0.80
NN from scratch (ct)	0.70	0.62	0.74	0.81

Table 2: Comparison between results on our curated test (ct) split and another blind test set (bt) weighted F1

5 Results

5.1 Comparison between Simple from scratch NN and Pytorch implemented NN

Keeping the overall framework and procedure of training and evaluation the same, we compared the performance of our simple NN from scratch and the Pytorch implemented model. Please consider the comparison in this section solely as a mean to check how similar the two implementations are since the model implemented using Pytorch is also using the identical set of hyper-parameters, leaving only the tiny implementation details to differ. Looking at Table 3, the closeness of the results are conspicuous which illustrates the resemblance of our NN and Pytorch implemented model even in the finer grained levels of implementation. However, another observation in Table 3 is the comparison of runtime for the experiments which clearly shows that our implementation is faster than Pytorch.

model (runtime)	products	4dim	questions	odiya
Pytorch	0.70 (13m)	0.59 (3m)	0.72 (34s)	0.80 (1m)
simple NN	**0.70 (7m)**	**0.62 (2m)**	**0.74 (21s)**	**0.81 (1m)**

Table 3: Comparison between simple NN from scratch and the equivalent Pytorch implemented model in terms of test split weighted F1

5.2 Comparison between Simple from scratch NN and linear models

As opposed to experiments and results presented in Section 5.1, we also tried simpler linear models such as perceptron, naive bayes, and logistic regression and the comparison between the performance of these models and our simple NN is shown in Table 4. You can see that linear models are performing slightly better than our simple NN. There are numerous justification for this observation. NNs are great at picking up patterns, but having generalizable models that can perform well both on train and unseen data would require more human steered hyper-parameter tuning, which we did not do extensively since it was out of scope of this study. Plus, NNs can be too complicated for some datasets and simply using more basic models would perform better. Last but not least, the way we tokenized our data cutting off the words occurring at the end of sequence is definitely an influential factor.

model	products	4dim	questions	odiya
naive bayes	0.82	0.75	0.81	0.92
perceptron	0.80	0.74	0.72	0.84
logistic regression	0.84	0.82	0.80	0.87
simple NN from scratch	0.70	0.62	0.74	0.81

Table 4: Comparison between simple NN from scratch and simple linear models in terms of test split weighted F1

5.3 Effect of activation function

One of the biggest points that makes neural network more capable of simple linear models to pick up complex patterns and structures is the non-linearity in their design which is implemented through activation functions. Different activation functions would have different properties. The most important point about them is to focus on their active area so that they don't die or get saturated which is tightly connected to the normalization of their

[2]You can find all our hyper-parameters for different datasets and models at https://drive.google.com/drive/folders/1X7kPdv7_Nm4AN02XOPT98PG-iZR3jbeM?usp=sharing

inputs. Comparing ReLU (Nair and Hinton, 2010), tanh, and sigmoid, our results are presented in Table 5 keeping the other hyper-parameters fixed. Our results demonstrate the superiority of sigmoid over the other two functions. One of the reasons behind ReLU not performing as well can be its overfitting properties without regularization (Nwankpa et al., 2018). Also, having a shallow network can account for ReLU not showing its promising anti-vanishing properties over the other two activation functions.

Activation Function	products	4dim	questions	odiya
ReLU	0.69	0.57	0.77	0.80
tanh	0.67	0.56	0.77	0.79
sigmoid	**0.70**	**0.62**	0.74	**0.81**

Table 5: Comparison between different activation functions in terms of test split weighted F1

5.4 Effect of regularization

One of the most important issues that neural networks face while being trained is capturing the unnecessary and ungeneralizable nuances which can be reflected in their inner parameters. The consequence of over-fitted parameters development is not having generalizability on the evaluation and test split. Different techniques can be used in order to prevent this from happening. L2 regularization tries to lower the complexity of the model by defining another objective to minimize alongside the classification loss. Here, we define the complexity of the model with Eq 4 which changes the overall loss as well (λ or weight_decay is the effect of regularization term.)

$$l2\ loss = \lambda||W||_2^2 = \lambda \sum_i W_i^2 \tag{4}$$

$$Loss = -\sum_{k=1}^{c} y_i log(a_k) + \lambda \sum_i W_i^2$$

weight_decay	products	4dim	questions	odiya
1e-3	0.70	0.57	0.74	0.81
1e-2	0.70	0.61	0.74	0.81
1e-1	0.70	0.29	0.73	0.80
0	0.70	0.62	0.74	0.81

Table 6: Comparison between different regularization effects in terms of test split weighted F1

Considering the results in Table 6, we can see that the effect is much stronger for 4dim than the other datasets, which can be attributed to the different models having different weight vectors and distinct effect of regularization as a result.

5.5 Effect of initialization

Weight initialization is crucial design choice when developing neural network models. They are even considered as the reason why NNs did not attract much attraction in the field of machine learning in the first place. Their effect is tightly connected to the activation function as they try to create values within a proper range so that they would fall into the active area of activation functions. Intuitively they try to initialize the weights in a way to account for the number of neurons that are attached to them from the previous layer and the fact that they can sum up to a large number. We tried two initialization techniques, one being uniform distribution of numbers picked from the range (0, 1), and the other one being He initialization (He et al., 2015). We can see in Table 7 that He initialization works better than former initialization as it results in activated values being saturated, having dead neurons and eventually not learning through back propagation.

Initialization	products	4dim	questions	odiya
Uniform (0, 1)	0.43	0.09	0.08	0.28
He initialization	**0.70**	**0.62**	**0.74**	**0.81**

Table 7: Comparison between different weight initialization strategies in terms of test split weighted F1

6 Conclusion

We analyze the potentials and limitations of neural networks, alongside various hyper-parameters and design options that can affect their performance on four datasets associated with distinct domains. Although neural networks we used are pretty simple and only have two layers, they demonstrate reasonable learning potentials, and due to their generalizability can be used in any other context and domain, as can be seen in our study as well that one architecture and model should only slightly change to perform well on each different dataset. Looking closely at the design options, particularly we analyzed the effect of different activation functions, weight initialization techniques as well as regularization. We argue that not all the neural networks can perform well on every dataset as can be seen in our results, and their hyper-parameters should be chosen carefully to reach their ideal performance. While this work tries to have a broad overview of design choices and hyper-parameters more intuitively, further work can be done to focus on the nuances of the combination of different architectures and hyper-parameters with more theoretical groundings.

References

Lukas Biewald. 2020. Experiment tracking with weights and biases. Software available from wandb.com.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification.

Walaa Medhat, Ahmed Hassan, and Hoda Korashy. 2014. Sentiment analysis algorithms and applications: A survey. *Ain Shams Engineering Journal*, 5(4):1093–1113.

Vinod Nair and Geoffrey E. Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, page 807–814, Madison, WI, USA. Omnipress.

Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. 2018. Activation functions: Comparison of trends in practice and research for deep learning.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

Mazhar Iqbal Rana, Shehzad Khalid, and Muhammad Usman Akbar. 2014. News classification based on their headlines: A review. In *17th IEEE International Multi Topic Conference 2014*, pages 211–216.