Joseph Huang, Nithi Subbaian
Prof. Curro

Midterm Project

Assignment:

Research paper to replicate: http://ee.cooper.edu/~curro/cgml/week4/paper8.pdf
(Understanding intermediate layers using linear classifier probes)

Images to Replicate: Figures 5 and 8

Figure 5:

Information for Figure 5:

For this part of the assignment, we had to replicate figure 5 which uses the MNIST convolutional model given in *tensorflow/models/image/mnist/convolutional.py*. In this figure, the test prediction error is plotted at the beginning and end of training for a probe inserted at each layer. Things to note: For Figure 5a, there is a decrease in the first couple of layers as the first ReLU has a big impact. For Figure 5b, there prediction error mostly decreases at every layer.

Code for Figure 5:

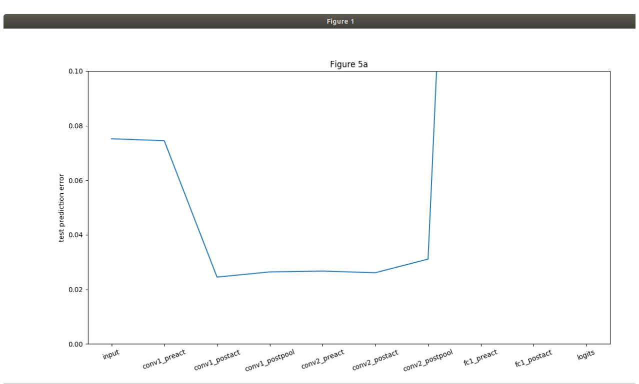
```
import tensorflow as tf
from tensorflow.keras import Model, Sequential, layers
from tensorflow.keras.layers import Conv2D, Flatten, Dense, ReLU, MaxPool2D, Softmax, Dropout
from tensorflow.keras.regularizers import l2
import numpy as np
import matplotlib
matplotlib.use('tkagg')
import matplotlib.pyplot as plt
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
import logging
tf.get_logger().setLevel(logging.ERROR)

IMAGE_DIM = 28
NUM_CHANNELS = 1
NUM_LABELS = 10
NUM_EPOCHS = 10
BATCH_SIZE = 512

mnist = tf.keras.datasets.mnist
(x_train, y_train),(x_test, y_test) = mnist.load_data()

rand_index = np.arange(50000)
np.random.shuffle(rand_index)
x_train = x_train[rand_index]
y_train = y_train[rand_index]
rand_index = np.arange(10000)
np.random.shuffle(rand_index)
x_test = x_test[rand_index]
y_test = y_test[rand_index]

x_test = x_test.reshape(x_test.shape[0], IMAGE_DIM, IMAGE_DIM, NUM_CHANNELS).astype('float32')/255.0
x_train = x_train.reshape(x_train.shape[0], IMAGE_DIM, IMAGE_DIM, NUM_CHANNELS).astype('float32')/255.0
```

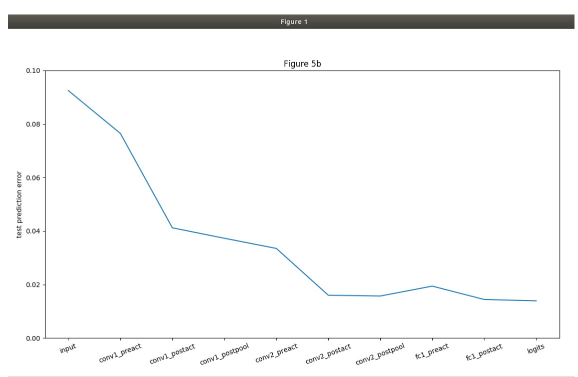
```
XTRAIN_LENGTH = len(x_train)

probe_layer = {0:"input", 1:"conv1_preact", 2:"conv1_postact", 3:"conv1_postpool", 4:"conv2_preact",
 5:"conv2_postact", 6:"conv2_postpool", 8:"fc1_preact", 9:"fc1_postact", 10:"logits"}

probe_for_graph = ["input", "conv1_preact", "conv1_postact", "conv1_postpool", "conv2_preact",
 "conv2_postact", "conv2_postpool", "fc1_preact", "fc1_postact", "logits"]

class linearClassifier(layers.Layer):
	def __init__(self):
	super(linearClassifier, self).__init__()
	self.f1 = Flatten()
	self.d1 = Dense(NUM_LABELS)

	def call(self, x):
	return self.d1(self.f1(x))

class MyModel(Model):
	def __init__(self):
	super(MyModel, self).__init__()

	# list of layers
	self.my_layers = []

	# the i-th entry represents a probe inserted before the i-th layer
	self.probes = {}

	# index of probe being trained
	self.probe_layer_num = -1

	self.add_probe(0)

	self.my_layers.append(Conv2D(32, [5, 5], strides=(1, 1), padding='same'))
	self.add_probe(1)

	self.my_layers.append(ReLU())
	self.add_probe(2)

	self.my_layers.append(MaxPool2D(pool_size=(2, 2), padding='same'))
	self.add_probe(3)

	self.my_layers.append(Conv2D(64, [5, 5], strides=(1, 1), padding='same'))
	self.add_probe(4)

	self.my_layers.append(ReLU())
	self.add_probe(5)

	self.my_layers.append(MaxPool2D(pool_size=(2, 2), padding='same'))
	self.add_probe(6)

	self.my_layers.append(Flatten())
	self.my_layers.append(Dense(512, kernel_regularizer=l2(5e-4), \
	bias_regularizer=l2(5e-4)))
	self.add_probe(8)

	self.my_layers.append(ReLU())
	self.d1 = Dropout(0.5)
```

```
        self.add_probe(9)

        self.my_layers.append(Dense(NUM_LABELS, kernel_regularizer=l2(5e-4), \
        bias_regularizer=l2(5e-4)))
        self.add_probe(10)

        def add_probe(self, key):
        self.probes[key] = linearClassifier()

        def call(self, x):
        if self.probe_layer_num == -1: # for network training
        for (i, layer) in enumerate(self.my_layers):
        if i == 9:
        x = self.d1(x)
        x = layer(x)
        return x
        else: # for probe training
        for layer in self.my_layers[0:self.probe_layer_num]:
        x = layer(x)
        x = tf.stop_gradient(x)
        probe = self.probes[self.probe_layer_num]
        return probe(x)

model = MyModel()

# Optimizer for probes
# optim_probe = tf.keras.optimizers.RMSprop(learning_rate=0.01, decay=0.9, momentum=0.9, \
#       epsilon=1e-6, centered=True)

def train_probes(weights):
        probe_errors = []

        # early_stop = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=3, \
        #       restore_best_weights=True)
        # We previously implemented early stopping, but the results were not as pleasant,
        # so we resulted to training for many many epochs

        for probe_layer_num in model.probes.keys():
        model.reset_metrics()
        model.probe_layer_num = probe_layer_num
        model.compile(optimizer=optimizer, loss=loss_object, metrics=['accuracy'])
        model.set_weights(weights)

        model.fit(x_train, y_train, batch_size=BATCH_SIZE, epochs=500, \
        verbose=2, validation_split=1/6)
        test_loss, test_accuracy = model.evaluate(x_test, y_test, verbose=2)
        probe_errors.append(1 - test_accuracy)
        print("Error for probe ", probe_layer[probe_layer_num], ":", 1 - test_accuracy)

        return probe_errors

lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(0.01, XTRAIN_LENGTH, 0.95, staircase=True)
optimizer = tf.optimizers.SGD(learning_rate=lr_schedule, momentum=0.9)
loss_object = tf.losses.SparseCategoricalCrossentropy(from_logits=True)
model.run_eagerly = True

# compile model and save initial weights
```

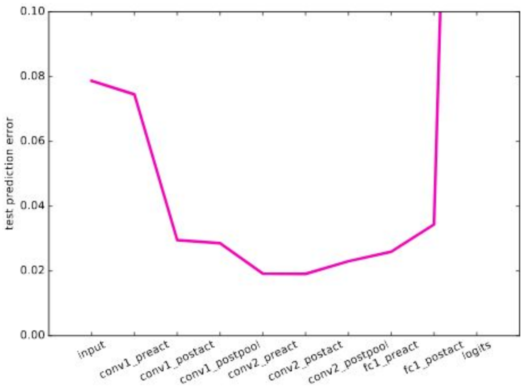
```
model.compile(optimizer=optimizer, loss=loss_object, metrics=['accuracy'])
for probe_layer_num in model.probes.keys():
        model.probe_layer_num = probe_layer_num
        model(x_train[0:BATCH_SIZE])
weights = model.get_weights()

# train the probes with pre-trained weights
probe_errors = train_probes(weights)
plt.figure(figsize=(20,10))
index = range(len(probe_for_graph))
plt.plot(index, probe_errors)
plt.xticks(index, probe_for_graph, rotation=20)
axes = plt.gca()
axes.set(ylim=(0,0.1))
plt.ylabel("test prediction error")
plt.title("Figure 5a")
plt.show()

#train model and save weights
model.probe_layer_num = -1
model.compile(optimizer=optimizer, loss=loss_object, metrics=['accuracy'])
model.fit(x_train, y_train, batch_size=BATCH_SIZE, epochs=NUM_EPOCHS, verbose=2)
weights = model.get_weights()

# train the probes with post-trained weights
probe_errors_trained = train_probes(weights)
plt.figure(figsize=(20,10))
index = range(len(probe_for_graph))
plt.plot(index, probe_errors_trained)
plt.xticks(index, probe_for_graph, rotation=20)
axes = plt.gca()
axes.set(ylim=(0,0.1))
plt.ylabel("test prediction error")
plt.title("Figure 5b")
plt.show()

# fig1.savefig('midterm_figure1.png')
# fig2.savefig('midterm_figure2.png')
```

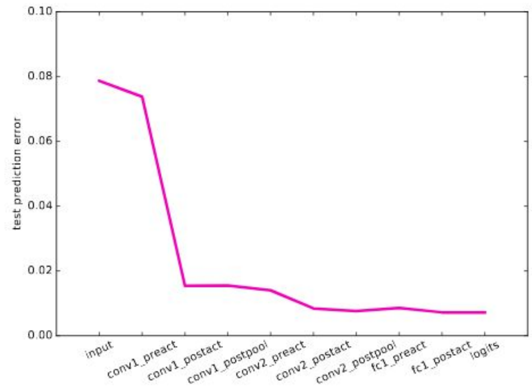
Obtained Figures for Figure 5:





Note that the two figures above are from different runs of the program - the best figures were chosen. With different optimization and more time for us to train, we believe the figures will appear more like that in the research paper, as shown below:





<u>Figure 8:</u>

Information for Figure 8:

For this part of the assignment, we had to replicate figure 8. In this figure is a model with 128 layers with a skip connection from layer 0 to layer 64. The figure visualizes a probe at every layer to see how well each layer would perform if its values were used as a linear classifier. The probes allow us to observe how the first 64 layers are ignored even after copious training.

Code for Figure 8:

```
import tensorflow as tf
from tensorflow.keras import Model, layers
from tensorflow.keras.layers import Flatten, Dense, Concatenate
from tensorflow.keras.datasets import mnist
import sys
import numpy as np
import matplotlib.pyplot as plt
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
import logging
tf.get_logger().setLevel(logging.ERROR)

NUM_LABELS = 10
SKIP_LAYER = 64
LAYER_COUNT = 128
BATCH_SIZE = 512
(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_test = x_test.astype('float32')/255.0
x_train = x_train.astype('float32')/255.0
XTRAIN_LENGTH = len(x_train)

class MyModel(Model):
        def __init__(self):
        super(MyModel, self).__init__()

        # applied at the first layer
        self.f1 = Flatten()
```

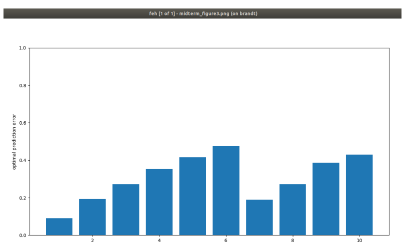
```
        # list of layers
        self.my_layers = []

        # applied before 64th layer
        self.c1 = Concatenate()

        # the i-th entry represents a probe inserted before the i-th layer
        # Each probe is basically a dense layer
        self.probes = []

        # index of probe being trained
        self.probe_num = -1

        # Pathologically deep model with
        # 128 fully-connected layers & 128 hidden units
        # activation function is leaky ReLU
        lrelu = lambda x: tf.keras.activations.relu(x, alpha=0.01)
        for _ in range(LAYER_COUNT):
        self.my_layers.append(Dense(128, activation=lrelu))
        self.probes.append(Dense(NUM_LABELS))

        self.my_layers.append(Dense(NUM_LABELS))

        def call(self, x):
        x = self.f1(x)
        r = x # residual / short-cut / skip connection
        if self.probe_num == -1: # for network training
        for (i, layer) in enumerate(self.my_layers):
        if i == SKIP_LAYER:
        x = self.c1([x, r])
        x = layer(x)
        return x
        else: # for probe training
        for (i, layer) in enumerate(self.my_layers[0:self.probe_num]):
        if i == SKIP_LAYER:
        x = self.c1([x, r])
        x = layer(x)
        x = tf.stop_gradient(x)
        probe = self.probes[self.probe_num]
        return probe(x)

model = MyModel()

def train_probes(weights):
        probe_errors = []
        callback = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=2,verbose=1)

        for probe_num in range(len(model.probes)):

        optim_probe = tf.keras.optimizers.Adam()
        model.reset_metrics()
        model.probe_num = probe_num
        model.compile(optimizer=optim_probe, loss=loss_object, metrics=['accuracy'])
        model.set_weights(weights)

        print("Starting Training and Evaluation of probe number: ", probe_num, flush=True)
```

```
        model.fit(x_train, y_train, batch_size=BATCH_SIZE, epochs=200, \
        verbose=0, validation_split=1/6, callbacks=[callback])
        _, test_accuracy = model.evaluate(x_test, y_test, verbose=0)

        probe_errors.append(1 - test_accuracy)
        print("Probe Number:", probe_num, " Probe Error: ", 1 - test_accuracy, flush=True)

        return probe_errors

lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(0.01,XTRAIN_LENGTH, 0.95, staircase=True)
optimizer = tf.optimizers.SGD(learning_rate=lr_schedule, momentum=0.9)
loss_object = tf.losses.SparseCategoricalCrossentropy(from_logits=True)

# compile model and save initial weights
model.compile(optimizer=optimizer, loss=loss_object, metrics=['accuracy'])
model(x_train[0:BATCH_SIZE])
for probe_num in range(len(model.probes)):
        model.probe_num = probe_num
        model(x_train[0:BATCH_SIZE])
weights = model.get_weights()

probe_errors = train_probes(weights)

print("Training Model for 500 minibatches", flush=True)
model.probe_num = -1
model.compile(optimizer=optimizer, loss=loss_object, metrics=['accuracy'])
model.fit(x_train, y_train, batch_size=BATCH_SIZE, epochs=int(500*BATCH_SIZE/XTRAIN_LENGTH), verbose=0)
weights = model.get_weights()

probe_errors_trained = train_probes(weights)

print("Training Model for 1500 more minibatches", flush=True)
model.probe_num = -1
model.compile(optimizer=optimizer, loss=loss_object, metrics=['accuracy'])
model.set_weights(weights)
model.fit(x_train, y_train, batch_size=BATCH_SIZE, epochs=int(1500*BATCH_SIZE/XTRAIN_LENGTH), verbose=0)
weights = model.get_weights()

probe_errors_trained2 = train_probes(weights)

x = np.arange(1,LAYER_COUNT +1)
fig1 = plt.figure(figsize=(20,10))
plt.bar(x, probe_errors)
plt.xlabel("Probes after 0 minibatches")
plt.ylabel("Optimal Prediction Error")
axes = plt.gca()
axes.set_ylim([0.0,1.0])
# plt.show()

fig2 = plt.figure(figsize=(20,10))
plt.bar(x, probe_errors_trained)
plt.xlabel("Probes after 500 minibatches")
plt.ylabel("Optimal Prediction Error")
axes = plt.gca()
axes.set_ylim([0.0,1.0])
# plt.show()
```

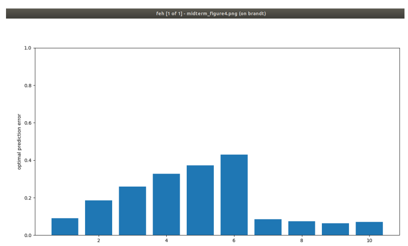
```
fig3 = plt.figure(figsize=(20,10))
plt.bar(x, probe_errors_trained2)
plt.xlabel("Probes after 2000 minibatches")
plt.ylabel("Optimal Prediction Error")
axes = plt.gca()
axes.set_ylim([0.0,1.0])
# plt.show()

fig1.savefig('midterm_figure3_128.png')
fig2.savefig('midterm_figure4_128.png')
fig3.savefig('midterm_figure5_128.png')
```

Obtained Figures for Figure 8:

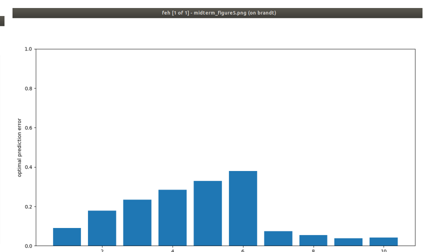
Here are images for 10 layers:



Probes after 0 minibatches

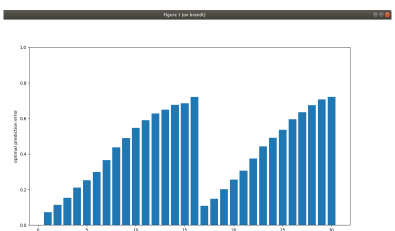


Probes after 500 minibatches

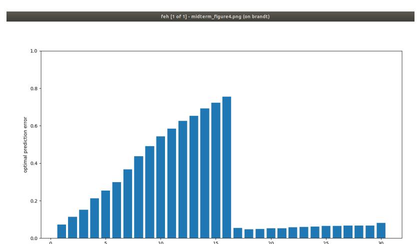


Probes after 2000 minibatches

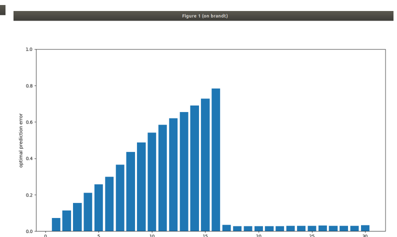
Here are images for 30 layers:



Probes after 0 minibatches

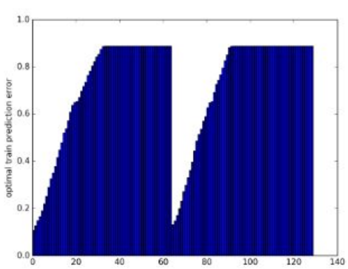


Probes after 500 minibatches

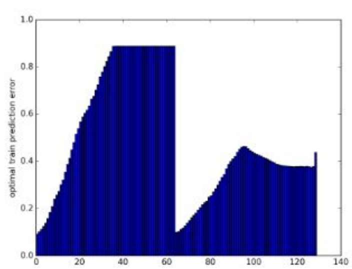


Probes after 2000 minibatches

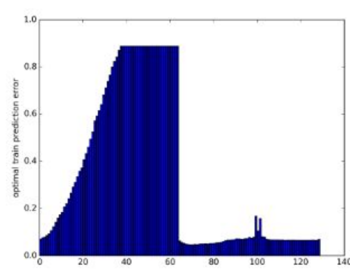
The article's figures shown below:



(a) probes after 0 minibatches



(b) probes after 500 minibatches



(c) probes after 2000 minibatches