CMPT 898 Assignment 3 Solution

<u>Question 1:</u> This question required us to reimplement LeNet-5 with a ReLu activated hidden layers and a softmax output layer. For my implementation (and for all successive questions) I used the Cifar10 dataset.

First, I had to lay out some generalized parameters and functions. To start, I imported and normalized the test data:

```
import tensorflow as tf
import numpy as np
import os, datetime
import math

# Hyper-parameters that remain constant throughout all questions
EPOCHS = 10
BATCH = 256
OPTIMIZER = 'adam'
LOSS = 'sparse_categorical_crossentropy'
METRICS = 'accuracy'

# Grab the Cifar10 dataset, which is a color image database consisiting of
# 10 different classes representing airplanes, cars, birds, cats, deer, dogs, fro
gs, horses, ships, and trucks
(x_train, y_train),(x_test, y_test) = tf.keras.datasets.cifar10.load_data()

#Normalization of x_train and x_test and split into training dataset and testing
dataset
x_train, x_test = x_train / 255.0, x_test / 255.0
```

From this block of code we can also see I set up some constants that remained the same for all models: **training epochs** (EPOCHS) = 10, **batch size** (BATCH) = 256, **network optimizer** (OPTIMIZER) = ‘adam’, **loss function** (LOSS) = ‘sparse_categorical_crossentropy’, and **network optimization metric** (METRICS) = ‘accuracy’. With these constant throughout all network, it allows easier comparison of results.

Next, I created a generalized model training function that takes in a model and trains it given the above constant parameters:

```
# Trains the model given the constant parameters and returns the model and the fi
t history
def train_model(model):
  model.compile(optimizer=OPTIMIZER,
                loss=LOSS,
                metrics=[METRICS])

  logdir = os.path.join("logs", datetime.datetime.now().strftime("%Y%m%d-
%H%M%S"))
  tensorboard_callback = tf.keras.callbacks.TensorBoard(logdir, histogram_freq=1)
  history = model.fit(x=x_train,
                      y=y_train,
                      epochs=EPOCHS,
                      batch_size=BATCH,
                      validation_data=(x_test, y_test),
                      callbacks=[tensorboard_callback])

  return (model, history)
```

With the model training generalized as above, it allows easy swapping in and out of model. Additionally, this training function returns both the trained model and the training history, which allows easy training performance analysis and model prediction.

Finally, to satisfy question 1, I reimplemented and trained LetNet-5 as it was described in lecture:

```
# LeNet-5 as described in the lecture notes
def create_baseline_model():
  return tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(
        filters=6,
        kernel_size=5,
        activation='relu',
        input_shape=(32, 32, 3)),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Conv2D(
        filters=16,
        kernel_size=5,
        activation='relu'),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(120, activation='relu'),
    tf.keras.layers.Dense(84, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')])

train_model(create_baseline_model())
```

When trained for 10 epochs with the above parameters on the Cifar10 dataset, the baseline model achieved a **training accuracy** of **0.6191** and a **test accuracy** of **0.6022.**

Question 2: This involved applying a L2 regularization of two specific strengths to the baseline model. As no indication of when to apply the regularization was given, I chose to apply it after every layer. The two lambda levels I chose were 0.001 and 0.01. My decision to apply a regularization after each layer likely affected the accuracy of the final, as L2 regularization models preformed worse than the baseline.

Firstly, I implemented a generalized regularized LetNet-5 model function, which allows question 2 and 3 to reuse the same code (while swapping out the regularization function):

```python
# A model that applies an regularization function at a specific strength at every
 layer
# Reused by both question 2 and 3
def create_regularized_model(regularizer, reg_strength):
  return tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(
        filters=6,
        kernel_size=5,
        activation='relu',
        input_shape=(32, 32, 3),
        kernel_regularizer=regularizer(reg_strength)),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Conv2D(
        filters=16,
        kernel_size=5,
        activation='relu',
        kernel_regularizer=regularizer(reg_strength)),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(
        120,
        activation='relu',
        kernel_regularizer=regularizer(reg_strength)),
    tf.keras.layers.Dense(
        84,
        activation='relu',
        kernel_regularizer=regularizer(reg_strength)),
    tf.keras.layers.Dense(10,
        activation='softmax',
        kernel_regularizer=regularizer(reg_strength))])
```

This function has identical topology as the baseline model, but with regularization applied after each step.

Next two L2 regularized models were created and trained with two different values for lambdas (0.001 and 0.01 respectively):

```
LAMBDA1 = 0.001
LAMBDA2 = 0.01

print(f"Training model 1 with L2 regularization and a lambda of {LAMBDA1}")
train_model(create_regularized_model(tf.keras.regularizers.l2, LAMBDA1))
print(f"Training model 2 with L2 regularization and a lambda of {LAMBDA2}")
train_model(create_regularized_model(tf.keras.regularizers.l2, LAMBDA2))
%load_ext tensorboard
%tensorboard --logdir logs
```

The first L2 model, using a lambda of 0.001, achieved a **training accuracy** of **0.5359** and a **test accuracy** of **0.5328.** The second L2 model, using a lambda of 0.01, achieved a **training accuracy** of **0.4507** and a **test accuracy** of **0.4570.** As both models performed worse in respect to accuracy when compared to the baseline, I would suggest reducing the number of times the model is regularized. The over regularization appears to be adding to much noise to the model, reducing overall accuracy.

Question 3: This question essentially asked to reimplement question 2 but with an L1 regularize. As a generalized regularized model generating function was already created in the last question, all this question involved was passing it a different regularization function:

```
LAMBDA1 = 0.001
LAMBDA2 = 0.01

print(f"Training model 1 with L1 regularization and a lambda of {LAMBDA1}")
train_model(create_regularized_model(tf.keras.regularizers.l1, LAMBDA1))
print(f"Training model 2 with L1 regularization and a lambda of {LAMBDA2}")
train_model(create_regularized_model(tf.keras.regularizers.l1, LAMBDA2))

%load_ext tensorboard
%tensorboard --logdir logs
```

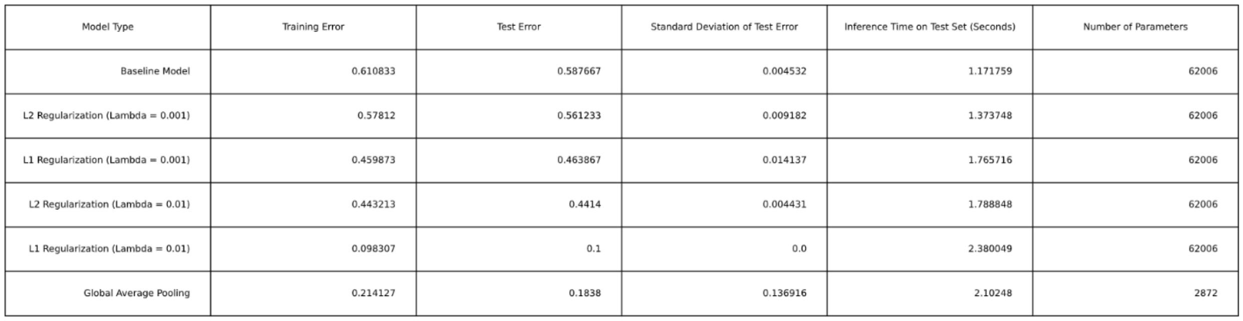
The first L1 model, using a lambda of 0.001, achieved a **training accuracy** of **0.4605** and a **test accuracy** of **0.4492.** The second L1 model, using a lambda of 0.01, achieved a **training accuracy** of **0.0987** and a **test accuracy** of **0.1000.** Over regularization is a problem with this model scheme as well. It is worth noting that when the L1 regularization method was given a lambda of 0.01, it achieved accuracy consistent with randomly guessing.

<u>Question 4:</u> This question involved removing the fully connected end layer to of the network and replacing it with a global average pooling. The model is identical to the start of the baseline network, before the fully connected portion:

```
#A model that applies an L1 regularization at every layer
def create_average_pooling_model():
  return tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(
        filters=6,
        kernel_size=5,
        activation='relu',
        input_shape=(32, 32, 3)),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Conv2D(
        filters=16,
        kernel_size=5,
        activation='relu'),
    tf.keras.layers.GlobalAveragePooling2D()])

print(f"Training model with Global Average Pooling")
train_model(create_average_pooling_model())

%load_ext tensorboard
%tensorboard --logdir logs
```

This Global Average Pooling model achieved a **training accuracy** of **0.3553** and a **test accuracy** of **0.3571.** This is significantly worse than the baseline or most of the regularized models.

<u>Question 5:</u> For this question, a performance table was generated comparing all the models:



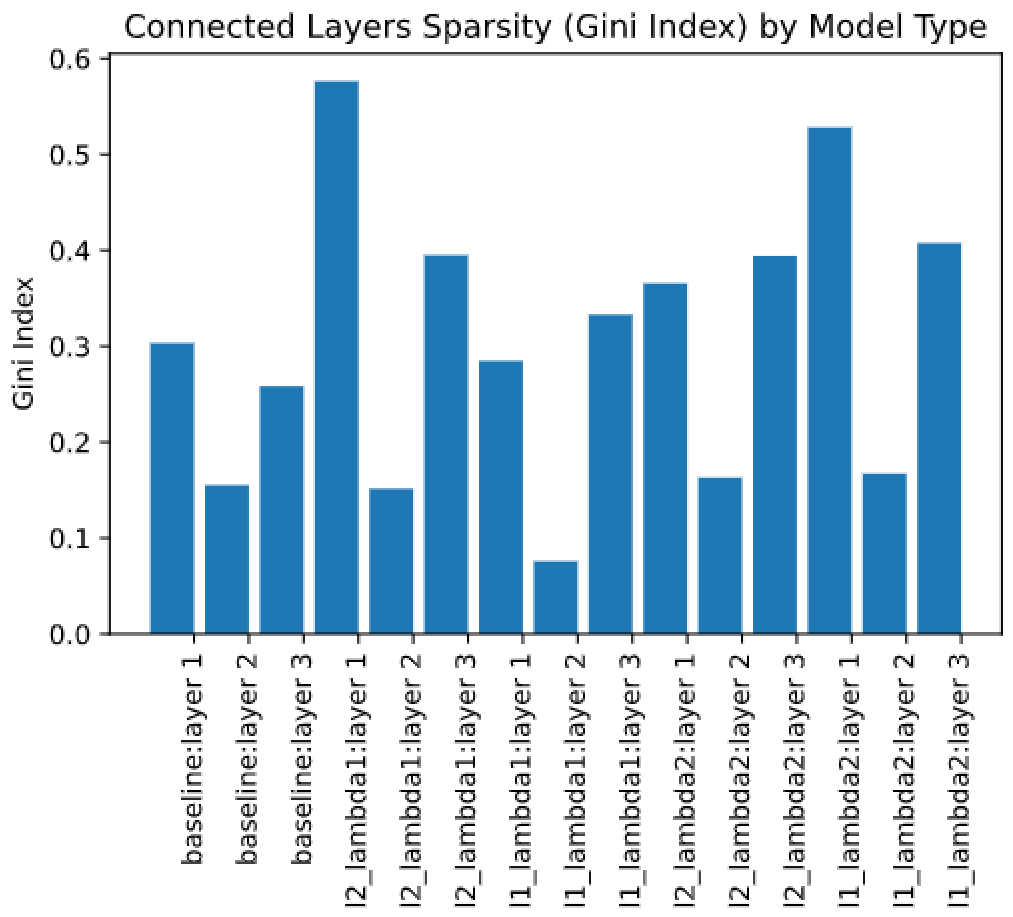
Model Type	Training Error	Test Error	Standard Deviation of Test Error	Inference Time on Test Set (Seconds)	Number of Parameters
Baseline Model	0.610833	0.587667	0.004532	1.171759	62006
L2 Regularization (Lambda = 0.001)	0.57812	0.561233	0.009182	1.373748	62006
L1 Regularization (Lambda = 0.001)	0.459873	0.463867	0.014137	1.765716	62006
L2 Regularization (Lambda = 0.01)	0.443213	0.4414	0.004431	1.788848	62006
L1 Regularization (Lambda = 0.01)	0.098307	0.1	0.0	2.380049	62006
Global Average Pooling	0.214127	0.1838	0.136916	2.10248	2872

Each value within the able is an average of all three of the training sessions. Additionally, all parameters of all the models were trainable, with the only one to have a different number of parameters being the global average pooling (as was expected). The best performing model was the baseline. For the L2 model (with lambda = 0.001) with reduced regularization, its overall performance would likely be better than it is shown above.

Question 6: Finally, for question 6, the sparsity of the weights of all models with a fully connected layer was to be compared. Based on the included paper (https://arxiv.org/pdf/0811.4706.pdf), the Gini Index was chosen as the best measure of sparcity. My implementation of the Gini Index is based off of that of GitHub user oliviaguest (https://github.com/oliviaguest/gini/blob/master/gini.py):

```
# Gini Index as described in https://github.com/oliviaguest/gini/blob/master/gini
.py. Expects a tf.Variable of weights as input
def gini_index(weight_vect):
    flat_weight = tf.keras.backend.flatten(weight_vect).numpy()

    # Remove negative values
    if np.amin(flat_weight) < 0:
        flat_weight -= np.amin(flat_weight)

    # Gini index cannot work with values of 0
    flat_weight += 0.0000001

    sorted_weight = np.sort(flat_weight)
    N = len(sorted_weight)
    i = np.arange(1, N+1)

    return ((np.sum((2 * i - N - 1) * sorted_weight)) / (N * np.sum(sorted_weight
)))

# This max depth will be the same for all models
max_connected_depth = len(model_results[key][0][0].weights)

# Min depth is found using magic number 3. This comes from the number of fully co
nnected layers at the end of each network
min_connected_depth = max_connected_depth - 3

plt_labels = list()
plt_vals = list()

# Collects values for figure, skipping global average pooling
for key, model_tuple in model_results.items():
    model = model_tuple[0][0]
    if key == 'global_average_pooling':
        continue
    for i in range(min_connected_depth, max_connected_depth):
        plt_labels.append(f"{key}:layer {i}")
        plt_vals.append(gini_index(model.weights[i]))

plt.figure()
barwidth= 0.8
plt.bar(np.arange(len(plt_vals)),plt_vals, barwidth)
plt.gca().set_xticks(np.arange(len(plt_vals))+barwidth/2.)
plt.gca().set_xticklabels(plt_labels)
plt.xticks(rotation='vertical')
```

The above code produced the following sparsity bar chart:



For interpretation, a higher Gini Index indicates a higher sparsity. The most uniform/least sparse model layer was that of L1 regularization (with lambda of 0.001) on layer 2. The sparsest model was L2 regularization (with lambda of 0.001) on layer, with the largest Gini Index of just under 0.60.