Assignment 1

About

In this assignment, you will gain experience implementing a linear model, a neural network, and a deep neural network using TensorFlow 2.0.

- You will use two different development styles. I thought it'd be helpful for you to see both of these early (if you're familiar with them, you can branch out to any major framework that exist today).
- Along the way, you'll add code to visualize the weights of a linear model, and provide your own implementation of softmax (so you learn to extend the built-in functionality right off the bat).

This assignment has several parts, plan ahead and get started early (and come to office hours if you're stuck, the CAs are happy to help).

Instructions

Complete the code in this notebook by searching for the text **"TODO"**.

Submission instructions

Please submit this assignment on CourseWorks by uploading a Jupyter notebook that includes saved output. If you are working in Colab, you can prepare your notebook for submission by ensuring that runs end-to-end, then saving and downloading it:

1. `Runtime -> Restart and run all`
2. `File -> Save`
3. `File -> Download.ipynb`

Resources

You can find all the latest tutorials for TensorFlow 2.0 here (https://www.tensorflow.org/tutorials). Code examples that will help you with each part of the assignment are linked below.

Setup

Install TensorFlow 2.0

The most recent version of TensorFlow is alreadey installed in Colab. If you prefer working in Jupyter locally, you will need to install TensorFlow following these instructions (http://tensorflow.org/install).

Check which TF version is installed

It's good practice to check which version you have installed.

In [1]:

```
import tensorflow as tf
import numpy as np
print(tf.__version__)
```

```
2.5.0
```

In [2]:

```
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras import Model
import matplotlib.pyplot as plt
```

Part 1: First steps with Sequential models

You will work with the Sequential API in this section. This is the easiest way to develop models with TF 2.0, and is the most common in practice.

Here are a few code examples that will help you with this part of the assignment:

- Get started for beginners (https://www.tensorflow.org/tutorials/quickstart/beginner)
- Classify images (https://www.tensorflow.org/tutorials/keras/basic_classification)
- Explore overfitting and underfitting (https://www.tensorflow.org/tutorials/keras/overfit_and_underfit)

Download and prepare a dataset

In [3]:

```
mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
```

Define, train, and evaluate a linear model

In [4]:

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(10, activation='softmax')
])

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

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

```
Epoch 1/5
1875/1875 [==============================] - 1s 560us/step - loss: 0.4
707 - accuracy: 0.8771
Epoch 2/5
1875/1875 [==============================] - 1s 531us/step - loss: 0.3
035 - accuracy: 0.9159
Epoch 3/5
1875/1875 [==============================] - 1s 525us/step - loss: 0.2
835 - accuracy: 0.9214
Epoch 4/5
1875/1875 [==============================] - 1s 530us/step - loss: 0.2
727 - accuracy: 0.9244
Epoch 5/5
1875/1875 [==============================] - 1s 537us/step - loss: 0.2
669 - accuracy: 0.9260
313/313 [==============================] - 0s 439us/step - loss: 0.269
5 - accuracy: 0.9270
```

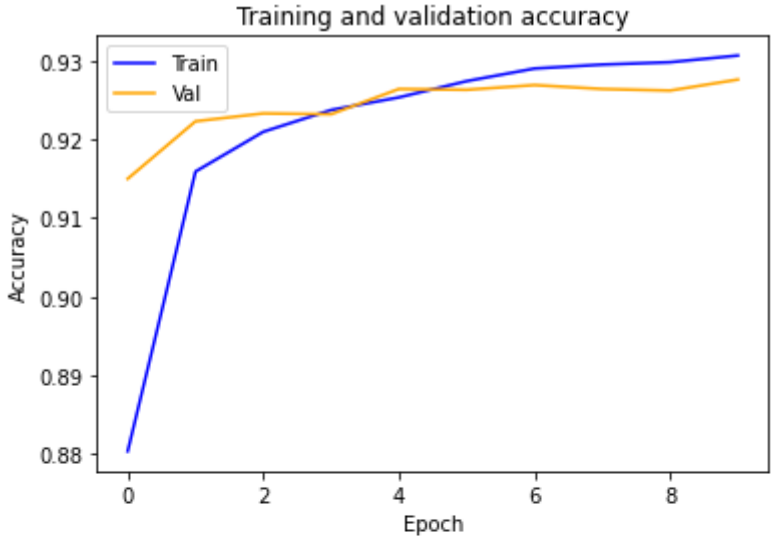
Out[4]:

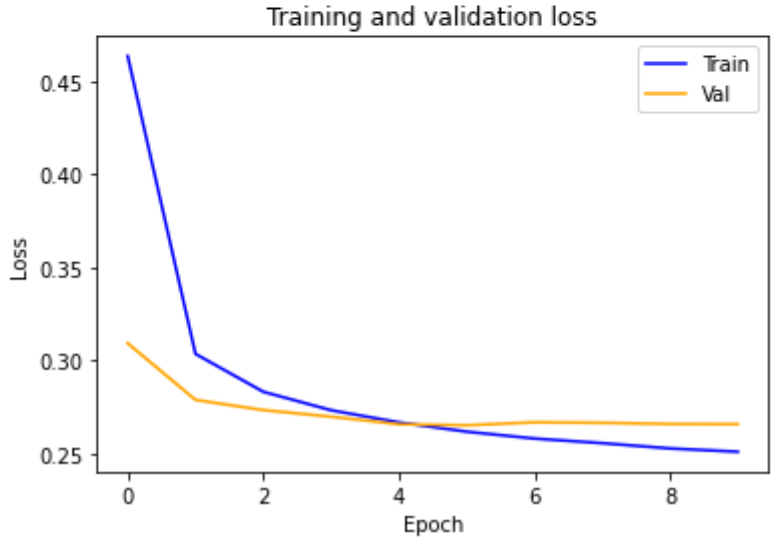
```
[0.2694820165634155, 0.9269999861717224]
```

1a: Plot loss and accuracy

TODO

Modify the code below to produce plots showing loss and accuracy as a function of epochs on training and validation data (it's fine to use x_test and y_test as validation data for this assignment). To do so, you will need to add validation data to the call for `model.fit` , and capture the results in a history object. Code for plotting is provided for you, you can pass your history object to this. You can find additional example code here (https://www.tensorflow.org/tutorials/keras/text_classification).


Training and validation accuracy
Train
Val
Accuracy
0.93
0.92
0.91
0.90
0.89
0.88
0
2
4
6
8
Epoch

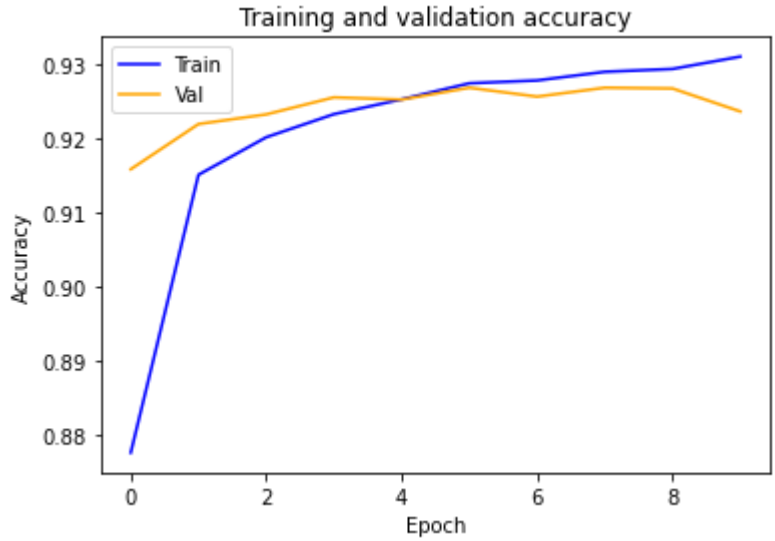

Training and validation loss
Train
Val
Loss
0.45
0.40
0.35
0.30
0.25
0
2
4
6
8
Epoch

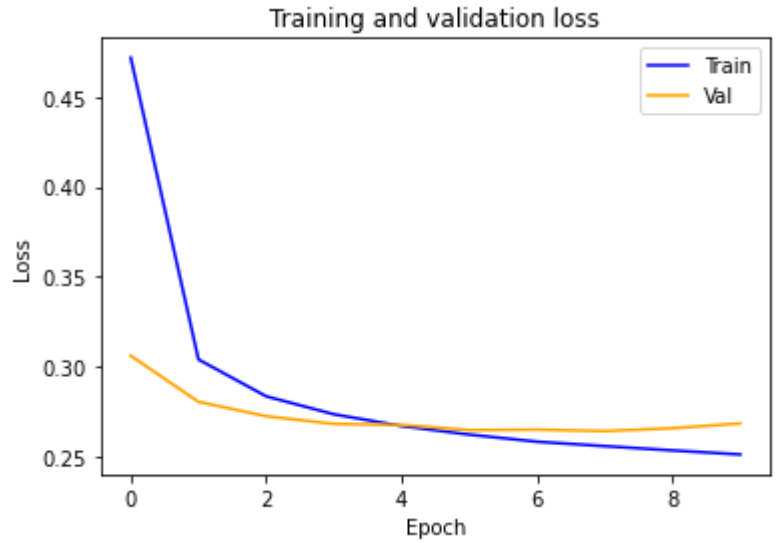
In [5]:

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(10, activation='softmax')
])

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


# TODO
#
# 1. create a history object to store the results of model.fit
# ```history = model.fit(...)```
#
# 2. add another parameter to model.fit for validation data
# https://keras.io/models/sequential/

history = model.fit(x_train, y_train, validation_data = (x_test, y_test), epochs=10)

# A plotting function you can reuse
def plot(history):

  # The history object contains results on the training and test
  # sets for each epoch
  acc = history.history['accuracy']
  val_acc = history.history['val_accuracy']
  loss = history.history['loss']
  val_loss = history.history['val_loss']

  # Get the number of epochs
  epochs = range(len(acc))

  plt.title('Training and validation accuracy')
  plt.plot(epochs, acc, color='blue', label='Train')
  plt.plot(epochs, val_acc, color='orange', label='Val')
  plt.xlabel('Epoch')
  plt.ylabel('Accuracy')
  plt.legend()

  _ = plt.figure()
  plt.title('Training and validation loss')
  plt.plot(epochs, loss, color='blue', label='Train')
  plt.plot(epochs, val_loss, color='orange', label='Val')
  plt.xlabel('Epoch')
  plt.ylabel('Loss')
  plt.legend()

plot(history)
```

```
Epoch 1/10
1875/1875 [==============================] - 1s 635us/step - loss: 0.4
720 - accuracy: 0.8777 - val_loss: 0.3060 - val_accuracy: 0.9159
Epoch 2/10
1875/1875 [==============================] - 1s 621us/step - loss: 0.3
039 - accuracy: 0.9151 - val_loss: 0.2803 - val_accuracy: 0.9220
Epoch 3/10
1875/1875 [==============================] - 1s 603us/step - loss: 0.2
```

```
833 - accuracy: 0.9202 - val_loss: 0.2722 - val_accuracy: 0.9233
Epoch 4/10
1875/1875 [==============================] - 1s 594us/step - loss: 0.2
733 - accuracy: 0.9233 - val_loss: 0.2680 - val_accuracy: 0.9256
Epoch 5/10
1875/1875 [==============================] - 1s 598us/step - loss: 0.2
668 - accuracy: 0.9253 - val_loss: 0.2674 - val_accuracy: 0.9253
Epoch 6/10
1875/1875 [==============================] - 1s 599us/step - loss: 0.2
620 - accuracy: 0.9275 - val_loss: 0.2645 - val_accuracy: 0.9269
Epoch 7/10
1875/1875 [==============================] - 1s 612us/step - loss: 0.2
580 - accuracy: 0.9279 - val_loss: 0.2648 - val_accuracy: 0.9257
Epoch 8/10
1875/1875 [==============================] - 1s 623us/step - loss: 0.2
556 - accuracy: 0.9291 - val_loss: 0.2640 - val_accuracy: 0.9269
Epoch 9/10
1875/1875 [==============================] - 1s 602us/step - loss: 0.2
532 - accuracy: 0.9294 - val_loss: 0.2656 - val_accuracy: 0.9268
Epoch 10/10
1875/1875 [==============================] - 1s 610us/step - loss: 0.2
509 - accuracy: 0.9311 - val_loss: 0.2682 - val_accuracy: 0.9237
```





1b: Implement a neural network

TODO

Modify the code below to create a neural network (with a single hidden layer). Add a Dense layer with 128 units and ReLU activation. Train and evaluate your model. It is not necessary to produce plots for this section.

In [6]:

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation = 'relu'),
  # TODO: Add a layer here
  tf.keras.layers.Dense(10, activation='softmax')
])

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

model.fit(x_train, y_train, epochs=5)
```

```
Epoch 1/5
1875/1875 [==============================] - 2s 711us/step - loss: 0.2
568 - accuracy: 0.9268
Epoch 2/5
1875/1875 [==============================] - 1s 692us/step - loss: 0.1
135 - accuracy: 0.9664
Epoch 3/5
1875/1875 [==============================] - 1s 718us/step - loss: 0.0
765 - accuracy: 0.9764
Epoch 4/5
1875/1875 [==============================] - 1s 683us/step - loss: 0.0
578 - accuracy: 0.9822
Epoch 5/5
1875/1875 [==============================] - 1s 708us/step - loss: 0.0
441 - accuracy: 0.9866
```

Out[6]:

```
<tensorflow.python.keras.callbacks.History at 0x7fa1affa1b20>
```

1c: Implement a deep neural network

TODO

Modify the code below to create and train a deep neural network with at least two hidden layers.

In [7]:

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  # TODO: Add two layers here
  tf.keras.layers.Dense(128, activation = 'relu'),
  tf.keras.layers.Dense(64, activation = 'relu'),
  tf.keras.layers.Dense(32, activation = 'relu'),
  tf.keras.layers.Dense(10, activation='softmax')
])

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

model.fit(x_train, y_train, epochs=10)
```

```
Epoch 1/10
1875/1875 [==============================] - 2s 788us/step - loss: 0.2
553 - accuracy: 0.9238
Epoch 2/10
1875/1875 [==============================] - 2s 860us/step - loss: 0.1
058 - accuracy: 0.9685
Epoch 3/10
1875/1875 [==============================] - 2s 827us/step - loss: 0.0
741 - accuracy: 0.9766
Epoch 4/10
1875/1875 [==============================] - 1s 770us/step - loss: 0.0
571 - accuracy: 0.9822
Epoch 5/10
1875/1875 [==============================] - 1s 785us/step - loss: 0.0
485 - accuracy: 0.9839
Epoch 6/10
1875/1875 [==============================] - 1s 770us/step - loss: 0.0
388 - accuracy: 0.9877
Epoch 7/10
1875/1875 [==============================] - 1s 771us/step - loss: 0.0
313 - accuracy: 0.9898
Epoch 8/10
1875/1875 [==============================] - 1s 775us/step - loss: 0.0
302 - accuracy: 0.9901
Epoch 9/10
1875/1875 [==============================] - 1s 769us/step - loss: 0.0
258 - accuracy: 0.9914
Epoch 10/10
1875/1875 [==============================] - 1s 768us/step - loss: 0.0
206 - accuracy: 0.9934
```

Out[7]:

```
<tensorflow.python.keras.callbacks.History at 0x7fa1affa1d90>
```

1d: Display predictions and their confidence

TODO

1. Choose one of your models above. Use it to make predictions on the entire test set using `model.predict`

2. Next, identify one image from the testing set the model classifies correctly, and another that it classifies incorrectly. Add code to display both of these images below, the correct labels for them, the predicted labels, and the confidence scores.

In [8]:

```
# TODO: add your code here
y_pred = model.predict(x_test)
y_class = np.argmax(y_pred)
```

In [9]:

```
x_test.shape
```

Out[9]:

```
(10000, 28, 28)
```

In [10]:

```
y_pred
```

Out[10]:

```
array([[3.2367626e-11, 9.0012868e-09, 1.0867403e-08, ..., 9.9999964e-0
1,
        7.5186621e-09, 2.3427242e-07],
       [4.1757954e-07, 3.2842400e-09, 9.9999952e-01, ..., 3.8791267e-1
0,
        1.6693984e-09, 5.5240023e-14],
       [1.0818306e-08, 9.9999201e-01, 1.2265028e-06, ..., 8.6834325e-0
7,
        3.1369887e-06, 4.8871822e-08],
       ...,
       [1.2576627e-16, 1.5399065e-09, 2.3776180e-17, ..., 1.0075869e-1
2,
        4.2660824e-13, 7.5756141e-08],
       [4.3080709e-11, 7.3970968e-12, 4.1059319e-11, ..., 1.2704471e-1
2,
        4.8901115e-06, 7.0871981e-13],
       [4.4730070e-10, 8.6800679e-12, 1.8843297e-14, ..., 9.1938627e-2
3,
        4.4445926e-14, 7.6505989e-12]], dtype=float32)
```

In [11]:

```
y_test
```

Out[11]:

```
array([7, 2, 1, ..., 4, 5, 6], dtype=uint8)
```

In [12]:

```
correct, false = list(), list()
for i in range( len(y_pred)):
    if np.argmax(y_pred[i]) == y_test[i]:
        correct.append(i)
    else:
        false.append(i)
print(correct[0], false[0])
```

```
0 8
```

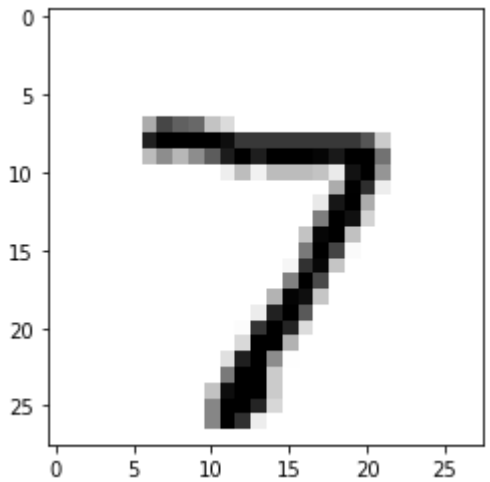
In [13]:

```
# we choose the first prediction as correct example and the 115th prediciton as fals
```

In [14]:

```
plt.imshow(x_test[0], cmap=plt.cm.binary)
print("ID: 0")
print("Predicted Label", np.argmax(y_pred[0]), "True Label:", y_test[0])
print("Confidence Scores for 10 classes:", y_pred[0])
```

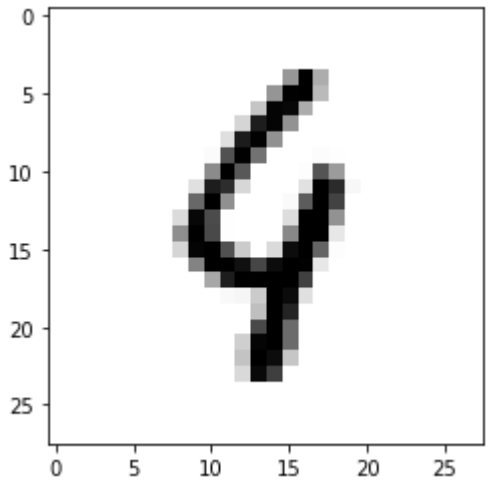
```
ID: 0
Predicted Label 7 True Label: 7
Confidence Scores for 10 classes: [3.2367626e-11 9.0012868e-09 1.08674
03e-08 1.3637430e-07 3.3999654e-09
 4.3663428e-12 5.1253345e-17 9.9999964e-01 7.5186621e-09 2.3427242e-0
7]
```



In [15]:

```
plt.imshow(x_test[115], cmap=plt.cm.binary)
print("ID: 115")
print("Predicted Label", np.argmax(y_pred[115]), "True Label:", y_test[115])
print("Confidence Scores for 10 classes:", y_pred[115])
```

```
ID: 115
Predicted Label 9 True Label: 4
Confidence Scores for 10 classes: [2.8530524e-06 1.8787830e-07 3.27949
89e-10 4.5732691e-08 3.2518097e-04
 5.1783296e-08 5.0464268e-07 1.8771158e-09 1.9778712e-08 9.9967122e-0
1]
```



Part 2: Subclassed models

In this part of the assignment, you'll work with the Keras Subclassing API. Instead of using a built-in method (`model.fit`) you will train models using a GradientTape.

Here are a few code examples that will help you with this part of the assignment:

- Get started for experts (https://www.tensorflow.org/tutorials/quickstart/advanced)
- TensorFlow basics (https://www.tensorflow.org/guide/basics)
- Keras overview (https://www.tensorflow.org/guide/keras/overview)
- Writing custom models (https://www.tensorflow.org/guide/keras/custom_layers_and_models)

Download and prepare a dataset

This is similar to the above, except now we'll use `tf.data` to batch and shuffle the data, instead of the utilities baked into `model.fit` . It's not necessary for this assignment, but if you wish, you can learn how to use tf.data here (https://www.tensorflow.org/guide/data).

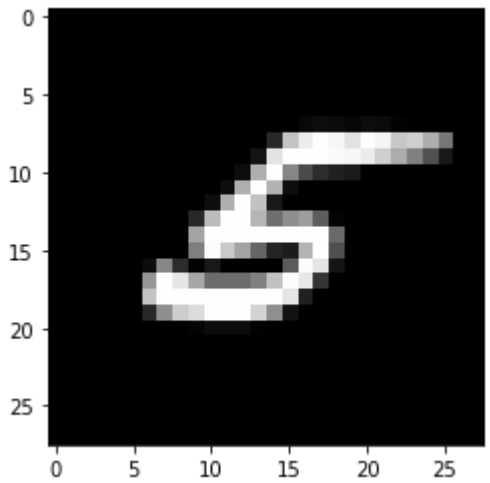
In [16]:

```
# Download a dataset
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# Batch and shuffle the data
train_ds = tf.data.Dataset.from_tensor_slices(
    (x_train.astype('float32') / 255, y_train)).shuffle(1024).batch(32)

test_ds = tf.data.Dataset.from_tensor_slices(
    (x_test.astype('float32') / 255, y_test)).batch(32)
```

In [17]:

```
# A quick example of iterating over a dataset object
for image, label in train_ds.take(1):
  plt.imshow(image[0], plt.get_cmap('gray'))
  print(label[0])
```

```
tf.Tensor(5, shape=(), dtype=uint8)
```



Define and train a linear model

In [18]:

```
class MyLinearModel(Model):
  def __init__(self):
    super(MyLinearModel, self).__init__()
    self.flatten = Flatten()
    self.d1 = Dense(10, activation='softmax', name="dense1")

  def call(self, x):
    x = self.flatten(x)
    return self.d1(x)
```

In [19]:

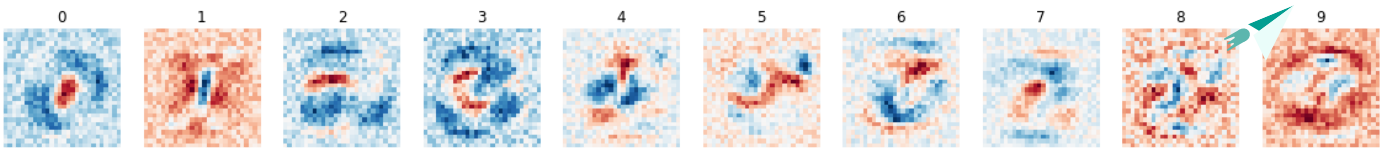
```
model = MyLinearModel()

loss_fn = tf.keras.losses.SparseCategoricalCrossentropy()
optimizer = tf.keras.optimizers.SGD()

# For each epoch
for epoch in range(5):

  # For each batch of images and labels
  for images, labels in train_ds:

    # Open a GradientTape.
    with tf.GradientTape() as tape:

      # Forward pass
      predictions = model(images)

      # Calculate loss
      loss = loss_fn(labels, predictions)

    # Backprop to calculate gradients
    gradients = tape.gradient(loss, model.trainable_variables)

    # Gradient descent step
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))

  # Calculate loss on the test data
  test_loss = []
  for images, labels in test_ds:
    predictions = model(images)
    loss_on_batch = loss_fn(labels, predictions)
    test_loss.append(loss_on_batch)

  print("Epoch {}, Test loss: {}".format(epoch, tf.reduce_mean(test_loss)))
```

```
Epoch 0, Test loss: 0.4802393913269043
Epoch 1, Test loss: 0.4000180959701538
Epoch 2, Test loss: 0.3671649694442749
Epoch 3, Test loss: 0.3483828604221344
Epoch 4, Test loss: 0.3359007239341736
```

Note: you may have noticed that the above code runs slowly (it's executing eagerly). Later in this notebook, you will compile your code (to run it in graph mode) using `@tf.function`. The general workflow is to write your code without using tf.function (as shown above, which makes for easier debugging). Once you've finished debugging your model, you can add `@tf.function` for performance if necessary.

2a: Visualize the learned weights

We can interpret a linear model by looking at the weights of the fully connected layer. Modify the below code to create a plot similar to this one from lecture 1:



TODO

Modify the below code to retrieve the learned weights. You can use either the public API of a model `model.get_layer(name)` then retrieve the weights from that, or (because our model is defined using the Subclassing API), you can access the dense layer directly `model.d1` .

In [20]:

```
# There are two ways to retrieve the weights. You can use the public API
# (model.get_layer(name).get_weights()), or, you can access the dense layer
# directly (model.d1) then find the accessor method, or again, access the
# variable directly.
# Python tip: try ```dir(model.d1)```

# TODO: modify this code to get the weights

weights, bias = (model.get_layer('dense1').get_weights()[0], model.get_layer('dense1
print(weights.shape)
```

```
(784, 10)
```

In [21]:

```
weights[:,9]
```

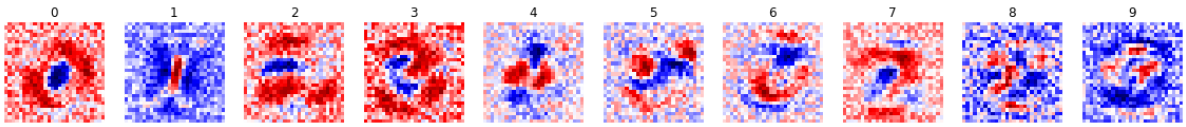
Out[21]:

```
array([ 8.66446421e-02,  6.53590485e-02,  2.67056227e-02,  8.53380188e
-02,
       -7.24661350e-03, -8.22667629e-02,  3.75542045e-03, -7.13985264e
-02,
        7.76515156e-03, -3.51574458e-02,  5.82865626e-03, -6.91174343e
-02,
        3.95607129e-02, -7.81011283e-02,  5.17939664e-02,  2.69237421e
-02,
       -2.97806263e-02,  1.47408620e-02,  6.21504113e-02, -2.00107172e
-02,
        6.63512945e-03,  2.45834142e-02, -8.60240906e-02,  6.50666133e
-02,
       -4.89614308e-02,  5.29532060e-02, -5.91285005e-02,  5.68334237e
-02,
       -3.74184363e-02,  2.63555869e-02, -2.29353383e-02, -8.10991302e
-02,
        7.23325908e-02, -8.65476727e-02,  1.77622326e-02, -4.56073694e
-02.
```

In [ ]:

In [22]:

```
fig, axs = plt.subplots(1,10, figsize=(20,20))
for i in range(10):
  subplot = axs[i]
  subplot.set_title(i)
  subplot.axis('off')
  # TODO: modify this code to complete the plotting function
  i_weights = weights[:,i] # Select the weights for the i'th output
  img = tf.reshape(i_weights, [28,28]) # Reshape the weights into a 28x28 array
  subplot.imshow(img, plt.get_cmap('seismic'))
plt.show()
```



2b: Implement a deep neural network

TODO

Modify this code to create a deep neural network. Train your model using the code below, and compare the accuracy to the linear model above.

Note: you do not need to modify any sections other than the model definition and the call method.

The code below uses compiled versions of the training and evaluation loops (remove the `@tf.function` annotations if you need to debug).

In [23]:

```
class MyDNN(Model):
    def __init__(self):
        super(MyDNN, self).__init__()
        self.flatten = Flatten()
        # Modify me
        self.d1 = Dense(128, activation='relu')
        self.d2 = Dense(64, activation='relu')
        self.d3 = Dense(32, activation='relu')
        self.d4 = Dense(10, activation='softmax')

    def call(self, x):
        x = self.flatten(x)
        x = self.d1(x)
        x = self.d2(x)
        x = self.d3(x)
        return self.d4(x)

model = MyDNN()
```

In [24]:

```
loss_object = tf.keras.losses.SparseCategoricalCrossentropy()
optimizer = tf.keras.optimizers.Adam()
```

These are helper functions we'll use to record loss and accuracy while your model is trained.

In [25]:

```
train_loss = tf.keras.metrics.Mean(name='train_loss')
train_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name='train_accuracy')

test_loss = tf.keras.metrics.Mean(name='test_loss')
test_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name='test_accuracy')
```

This method trains the model on a batch of data.

In [26]:

```
@tf.function
def train_step(images, labels):
  with tf.GradientTape() as tape:
    predictions = model(images)
    loss = loss_object(labels, predictions)
  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))

  train_loss(loss)
  train_accuracy(labels, predictions)
```

This method evaluates the model on a batch of data.

In [27]:

```
@tf.function
def test_step(images, labels):
  predictions = model(images)
  t_loss = loss_object(labels, predictions)

  test_loss(t_loss)
  test_accuracy(labels, predictions)
```

Training and evaluation loop.

In [28]:

```
EPOCHS = 5

for epoch in range(EPOCHS):
  for images, labels in train_ds:
    train_step(images, labels)

  for test_images, test_labels in test_ds:
    test_step(test_images, test_labels)

  template = 'Epoch {}, Loss: {}, Accuracy: {}, Test Loss: {}, Test Accuracy: {}'
  print(template.format(epoch+1,
                        train_loss.result(),
                        train_accuracy.result()*100,
                        test_loss.result(),
                        test_accuracy.result()*100))

  # Reset the metrics for the next epoch
  train_loss.reset_states()
  train_accuracy.reset_states()
  test_loss.reset_states()
  test_accuracy.reset_states()
```

```
Epoch 1, Loss: 0.250449538230896, Accuracy: 92.67166900634766, Test Lo
ss: 0.1425177901983261, Test Accuracy: 95.45000457763672
Epoch 2, Loss: 0.10370540618896484, Accuracy: 96.86499786376953, Test
Loss: 0.10170017927885056, Test Accuracy: 96.75
Epoch 3, Loss: 0.0739724263548851, Accuracy: 97.68999481201172, Test L
oss: 0.1000342071056366, Test Accuracy: 96.94000244140625
Epoch 4, Loss: 0.05677380412817001, Accuracy: 98.21166229248047, Test
Loss: 0.08949826657772064, Test Accuracy: 97.22999572753906
Epoch 5, Loss: 0.04504245147109032, Accuracy: 98.62166595458984, Test
Loss: 0.09845070540904999, Test Accuracy: 97.15999603271484
```

2c: Provide your own implementation of softmax and use it to train a model

In your linear model above, the starter code looked similar to:

```
class LinearModel(Model):
  def __init__(self):
    super(LinearModel, self).__init__()
    self.flatten = Flatten()
    self.d1 = Dense(10, activation='softmax')

  def call(self, x):
    x = self.flatten(x)
    return self.d1(x)
```

Now, create a function:

```
def my_softmax(logits):
  # ...
```

and use it in your model as follows:

```
    class LinearModel(Model):
      def __init__(self):
        super(LinearModel, self).__init__()
        self.flatten = Flatten()
        self.d1 = Dense(10)

      def call(self, x):
        x = self.flatten(x)
        x = self.d1(x)
        return my_softmax(x)
```

Notice, we've removed the built-in activation method on the Dense layer, and added our own to the call method.

Tip: You can implement softmax first using NumPy, if you like, the gradually convert your code to use TensorFlow ops (which begin with tf.* instead of np.*).

Notes:

- Your softmax implementation should be numerically stable.
- You will need to use tf.* ops in order to use your code to train a model (TF cannot backprop through NumPy operations).

In [29]:

```
# TODO: your code here

def my_softmax(x):
    X_exp = tf.exp(x)
    return  X_exp/tf.reduce_sum(X_exp)


class MyModel(Model):
  def __init__(self):
    super(MyModel, self).__init__()
    self.flatten = Flatten()
    self.d1 = Dense(10)

  def call(self, x):
    x = self.flatten(x)
    x = self.d1(x)
    return my_softmax(x)

model = MyModel()


# TODO
# Add code to train this model
# When it's trained, the accuracy should be similar to the linear
# model from part one (but not identical, the weights are initialized randomly)
```

In [30]:

```
loss_object = tf.keras.losses.SparseCategoricalCrossentropy()
optimizer = tf.keras.optimizers.Adam()
train_loss = tf.keras.metrics.Mean(name='train_loss')
train_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name='train_accuracy')

test_loss = tf.keras.metrics.Mean(name='test_loss')
test_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name='test_accuracy')
@tf.function
def train_step(images, labels):
  with tf.GradientTape() as tape:
    predictions = model(images)
    loss = loss_object(labels, predictions)
  gradients = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(gradients, model.trainable_variables))

  train_loss(loss)
  train_accuracy(labels, predictions)
@tf.function
def test_step(images, labels):
  predictions = model(images)
  t_loss = loss_object(labels, predictions)

  test_loss(t_loss)
  test_accuracy(labels, predictions)
```

In [31]:

```
EPOCHS = 5

for epoch in range(EPOCHS):
  for images, labels in train_ds:
    train_step(images, labels)

  for test_images, test_labels in test_ds:
    test_step(test_images, test_labels)

  template = 'Epoch {}, Loss: {}, Accuracy: {}, Test Loss: {}, Test Accuracy: {}'
  print(template.format(epoch+1,
                        train_loss.result(),
                        train_accuracy.result()*100,
                        test_loss.result(),
                        test_accuracy.result()*100))

  # Reset the metrics for the next epoch
  train_loss.reset_states()
  train_accuracy.reset_states()
  test_loss.reset_states()
  test_accuracy.reset_states()
```

```
Epoch 1, Loss: 0.4669547975063324, Accuracy: 87.66166687011719, Test L
oss: 0.3107800781726837, Test Accuracy: 91.4699935913086
Epoch 2, Loss: 0.3026660084724426, Accuracy: 91.52999877929688, Test L
oss: 0.2801017463207245, Test Accuracy: 92.04999542236328
Epoch 3, Loss: 0.28079625964164734, Accuracy: 92.07500457763672, Test
Loss: 0.27176058292388916, Test Accuracy: 92.15999603271484
Epoch 4, Loss: 0.2687476873397827, Accuracy: 92.5183334350586, Test Lo
ss: 0.26661887764930725, Test Accuracy: 92.44999694824219
Epoch 5, Loss: 0.2616611123085022, Accuracy: 92.69667053222656, Test L
oss: 0.26668667793273926, Test Accuracy: 92.29999542236328
```

In [ ]:

In [ ]: