Introduction

A brief introduction about this project. I started out using google colab to try and take advantage of using their GPU for faster training and grid searching. I then learned about their other option for runtime, TPU (Tensorflow processing unit) and was eager to play around with it as it is designed for running this kind of stuff. However, in the long run, I ended up having a lot of trouble when it came time to grid search as I lost a night of grid searching twice, due to restrictions put on kernals running over long periods of time. I thought I was checkpointing my grid searching efforts, but it turned out I misunderstood how Keras callbacks interact when used with GridSearchcv and I ended up not being able to restore my checkpointed progress after many hours of running.

After this disaster I chose to ditch trying to checkpoint my grid searching efforts and to move the project to a local anaconda environment with minimal packages installed apart from what I needed. This allowed me to use the parameter n\_jobs=-1 during my grid searching, which would account for about 90% of my running time over the course of the project. However, I ran into memory leak issues, causing my grid search to crash after hours of running. I then learned to use the pre\_dispatch parameter in the grid search along with n\_jobs in order to properly manage memory and cause the crashing to stop. These parameters were not possible to run on colab or azure notebooks due to the architecture of those softwares. Once I had figured out how to utilize these parameters locally, grid searching became a breeze thanks to my 6 core 12 thread CPU allowing me to run searches rather fast and make improvements in my model.

If I learned anything from this project, it’s that if I were to do it again, I would start locally with anaconda instead of trying to make use of GPU’s and TPU’s hosted in the cloud. I’ve learned that thankfully my machine is strong enough to be able to train deep learning algorithms locally. Now on to the project.

Imports

Here are our imports, as is expected we are using NumPy, pandas and popular graphing library matplotlib. We are generating our dataset using sklearn’s make\_blobs from their sample generator library and we are building our neural net using Keras. Some of the more uncommon imports here would be seaborn, and google.colab.

Seaborn is a python statistical data visualization library based on matplotlib, it provides a high level interface for drawing attractive and informative statistical graphs. I quite like some of the graphs that can be made using Seaborn so I may use this over matplotlib at times, hopefully to enable better explanation of the data and findings.

I use google colab import drive in order to properly implement checkpointing in my grid searches, due to the nature of colab, it is isolated from your personal google drive in its runtime. So by importing drive I can save all the .hdf5 files that I want, containing my best weights and parameters for my grid searches.

Creating the dataset

To create the dataset we will be using sklearn.datasets make blobs method, this method generates isotropic gaussian blobs designed for clustering. I am using the following assigned parameters for creation of the dataset

* n\_samples = 100,000
* centers = 5
* n\_features = 86
* cluster\_std = 123
* seed = 23

I then split the data in a 80:20 train test split, where we shall develop and train our model on the 80% and keep the remaining 20% of the dataset for unseen testing once we have our finished model

Data Exploration

Here we will attempt to do some exploration of the blob dataset. First, I will turn the data into a pandas data frame in order to make exploratory data analysis easier.

We will look at the shape of our data, you might notice we have 87 columns, this is perfectly fine as one of our columns is Y, not part of the 86 features.

Below we can see some of the statistics of our data, including the mean and std. However, this isn’t incredibly useful as from what we can tell by looking, each feature is rather similar in values. None seem to be incredible outliers from our initial inspection

To try get a bit of a better visualisation of the data, below I take five feature columns and the mean of their five clusters. Then I fit them to a barplot for visualising. We can see that features x81 and x20 and x43 have seemingly randomly distributed values for means across all five of the clusters, where features x81 and x23 have mostly positive means. This doesn’t tell us much of anything, however. Just that the data we’re working with has indeed got some variance when it comes to features mean values.

Here we try to go into a bit more detail, investigating various metrics of one of the clusters to see can we identify anything that would be telling of any traits about our data. The first thing I see is that we have a bell curve standard deviation which is to be expected. We have a bimodal graph for our mean, this strikes me as an interesting finding and we will keep it in mind in the future.

I thought we should do some principal component analysis to perhaps get a better understanding of the data however we can see there are no gains to be made from dimensionality reduction. All of the X features seem to be linearly independent, meaning PCA is no use in this case.

Modelling

Here we will begin to create our model, first we must pre-process our X data by scaling it using standard scaler. We then change our Y data from integer data to categorical data.

After playing around with the network topology for a while and checking accuracies. I landed on using 7 hidden dense layers. I also stuck with Adam as I was seeing promising results using this optimizer over a few others I tested, however Adamax and Nadam were also showing decent results. This seemed to provide the best learning model as far as I could tell at this point. We are using Relu activation function and our number of neurons starts at 1000, then progressively lowers to 750, 250 then to 100. We have three layers with 100 neurons as I found this helped, then to 50, 20 and finally 5 for our classification using activation function SoftMax. Due to having 5 classes, we shall be using categorical cross entropy for our loss when compiling our model. I make use of Keras ModelCheckpoint to checkpoint any improvements in my weights and fit the model with a validation split of 20%. I was initially using 10 epochs with the default batch size of 32. We can see from the graph that is plotted below the code cell that the model quickly becomes overfitted as it begins to learn the data. Straight away from seeing this graph my initial thoughts were to reduce overfitting.

I decided it would be a good idea to make use of sklearns dummy classifier model to test against my chosen network topology and see how it stacks up. The dummy classifier will provide us with an idea of baseline performance, for example, a success rate that you should expect to receive by simply guessing. I was quite happy with the results of this experiment as my model outperformed the dummy classifier by roughly 8% accuracy, meaning my model was working. Our dummy classifer had a test accuracy score of 19.5% and our model had a test accuracy score of 27.2%! Taking into consideration the size and size of parameters of our blob dataset, and the nature of generated isotropic gaussian blob data in general, this is quite a good result to achieve. I was happy with our initial network topology after this comparison, and decided it was time to move onto finer hyperparameter tuning using GridSearchCV.

From looking at our previously plotted graph using our initial model, we can tell the model is becoming overfitted, we want to try our best to reduce this if we can. This led me to thinking about adding dropout layers after our dense layers to try combat this issue. I also thought that we could maybe try out the three different optimizers that I found were outputting the best results for me. I also wanted to try a few different learning rates to see if we could get our optimizers to train our model more accurately with dropout in the equation. So I created a parameter grid for our grid search, made up of my three choices of optimizer, a few logarithmic learning rates and some drop out values (including a drop out rate of 0 just in case drop out layers proved to be of no benefit to model accuracy). We ran the grid search locally utilizing n\_jobs parameter and setting it to -1 to run the grid search using all 12 threads of my CPU’s 6 cores. We also made sure to set a safe value for pre\_dispatch parameter (I went with 8 in this case) to make sure we avoided running into a nasty memory leak as I found this was a problem initially, unfortunately resulting in a couple of failed grid searches. Thankfully (after a few hours) our grid search completed, giving us a best accuracy of 29.5% using the following parameters

* 0.0 dropout rate
* 0.0001 learning rate
* Adamax optimizer

To my surprise, our dropout layers didn’t help combat overfitting at all, and gave us poorer accuracy in general.

After reaching our optimal parameters from the last grid search, it appeared that using dropout layers had proved to not be of any benefit to our model training. Due to this we cut out the dropout layers and decided to do a grid search for optimal batch size and epochs using our best optimizer, Adamax.

Our second grid search gave us a best accuracy of 30%! Our model is now 11% more efficient than the dummy model. This might not sound like a lot, but every improvement no matter how small, is still an improvement in our model and positive overall. The best parameters for this grid search was to run just 2 epochs with a batch size of 50 using optimizer Adamax

I was happy to use these best parameters from our two grid searches in our final model. We created our final model, and made sure to save our model as a JSON file, along with saving our best weights improvement to a hierarchical data format file for use in our prediction later. It’s now time to implement early stopping in our final model in order to get the best accuracy. The model stopped early after 4 epochs, selecting the best accuracy as epoch 3 with a test accuracy score of 30.2%

It’s time to predict our model using our final model we saved to a json file and our best weights from our early stopping in the final run of our model. We run the model on our unseen test data that we put aside during data preprocessing and see an accuracy of 30.5% along with a loss of 1.545. We have successfully trained our model to perform upwards 50% more efficiently than guessing or using a dummy model.

Conclusion: Our model can predict which of the 5 clusters a value belongs to, with 30% certainty. I learned a lot of valuable information from this project. I had a lot of trouble with many parts of the project including using cloud hosted kernals, file output, grid searches and much more. I now know how to do all of these things and have a better understanding of how to train a deep neural network using Keras.