Pulsar Classification using Deep Neural Networks

Kyle W. MacMillan Department of Computer Science
South Dakota School of Mines and Technology
Rapid City, SD 57701
kyle.macmillan@mines.sdsmt.edu

Abstract

Pulsars are rapidly rotating, highly magnetized, neutron stars and white dwarfs that emit focused electromagnetic radiation in a beam. The beam is only visible to us when it is directly facing Earth and is the reason for their pulsed nature. This paper approaches classification of pulsars and non-pulsars with the intent of learning more about TensorFlow. The classification method used is Deep Neural Networks and an acceptable level of accuracy was obtained.

1 Introduction

We were tasked with picking a dataset and training a deep learning network. First thing to do was to pick a dataset. Being an open-ended assignment there was uncertainty as to what particular dataset to explore. Initial looks involved different image classification datasets such as CIFAR-10, Caltech 101, and NORB. Down the road there is no desire to work with image data classification, work with files and datasets of continuous and discrete data are preferred. After thorough search of the available datasets it was decided pulsar classification provided by Lyon et al. [2015] was best suited this paper.

The last decision before pressing on to data format was to pick the correct deep learning framework for the job. There were several options but TensorFlow seemed like a good choice. Keras and PyTorch are other tools that utilize TensorFlow but were not used because those are often considered a "front end" for TensorFlow, which was not part of the goals of the assignment.

2 Method

2.1 Main

Main.py is where the project is ran from. There are several hyperparameters that a user can pick, beginning with $\_TEST\_PERCENTAGE$, which sets the percentage of comma-separated values (CSV) data to be utilized for testing. Industry standard is 20%, so that is the default setting. The training data $\_TRAIN$ is, as a result, 80% of the CSV data.

$$\_\mathcal{TRAIN} = \_NUM\_ELEMENTS * (1 - \_TEST\_PERCENTAGE)$$

It was necessary to hard-code in the number of elements in the CSV file because there was trouble when trying to run update on the CSV class due to an incomplete understanding of python abilities such as $class.update()$.

$\_BATCH\_SIZE$ is set as

$$\_\mathcal{BATCH\_SIZE} = max\{\_TRAIN * 0.1, 1\}$$

This is part of ensuring the training does not get stuck in local minima. _MODEL is then set as whichever optimizer the user would like, in this case, RMSPropOptimizer was selected. A CSV class is created based on the dataset to be classified.

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

Hyperparameter choices are part of the art of machine learning but in this project it was not the focus to make perfect hyperparameters. That being said, that was explored briefly for this project as seen in section 2.5. $\_NODES\_PER\_LAYER$ was one of such hyperparameters, as was $\_LAYERS$, and $\_HIDDEN\_LAYERS$. The final implementation of these parameters are as follows:

$$\_\mathcal{NODES\_PER\_LAYER} = (num\_features - 1) * 1.5$$

$$\_\mathcal{LAYERS} = max\{sqrt(\_NODES\_PER\_LAYER), 2\}$$

$$\_\mathcal{HIDDEN\_LAYERS} = [\_NODES\_PER\_LAYER] * \_LAYERS$$

This lead to 3 hidden layers with 12 nodes each for the HTRU_2 Dataset.

After all parameters have been setup main gets run. Users can select the number of runs to perform as an argument when they begin the program e.g.

python main.py *X*

The program will then run the classifier *X* times. After finishing the program will output classification accuracy of the given dataset.

2.2 Data format

After determining the dataset and the framework to be utilized it is necessary to figure out how to read data from the dataset. A modular approach was taken with this project. Users will be able to select a CSV file to load in. By specifying a class to contain the CSV data it is possible to run multiple datasets while making only minor modifications to the code. The format is also much easier to read as a result of the class format.

Rather than code up a custom method of moving CSV data into a format usable by the deep learning framework it was determined to "stand on the shoulders of giants". Pandas was used to bring CSV data into usable form. After pulling the data into a single structure it was necessary to split the data into training and testing data. Again, rather than code that from scratch it was determined best practice to utilize existing tools.

For that endevour scikit-learn was used, specifically the $train\_test\_split()$ method. As is standard for classification problems the data was segmented into 80% training and 20% testing data. It was then necessary to place the $class$ label into a data structure for later. For both the train and test data, associated labels are passed, along with the data, as tuples. The psuedocode for this section can be found under Algorithm 1.

Algorithm 1 $input\_fn()$

$data \leftarrow CSV$
$test, train \leftarrow train\_test\_split(data)$
$test\_label \leftarrow test.pop(class)$
$train\_label \leftarrow train.pop(class)$
return $(train, train\_label), (test, test\_label)$

2.3 TensorFlow estimator

TensorFlow has built in estimators such as:

- DNNClassifier
- BaselineClassifier
- BaselineRegressor
- DNNLinearCombinedClassifier
- LinearClassifier
- LinearRegressor

You also have the ability to build your own estimator, or classifier, and this is the route that was selected for this project. The custom estimator selected utilizes the $model$ specified at the start of the program, a $model\_dir$ for TensorBoard output which will be discussed in section 2.6, and the $params$ which consist of $feature\_columns$, $hidden\_units$, and $n\_classes$.

Estimator also requires three modes depending on the goal:

- ModeKeys.TRAIN
- ModeKeys.EVAL
- ModeKeys.PREDICT

Lastly, the estimator requires a return of $EstimatorSpec$. Documentation for the EstimatorSpec can be found on TensorFlow's website. It is important to note that each of the different $ModeKeys$ requires a different set of parameters for the returned $EstimatorSpec$.

2.4 Training

TensorFlow offers the ability to train estimators by calling a $train()$ method. For this paper the $input\_fn$ and $steps$ arguments are set by a function call to $train\_input\_fn$ and $csv.num\_examples['train']$ respectively. This allows the data to be segmented for training.

The model that was instantiated during estimator creation is detailed under model.model_RMSProp. For each of the layers that was passed to it under $params['hidden\_units']$ there is a fully connected layer built with selu activation. Klambauer et al. [2017] found with the proper math you can obtain self-normalizing exponential linear units, meaning they don't require batch normalization and that is why they were chosen over all of the other activation functions. There is also the final fully connected layer which has no activation layer. This final layer aids in determining the classification.

During training ModeKeys.TRAIN is called, which specifies RMSProp gradient descent as our optimizer and then calls $minimize()$. The $minimize()$ method will automatically compute the gradients and apply them to the variables, but if you need to tune the network there are additional steps that can be taken instead:

1. Compute the gradients with $compute\_gradients()$
2. Process the gradients as you wish
3. Apply the processed gradients with $apply\_gradients()$

Further documentation on this process can be found on TensorFlow's website under the class Optimizer. It was not necessary to have that level of fidelity with this particular dataset, so the base $minimize()$ was all that was necessary.

With $minimize()$ minimizing the loss we are able to make progress towards a classification. This dataset in particular minimizes loss to near-zero after just the first episode of 100 training steps as can be seen in Figure 5.

2.5 Hyperparameters

Hyperparameters are normally very important but in this dataset it was discovered that they had very little impact on the efficacy of this classifier.

An example of hyperparameter randomization can be found under:

python hyperparams.py

That file will automatically generate randomized hyperparameters. During testing of this project various hyperparameters were tested with no significant improvements over the initial parameters set so the associated code was removed from primary operation of the classifier. The file described would allow for random testing of hyperparameters across any number of computers. Bergstra and Bengio [2012] found random search of hyperparameters to be superior to grid search and was the method used while exploring hyperparameter tuning during this project.

2.6 TensorBoard

One of the other reasons for utilizing TensorFlow is the TensorBoard project. TensorBoard allows for visualization of various parameters as defined by the user. Figure 1 and 2 are the same results as shown in Figure 4 and 5 respectively. The smooothing helps in understanding what may be going on, especially in more turbulant graphs. Another benefit is being able to visualize the graph structure. To use TensorBoard on this project ensure it was installed with TensorFlow and type

tensorboard --logdir model

and go to the link it provides (by ctrl clicking or copy/paste).

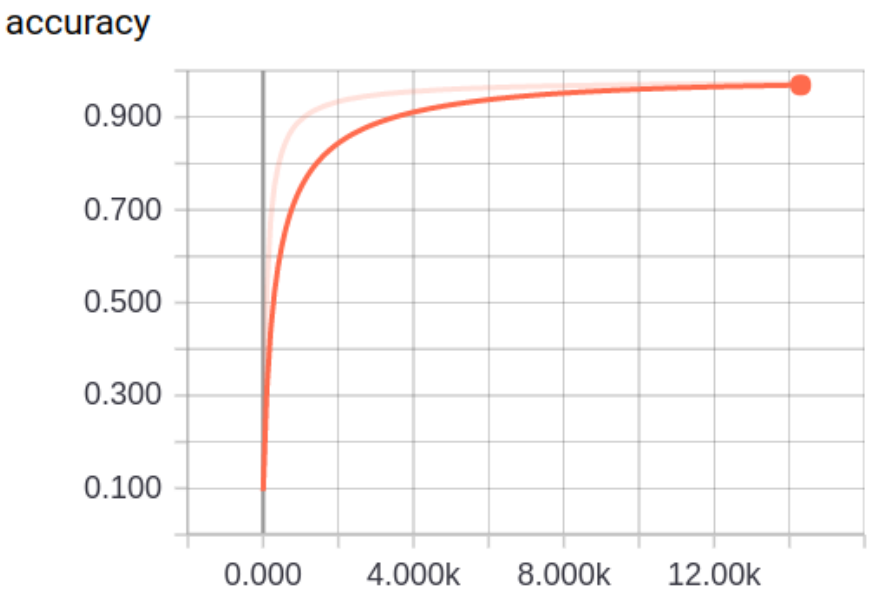

Figure 1: Smoothed accuracy

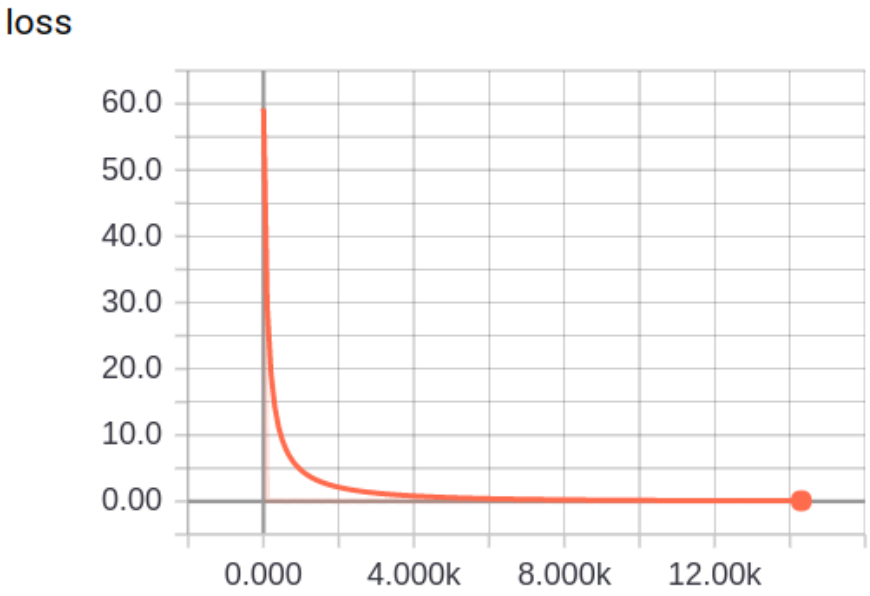

Figure 2: Smoothed loss

3 Results

RMSProp gradient descent with 0.001 learning rate, performing batch runs on 10% of the training samples every 100 steps produced an accuracy of 97.8% averaged over 20 tests. These were taken with 80% train and 20% test samples. Other hyperparameters were three fully connected layers with 12 nodes each using selu activation. Increasing hidden layers led to noisier loss after the large initial drop. The following had no measurable improvement on the classification accuracy:

- Changing from RMSProp to Adagrad
- Changing batch size
- Modified learning rate
- Depth of layers or number of neurons

```
INFO:tensorflow:Finished evaluation at 2018-04-19-01:46:51
INFO:tensorflow:Saving dict for global step 14318: accuracy
Accuracy =  0.978687149286
kyle@:ResearchProject_HTRU2$
```

Figure 3: Detailed accuracy

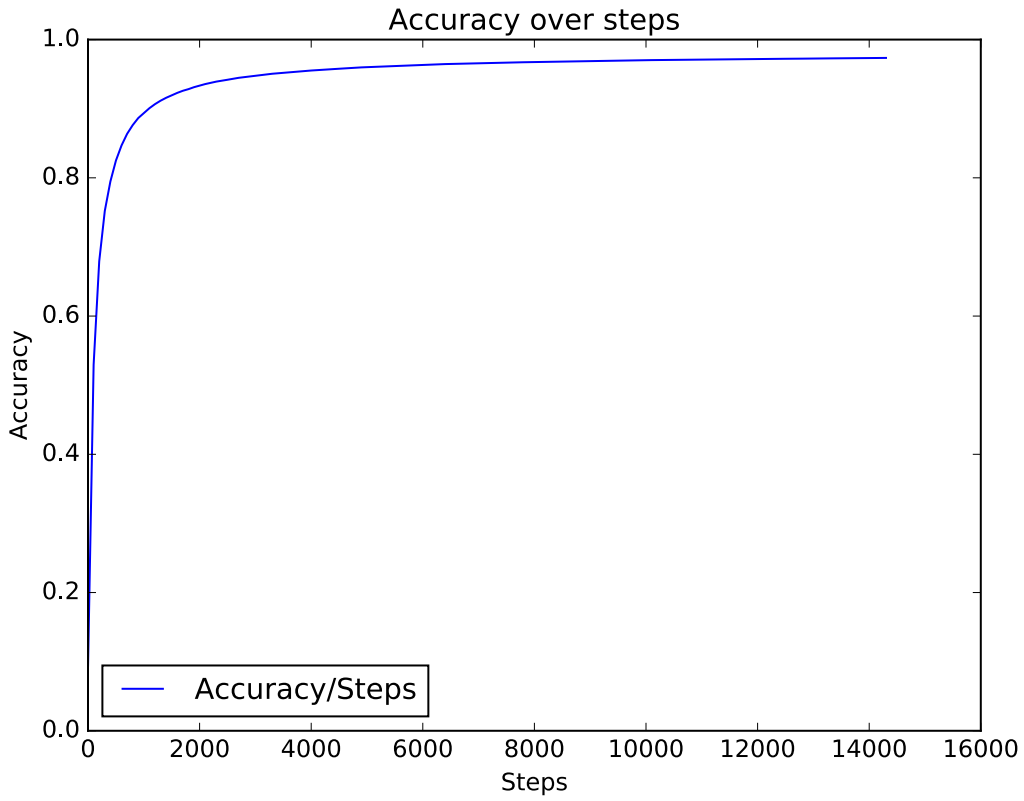

Figure 4: Accuracy

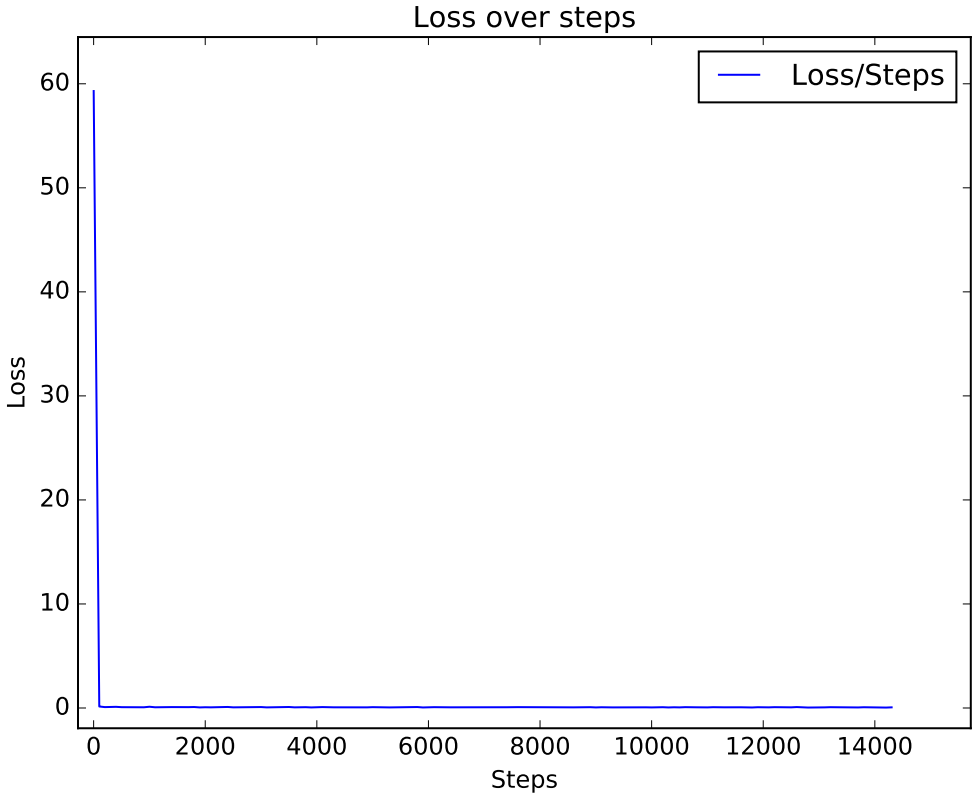

Figure 5: Loss

4 Conclusion and future work

In conclusion, this particular dataset ended up being relatively simple to classify with modern methods. The original paper reported the exact same accuracy of 97.8%. Room for improvement could be a better interface for the CSV files and possible command line arguments. Additionally, the accuracy would occasionally break 98%, but was rare. There is possible room for improvement on the hyperparameters to increase the accuracy consistently beyond 98%. The loss function dropped off almost immediately, so it could probably deal with fewer hidden layers and/or neurons. Another possible improvement may be adding extra feature layers that are extrapolated from the pulsars (or possibly non-pulsars).

References

James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Machine Learning Research*, 13(10):281–305, 2012. doi: http://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf.

Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. *arXiv*, 2017. doi: https://arxiv.org/pdf/1706.02515.pdf.

R.J. Lyon, B.W. Stappers, S. Cooper, J.M. Brooke, and J.D. Knowles. Fifty years of pulsar candidate selection: From simple filters to a new principled real-time classification approach. *arXiv*, 2015. doi: https://arxiv.org/abs/1603.05166.