HW1

February 6, 2020

0.1 # Assignment1, Deep Learning

- Seyedeh Mina Mousavifar
- 11279515
- sem311

0.2 1. TensorFlow

[1]:
```
try:
  # %tensorflow_version only exists in Colab.
  %tensorflow_version 2.x
except Exception:
  pass
```

```
TensorFlow 2.x selected.
```

0.2.1 Loading Data

[0]:
```
import tensorflow as tf

fashion_mnist = tf.keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.
 ↪load_data()

# convert 0-255 pixels to 0-1 pixels for nn input
train_images, test_images = train_images / 255.0, test_images / 255.0
```

0.2.2 Image Classifier with Keras

[3]:
```
###############################
# Image Classifier with Keras - given

import os, datetime

def create_model_keras():
  return tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10, activation='softmax')
  ])


def train_model_keras():
  model = create_model_keras()
  model.compile(optimizer='adam',
                loss='sparse_categorical_crossentropy',
                metrics=['accuracy'])

  logdir = os.path.join("logs/fit/", datetime.datetime.now().
 ↪strftime("%Y%m%d-%H%M%S"))
  tensorboard_callback = tf.keras.callbacks.TensorBoard(logdir,␣
 ↪histogram_freq=1)

  model.fit(x=train_images,
            y=train_labels,
            epochs=70,
            validation_data=(test_images, test_labels),
            callbacks=[tensorboard_callback])

train_model_keras()
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/70
60000/60000 [==============================] - 8s 125us/sample - loss: 0.4980 -
accuracy: 0.8213 - val_loss: 0.4093 - val_accuracy: 0.8514
Epoch 2/70
60000/60000 [==============================] - 7s 116us/sample - loss: 0.3817 -
accuracy: 0.8613 - val_loss: 0.3799 - val_accuracy: 0.8656
Epoch 3/70
60000/60000 [==============================] - 7s 112us/sample - loss: 0.3513 -
accuracy: 0.8711 - val_loss: 0.3726 - val_accuracy: 0.8629
Epoch 4/70
60000/60000 [==============================] - 7s 111us/sample - loss: 0.3292 -
accuracy: 0.8787 - val_loss: 0.3640 - val_accuracy: 0.8657
Epoch 5/70
60000/60000 [==============================] - 7s 112us/sample - loss: 0.3138 -
accuracy: 0.8853 - val_loss: 0.3387 - val_accuracy: 0.8768
Epoch 6/70
60000/60000 [==============================] - 7s 119us/sample - loss: 0.3016 -
accuracy: 0.8888 - val_loss: 0.3466 - val_accuracy: 0.8749
Epoch 7/70
60000/60000 [==============================] - 7s 114us/sample - loss: 0.2893 -
accuracy: 0.8912 - val_loss: 0.3437 - val_accuracy: 0.8770
```

```
Epoch 8/70
60000/60000 [==============================] - 7s 112us/sample - loss: 0.2779 -
accuracy: 0.8961 - val_loss: 0.3483 - val_accuracy: 0.8761
Epoch 9/70
60000/60000 [==============================] - 7s 113us/sample - loss: 0.2716 -
accuracy: 0.8979 - val_loss: 0.3335 - val_accuracy: 0.8817
Epoch 10/70
60000/60000 [==============================] - 7s 116us/sample - loss: 0.2633 -
accuracy: 0.9005 - val_loss: 0.3242 - val_accuracy: 0.8867
Epoch 11/70
60000/60000 [==============================] - 7s 116us/sample - loss: 0.2559 -
accuracy: 0.9022 - val_loss: 0.3326 - val_accuracy: 0.8838
Epoch 12/70
60000/60000 [==============================] - 7s 114us/sample - loss: 0.2495 -
accuracy: 0.9054 - val_loss: 0.3213 - val_accuracy: 0.8875
Epoch 13/70
60000/60000 [==============================] - 7s 114us/sample - loss: 0.2411 -
accuracy: 0.9089 - val_loss: 0.3387 - val_accuracy: 0.8900
Epoch 14/70
60000/60000 [==============================] - 7s 115us/sample - loss: 0.2374 -
accuracy: 0.9116 - val_loss: 0.3415 - val_accuracy: 0.8856
Epoch 15/70
60000/60000 [==============================] - 7s 118us/sample - loss: 0.2325 -
accuracy: 0.9123 - val_loss: 0.3311 - val_accuracy: 0.8873
Epoch 16/70
60000/60000 [==============================] - 7s 114us/sample - loss: 0.2247 -
accuracy: 0.9141 - val_loss: 0.3416 - val_accuracy: 0.8894
Epoch 17/70
60000/60000 [==============================] - 7s 115us/sample - loss: 0.2216 -
accuracy: 0.9154 - val_loss: 0.3491 - val_accuracy: 0.8824
Epoch 18/70
60000/60000 [==============================] - 7s 113us/sample - loss: 0.2169 -
accuracy: 0.9173 - val_loss: 0.3356 - val_accuracy: 0.8914
Epoch 19/70
60000/60000 [==============================] - 7s 117us/sample - loss: 0.2124 -
accuracy: 0.9197 - val_loss: 0.3287 - val_accuracy: 0.8882
Epoch 20/70
60000/60000 [==============================] - 7s 121us/sample - loss: 0.2090 -
accuracy: 0.9208 - val_loss: 0.3472 - val_accuracy: 0.8906
Epoch 21/70
60000/60000 [==============================] - 7s 113us/sample - loss: 0.2067 -
accuracy: 0.9220 - val_loss: 0.3476 - val_accuracy: 0.8883
Epoch 22/70
60000/60000 [==============================] - 7s 113us/sample - loss: 0.2000 -
accuracy: 0.9247 - val_loss: 0.3386 - val_accuracy: 0.8943
Epoch 23/70
60000/60000 [==============================] - 7s 116us/sample - loss: 0.1979 -
accuracy: 0.9252 - val_loss: 0.3495 - val_accuracy: 0.8937
```

```
Epoch 24/70
60000/60000 [==============================] - 7s 117us/sample - loss: 0.1943 -
accuracy: 0.9261 - val_loss: 0.3432 - val_accuracy: 0.8912
Epoch 25/70
60000/60000 [==============================] - 7s 123us/sample - loss: 0.1908 -
accuracy: 0.9275 - val_loss: 0.3426 - val_accuracy: 0.8929
Epoch 26/70
60000/60000 [==============================] - 7s 112us/sample - loss: 0.1909 -
accuracy: 0.9265 - val_loss: 0.3584 - val_accuracy: 0.8882
Epoch 27/70
60000/60000 [==============================] - 7s 112us/sample - loss: 0.1842 -
accuracy: 0.9300 - val_loss: 0.3613 - val_accuracy: 0.8910
Epoch 28/70
60000/60000 [==============================] - 7s 117us/sample - loss: 0.1824 -
accuracy: 0.9302 - val_loss: 0.3597 - val_accuracy: 0.8962
Epoch 29/70
60000/60000 [==============================] - 7s 115us/sample - loss: 0.1770 -
accuracy: 0.9319 - val_loss: 0.3638 - val_accuracy: 0.8915
Epoch 30/70
60000/60000 [==============================] - 7s 111us/sample - loss: 0.1779 -
accuracy: 0.9322 - val_loss: 0.3814 - val_accuracy: 0.8841
Epoch 31/70
60000/60000 [==============================] - 7s 112us/sample - loss: 0.1732 -
accuracy: 0.9341 - val_loss: 0.3822 - val_accuracy: 0.8901
Epoch 32/70
60000/60000 [==============================] - 7s 116us/sample - loss: 0.1708 -
accuracy: 0.9352 - val_loss: 0.3831 - val_accuracy: 0.8918
Epoch 33/70
60000/60000 [==============================] - 7s 112us/sample - loss: 0.1661 -
accuracy: 0.9370 - val_loss: 0.4026 - val_accuracy: 0.8897
Epoch 34/70
60000/60000 [==============================] - 7s 112us/sample - loss: 0.1674 -
accuracy: 0.9361 - val_loss: 0.3823 - val_accuracy: 0.8911
Epoch 35/70
60000/60000 [==============================] - 7s 112us/sample - loss: 0.1641 -
accuracy: 0.9377 - val_loss: 0.3789 - val_accuracy: 0.8929
Epoch 36/70
60000/60000 [==============================] - 7s 111us/sample - loss: 0.1632 -
accuracy: 0.9387 - val_loss: 0.3948 - val_accuracy: 0.8927
Epoch 37/70
60000/60000 [==============================] - 7s 117us/sample - loss: 0.1595 -
accuracy: 0.9393 - val_loss: 0.3697 - val_accuracy: 0.8971
Epoch 38/70
60000/60000 [==============================] - 7s 114us/sample - loss: 0.1590 -
accuracy: 0.9386 - val_loss: 0.3935 - val_accuracy: 0.8943
Epoch 39/70
60000/60000 [==============================] - 7s 112us/sample - loss: 0.1546 -
accuracy: 0.9403 - val_loss: 0.4038 - val_accuracy: 0.8946
```

```
Epoch 40/70
60000/60000 [==============================] - 7s 114us/sample - loss: 0.1570 -
accuracy: 0.9406 - val_loss: 0.4234 - val_accuracy: 0.8900
Epoch 41/70
60000/60000 [==============================] - 7s 117us/sample - loss: 0.1564 -
accuracy: 0.9408 - val_loss: 0.3975 - val_accuracy: 0.8958
Epoch 42/70
60000/60000 [==============================] - 7s 115us/sample - loss: 0.1486 -
accuracy: 0.9434 - val_loss: 0.4113 - val_accuracy: 0.8977
Epoch 43/70
60000/60000 [==============================] - 7s 113us/sample - loss: 0.1500 -
accuracy: 0.9428 - val_loss: 0.4046 - val_accuracy: 0.8913
Epoch 44/70
60000/60000 [==============================] - 7s 113us/sample - loss: 0.1483 -
accuracy: 0.9439 - val_loss: 0.4044 - val_accuracy: 0.8979
Epoch 45/70
60000/60000 [==============================] - 7s 116us/sample - loss: 0.1474 -
accuracy: 0.9438 - val_loss: 0.4088 - val_accuracy: 0.8976
Epoch 46/70
60000/60000 [==============================] - 7s 115us/sample - loss: 0.1468 -
accuracy: 0.9444 - val_loss: 0.4370 - val_accuracy: 0.8893
Epoch 47/70
60000/60000 [==============================] - 7s 113us/sample - loss: 0.1419 -
accuracy: 0.9469 - val_loss: 0.4194 - val_accuracy: 0.8957
Epoch 48/70
60000/60000 [==============================] - 7s 112us/sample - loss: 0.1417 -
accuracy: 0.9449 - val_loss: 0.4407 - val_accuracy: 0.8922
Epoch 49/70
60000/60000 [==============================] - 7s 116us/sample - loss: 0.1409 -
accuracy: 0.9452 - val_loss: 0.4372 - val_accuracy: 0.8942
Epoch 50/70
60000/60000 [==============================] - 7s 114us/sample - loss: 0.1386 -
accuracy: 0.9474 - val_loss: 0.4372 - val_accuracy: 0.8984
Epoch 51/70
60000/60000 [==============================] - 7s 113us/sample - loss: 0.1367 -
accuracy: 0.9489 - val_loss: 0.4209 - val_accuracy: 0.8964
Epoch 52/70
60000/60000 [==============================] - 7s 114us/sample - loss: 0.1373 -
accuracy: 0.9489 - val_loss: 0.4300 - val_accuracy: 0.8969
Epoch 53/70
60000/60000 [==============================] - 7s 114us/sample - loss: 0.1324 -
accuracy: 0.9496 - val_loss: 0.4300 - val_accuracy: 0.8965
Epoch 54/70
60000/60000 [==============================] - 7s 115us/sample - loss: 0.1321 -
accuracy: 0.9484 - val_loss: 0.4390 - val_accuracy: 0.8973
Epoch 55/70
60000/60000 [==============================] - 7s 117us/sample - loss: 0.1280 -
accuracy: 0.9503 - val_loss: 0.4565 - val_accuracy: 0.8915
```

```
Epoch 56/70
60000/60000 [==============================] - 7s 114us/sample - loss: 0.1332 -
accuracy: 0.9488 - val_loss: 0.4557 - val_accuracy: 0.8987
Epoch 57/70
60000/60000 [==============================] - 7s 114us/sample - loss: 0.1281 -
accuracy: 0.9516 - val_loss: 0.5052 - val_accuracy: 0.8918
Epoch 58/70
60000/60000 [==============================] - 7s 118us/sample - loss: 0.1250 -
accuracy: 0.9521 - val_loss: 0.4844 - val_accuracy: 0.8923
Epoch 59/70
60000/60000 [==============================] - 7s 113us/sample - loss: 0.1248 -
accuracy: 0.9525 - val_loss: 0.5148 - val_accuracy: 0.8951
Epoch 60/70
60000/60000 [==============================] - 7s 113us/sample - loss: 0.1289 -
accuracy: 0.9510 - val_loss: 0.4990 - val_accuracy: 0.8956
Epoch 61/70
60000/60000 [==============================] - 7s 114us/sample - loss: 0.1230 -
accuracy: 0.9521 - val_loss: 0.4800 - val_accuracy: 0.8960
Epoch 62/70
60000/60000 [==============================] - 7s 112us/sample - loss: 0.1247 -
accuracy: 0.9526 - val_loss: 0.4899 - val_accuracy: 0.8956
Epoch 63/70
60000/60000 [==============================] - 7s 115us/sample - loss: 0.1241 -
accuracy: 0.9529 - val_loss: 0.4688 - val_accuracy: 0.8927
Epoch 64/70
60000/60000 [==============================] - 7s 120us/sample - loss: 0.1223 -
accuracy: 0.9538 - val_loss: 0.4688 - val_accuracy: 0.8972
Epoch 65/70
60000/60000 [==============================] - 7s 112us/sample - loss: 0.1203 -
accuracy: 0.9535 - val_loss: 0.4978 - val_accuracy: 0.8943
Epoch 66/70
60000/60000 [==============================] - 7s 112us/sample - loss: 0.1174 -
accuracy: 0.9552 - val_loss: 0.4965 - val_accuracy: 0.8959
Epoch 67/70
60000/60000 [==============================] - 7s 116us/sample - loss: 0.1209 -
accuracy: 0.9536 - val_loss: 0.4832 - val_accuracy: 0.8965
Epoch 68/70
60000/60000 [==============================] - 7s 111us/sample - loss: 0.1202 -
accuracy: 0.9545 - val_loss: 0.5412 - val_accuracy: 0.8936
Epoch 69/70
60000/60000 [==============================] - 7s 113us/sample - loss: 0.1155 -
accuracy: 0.9557 - val_loss: 0.5332 - val_accuracy: 0.8933
Epoch 70/70
60000/60000 [==============================] - 7s 122us/sample - loss: 0.1144 -
accuracy: 0.9559 - val_loss: 0.4992 - val_accuracy: 0.8954
```

0.2.3 Image Classifier without keras

Defining Neural Network Operations

[0]:
```
import tensorflow as tf

DROPOUT_RATE = 0.2
OUTPUT_CLASSES = 10

def dense_relu(inputs , weights):
    """
    This function fully connect inputs to weights with relu activation function
    :param inputs: inputs of neural network
    :param weights: weights of neural network
    :return tensor: tensor network
    """
    return tf.nn.relu(tf.matmul(inputs , weights))


def dense_softmax(inputs , weights):
    """
    This function fully connect inputs to weights with softmax activation␣
 ↪function
    :param inputs: inputs of neural network
    :param weights: weights of neural network
    :return tensor: tensor network
    """
    return tf.nn.softmax(tf.matmul(inputs , weights))


def drop_out(inputs, dropout_rate):
    """
    This function fully connect inputs to weights with softmax activation␣
 ↪function
    :param inputs: inputs of neural network
    :param dropout_rate: dropout rate of neural network
    :return tensor: tensor network
    """
    return tf.nn.dropout(inputs, rate=dropout_rate)


def flatten(inputs):
    """
    This function transforms the format of the images from
    a two-dimensional array (of 28 by 28 pixels) to
    a one-dimensional array (of 28 * 28 = 784 pixels)
    :param inputs: two-dimensional array of 28*28 pixels
    :return tensor: reformatted tensor
    """
    data_batch_size = tf.shape(inputs)[0]
```

```
  return tf.reshape(inputs, shape=[data_batch_size,784])
```

Initializing weights

[0]:
```
# initializing weights
initializer = tf.initializers.glorot_uniform()

def get_weight(shape, name):
  """
  This function initializes weights for neural network
  :param shape: shape of weights to initialize
  :param name: name of layer
  :return weights_out: list of weight tensors
  """
  return tf.Variable(initializer(shape), name=name, trainable=True, dtype=tf.
 ↪float32)


# first layer with 512 nodes
# second layer with 10 nodes
shapes = [[784, 512],
          [512, 10]]

weights = []
# obtatining weights
for i in range(len(shapes)):
  weights.append(get_weight(shapes[i], 'weight{}'.format(i)))
```

Creating Model

[0]:
```
def model(x):
  """
  This function creates three layer neural network
  :param x: image data
  :return x: image classifier neural network
  """
  x = tf.cast(x, dtype=tf.float32)

  # transformation layer, flatting 28*28 matrix to 1*784
  x = flatten(x)

  # first layer, dense layer with relu activation
  x = dense_relu(inputs=x, weights=weights[0])
  # second layer, prunning data
  x = drop_out(inputs=x, dropout_rate=DROPOUT_RATE)
  # third layer, dense layer with softmax activation
  x = dense_softmax(inputs=x, weights=weights[1])
```

```python
    return x
```

Defining the loss function and optimization

[0]:
```python
LEARNING_RATE = 0.001

def loss(pred, target):
  """
  This function calculates loss between predicted value and original value␣
 ↪based on categorical cross entropy method
  :param pred: predicted value
  :param target: original value
  :return tensor: reduced tensor
  """
  return tf.losses.sparse_categorical_crossentropy(target, pred)

# adding optimizer
optimizer = tf.optimizers.Adam(LEARNING_RATE)
```

Creating training and test functions

[0]:
```python
import datetime

# Define our metrics

# training loss, which is mean of train loss tensor
train_loss = tf.metrics.Mean('train_loss', dtype=tf.float32)
# train accuracy, which calculates accuracy on sparse categorical data
train_accuracy = tf.metrics.SparseCategoricalAccuracy('train_accuracy')
# test loss, which is mean of test loss tensor
test_loss = tf.metrics.Mean('test_loss', dtype=tf.float32)
test_accuracy = tf.metrics.SparseCategoricalAccuracy('test_accuracy')

def train_step(model, optimizer, inputs, outputs):
  """
  This function train the model by calculating gradient and optimizing weights
  :param model: image classification model
  :param inputs: train images
  :param outputs: original labels of train set
  """
  # calculating gradient
  with tf.GradientTape() as tape:
    prediction = model(inputs)
    current_loss = loss(prediction, outputs)
  grads = tape.gradient(current_loss, weights)
  optimizer.apply_gradients(zip(grads, weights))
  # saving loss and accuracy
  train_loss(current_loss)
  train_accuracy(outputs, prediction)


def test_step(model, inputs, outputs):
  """
  This function test the model by its predictions
  :param model: image classification model
  :param inputs: test images
  :param outputs: original labels of test set
  """
  # predict output
  prediction = model(inputs)
  # calculate loss
  current_loss = loss(prediction, outputs)

  # saving loss and accuracy
  test_loss(current_loss)
  test_accuracy(outputs, prediction)


# logging info for summarizing data and visualizing on tensorboard
current_time = datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
train_log_dir = 'logs/fit/' + current_time + '/my_train'
test_log_dir = 'logs/fit/' + current_time + '/my_test'
train_summary_writer = tf.summary.create_file_writer(train_log_dir)
test_summary_writer = tf.summary.create_file_writer(test_log_dir)
```

Training

[9]:
```
BATCH_SIZE = 64
NUM_EPOCHS = 70

# Using batching capabilities
train_dataset = tf.data.Dataset.from_tensor_slices((train_images, train_labels))
test_dataset = tf.data.Dataset.from_tensor_slices((test_images, test_labels))

# converting to batches
train_dataset = train_dataset.shuffle(60000).batch(BATCH_SIZE)
test_dataset = test_dataset.batch(BATCH_SIZE)


for epoch in range(NUM_EPOCHS):
  # model computation starting time
  start_timer = datetime.datetime.now()
```

```
# training for each batch
for (image, label) in train_dataset:
  train_step(model, optimizer, image, label)

# logging train loss and accuracy after one epoch for training set
with train_summary_writer.as_default():
  tf.summary.scalar('loss', train_loss.result(), step=epoch)
  tf.summary.scalar('accuracy', train_accuracy.result(), step=epoch)

# predicting for each batch
for (x_test, y_test) in test_dataset:
  test_step(model, x_test, y_test)

# logging test loss and accuracy after one epoch for training set
with test_summary_writer.as_default():
  tf.summary.scalar('loss', test_loss.result(), step=epoch)
  tf.summary.scalar('accuracy', test_accuracy.result(), step=epoch)

# modeling end time after one epoch and logging
taken_time = datetime.datetime.now() - start_timer

# printing summary of data for each epoch
template = 'Epoch {} - time: {}s, {}s per sample - loss: {} - accuracy: {} -␣
↪test loss: {} - test accuracy: {}'
print (template.format(epoch+1,
                       round(taken_time.total_seconds(),2),
                       round(taken_time.total_seconds()*100/6,0),
                       train_loss.result(),
                       train_accuracy.result()*100,
                       test_loss.result(),
                       test_accuracy.result()*100))

# Reset metrics every epoch
train_loss.reset_states()
test_loss.reset_states()
train_accuracy.reset_states()
test_accuracy.reset_states()
```

```
Epoch 1 - time: 8.97s, 150.0s per sample - loss: 0.507331371307373 - accuracy:
82.0 - test loss: 0.43083393573760986 - test accuracy: 84.38999938964844
Epoch 2 - time: 8.94s, 149.0s per sample - loss: 0.3849656581878662 - accuracy:
86.01499938964844 - test loss: 0.4108673334121704 - test accuracy:
85.29999542236328
Epoch 3 - time: 8.66s, 144.0s per sample - loss: 0.34882697463035583 -
accuracy: 87.18167114257812 - test loss: 0.39731067419052124 - test accuracy:
85.86000061035156
```

```
Epoch 4 - time: 8.89s, 148.0s per sample - loss: 0.327258437871933 - accuracy:
88.01333618164062 - test loss: 0.3681788146495819 - test accuracy:
86.69999694824219
Epoch 5 - time: 8.7s, 145.0s per sample - loss: 0.3132924437522888 - accuracy:
88.38999938964844 - test loss: 0.35258716344833374 - test accuracy:
87.30999755859375
Epoch 6 - time: 8.63s, 144.0s per sample - loss: 0.2994932234287262 - accuracy:
88.96333312988281 - test loss: 0.3583742082118988 - test accuracy:
87.11000061035156
Epoch 7 - time: 8.59s, 143.0s per sample - loss: 0.28688740730285645 -
accuracy: 89.44332885742188 - test loss: 0.3442571461200714 - test accuracy:
87.70999908447266
Epoch 8 - time: 8.76s, 146.0s per sample - loss: 0.27845898270606995 -
accuracy: 89.56500244140625 - test loss: 0.3503366708755493 - test accuracy:
87.5
Epoch 9 - time: 8.87s, 148.0s per sample - loss: 0.26812148094177246 -
accuracy: 89.99333190917969 - test loss: 0.3450849652290344 - test accuracy:
88.19000244140625
Epoch 10 - time: 8.53s, 142.0s per sample - loss: 0.2601473033428192 -
accuracy: 90.30000305175781 - test loss: 0.3413485586643219 - test accuracy:
87.81999969482422
Epoch 11 - time: 8.92s, 149.0s per sample - loss: 0.2533170282840729 -
accuracy: 90.43333435058594 - test loss: 0.3369845449924469 - test accuracy:
88.0
Epoch 12 - time: 8.62s, 144.0s per sample - loss: 0.24595242738723755 -
accuracy: 90.79499816894531 - test loss: 0.34140732884407043 - test accuracy:
88.05000305175781
Epoch 13 - time: 8.69s, 145.0s per sample - loss: 0.2395244687795639 -
accuracy: 90.99666595458984 - test loss: 0.3489454388618469 - test accuracy:
87.5999984741211
Epoch 14 - time: 9.09s, 151.0s per sample - loss: 0.23220324516296387 -
accuracy: 91.28999328613281 - test loss: 0.3558362126350403 - test accuracy:
87.55000305175781
Epoch 15 - time: 8.72s, 145.0s per sample - loss: 0.22802360355854034 -
accuracy: 91.35333251953125 - test loss: 0.3636306822299957 - test accuracy:
87.52999877929688
Epoch 16 - time: 8.67s, 144.0s per sample - loss: 0.2234516590833664 -
accuracy: 91.56999969482422 - test loss: 0.3329439163208008 - test accuracy:
89.06999969482422
Epoch 17 - time: 8.44s, 141.0s per sample - loss: 0.21532513201236725 -
accuracy: 91.95999908447266 - test loss: 0.35293760895729065 - test accuracy:
88.34000396728516
Epoch 18 - time: 8.78s, 146.0s per sample - loss: 0.2110263705253601 -
accuracy: 92.11166381835938 - test loss: 0.34767845273017883 - test accuracy:
88.54000091552734
Epoch 19 - time: 8.52s, 142.0s per sample - loss: 0.20905645191669464 -
accuracy: 92.32666778564453 - test loss: 0.34441766142845154 - test accuracy:
88.47000122070312
```

```
Epoch 20 - time: 8.52s, 142.0s per sample - loss: 0.2003278136253357 -
accuracy: 92.39500427246094 - test loss: 0.35860133171081543 - test accuracy:
88.30999755859375
Epoch 21 - time: 8.58s, 143.0s per sample - loss: 0.20083996653556824 -
accuracy: 92.32166290283203 - test loss: 0.34785231947898865 - test accuracy:
88.52999877929688
Epoch 22 - time: 8.74s, 146.0s per sample - loss: 0.19438208639621735 -
accuracy: 92.788330078125 - test loss: 0.35635045170783997 - test accuracy: 88.5
Epoch 23 - time: 8.69s, 145.0s per sample - loss: 0.19295404851436615 -
accuracy: 92.72000122070312 - test loss: 0.3589054346084595 - test accuracy:
88.0999984741211
Epoch 24 - time: 8.48s, 141.0s per sample - loss: 0.18834653496742249 -
accuracy: 92.8566665649414 - test loss: 0.3526705205440521 - test accuracy:
88.70000457763672
Epoch 25 - time: 8.86s, 148.0s per sample - loss: 0.18525581061840057 -
accuracy: 93.12999725341797 - test loss: 0.3491140604019165 - test accuracy:
88.59000396728516
Epoch 26 - time: 8.63s, 144.0s per sample - loss: 0.17962025105953217 -
accuracy: 93.26166534423828 - test loss: 0.3502592444419861 - test accuracy:
89.09000396728516
Epoch 27 - time: 8.66s, 144.0s per sample - loss: 0.17554374039173126 -
accuracy: 93.35333251953125 - test loss: 0.38217344880104065 - test accuracy:
88.5999984741211
Epoch 28 - time: 8.63s, 144.0s per sample - loss: 0.17437992990016937 -
accuracy: 93.41333770751953 - test loss: 0.36356955766677856 - test accuracy:
88.77000427246094
Epoch 29 - time: 8.97s, 150.0s per sample - loss: 0.16983026266098022 -
accuracy: 93.6500015258789 - test loss: 0.3814142942428589 - test accuracy:
88.69000244140625
Epoch 30 - time: 8.76s, 146.0s per sample - loss: 0.16931642591953278 -
accuracy: 93.50167083740234 - test loss: 0.3763173222541809 - test accuracy:
88.63999938964844
Epoch 31 - time: 8.79s, 146.0s per sample - loss: 0.16248761117458344 -
accuracy: 93.88666534423828 - test loss: 0.3783556818962097 - test accuracy:
88.77999877929688
Epoch 32 - time: 9.0s, 150.0s per sample - loss: 0.16294065117835999 -
accuracy: 93.83000183105469 - test loss: 0.3836919069290161 - test accuracy:
88.59000396728516
Epoch 33 - time: 8.66s, 144.0s per sample - loss: 0.16247014701366425 -
accuracy: 93.85333251953125 - test loss: 0.39633017778396606 - test accuracy:
88.70000457763672
Epoch 34 - time: 8.6s, 143.0s per sample - loss: 0.1569889336824417 - accuracy:
94.05999755859375 - test loss: 0.36087939143180847 - test accuracy:
89.16000366210938
Epoch 35 - time: 8.95s, 149.0s per sample - loss: 0.15645885467529297 -
accuracy: 94.13166809082031 - test loss: 0.40210452675819397 - test accuracy:
88.80000305175781
Epoch 36 - time: 9.38s, 156.0s per sample - loss: 0.14876341819763184 -
```

```
accuracy: 94.45499420166016 - test loss: 0.392085999250412 - test accuracy:
89.01000213623047
Epoch 37 - time: 8.49s, 142.0s per sample - loss: 0.15185752511024475 -
accuracy: 94.16666412353516 - test loss: 0.388346403837204 - test accuracy:
88.59000396728516
Epoch 38 - time: 8.47s, 141.0s per sample - loss: 0.14932046830654144 -
accuracy: 94.33000183105469 - test loss: 0.39366477727890015 - test accuracy:
88.77999877929688
Epoch 39 - time: 8.81s, 147.0s per sample - loss: 0.1461104303598404 -
accuracy: 94.47833251953125 - test loss: 0.3987484276294708 - test accuracy:
88.61000061035156
Epoch 40 - time: 8.58s, 143.0s per sample - loss: 0.14490041136741638 -
accuracy: 94.48999786376953 - test loss: 0.4091548025608063 - test accuracy:
89.22000122070312
Epoch 41 - time: 8.59s, 143.0s per sample - loss: 0.13909019529819489 -
accuracy: 94.72167205810547 - test loss: 0.3957258462905884 - test accuracy:
88.9000015258789
Epoch 42 - time: 8.65s, 144.0s per sample - loss: 0.14433595538139343 -
accuracy: 94.52000427246094 - test loss: 0.40927982330322266 - test accuracy:
88.6500015258789
Epoch 43 - time: 8.75s, 146.0s per sample - loss: 0.13712304830551147 -
accuracy: 94.77999877929688 - test loss: 0.4155561625957489 - test accuracy:
88.77000427246094
Epoch 44 - time: 8.47s, 141.0s per sample - loss: 0.1360083669424057 -
accuracy: 94.71833038330078 - test loss: 0.4016106426715851 - test accuracy:
89.12000274658203
Epoch 45 - time: 8.43s, 141.0s per sample - loss: 0.13382649421691895 -
accuracy: 94.93167114257812 - test loss: 0.4212495684623718 - test accuracy:
89.41000366210938
Epoch 46 - time: 8.81s, 147.0s per sample - loss: 0.13187140226364136 -
accuracy: 94.93167114257812 - test loss: 0.44370365142822266 - test accuracy:
88.59000396728516
Epoch 47 - time: 8.55s, 143.0s per sample - loss: 0.1315891444683075 -
accuracy: 95.05166625976562 - test loss: 0.4262983500957489 - test accuracy:
88.79000091552734
Epoch 48 - time: 8.49s, 141.0s per sample - loss: 0.12988314032554626 -
accuracy: 95.125 - test loss: 0.4048975110054016 - test accuracy:
89.13999938964844
Epoch 49 - time: 8.68s, 145.0s per sample - loss: 0.12766620516777039 -
accuracy: 95.11666870117188 - test loss: 0.4261811077594757 - test accuracy:
89.20999908447266
Epoch 50 - time: 8.79s, 146.0s per sample - loss: 0.12800233066082 - accuracy:
95.08167266845703 - test loss: 0.4585956931114197 - test accuracy:
88.63999938964844
Epoch 51 - time: 8.59s, 143.0s per sample - loss: 0.1249021515250206 -
accuracy: 95.24666595458984 - test loss: 0.41700342297554016 - test accuracy:
89.1500015258789
Epoch 52 - time: 8.6s, 143.0s per sample - loss: 0.12376584112644196 -
```

```
accuracy: 95.29833221435547 - test loss: 0.4135459363460541 - test accuracy:
89.33000183105469
Epoch 53 - time: 8.74s, 146.0s per sample - loss: 0.12060042470693588 -
accuracy: 95.39167022705078 - test loss: 0.42278334498405457 - test accuracy:
89.19000244140625
Epoch 54 - time: 8.56s, 143.0s per sample - loss: 0.11942566186189651 -
accuracy: 95.44499969482422 - test loss: 0.45119354128837585 - test accuracy:
88.51000213623047
Epoch 55 - time: 8.52s, 142.0s per sample - loss: 0.11673358082771301 -
accuracy: 95.56999969482422 - test loss: 0.4437153935432434 - test accuracy:
88.81000518798828
Epoch 56 - time: 8.64s, 144.0s per sample - loss: 0.11759279668331146 -
accuracy: 95.50833129882812 - test loss: 0.4502458870410919 - test accuracy:
88.81999969482422
Epoch 57 - time: 8.66s, 144.0s per sample - loss: 0.11492475122213364 -
accuracy: 95.625 - test loss: 0.47613632678985596 - test accuracy:
88.72000122070312
Epoch 58 - time: 8.43s, 140.0s per sample - loss: 0.11553814262151718 -
accuracy: 95.56000518798828 - test loss: 0.446164071559906 - test accuracy:
89.19000244140625
Epoch 59 - time: 8.47s, 141.0s per sample - loss: 0.1138080582022667 -
accuracy: 95.59166717529297 - test loss: 0.44127774238586426 - test accuracy:
89.08000183105469
Epoch 60 - time: 8.91s, 148.0s per sample - loss: 0.11393597722053528 -
accuracy: 95.70833587646484 - test loss: 0.4778379797935486 - test accuracy:
89.11000061035156
Epoch 61 - time: 8.53s, 142.0s per sample - loss: 0.10764867812395096 -
accuracy: 95.78666687011719 - test loss: 0.47073090076446533 - test accuracy:
89.16000366210938
Epoch 62 - time: 8.58s, 143.0s per sample - loss: 0.1096615120768547 -
accuracy: 95.70333099365234 - test loss: 0.4392678737640381 - test accuracy:
89.26000213623047
Epoch 63 - time: 8.64s, 144.0s per sample - loss: 0.10603305697441101 -
accuracy: 95.97000122070312 - test loss: 0.46711090207099915 - test accuracy:
88.76000213623047
Epoch 64 - time: 8.61s, 144.0s per sample - loss: 0.10744874179363251 -
accuracy: 95.87666320800781 - test loss: 0.46512866020202637 - test accuracy:
88.77000427246094
Epoch 65 - time: 8.36s, 139.0s per sample - loss: 0.10453688353300095 -
accuracy: 96.04000091552734 - test loss: 0.48344454169273376 - test accuracy:
88.34000396728516
Epoch 66 - time: 8.71s, 145.0s per sample - loss: 0.10423196107149124 -
accuracy: 96.086669921875 - test loss: 0.48628613352775574 - test accuracy:
89.0999984741211
Epoch 67 - time: 8.7s, 145.0s per sample - loss: 0.10444384068250656 -
accuracy: 96.03833770751953 - test loss: 0.46290209889411926 - test accuracy:
89.1500015258789
Epoch 68 - time: 8.41s, 140.0s per sample - loss: 0.10377102345228195 -
```

```
accuracy: 96.04499816894531 - test loss: 0.4816974103450775 - test accuracy:
89.0
Epoch 69 - time: 8.59s, 143.0s per sample - loss: 0.09978978335857391 -
accuracy: 96.22666931152344 - test loss: 0.49389901757240295 - test accuracy:
88.6300048828125
Epoch 70 - time: 8.53s, 142.0s per sample - loss: 0.10161121934652328 -
accuracy: 96.18333435058594 - test loss: 0.49112969636917114 - test accuracy:
88.81999969482422
```

0.2.4 Tensorboard

[10]:
```
# running tensorboard
%load_ext tensorboard
%tensorboard --logdir logs

#%reload_ext tensorboard
```

```
Output hidden; open in https://colab.research.google.com to view.
```

[0]:
```
#!rm -rf ./logs/
```

This code is a mixture of tensorflow website, cityscape image segmentation tutorial, and image classification from scratch tutorial.

0.2.5 Comparison

The primary usage of Keras is facilitating modelling, training and validating procedures, in which I needed to implement various functions as mentioned above.

In the tensorboard above, the result of image classification without Keras is described as my_train suffix for the training set(light blue line) and my_test suffix for the test set(red line). The metrics for Keras image classification is described as epoch_accuracy(dark blue line) and epoch_loss(dark blue line).

Accuracy

As we can see in the tensorboard above the first plot, our model gain accuracy of 96% on the training data after 70 epochs, which shows obvious overfitting, as after 31 epochs, the test accuracy won't increase from 88%, but our model continues to improve its train accuracy. However, our final test accuracy is 88.82%.

Keras model gain 95.59% accuracy on the training data and final accuracy of test set after 70 epochs is 89.54%, which shows Keras is better in the modelling procedure.

Loss

In the above tensorboard, the second plots show the loss for Keras classification and the third plot shows the loss for our non-Keras model. We can see that our model has less loss compared to Keras. Our model test loss after 70 Epochs is 49.11%, and Keras' loss on test set is 49.92%. Consequently, our model and Keras doesn't have significant different in final loss on the test set.

Keras' loss is 11.44% on the training data and our model loss is 10.16% after 70 epochs. I think this is caused because of my learning rate, which is 0.001 for the model.

Moreover, we can see that after 14 epochs, both models manage to get to the optimal point during learning, which is shown by the least loss, and after 20 epochs,testing accuracy fluctuates only in 1% range.

Time

In the prints above, we can see that our model epoch time is between 8 to 9 seconds and fluctuates slightly. Keras epoch time is fixed for 7 seconds, which results in 2 minutes difference between Keras and without Keras modelling.

In conclusion, Keras is easier, and faster in modelling neural networks.

Summary

Here is a short comparison between these two models:

Metric	Without Keras	Keras
Accuracy - Test	88.82%	89.54%
Loss - Test	49.11%	49.92%
Time per Epoch	8.7s	7s
Time per Sample	145s	113s
Total Time	9m 58s	7m 54s
Accuracy - Train	96.11%	95.59%
Loss - Train	10.16%	11.44%

0.3 2. CPU vs. GPU computation

[12]:
```
import tensorflow as tf
import numpy as np
import pandas as pd
import time

def create_random_matrix(size_in):
  """
  This function creates two square matrices based on given size of the matrix
  :param size_in: size of the matrix
  :return first_matrix, second_matrix: Returns two random square matrices with␣
 ↪the same size
  """

  # setting seed for having different random numbers for the other matrix
  first_matrix = tf.random.uniform([size_in, size_in])

  tf.random.set_seed(time.time())
  second_matrix = tf.random.uniform([size_in, size_in])

  return first_matrix, second_matrix

def time_product(first_matrix, second_matrix):
  """
  This function multiplicates two square matrices and calculates execution time
```

```
    :param first_matrix: first random square matrix
    :param second_matrix: second random square matrix
    :return elapsed: Returns elapsed time for multiplicating matrices
    :return result_product: Returns product of two matrices
    """

    # starting time
    start = time.perf_counter()

    result_product = tf.matmul(first_matrix, second_matrix)

    # calculating elapsed time from execution
    elapsed = time.perf_counter() - start

    return elapsed, result_product

def compute_cpu(first_matrix, second_matrix):
    """
    This function uses CPU to multiplicate two square matrices and calculate␣
 ↪execution time
    :param first_matrix: first random square matrix
    :param second_matrix: second random square matrix
    :return elapsed_time: Returns elapsed time for multiplicating matrices on CPU
    """

    # Force execution on CPU
    with tf.device("CPU:0"):
        elapsed_time, product = time_product(first_matrix, second_matrix)
        # throws exception if the code is not executed on CPU
        assert product.device.endswith("CPU:0")

    return elapsed_time

def compute_gpu(first_matrix, second_matrix):
    """
    This function uses GPU to multiplicate two square matrices and calculate␣
 ↪execution time
    :param first_matrix: first random square matrix
    :param second_matrix: second random square matrix
    :return elapsed_time: Returns elapsed time for multiplicating matrices on GPU
    """

    # Force execution on GPU #0 if available
    if tf.config.experimental.list_physical_devices("GPU"):
        with tf.device("GPU:0"):
            elapsed_time, product = time_product(matrix1, matrix2)
            # throws exception if the code is not executed on GPU
```

```
      assert product.device.endswith("GPU:0")

  return elapsed_time


# main program
rounds = 10
square_size = [500, 1000, 5000, 10000]

answer_timer = {500:{}, 1000:{}, 5000:{}, 10000:{}}

for size in square_size:
  # calculating for each size

  elapsed_time_cpu = 0
  elapsed_time_gpu = 0

  for i in range(rounds):
    # calculating computation for a specific rounds for each size
    matrix1, matrix2 = create_random_matrix(size)

    elapsed_time_cpu += compute_cpu(matrix1, matrix2)

    elapsed_time_gpu += compute_gpu(matrix1, matrix2)

  # calculating average for computation in ms
  answer_timer[size]['CPU'] = elapsed_time_cpu*1000/rounds
  answer_timer[size]['GPU'] = elapsed_time_gpu*1000/rounds


# printing result in tabular format
print("Computing computation time for 10 round of multiplication")
print("{:<8} {:<15} {:<10}".format('size','processor', 'execution time(ms)'))
for size, item in answer_timer.items():
    for proc, timer in item.items():
      print("{:<8} {:<15} {:<10}".format(size, proc, timer))
```

```
Computing computation time for 10 round of multiplication
size     processor       execution time(ms)
500      CPU             8.90463819996512
500      GPU             0.1385243000186165
1000     CPU             31.14645260029647
1000     GPU             0.2542460002587177
5000     CPU             3473.367635300201
5000     GPU             0.3389669003809104
```

```
10000    CPU              28565.83050890004
10000    GPU              0.3881196995280334
```

Sample Output for multiplication computation for average in 10 round of multiplication in m second:

Size	Processor	Execution time(ms)
500	CPU	9.760756099922219
500	GPU	0.15373470009762968
1000	CPU	32.54002580001725
1000	GPU	0.24147199987964996
5000	CPU	3449.983324600135
5000	GPU	0.3366110000115441
10000	CPU	28675.065360599954
10000	GPU	0.38258300000961754

Because the matrices were sampled from uniform distribution, for each size the computation time was averaged between specific number of rounds.

As we can see in this table, the computation time for GPU is significantly less than CPU. But one important result is that as size of matrices increase dramatically, the computation time for CPU increases significantly but for GPU there is a slight rise in computation time.

This code was obtained from tensorflow website.

0.4 3. Differentiation

Jaconbian Matrix **J**:

$$\mathbf{f} : R^m \to R^n \,, \mathbf{J} \in R^{n*m} \;\; : J_{i,j} = \frac{\partial}{\partial x_j} f(\mathbf{x})_i$$

$$\mathbf{y} = 3\mathbf{x}^2 + 2\mathbf{x} + 3 = 3(\mathbf{x} \odot \mathbf{x}) + 2\mathbf{x} + 3 \Rightarrow \frac{\partial y}{\partial x} = 3(\frac{\partial x}{\partial x} \odot x + x \odot \frac{\partial x}{\partial x}) + 2\frac{\partial x}{\partial x} + 3\frac{\partial}{\partial x}$$

$$\mathbf{J} = \big(3(x+x) + 2 + 0\big) = \big(\underline{6x+2}\big)$$

for $x = \begin{pmatrix} 1 \\ 3 \end{pmatrix} \to \mathbf{J} = \begin{pmatrix} 8 \\ 20 \end{pmatrix}$

This following code was obtained from tensorflow website.

[2]:
```
import tensorflow as tf
import numpy as np

# creating x matrix
x = tf.constant([[1.0], [3.0]])

with tf.GradientTape(persistent=True) as t:
  t.watch(x)
  y = 3* x * x + 2 * x + 3
```

```
dy_dx = t.gradient(y, x)

print("Jacobian of y = 3x^2 + 2x + 3 for x = [[1][3]]:")
with tf.Session() as sess:  print("J = ", dy_dx.eval())
```

```
Jacobian of y = 3x^2 + 2x + 3 for x = [[1][3]]:
J =  [[ 8.]
 [20.]]
```

As we can see above, by hand and by code the same result is obtained.

0.5 4. Eigenvectors

$$Av = \lambda v \quad \textit{where } \lambda : \textit{eigenvalue} \;,\; v : \textit{eigenvector}$$

$$Av - \lambda v = 0 \rightarrow (A - \lambda I)v = 0 \,, v \neq 0 \rightarrow A - \lambda I = 0$$

Calculating eigenvalues)

$$A - \lambda I = 0 \,, ker(A - \lambda I) \neq 0 \rightarrow A - \lambda I \;: not - invertible \Rightarrow \underline{det(A - \lambda I) = 0}$$

$$det(\begin{pmatrix} 3 & 2 \\ 2 & 3 \end{pmatrix} - \lambda I) = det(\begin{pmatrix} 3 & 2 \\ 2 & 3 \end{pmatrix} - \lambda \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}) = det(\begin{pmatrix} 3 & 2 \\ 2 & 3 \end{pmatrix} - \begin{pmatrix} \lambda & 0 \\ 0 & \lambda \end{pmatrix}) = det(\begin{pmatrix} 3-\lambda & 2 \\ 2 & 3-\lambda \end{pmatrix}) \rightarrow$$

$$det(A - \lambda I) = \lambda^2 - 6\lambda + 5 = 0 \rightarrow (\lambda - 5)(\lambda - 1) = 0 \rightarrow \underline{\lambda = 5} \,, \underline{\lambda = 1}$$

Calculating eigenvectors)

$$A - \lambda I = \begin{pmatrix} 3-\lambda & 2 \\ 2 & 3-\lambda \end{pmatrix} \,, (A - \lambda I)v = 0$$

$\underline{\lambda = 5}$)

$$\begin{pmatrix} 3-\lambda & 2 \\ 2 & 3-\lambda \end{pmatrix} = \begin{pmatrix} -2 & 2 \\ 2 & -2 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

By adding first row to the second row, we have:

$$\begin{pmatrix} -2 & 2 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \rightarrow -2v_1 + 2v_2 = 0 \rightarrow \underline{v_1 = v_2}$$

$\underline{\lambda = 1}$)

$$\begin{pmatrix} 3-\lambda & 2 \\ 2 & 3-\lambda \end{pmatrix} = \begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

By subtracting first row from the second row, we have:

$$\begin{pmatrix} 2 & 2 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \rightarrow 2v_1 + 2v_2 = 0 \rightarrow \underline{v_1 = -v_2}$$

Examples)

We can check correctness of our eigen values and eigenvectors by checking the following equation:

$$Av = \lambda v$$

1. $\underline{\lambda = 5, v_1 = v_2}$) $v_1 = 1,\ v_2 = 1$

$$\begin{pmatrix} 3 & 2 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = 5 \begin{pmatrix} 1 \\ 1 \end{pmatrix} \rightarrow \begin{pmatrix} 5 \\ 5 \end{pmatrix} = \begin{pmatrix} 5 \\ 5 \end{pmatrix} \quad \checkmark$$

2. $\underline{\lambda = 1, v_1 = -v_2}$) $v_1 = 1,\ v_2 = -1$

$$\begin{pmatrix} 3 & 2 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} 1 \\ -1 \end{pmatrix} = 1 \begin{pmatrix} 1 \\ -1 \end{pmatrix} \rightarrow \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad \checkmark$$