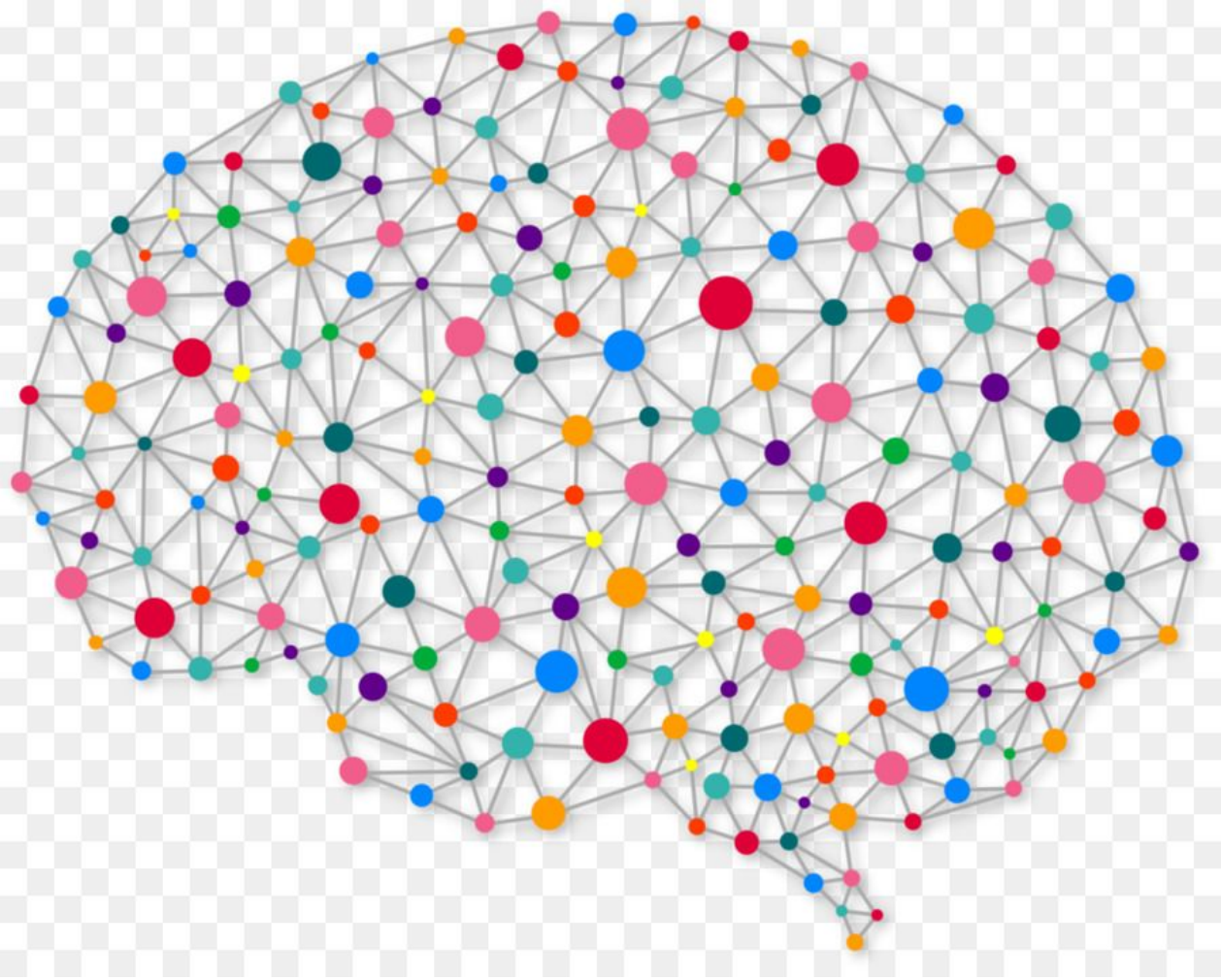
CMPT 412

Project 6

Deep Learning for Scene Recognition

USING 1 FREE LATE DAY



3.1 Tensorflow Installation

In this part, we will install the packages and requirements needed to train and test our model. We will be using Google Colab throughout the project as it provides unique features that are provided to us for free. Colab also takes advantage of Python Notebooks which is a natural and popular coding environment.

Initially, I uploaded the provided project package to a Google Drive account which can be integrated and used in Colab. Then, we create a new Google Colaboratory Notebook directly from Google Drive, and we select New Python 3 Notebook from the options. Also, since Colb provides free GPU, we make sure to select GPU as our hardware accelerator in the runtime options. Moreover, using the simple code snippet below, we are able to mount the Google Drive and use the drive in a natural way, as if we are working locally. This way, we don't need to worry about read/write permissions throughout the training process. As we can see, the drive is successfully mounted and ready to use after a simple authentication procedure.

```
[ ] from google.colab import drive
    drive.mount('/content/drive')

Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8qdgf4n4g3pfee64

Enter your authorization code:
··········
Mounted at /content/drive
```

We will now install the required packages which we are going to need to train our model. The following lines of code are used to install Tensorflow 1.15.0, Tensorpack 0.8.9, as well as NumPy 1.16.2. These packages are able to work together without any version mismatch issues. Note that '!' is required to tell Colab that we are writing using command line tools and not writing code.

```
1.  !pip3 install --upgrade tensorflow==1.15.0
2.  !pip3 install --upgrade tensorpack==0.8.9
3.  !pip3 install numpy==1.16.2
```

Now, we are ready to run our code. We will do so using the code snippet below in order to select Task 1 and tell the system to use GPU. Note that we need to specify the exact location of both run.py and the data directory.

```
!python "/content/drive/My Drive/project6_package/code/run.py" --task 1
--gpu 0 --data "/content/drive/My Drive/project6_package/data/"
```

Running the default code without any modifications resulted in an accuracy of ~30%. However, by applying a few techniques, I was able to boost the accuracy to ~53%. The techniques used to boost accuracy as well as experimental results will be discussed in more detail in the next section.

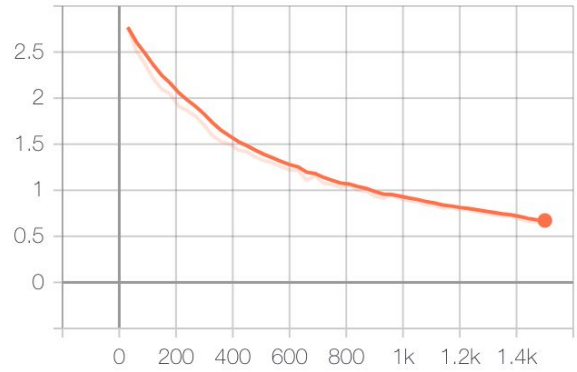
3.2 Training from Scratch

In this part, we are going to demonstrate the results of training our model from scratch after applying some techniques to improve the accuracy, using a few graphs generated by TensorBoard.

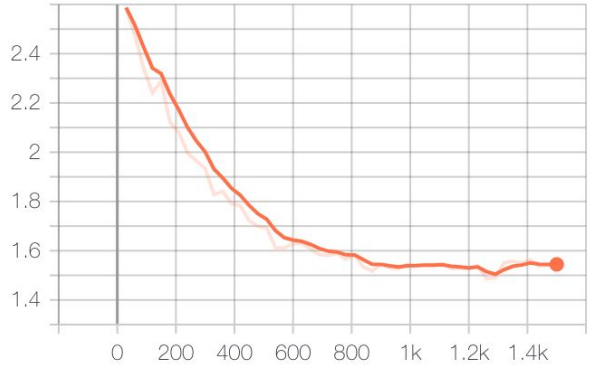
The methods used to boost the accuracy are as follows. Standardization (feature normalization) by subtracting the image by the mean and dividing by the standard deviation. Dropout regularization by adding a dense layer with its own ReLU layer, and a dropout layer with probability of 50% before the fully connected layer. Finally, data augmentation was used, specifically, flipping 60% of the images horizontally in order to “fake” more data that is still visually acceptable (flipping vertically would result in odd scenes).

The generated train_log folder is then passed to TensorBoard in order to visualize the results. The graphs can be found on the next page. A smoothing of 0.6 is applied in TensorBoard to make it easier to see the trend. However, the original results are shown behind the graph in a fainter colour. We can see that validation error is at 47% which puts the accuracy at 53%. Moreover, the learning rate used throughout the entire training is 0.1 and the number of epochs was set to 50.

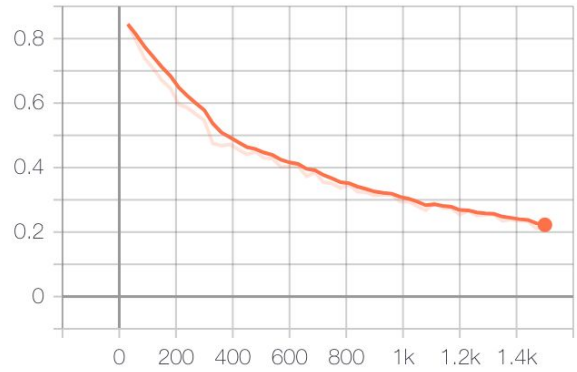
cross_entropy_loss



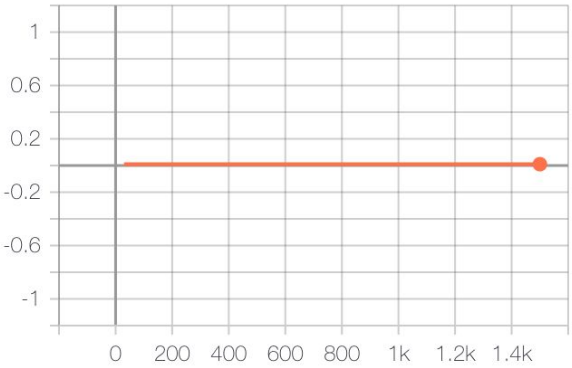
validation_cost



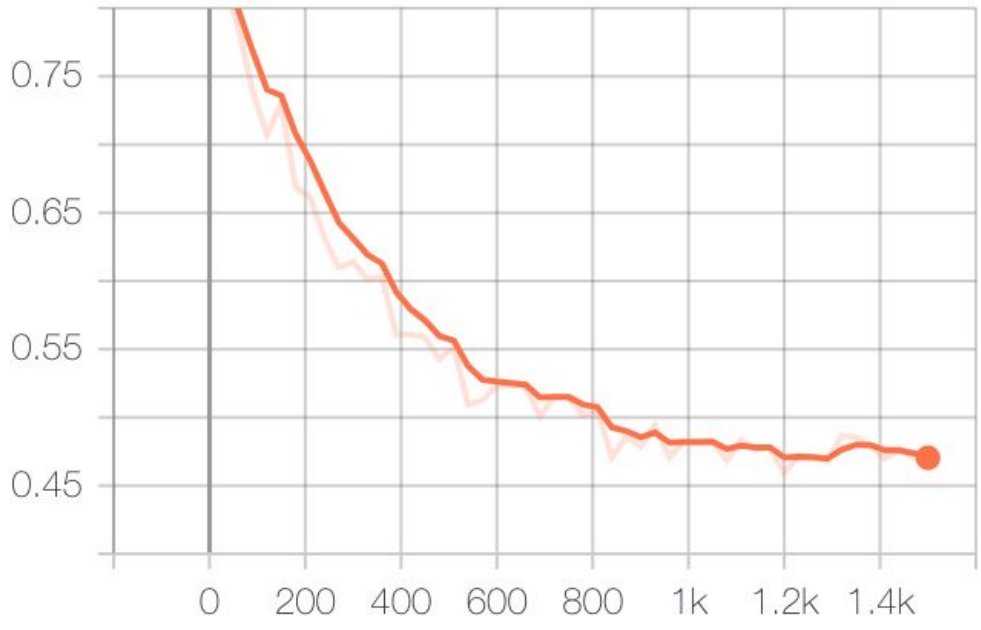
train_error



learning_rate



validation_error



3.3 Fine-tuning VGG

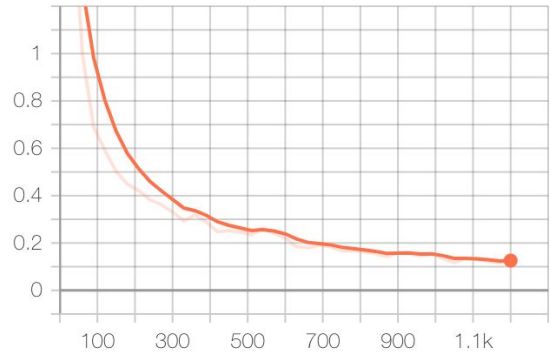
In this part, we are going to upload the pretrained VGG model to Google Drive and attempt to fine-tune the parameters in order to achieve much higher accuracy. The advantage of using the provided GPU from Google Colab is the speed of training. For Task 2, I have been able to achieve an outstanding speed of 15 - 20 seconds per epoch for training. This allowed me to experiment with a few different hyperparameters and optimizers and observe which is the most successful.

After trying many combinations of hyperparameters, optimizers, and data augmentations, I decided to use the following: an Adam Optimizer seemed to be the appropriate choice, especially due to its fast learning. I was surprised by how quickly the loss was dropping and I was achieving remarkable accuracy after only a few epochs! This was not the case with using the RMS optimizer as it seemed to be a very slow learner even when using momentum. In addition, I augmented more images, namely 75% of the images were flipped horizontally and standardization was applied as in part 3.2. The number of epochs was set to 40 and learning rate was reduced to 0.0015. Also, stop_gradient was used for all layers except the last FullyConnected layer which we are trying to optimize and fine-tune. Using all the methods above, I was able to achieve ~88% accuracy or 12% error as can be seen in the screenshot below. The screenshot shows the results after epoch 40 which is the last epoch.

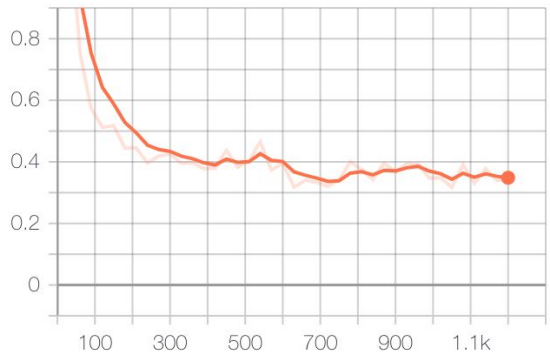
```
[1204 21:38:31 @base.py:250] Start Epoch 40 ...
100% 30/30 [00:14<00:00,  1.67s/it]
[1204 21:38:46 @base.py:260] Epoch 40 (global_step 1200) finished, time:14.9 seconds.
[1204 21:38:48 @saver.py:77] Model saved to train_log/run/model-1200.
100% 30/30 [00:09<00:00,  3.35it/s]
[1204 21:38:57 @monitor.py:440] cross_entropy_loss: 0.12899
[1204 21:38:57 @monitor.py:440] learning_rate: 0.0015
[1204 21:38:57 @monitor.py:440] train_error: 0.038013
[1204 21:38:57 @monitor.py:440] validation_cost: 0.34164
[1204 21:38:57 @monitor.py:440] validation_error: 0.11867
[1204 21:38:57 @group.py:48] Callbacks took 10.972 sec in total. InferenceRunner: 9.12 seconds
[1204 21:38:57 @base.py:264] Training has finished!
```

Below are the graphs of the above model generated by TensorBoard. Note that the same smoothing of 0.6 is applied to the graphs as before. Also, we can see that using Adam Optimizer results in much faster convergence, namely the loss and errors are very steep in the initial iterations and drop very quickly.

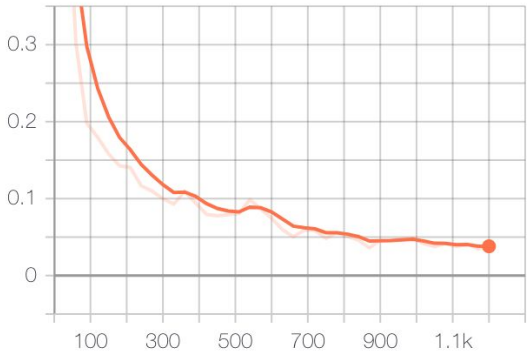
cross_entropy_loss



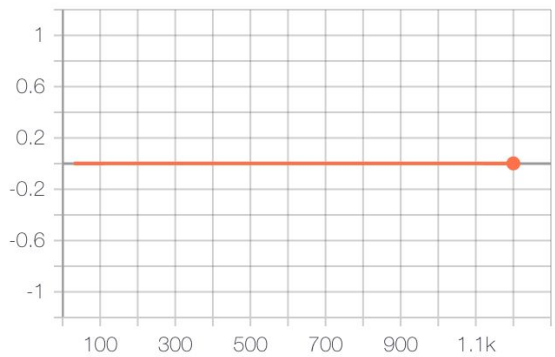
validation_cost



train_error



learning_rate



validation_error