```
#CGML Midterm - "Understanding intermediate layers using linear classifier probes"
#https://arxiv.org/pdf/1610.01644.pdf
#Camille Chow and Jacob Maarek
#Fall 2018
#Figure 3

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

NUM_SAMP = 1000
training_size = 900
test_size = 100

#dataset: two guassian distributions
class Data():
  def __init__(self):
    np.random.seed()
    self.mu1 = np.random.randn()
    self.mu2 = np.random.randn()
    self.sig = .5
    self.x = np.atleast_2d(np.linspace(np.min([self.mu1, self.mu2])-2*self.sig, np.max([s
elf.mu1, self.mu2])+2*self.sig,NUM_SAMP)).T
    self.g1 = np.atleast_2d(np.exp(-np.power(self.x - self.mu1, 2.) / (2 * np.power(self.
sig, 2.))))
    self.g2 = np.atleast_2d(np.exp(-np.power(self.x - self.mu2, 2.) / (2 * np.power(self.
sig, 2.))))

    self.coords1 = np.hstack((self.x, self.g1))
    self.coords2 = np.hstack((self.x, self.g2))

    self.data = np.vstack((self.coords1, self.coords2))
    self.labels = np.atleast_2d(np.hstack((np.zeros(NUM_SAMP), np.ones(NUM_SAMP)))).T

  def get_data(self):
    index = np.arange(2 * NUM_SAMP)
    choices1 = np.random.choice(index, size=training_size)
    choices2 = np.random.choice(index, size=test_size)
    return self.data[choices1], self.labels[choices1], self.data[choices2], self.labels[c
hoices2]

num_layers = 33
layers = []
probes = []
losses = []
optims = []
varlist = []
threshold = .35
fail_count = 0
max_fails = 10

tf.reset_default_graph()

points = tf.placeholder(tf.float32, [None, 2])
choice = tf.placeholder(tf.float32, [None, 1])

def my_leaky_relu(x):
  return tf.nn.leaky_relu(x, alpha=.5)

#model: 32 dense layers w/ 128 hidden units, probes at every layer
for i in range (num_layers):
  if i == 0:
    layers.append(tf.layers.dense(points, 128, activation=my_leaky_relu, kernel_initializ
er = tf.glorot_normal_initializer(seed = None, dtype=tf.float32), trainable=False))

  else:
    layers.append(tf.layers.dense(layers[i-1], 128, activation=my_leaky_relu, kernel_init
ializer = tf.glorot_normal_initializer(seed = None, dtype=tf.float32), trainable=False))

  probes.append(tf.layers.dense(layers[i], 1, activation=None, kernel_initializer = tf.gl
```

```
orot_normal_initializer(seed = None, dtype=tf.float32), name=("Probe"+str(i))))

  varlist.append(tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, ("Probe" + str(i))))

  losses.append(tf.losses.sigmoid_cross_entropy(choice, probes[i]))
  optims.append(tf.train.RMSPropOptimizer(learning_rate = 0.1, momentum=0.75).minimize(lo
sses[i], var_list=varlist[i]))

init = tf.global_variables_initializer()

probe_error_experiment = []

experiment_number = 0

#perform 100 experiments
while experiment_number < 100:
  sess = tf.Session()
  sess.run(init)

  print("Experiment", experiment_number)
  data = Data()
  x_train, y_train, x_test, y_test = data.get_data()
  probe_error = []

  layers_number = 0
  while layers_number < num_layers-1:

    for _  in range(0, 100):
      loss_np, _ = sess.run([losses[layers_number], optims[layers_number]], feed_dict={po
ints: x_train, choice: y_train})
    #if probe doesn't train sufficiently, start over with new data
    if layers_number == 0 and loss_np > threshold:
      fail_count += 1
      if fail_count >= max_fails:
          print("Dataset Failed")
          layers_number = 0
          break
    else:
      probe_results, choice_vals = (sess.run([tf.round(tf.nn.sigmoid(probes[layers_number
])), choice], feed_dict={points: x_test, choice: y_test}))
      probe_error.append (np.sum(np.round(np.abs(probe_results-choice_vals)))/(100))
      print("Layer", layers_number, "Loss", loss_np)
      layers_number += 1

  if fail_count >= max_fails:
    fail_count = 0
    sess.close()
    continue
  experiment_number += 1
  print(probe_error)
  probe_error_experiment.append(probe_error)
  sess.close()

#take average of experiments
probe_error_experiment = np.asarray(probe_error_experiment)
average_layer = np.mean(probe_error_experiment, axis=0)
#plot
x = np.arange(1,num_layers)
plt.bar(x, average_layer, tick_label=x)
plt.xlabel("linear probe at layer k")
plt.ylabel("optimal prediction error")
```

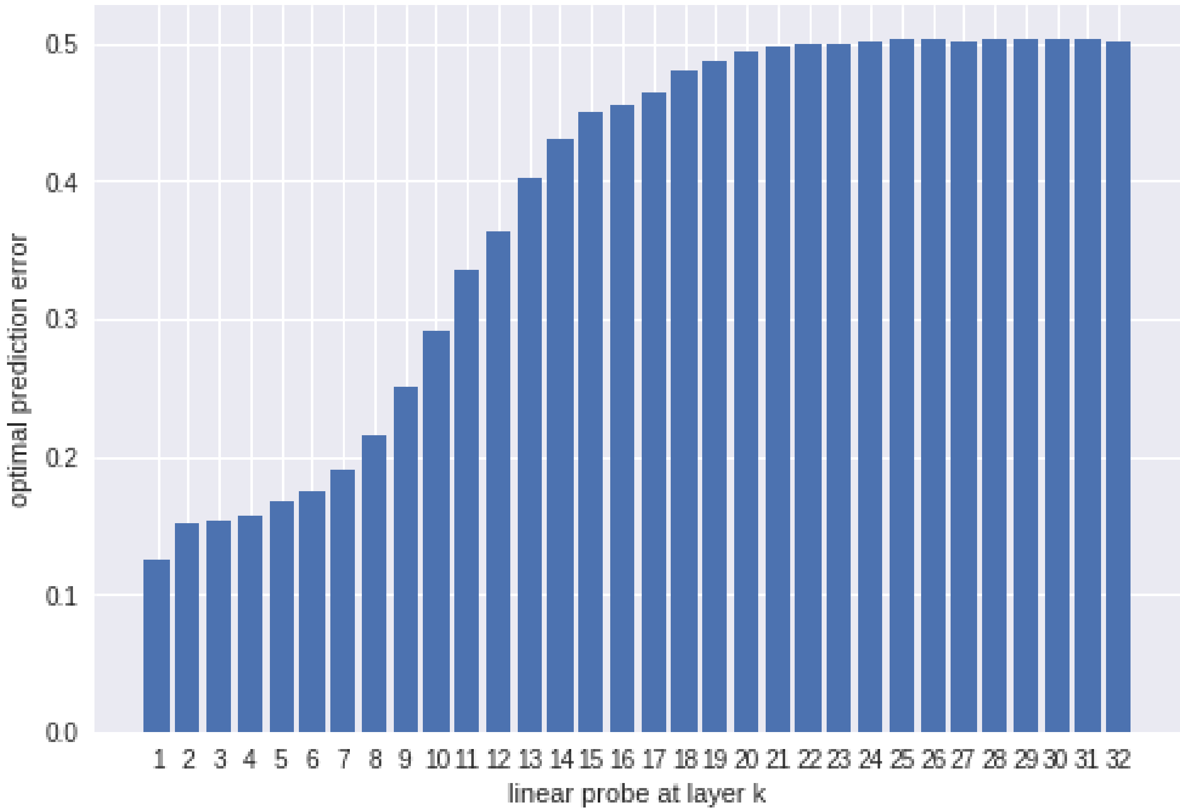
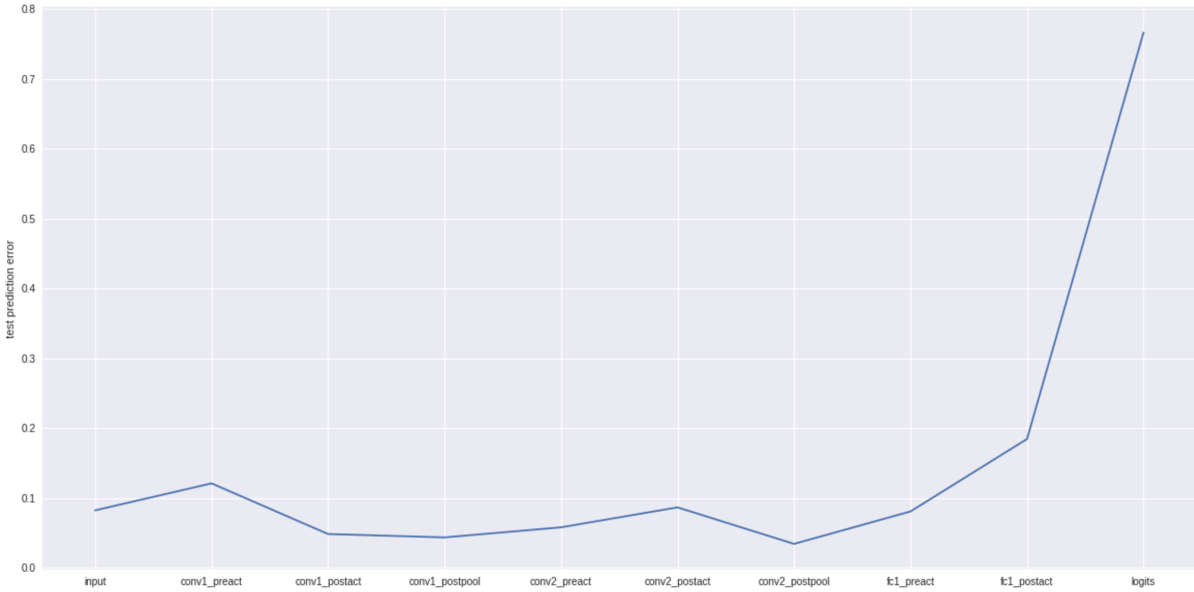


Figure 1: Toy experiment results

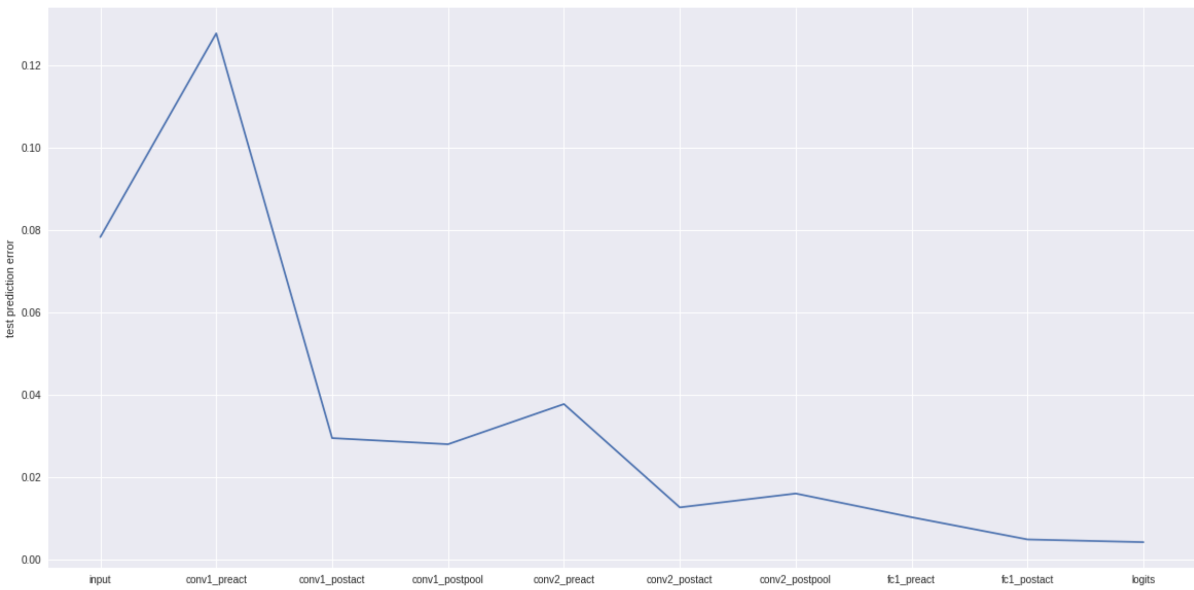
```
#CGML Midterm - Linear Probes
#Figure 5

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

SEED = 66478
IMAGE_SIZE = 28
NUM_CHANNELS = 1
NUM_LABELS = 10

NUM_EPOCHS = 10
BATCH_SIZE = 32

def get_batch(x_train, y_train):
  choices = np.random.choice(np.arange(len(x_train)), size=BATCH_SIZE)
  return x_train[choices], y_train[choices]

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

#shape test data
x_test = x_test.reshape(x_test.shape[0], IMAGE_SIZE, IMAGE_SIZE, NUM_CHANNELS)
x_test = x_test.astype('float32')
x_test /= 255.0
y_test = tf.keras.utils.to_categorical(y_test, NUM_LABELS)

#shape training data
x_train = x_train.reshape(x_train.shape[0], IMAGE_SIZE, IMAGE_SIZE, NUM_CHANNELS)
x_train = x_train.astype('float32')
x_train /= 255.0
y_train = tf.keras.utils.to_categorical(y_train, NUM_LABELS)

DATAPOINTS = len(x_train)
probes = []
probe_names = ["input", "conv1_preact", "conv1_postact", "conv1_postpool", "conv2_preact"
, "conv2_postact", "conv2_postpool", "fc1_preact", "fc1_postact", "logits"]
varlist = []
model_varlist = []
losses = []
optims = []

tf.reset_default_graph()

data = tf.placeholder(tf.float32, [None, IMAGE_SIZE, IMAGE_SIZE, NUM_CHANNELS])
labels = tf.placeholder(tf.int32, [None, NUM_LABELS])
eval_data = tf.placeholder(tf.float32, [None, IMAGE_SIZE, IMAGE_SIZE, NUM_CHANNELS])

def data_type():
  return tf.float32

def add_probe(input_layer, probe_num):
  probes.append(tf.layers.flatten(input_layer))
  probes[probe_num] = tf.layers.dense(probes[probe_num], NUM_LABELS, activation=None, nam
e=probe_names[probe_num], kernel_initializer=tf.glorot_normal_initializer(seed=None, dtyp
e=tf.float32))
  varlist.append(tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, probe_names[probe_nu
m]))
  losses.append(tf.losses.softmax_cross_entropy(labels, probes[probe_num]))
  optims.append(tf.train.RMSPropOptimizer(learning_rate=0.0005, decay=0.9, momentum=0.9,
epsilon=1e-6, centered = True).minimize(losses[probe_num], var_list=varlist[probe_num]))

#model taken from: https://github.com/tensorflow/models/blob/master/tutorials/image/mnist
/convolutional.py

# The variables below hold all the trainable weights. They are passed an
# initial value which will be assigned when we call:
# {tf.global_variables_initializer().run()}
conv1_weights = tf.Variable(tf.truncated_normal([5, 5, NUM_CHANNELS, 32], stddev=0.1, see
```

```
d=SEED, dtype=data_type()))
conv1_biases = tf.Variable(tf.zeros([32], dtype=data_type()))
conv2_weights = tf.Variable(tf.truncated_normal([5, 5, 32, 64], stddev=0.1, seed=SEED, dt
ype=data_type()))
conv2_biases = tf.Variable(tf.constant(0.1, shape=[64], dtype=data_type()))
fc1_weights = tf.Variable(tf.truncated_normal([IMAGE_SIZE // 4 * IMAGE_SIZE // 4 * 64, 51
2], stddev=0.1, seed=SEED, dtype=data_type()))
fc1_biases = tf.Variable(tf.constant(0.1, shape=[512], dtype=data_type()))
fc2_weights = tf.Variable(tf.truncated_normal([512, NUM_LABELS], stddev=0.1, seed=SEED, d
type=data_type()))
fc2_biases = tf.Variable(tf.constant(0.1, shape=[NUM_LABELS], dtype=data_type()))

def model(data, train=False):
  add_probe(data, 0)
  conv1 = tf.nn.conv2d(data, conv1_weights, strides=[1, 1, 1, 1], padding='SAME', name="c
onv1")
  add_probe(conv1, 1)
  relu1 = tf.nn.relu(tf.nn.bias_add(conv1, conv1_biases), name="relu1")
  add_probe(relu1, 2)
  pool1 = tf.nn.max_pool(relu1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME',
 name="pool1")
  add_probe(pool1, 3)
  conv2 = tf.nn.conv2d(pool1, conv2_weights, strides=[1, 1, 1, 1], padding='SAME', name="
conv2")
  add_probe(conv2, 4)
  relu2 = tf.nn.relu(tf.nn.bias_add(conv2, conv2_biases), name="relu2")
  add_probe(relu2, 5)
  pool2 = tf.nn.max_pool(relu2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME',
 name="pool2")
  add_probe(pool2, 6)

  # Reshape the feature map cuboid into a 2D matrix to feed it to the
  # fully connected layers.
  reshape = tf.layers.flatten(pool2)

  # Fully connected layer. Note that the '+' operation automatically
  # broadcasts the biases.
  fc1 = tf.matmul(reshape, fc1_weights, name="fc1") + fc1_biases
  add_probe(fc1, 7)
  hidden = tf.nn.relu(fc1, name="hidden")

  # Add a 50% dropout during training only. Dropout also scales
  # activations such that no rescaling is needed at evaluation time.
  if train:
    hidden = tf.nn.dropout(hidden, 0.5, seed=SEED)

  add_probe(hidden, 8)

  return tf.matmul(hidden, fc2_weights) + fc2_biases

# Training computation: logits + cross-entropy loss.
logits = model(data, True)
add_probe(logits, 9)
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=labels, logits=logit
s))

# L2 regularization for the fully connected parameters.
regularizers = (tf.nn.l2_loss(fc1_weights) + tf.nn.l2_loss(fc1_biases) +
                tf.nn.l2_loss(fc2_weights) + tf.nn.l2_loss(fc2_biases))
# Add the regularization term to the loss.
loss += 5e-4 * regularizers

# Optimizer: set up a variable that's incremented once per batch and
# controls the learning rate decay.
batch = tf.Variable(0, dtype=data_type())
# Decay once per epoch, using an exponential schedule starting at 0.01.
learning_rate = tf.train.exponential_decay(
    0.01,                # Base learning rate.
    batch * BATCH_SIZE,  # Current index into the dataset.
    DATAPOINTS,          # Decay step.
```

```
    0.95,                # Decay rate.
    staircase=True)
# Use simple momentum for the optimization.
optimizer = tf.train.MomentumOptimizer(learning_rate, 0.9).minimize(loss, global_step=bat
ch)

def get_probe_error():
  probe_errors = []

  for a in range(10):
    loss_np = []
    optim_np = []
    correct = []

    print("training probe: ", a)
    for _ in (range(int(DATAPOINTS/BATCH_SIZE))):
      x_batch, y_batch = get_batch(x_train, y_train)
      loss_np, optim_np = sess.run([losses[a], optims[a]], feed_dict={data: x_batch, labe
ls: y_batch})


    SET = int(DATAPOINTS/10)
    print("evaluating probe: ", a)
    for j in range (10):
      correct.append(sess.run([tf.nn.in_top_k(tf.nn.softmax(probes[a]), tf.argmax(labels,
 1), 1)], feed_dict={data: x_train[int(j*SET):int((j+1)*SET-1), :], labels: y_train[int(j
*SET):int((j+1)*SET-1), :]}))

    probe_errors.append((len(y_train)-np.sum(correct))/len(y_train))
    print(probe_names[a], "Probe_error:", (len(y_train)-np.sum(correct))/len(y_train))
  sess.close()
  return probe_errors

probe_errors = get_probe_error()

#plot accuracy at each probe - PRE TRAINING
plt.figure(figsize=(20,10))
index = range(len(probe_names))
plt.plot(index, probe_errors)
plt.xticks(index, probe_names)
plt.ylabel("test prediction error")

#train model
NUM_BATCHES = DATAPOINTS*NUM_EPOCHS//BATCH_SIZE
for i in range(NUM_BATCHES):
  x_batch, y_batch = get_batch(x_train, y_train)

  loss_np_layers, optim_np_layers = sess.run([loss, optimizer], feed_dict={data: x_batch,
 labels: y_batch})

probe_errors_trained = get_probe_error()

# plot accuracy at each probe - POST TRAINING
plt.figure(figsize=(20,10))
index = range(len(probe_names))
plt.plot(index, probe_errors_trained)
plt.xticks(index, probe_names)
plt.ylabel("test prediction error")
```



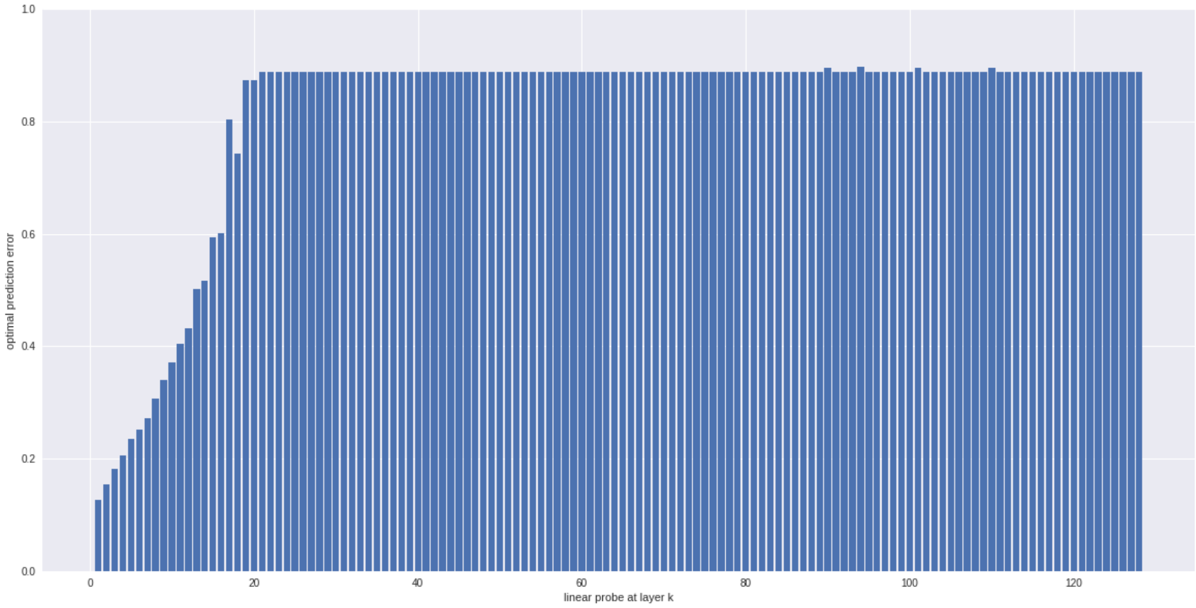
(a) After initialization, no training



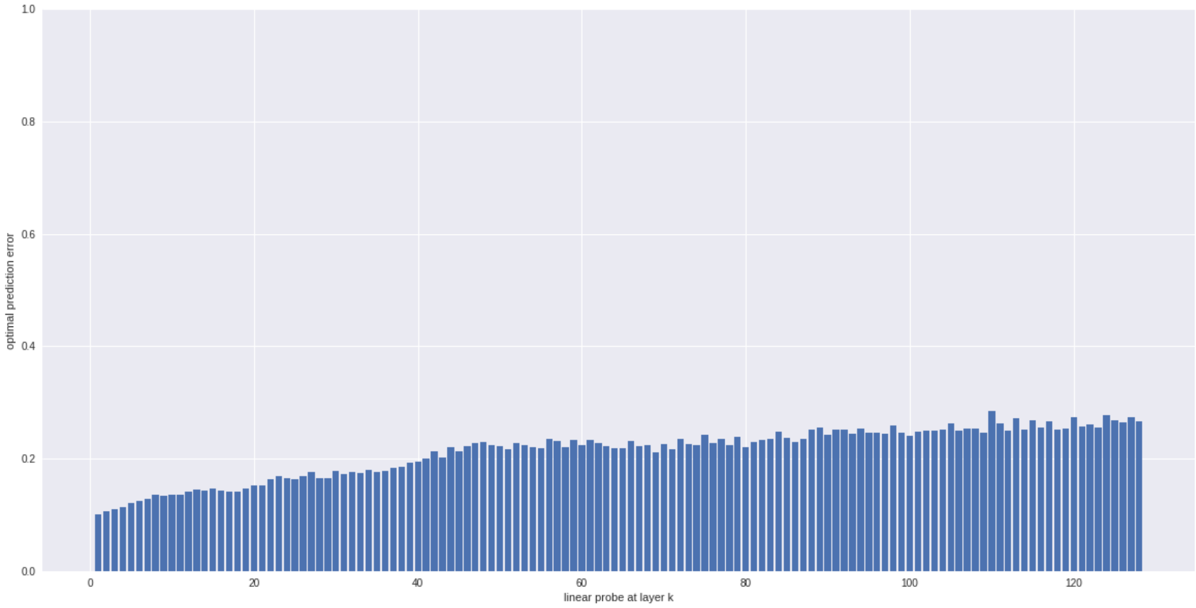
(b) After training for 10 epochs

```
#CGML Midterm - Linear Probes
#Figure 7

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from keras.datasets import mnist
from keras import backend as K
import keras

#Convert train data into val and train
num_classes = 10

(x_train, y_train), (x_test, y_test) = mnist.load_data()

y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

x_train = x_train/255
x_train = np.reshape(x_train, (len(x_train), 28*28))
x_test = x_test/255

DATAPOINTS = len(x_train)
NUM_MINIBATCHES = 5000
NUM_EPOCHS = 10
BATCH_SIZE = int(DATAPOINTS*NUM_EPOCHS/NUM_MINIBATCHES)

def get_batch(x_train, y_train):
  choices = np.random.choice(np.arange(len(x_train)), size=BATCH_SIZE)
  return x_train[choices], y_train[choices]

#model: 128 fully connected layers, auxillary heads at every 16th layer
num_layers = 128
layers = []
probes = []
probe_losses = []
probe_optims = []
varlist = []
layers_losses = []
linear_classifier = []

tf.reset_default_graph()

images = tf.placeholder(tf.float32, [None, 28*28])
image_labels = tf.placeholder(tf.float32, [None, 10])

def my_leaky_relu(x):
  return tf.nn.leaky_relu(x, alpha=.5)

for i in range (num_layers):
  if i == 0:
    layers.append(tf.layers.dense(images, 128, activation=my_leaky_relu, kernel_initializ
er = tf.glorot_normal_initializer(seed = None, dtype=tf.float32), name=("Layer"+str(i))))

  else:
    layers.append(tf.layers.dense(layers[i-1], 128, activation=my_leaky_relu, kernel_init
ializer = tf.glorot_normal_initializer(seed = None, dtype=tf.float32), name=("Layer"+str(
i))))

  if i%16 == 0 or i == num_layers-1 and i!=0:

    linear_classifier.append(tf.layers.dense(layers[i], 10, activation=None))
    layers_losses.append(tf.losses.softmax_cross_entropy(image_labels, linear_classifier[
len(linear_classifier)-1]))

  probes.append(tf.layers.dense(layers[i], 10, activation=None, kernel_initializer = tf.g
lorot_normal_initializer(seed = None, dtype=tf.float32), name=("Probe"+str(i))))

  varlist.append(tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, ("Probe" + str(i))))
```

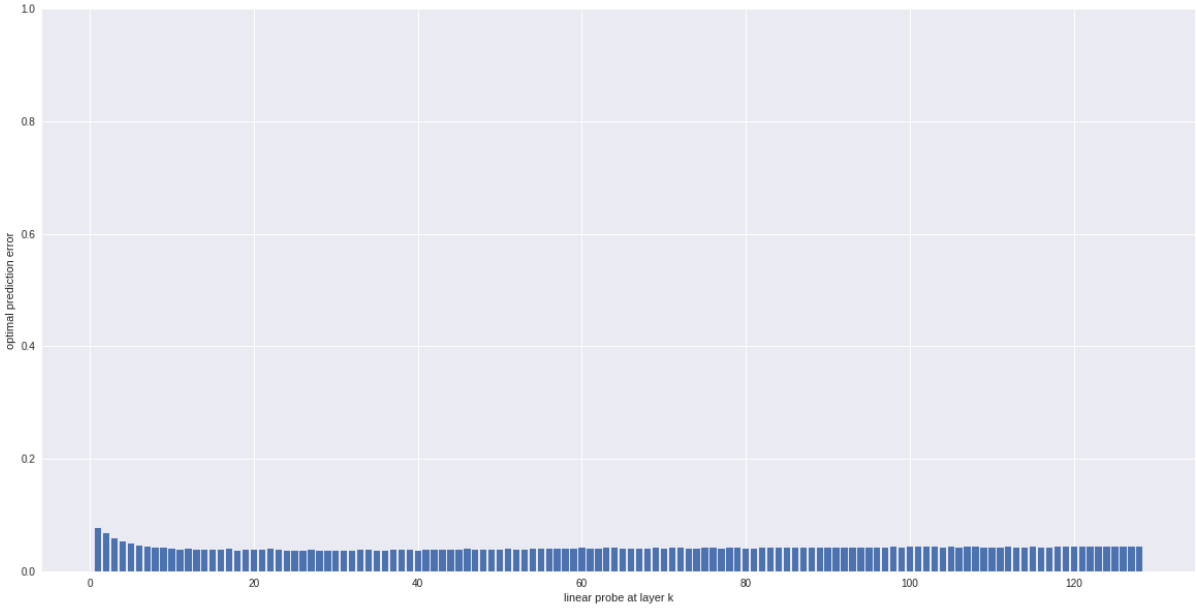
```
  probe_losses.append(tf.losses.softmax_cross_entropy(image_labels, probes[i]))
  probe_optims.append(tf.train.RMSPropOptimizer(learning_rate=0.0005, decay=0.9, momentum
=0.9, epsilon=1e-6, centered = True).minimize(probe_losses[i], var_list=varlist[i]))

layer_loss = tf.constant(0, dtype=tf.float32)
for j in range(len(linear_classifier)):
  layer_loss = layer_loss + layers_losses[j]

layer_varlist = list(filter(lambda a : "Layer" in a.name, [v for v in tf.trainable_variab
les()]))
layer_optim = tf.train.RMSPropOptimizer(learning_rate = 0.00001, momentum=0.9, centered =
 True).minimize(layer_loss, var_list = layer_varlist)

init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

probe_error_batch_num = []

for q in range(NUM_MINIBATCHES):
  x_batch, y_batch = get_batch(x_train, y_train)
  loss_np_layer, optim_np_layer = sess.run([layer_loss, layer_optim], feed_dict={images:
x_batch, image_labels: y_batch})

  if q%100 == 0:
    print("MiniBatch:", q, "MiniBatch Loss: ", loss_np_layer)

  if q == 0 or q == 499 or q == 4999:
    print("calculate probes")
    probe_error = []

    for a in range(num_layers):
      loss_np = []
      optim_np = []

      for t in range (int(DATAPOINTS/BATCH_SIZE)):
        x_batch, y_batch = get_batch(x_train, y_train)
        loss_np, optim_np = sess.run([probe_losses[a], probe_optims[a]], feed_dict={image
s: x_batch, image_labels: y_batch})

      correct = sess.run([tf.nn.in_top_k(tf.nn.softmax(probes[a]), tf.argmax(image_labels
, 1), 1)], feed_dict={images: x_train, image_labels: y_train})

      probe_error.append((len(y_train)-np.sum(correct))/len(y_train))
      print("Probe:", a+1, "Probe_error:", (len(y_train)-np.sum(correct))/len(y_train))

    print(probe_error)
    probe_error_batch_num.append(probe_error)

x = np.arange(1,num_layers+1)
plt.figure(figsize=(20,10))
plt.bar(x, probe_error_batch_num[0])
plt.xlabel("linear probe at layer k")
plt.ylabel("optimal prediction error")
axes = plt.gca()
axes.set_ylim([0.0,1.0])

plt.figure(figsize=(20,10))
plt.bar(x, probe_error_batch_num[1])
plt.xlabel("linear probe at layer k")
plt.ylabel("optimal prediction error")
axes = plt.gca()
axes.set_ylim([0.0,1.0])

plt.figure(figsize=(20,10))
plt.bar(x, probe_error_batch_num[2])
plt.xlabel("linear probe at layer k")
plt.ylabel("optimal prediction error")
axes = plt.gca()
axes.set_ylim([0.0,1.0])
```



(a) Probes after 0 minibatches



(b) After 500 minibatches

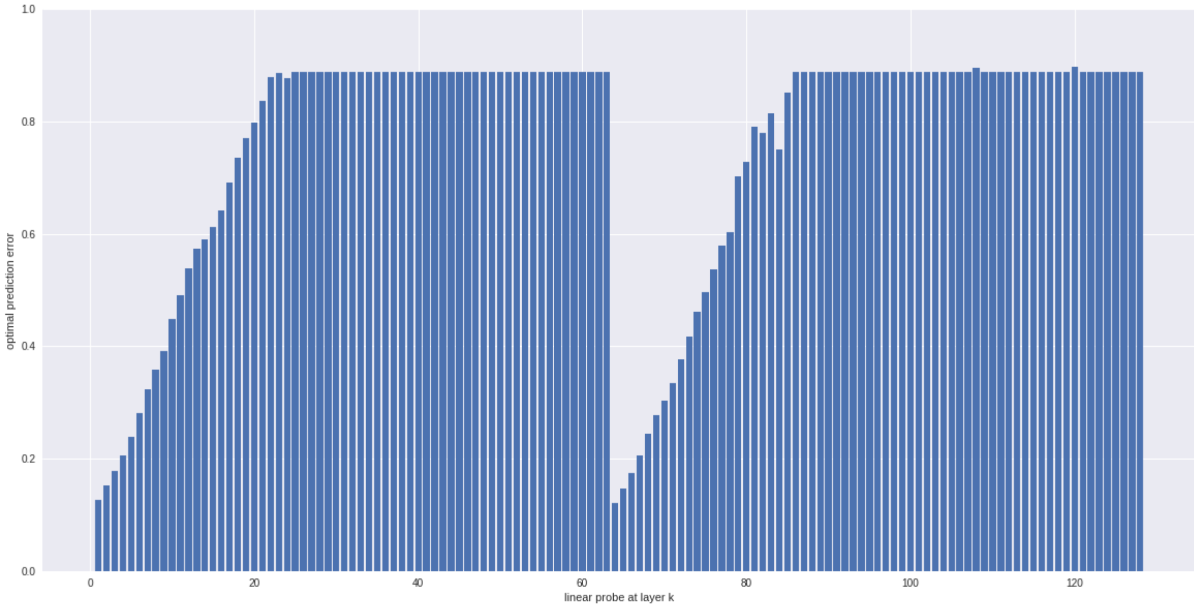


(c) After 5000 minibatches

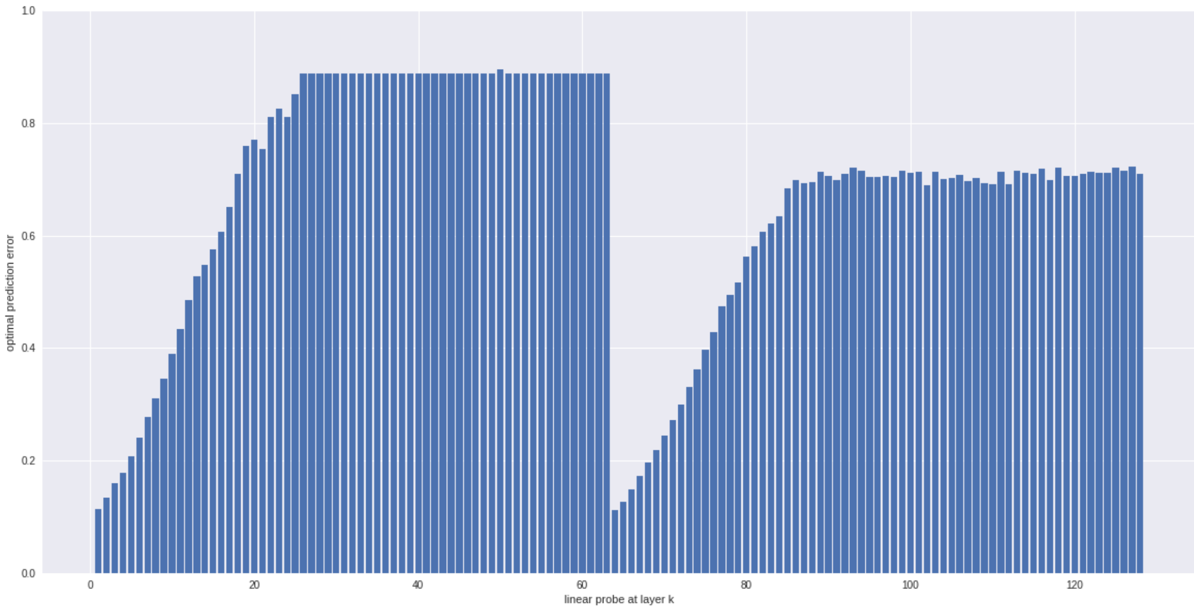
```
#CGML Midterm - Linear Probes
#Figure 8

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from keras.datasets import mnist
from keras import backend as K
import keras

#Convert train data into val and train
num_classes = 10

(x_train, y_train), (x_test, y_test) = mnist.load_data()

y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

x_train = x_train/255
x_train = np.reshape(x_train, (len(x_train), 28*28))
x_test = x_test/255

DATAPOINTS = len(x_train)
NUM_MINIBATCHES = 5000
NUM_EPOCHS = 10
BATCH_SIZE = int(DATAPOINTS*NUM_EPOCHS/NUM_MINIBATCHES)

def get_batch(x_train, y_train):
  choices = np.random.choice(np.arange(len(x_train)), size=BATCH_SIZE)
  return x_train[choices], y_train[choices]

#model: 128 fully connected layers, skip connection from input to layer 64
num_layers = 128
layers = []
probes = []
probe_losses = []
probe_optims = []
varlist = []
layers_losses = []
linear_classifier = []

tf.reset_default_graph()

images = tf.placeholder(tf.float32, [None, 28*28])
image_labels = tf.placeholder(tf.float32, [None, 10])

def my_leaky_relu(x):
  return tf.nn.leaky_relu(x, alpha=.5)

for i in range (num_layers):
  if i == 0:
    layers.append(tf.layers.dense(images, 128, activation=my_leaky_relu, kernel_initializ
er = tf.glorot_normal_initializer(seed = None, dtype=tf.float32), name=("Layer"+str(i))))

  elif i == 63:
    layers.append(tf.layers.dense(layers[i-1], 128, activation=my_leaky_relu, kernel_init
ializer = tf.glorot_normal_initializer(seed = None, dtype=tf.float32), name=("Layer_1_"+s
tr(i))))
    layers[i] = layers[i] + tf.layers.dense(images, 128, activation=None, use_bias = Fals
e, kernel_initializer = tf.glorot_normal_initializer(seed = None, dtype=tf.float32), name
=("Layer_2_"+str(i)))

  else:
    layers.append(tf.layers.dense(layers[i-1], 128, activation=my_leaky_relu, kernel_init
ializer = tf.glorot_normal_initializer(seed = None, dtype=tf.float32), name=("Layer"+str(
i))))

  probes.append(tf.layers.dense(layers[i], 10, activation=None, kernel_initializer = tf.g
lorot_normal_initializer(seed = None, dtype=tf.float32), name=("Probe"+str(i))))
```

```
  varlist.append(tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, ("Probe" + str(i))))

  probe_losses.append(tf.losses.softmax_cross_entropy(image_labels, probes[i]))
  probe_optims.append(tf.train.RMSPropOptimizer(learning_rate=0.0005, decay=0.9, momentum
=0.9, epsilon=1e-6, centered = True).minimize(probe_losses[i], var_list=varlist[i]))

linear_classifier = tf.layers.dense(layers[i], 10, activation=None, kernel_initializer =
tf.glorot_normal_initializer(seed = None, dtype=tf.float32), name=("Layer"+str(i+1)))
layer_loss = tf.losses.softmax_cross_entropy(image_labels, linear_classifier)

layer_varlist = list(filter(lambda a : "Layer" in a.name, [v for v in tf.trainable_variab
les()]))
layer_optim = tf.train.RMSPropOptimizer(learning_rate = 0.00001, momentum=0.9, centered =
 True).minimize(layer_loss, var_list = layer_varlist)

init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

probe_error_batch_num = []

for q in range(NUM_MINIBATCHES):
  x_batch, y_batch = get_batch(x_train, y_train)

  loss_np_layer, optim_np_layer = sess.run([layer_loss, layer_optim], feed_dict={images:
x_batch, image_labels: y_batch})

  if q%100 == 0:
    print("MiniBatch:", q, "MiniBatch Loss: ", loss_np_layer)

  if q == 0 or q == 499 or q == 4999:
    print("calculate probes")
    probe_error = []

    for a in range(num_layers):
      loss_np = []
      optim_np = []

      for t in range (int(DATAPOINTS/BATCH_SIZE)):
        x_batch, y_batch = get_batch(x_train, y_train)
        loss_np, optim_np = sess.run([probe_losses[a], probe_optims[a]], feed_dict={image
s: x_batch, image_labels: y_batch})

      correct = sess.run([tf.nn.in_top_k(tf.nn.softmax(probes[a]), tf.argmax(image_labels
, 1), 1)], feed_dict={images: x_train, image_labels: y_train})

      probe_error.append((len(y_train)-np.sum(correct))/len(y_train))
      print("Probe:", a+1, "Probe_error:", (len(y_train)-np.sum(correct))/len(y_train))

    print(probe_error)
    probe_error_batch_num.append(probe_error)

x = np.arange(1,num_layers+1)
plt.figure(figsize=(20,10))
plt.bar(x, probe_error_batch_num[0])
plt.xlabel("linear probe at layer k")
plt.ylabel("optimal prediction error")
axes = plt.gca()
axes.set_ylim([0.0,1.0])

plt.figure(figsize=(20,10))
plt.bar(x, probe_error_batch_num[1])
plt.xlabel("linear probe at layer k")
plt.ylabel("optimal prediction error")
axes = plt.gca()
axes.set_ylim([0.0,1.0])

plt.figure(figsize=(20,10))
plt.bar(x, probe_error_batch_num[2])
plt.xlabel("linear probe at layer k")
```

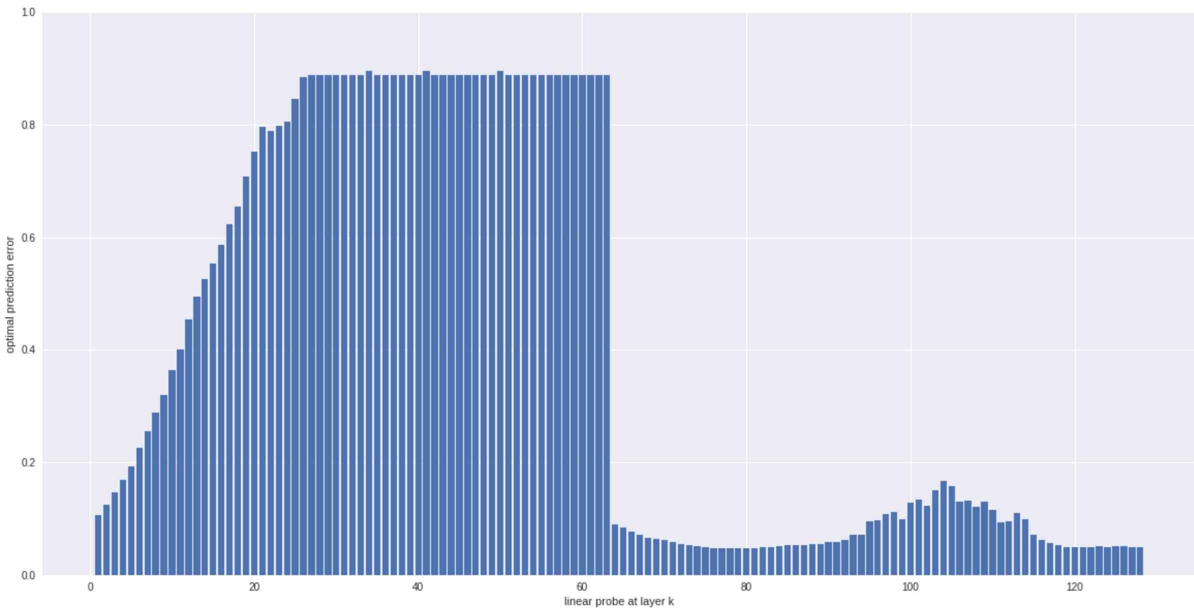
```
plt.ylabel("optimal prediction error")
axes = plt.gca()
axes.set_ylim([0.0,1.0])
```



(a) Probes after 0 minibatches



(b) After 500 minibatches



(c) After 5000 minibatches