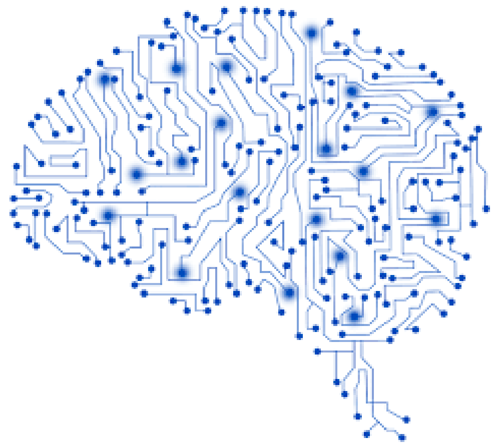
University of Minho
School of Engineering

ISLab



Deep Learning with TensorFlow™

Connective Systems and Classifiers

Perfil ML:FA @ MiEI/4º ano - 2º Semestre

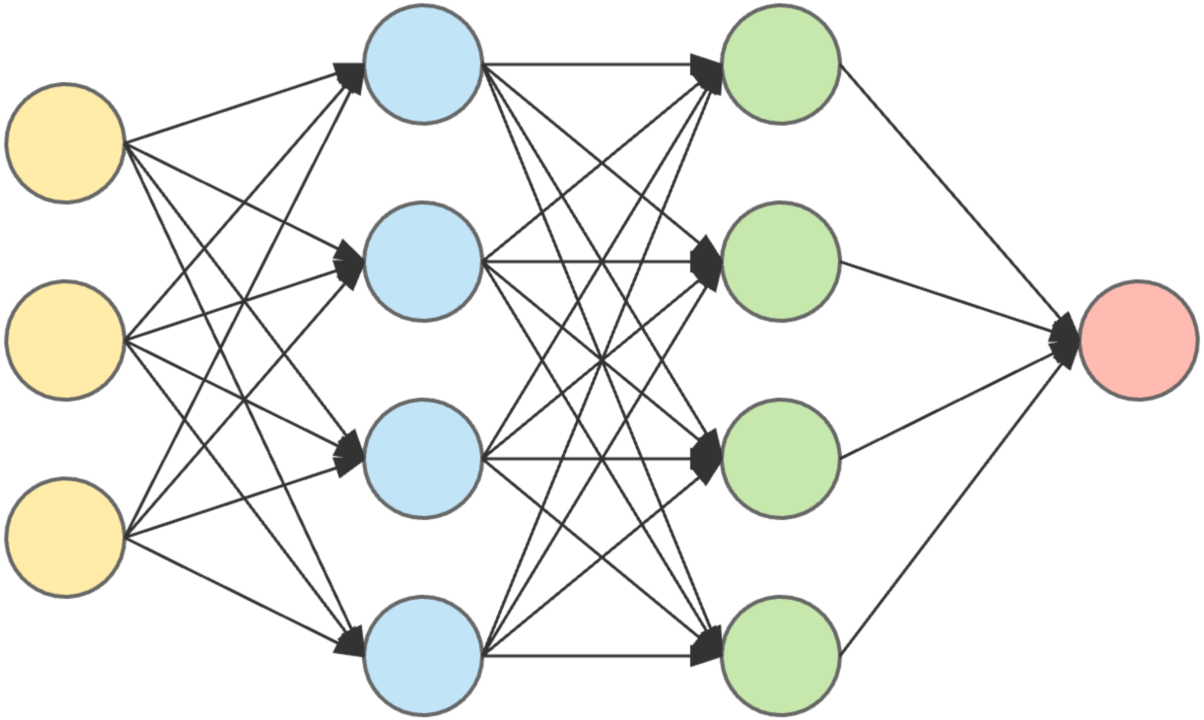
Bruno Fernandes, Victor Alves

27/02/2020

Contents

- Building a Multilayer Perceptron from scratch (Low-level Deep Learning)
- Deep Learning concepts:
  - Weights and Bias
  - Activation functions
  - Logits and Probs
  - Epochs and Batch Processing
  - Loss, Gradient and the Gradient Tape
  - Optimizers and Metrics
  - Vectorization
- Hands On

MLP from scratch with TF



MLP from scratch with TF

MLP = Multilayer Perceptron

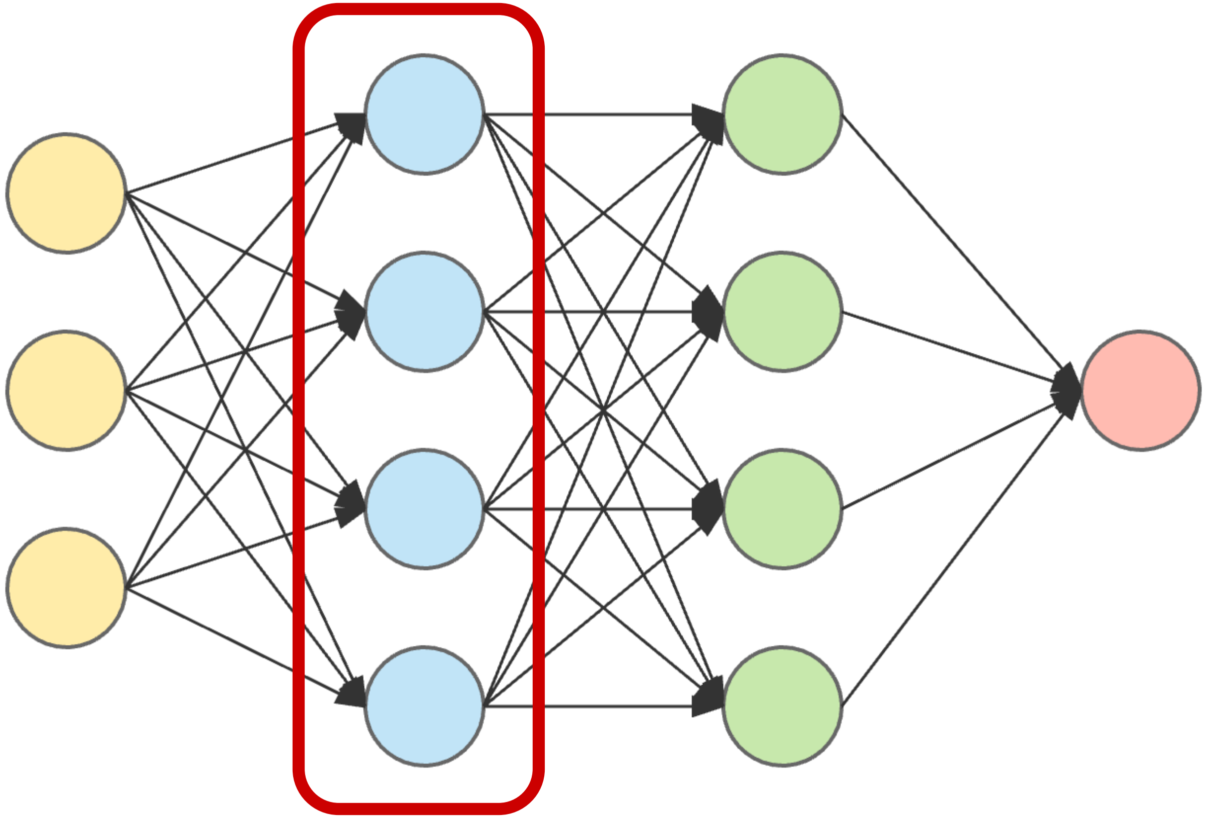
Q.: Have you heard about Deep Learning?
A.: Essentially, it is a neural network with one, or more, hidden layers!

As for a Multilayer Perceptron, it is a class of feedforward artificial neural network with, at least, one hidden layer - causing it to be deep!

In MLPs, a perceptron consists of one, or more, inputs, a processor and an output

MLP from scratch with TF

A Perceptron Layer



Some key concepts…

Weights

A coefficient for a feature in a linear model, or an edge in a deep network. The goal of training a linear model is to determine the ideal weight for each feature. If a weight is 0, then its corresponding feature does not contribute to the model.

Bias

An intercept or offset from an origin. Bias (also known as the bias term) is referred to as b in machine learning models.

$$Z = w \cdot x + b$$

Activation Function

A function that takes in the weighted sum of all of the inputs from the previous layer and then generates and passes an output value (typically nonlinear) to the next layer.

$$\sigma(Z)$$

(From: https://developers.google.com/machine-learning/glossary)

MLP from scratch with TF

Imports


```python
"""
Created on Thu Jan 23 2020

@author: brunofmf
"""

import logging
import tensorflow as tf
from tensorflow.keras.layers import Layer
from tensorflow.keras import Model
import matplotlib.pyplot as plt
import numpy as np

logging.getLogger('tensorflow').setLevel(logging.ERROR)
#for replicability purposes
tf.random.set_seed(91195003)
#for an easy reset backend session state
tf.keras.backend.clear_session()
```

MLP from scratch with TF

A Perceptron Layer


```
# ============================================================================
#
#         PerceptronLayer Class
#
# ============================================================================
class PerceptronLayer(Layer):

    '''
    The constructor in an object oriented perspective.
    Called when an object is created, allowing the class to initialize the attributes of a class.
    neurons corresponds to the number of neurons in this perceptron layer
    '''
    def __init__(self, neurons=16, **kwargs):
        super(PerceptronLayer, self).__init__(**kwargs)
        self.neurons = neurons

    '''
    We use the build function to deferr weight creation until the shape of the inputs is known
    '''
    def build(self, input_shape):
        pass

    '''
    Implements the function call operator (when an instance is used as a function).
    It will automatically run build the first time it is called, i.e., layer's weights are created dynamically
    '''
    def call(self, inputs):
        pass

    '''
    Enable serialization on our perceptron layer
    '''
    def get_config(self):
        pass
```

MLP from scratch with TF

A Perceptron Layer


```python
'''
We use the build function to deferr weight creation until the shape of the inputs is known
'''
def build(self, input_shape):
    #TODO: set the correct shape for the weights and the bias
    weights_init = tf.random_normal_initializer()
    self.w = tf.Variable(initial_value=weights_init(shape=(????, ????), dtype='float32'), trainable=True)
    bias_init = tf.zeros_initializer()
    self.b = tf.Variable(initial_value=bias_init(shape=(????,), dtype='float32'), trainable=True)
    #you will need to assert a fact, i.e., that weights and bias are to be automatically tracked by our layer
    #for example: assert perceptron.weights == [perceptron.w, perceptron.b]
```

```python
'''
We use the build function to deferr weight creation until the shape of the inputs is known
'''
def build(self, input_shape):
    #---
    #weights_init = tf.random_normal_initializer()
    #self.w = tf.Variable(initial_value=weights_init(shape=(????, ????), dtype='float32'), trainable=True)
    #bias_init = tf.zeros_initializer()
    #self.b = tf.Variable(initial_value=bias_init(shape=(????,), dtype='float32'), trainable=True)
    #you will need to assert a fact, i.e., that weights and bias are to be automatically tracked by our layer
    #for example: assert perceptron.weights == [perceptron.w, perceptron.b]
    #---
    #however, we can use a shortcut for adding weights to a layer
    #TODO: set the correct shape for the weights and the bias
    self.w = self.add_weight(shape=(????, ????), initializer='random_normal', trainable=True)
    self.b = self.add_weight(shape=(????,), initializer='random_normal', trainable=True)
```

MLP from scratch with TF

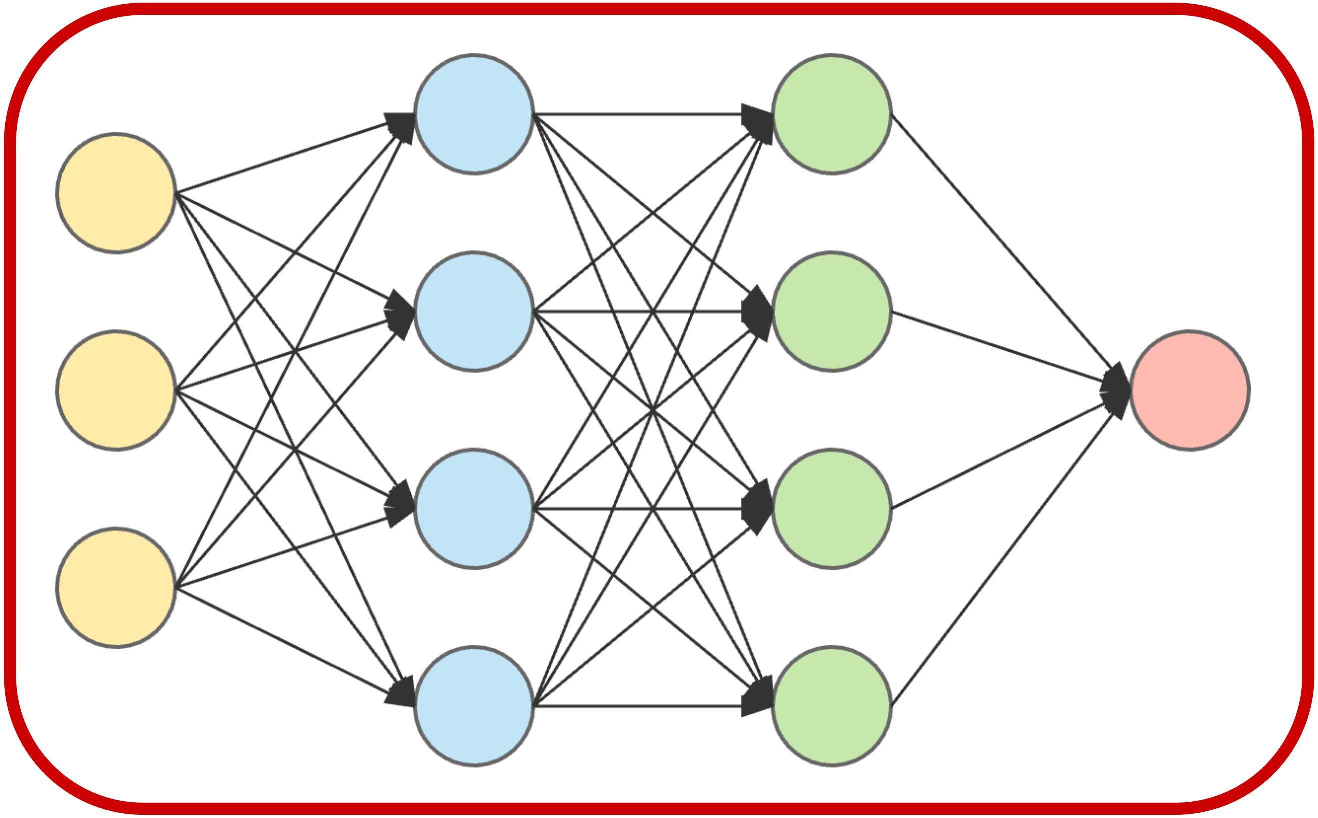
A Perceptron Layer


```
'''
Implements the function call operator (when an instance is used as a function).
It will automatically run build the first time it is called, i.e., layer's weights are created dynamically
'''
def call(self, inputs):
    #TODO: return the perceptron result
    pass

'''
Enable serialization on our perceptron layer
'''
def get_config(self):
    config = super(PerceptronLayer, self).get_config()
    config.update({'neurons': self.neurons})
    return config
```

$$Z = w \cdot x + b$$

MLP from scratch with TF

A Multilayer Perceptron (MLP)



MLP from scratch with TF

A Multilayer Perceptron (MLP)


```python
# ==============================================================================
#
#        Multilayer Perceptron Class
#
# ==============================================================================
class MultilayerPerceptron(Model):

    '''
    The Layers of our MLP (with a fixed number of neurons)
    '''
    def __init__(self, output_neurons=10, name='multilayerPerceptron', **kwargs):
        super(MultilayerPerceptron, self).__init__(name=name, **kwargs)
        self.perceptron_layer_1 = PerceptronLayer(16)
        self.perceptron_layer_2 = PerceptronLayer(32)
        self.perceptron_layer_3 = PerceptronLayer(output_neurons)

    '''
    Layers are recursively composable, i.e.,
    if you assign a Layer instance as attribute of another Layer, the outer layer will start tracking the weights of the inner layer.
    Remember that the build of each layer is called automatically (thus creating the weights).
    '''
    def feed_model(self, input_data):
        pass

    """
    Compute softmax values for the logits
    """
    def softmax(self, logits):
        pass

    def print_trainable_weights(self):
        pass

    def call(self, input_data):
        pass
```

MLP from scratch with TF

Layers Interaction


```python
'''
Layers are recursively composable, i.e.,
if you assign a Layer instance as attribute of another Layer, the outer layer will start tracking the weights of the inner layer.
Remember that the build of each layer is called automatically (thus creating the weights).
'''
def feed_model(self, input_data):
    x = self.perceptron_layer_1(input_data)
    #activation function applied to the output of the perceptron layer
    x = tf.nn.relu(x)
    #the output, now normalized, is fed as input to the second perceptron layer
    x = self.perceptron_layer_2(x)
    #again, activation function applied to the output of the second perceptron layer
    x = tf.nn.relu(x)
    #which, again, is fed as input to the third layer, which returns its output
    logits = self.perceptron_layer_3(x)
    #the output of the last layer going over a softmax activation
    #so, we will not be outputting logits but "probabilities"
    return self.softmax(logits) #equivalent of tf.nn.softmax(logits)
```

```python
"""
Compute softmax values for the logits
"""
def softmax(self, logits):
    #TODO: implement softmax
    pass
```

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

MLP from scratch with TF

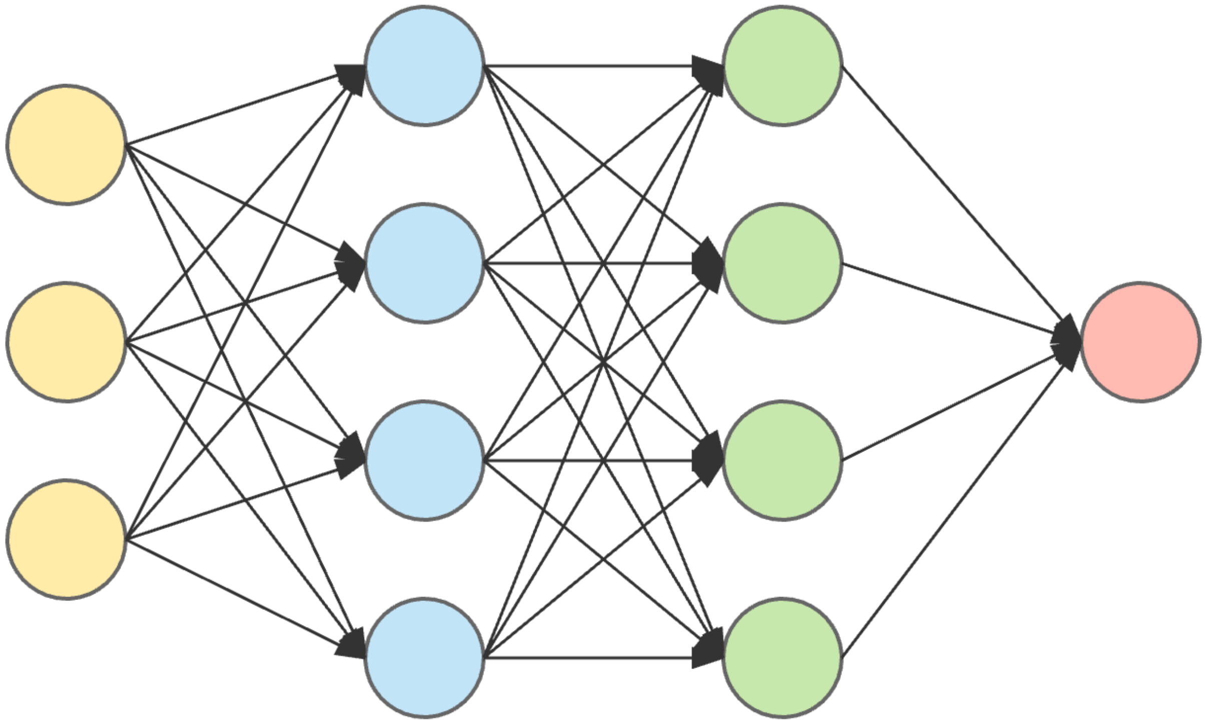
Calling the model


```python
    '''
    Layers are recursively composable, i.e.,
    if you assign a Layer instance as attribute of another Layer, the outer layer will start tracking the weights of the inner layer.
    Remember that the build of each layer is called automatically (thus creating the weights).
    '''
    def feed_model(self, input_data):
        x = self.perceptron_layer_1(input_data)
        #activation function applied to the output of the perceptron layer
        x = tf.nn.relu(x)
        #the output, now normalized, is fed as input to the second perceptron layer
        x = self.perceptron_layer_2(x)
        #again, activation function applied to the output of the second perceptron layer
        x = tf.nn.relu(x)
        #which, again, is fed as input to the third layer, which returns its output
        logits = self.perceptron_layer_3(x)
        #the output of the last layer going over a softmax activation
        #so, we will not be outputting logits but "probabilities"
        return self.softmax(logits) #equivalent of tf.nn.softmax(logits)
```

…

```python
    def print_trainable_weights(self):
        print('Weights:', len(self.weights))
        print('Trainable weights:', len(self.trainable_weights))
        print('Non-trainable weights:', len(self.non_trainable_weights))

    def call(self, input_data):
        #TODO: here, we want to feed the model and receive its output (i.e, the output of the last layer)
        probs = ????
        return probs
```

MLP from scratch with TF
Using it



Using our model...

MLP from scratch with TF

The structure


```
# ==============================================================================
#
#        Main Execution
#
# ==============================================================================

'''
Importing data
'''
def import_data():
    pass

'''
Preparing the model, the optimizers, the loss function and some metrics
'''
def prepare_model():
    pass

'''
Define a low level fit and predict making use of the tape.gradient
'''
def low_level_fit_and_predict():
   pass

'''
Define a low level fit and predict
making use of the tape.gradient
'''
def high_level_fit_and_predict():
    pass

'''
Run
'''
#hyperparameters
epochs = 5
batch_size = 32
learning_rate = 1e-3
output_neurons = 10

#load data
train_dataset, validation_dataset, x_test, y_test = import_data()
#init our model
mlp, optimizer, loss_object, train_metric, val_metric = prepare_model()
#use low-level or high-level fit and predict
#low_level_fit_and_predict()
high_level_fit_and_predict()
```

MLP from scratch with TF

Importing data


```
'''
Importing data
'''
def import_data():
    #load mnist training and test data
    (x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
```

MLP from scratch with TF

Importing data


```python
'''
Importing data
'''
def import_data():
    #load mnist training and test data
    (x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

    #some data exploration
    print('***** Log import_data *****')
    print('Train data shape', x_train.shape)
    print('Test data shape', x_test.shape)
    print('Number of training samples', x_train.shape[0])
    print('Number of testing samples', x_test.shape[0])
    for i in range(25):
        plt.subplot(5,5,i+1)    #Add a subplot as 5 x 5
        plt.xticks([])          #get rid of labels
        plt.yticks([])          #get rid of labels
        plt.imshow(x_test[i], cmap="gray")
    plt.show()
    print('***** Log import_data *****')
```

MLP from scratch with TF

Importing data


```python
'''
Importing data
'''
def import_data():
    #load mnist training and test data
    (x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

    #some data exploration
    print('***** Log import_data *****')
    print('Train data shape', x_train.shape)
    print('Test data shape', x_test.shape)
    print('Number of training samples', x_train.shape[0])
    print('Number of testing samples', x_test.shape[0])
    for i in range(25):
        plt.subplot(5,5,i+1)    #Add a subplot as 5 x 5
        plt.xticks([])          #get rid of labels
        plt.yticks([])          #get rid of labels
        plt.imshow(x_test[i], cmap="gray")
    plt.show()
    print('***** Log import_data *****')

    #reshape the input to have a list of self.batch_size by 28*28 = 784; and normalize (/255)
    x_train = x_train.reshape(x_train.shape[0], x_train.shape[1]*x_train.shape[2]).astype('float32')/255
    x_test = x_test.reshape(x_test.shape[0], x_test.shape[1]*x_test.shape[2]).astype('float32')/255
    #reserve 5000 samples for validation
    x_validation = x_train[-5000:]
    y_validation = y_train[-5000:]
    #do not use those same 5000 samples for training!
    x_train = x_train[:-5000]
    y_train = y_train[:-5000]
```

MLP from scratch with TF

Importing data


```
'''
Importing data
'''
def import_data():
    #load mnist training and test data
    (x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

    #some data exploration
    print('***** Log import_data *****')
    print('Train data shape', x_train.shape)
    print('Test data shape', x_test.shape)
    print('Number of training samples', x_train.shape[0])
    print('Number of testing samples', x_test.shape[0])
    for i in range(25):
        plt.subplot(5,5,i+1)    #Add a subplot as 5 x 5
        plt.xticks([])          #get rid of labels
        plt.yticks([])          #get rid of labels
        plt.imshow(x_test[i], cmap="gray")
    plt.show()
    print('***** Log import_data *****')

    #reshape the input to have a list of self.batch_size by 28*28 = 784; and normalize (/255)
    x_train = x_train.reshape(x_train.shape[0], x_train.shape[1]*x_train.shape[2]).astype('float32')/255
    x_test = x_test.reshape(x_test.shape[0], x_test.shape[1]*x_test.shape[2]).astype('float32')/255
    #reserve 5000 samples for validation
    x_validation = x_train[-5000:]
    y_validation = y_train[-5000:]
    #do not use those same 5000 samples for training!
    x_train = x_train[:-5000]
    y_train = y_train[:-5000]

    #create dataset iterator for training
    train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
    #shuffle in intervals of 1024 and slice in batchs of batch_size
    train_dataset = train_dataset.shuffle(buffer_size=1024).batch(batch_size)
    #create the validation dataset
    validation_dataset = tf.data.Dataset.from_tensor_slices((x_validation, y_validation))
    validation_dataset = validation_dataset.batch(batch_size)
    return train_dataset, validation_dataset, x_test, y_test
```

MLP from scratch with TF
Setting the model


```
'''
Preparing the model, the optimizers, the loss function and some metrics
'''
def prepare_model():
    mlp = MultilayerPerceptron(output_neurons=output_neurons)
```

MLP from scratch with TF
Setting the model


```
'''
Preparing the model, the optimizers, the loss function and some metrics
'''
def prepare_model():
    mlp = MultilayerPerceptron(output_neurons=output_neurons)
    #instantiate an optimizer
    optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
```

MLP from scratch with TF

Setting the model


```
'''
Preparing the model, the optimizers, the loss function and some metrics
'''
def prepare_model():
    mlp = MultilayerPerceptron(output_neurons=output_neurons)
    #instantiate an optimizer
    optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
    #instantiate a loss object (from_logits=False as we are applying a softmax activation over the last layer)
    loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False)
```

MLP from scratch with TF

Setting the model


```
'''
Preparing the model, the optimizers, the loss function and some metrics
'''
def prepare_model():
    mlp = MultilayerPerceptron(output_neurons=output_neurons)
    #instantiate an optimizer
    optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
    #instantiate a loss object (from_logits=False as we are applying a softmax activation over the last layer)
    loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False)
    #using a metric too
    train_metric = tf.keras.metrics.SparseCategoricalAccuracy()
    val_metric = tf.keras.metrics.SparseCategoricalAccuracy()
    return mlp, optimizer, loss_object, train_metric, val_metric
```

MLP from scratch with TF

Controlling the fit and predict


```
'''
Define a low level fit and predict making use of the tape.gradient
'''
def low_level_fit_and_predict():
```

MLP from scratch with TF

Controlling the fit and predict


```
'''
Define a low level fit and predict making use of the tape.gradient
'''
def low_level_fit_and_predict():
    #manually, let's iterate over the epochs and fit ourselves
    for epoch in range(epochs):
```

MLP from scratch with TF

Controlling the fit and predict


```python
'''
Define a low level fit and predict making use of the tape.gradient
'''
def low_level_fit_and_predict():
    #manually, let's iterate over the epochs and fit ourselves
    for epoch in range(epochs):
        print('Epoch %d/%d' %(epoch+1, epochs))

        #to store loss values
        loss_history = []

        #iterate over all batchs
        for step, (x_batch, y_batch) in enumerate(train_dataset):
```

MLP from scratch with TF

Controlling the fit and predict


```
'''
Define a low level fit and predict making use of the tape.gradient
'''
def low_level_fit_and_predict():
    #manually, let's iterate over the epochs and fit ourselves
    for epoch in range(epochs):
        print('Epoch %d/%d' %(epoch+1, epochs))

        #to store loss values
        loss_history = []

        #iterate over all batchs
        for step, (x_batch, y_batch) in enumerate(train_dataset):
            #use a gradien tape to save computations to calculate gradient later
            with tf.GradientTape() as tape:
```

MLP from scratch with TF

Controlling the fit and predict


```
'''
Define a low level fit and predict making use of the tape.gradient
'''
def low_level_fit_and_predict():
    #manually, let's iterate over the epochs and fit ourselves
    for epoch in range(epochs):
        print('Epoch %d/%d' %(epoch+1, epochs))

        #to store loss values
        loss_history = []

        #iterate over all batchs
        for step, (x_batch, y_batch) in enumerate(train_dataset):
            #use a gradien tape to save computations to calculate gradient later
            with tf.GradientTape() as tape:
                #running the forward pass of all layers
                #operations being recorded into the tape
                probs = mlp(x_batch)
```

MLP from scratch with TF

Controlling the fit and predict


```
'''
Define a low level fit and predict making use of the tape.gradient
'''
def low_level_fit_and_predict():
    #manually, let's iterate over the epochs and fit ourselves
    for epoch in range(epochs):
        print('Epoch %d/%d' %(epoch+1, epochs))

        #to store loss values
        loss_history = []

        #iterate over all batchs
        for step, (x_batch, y_batch) in enumerate(train_dataset):
            #use a gradien tape to save computations to calculate gradient later
            with tf.GradientTape() as tape:
                #running the forward pass of all layers
                #operations being recorded into the tape
                probs = mlp(x_batch)
                #computing the loss for this batch
                #how far are we from the correct labels?
                loss_value = loss_object(y_batch, probs)
```

MLP from scratch with TF

Controlling the fit and predict


```
'''
Define a low level fit and predict making use of the tape.gradient
'''
def low_level_fit_and_predict():
    #manually, let's iterate over the epochs and fit ourselves
    for epoch in range(epochs):
        print('Epoch %d/%d' %(epoch+1, epochs))

        #to store loss values
        loss_history = []

        #iterate over all batchs
        for step, (x_batch, y_batch) in enumerate(train_dataset):
            #use a gradien tape to save computations to calculate gradient later
            with tf.GradientTape() as tape:
                #running the forward pass of all layers
                #operations being recorded into the tape
                probs = mlp(x_batch)
                #computing the loss for this batch
                #how far are we from the correct labels?
                loss_value = loss_object(y_batch, probs)

            #store loss value
            loss_history.append(loss_value.numpy().mean())
            #use the tape to automatically retrieve the gradients of the trainable variables
            #with respect to the loss
            gradients = tape.gradient(loss_value, mlp.trainable_weights)
```

MLP from scratch with TF

Controlling the fit and predict

TensorFlow


```
'''
Define a low level fit and predict making use of the tape.gradient
'''
def low_level_fit_and_predict():
    #manually, let's iterate over the epochs and fit ourselves
    for epoch in range(epochs):
        print('Epoch %d/%d' %(epoch+1, epochs))

        #to store loss values
        loss_history = []

        #iterate over all batchs
        for step, (x_batch, y_batch) in enumerate(train_dataset):
            #use a gradien tape to save computations to calculate gradient later
            with tf.GradientTape() as tape:
                #running the forward pass of all layers
                #operations being recorded into the tape
                probs = mlp(x_batch)
                #computing the loss for this batch
                #how far are we from the correct labels?
                loss_value = loss_object(y_batch, probs)

            #store loss value
            loss_history.append(loss_value.numpy().mean())
            #use the tape to automatically retrieve the gradients of the trainable variables
            #with respect to the loss
            gradients = tape.gradient(loss_value, mlp.trainable_weights)
            #running one step of gradient descent by updating (going backwards now)
            #the value of the trainable variables to minimize the loss
            optimizer.apply_gradients(zip(gradients, mlp.trainable_weights))
```

MLP from scratch with TF

Controlling the fit and predict


```python
'''
Define a low level fit and predict making use of the tape.gradient
'''
def low_level_fit_and_predict():
    #manually, let's iterate over the epochs and fit ourselves
    for epoch in range(epochs):
        print('Epoch %d/%d' %(epoch+1, epochs))

        #to store loss values
        loss_history = []

        #iterate over all batchs
        for step, (x_batch, y_batch) in enumerate(train_dataset):
            #use a gradien tape to save computations to calculate gradient later
            with tf.GradientTape() as tape:
                #running the forward pass of all layers
                #operations being recorded into the tape
                probs = mlp(x_batch)
                #computing the loss for this batch
                #how far are we from the correct labels?
                loss_value = loss_object(y_batch, probs)

            #store loss value
            loss_history.append(loss_value.numpy().mean())
            #use the tape to automatically retrieve the gradients of the trainable variables
            #with respect to the loss
            gradients = tape.gradient(loss_value, mlp.trainable_weights)
            #running one step of gradient descent by updating (going backwards now)
            #the value of the trainable variables to minimize the loss
            optimizer.apply_gradients(zip(gradients, mlp.trainable_weights))
            # Update training metric.
            train_metric(y_batch, probs)

            #log every n batches
            if step%200 == 0:
                print('Step %s. Loss Value = %s; Mean loss = %s' %(step, str(loss_value.numpy()), np.mean(loss_history)))
```

MLP from scratch with TF

Controlling the fit and predict


```
'''
Define a low level fit and predict making use of the tape.gradient
'''
def low_level_fit_and_predict():
    #manually, let's iterate over the epochs and fit ourselves
    for epoch in range(epochs):
        print('Epoch %d/%d' %(epoch+1, epochs))

        #to store loss values
        loss_history = []

        #iterate over all batchs
        for step, (x_batch, y_batch) in enumerate(train_dataset):
            #use a gradien tape to save computations to calculate gradient later
            with tf.GradientTape() as tape:
                #running the forward pass of all layers
                #operations being recorded into the tape
                probs = mlp(x_batch)
                #computing the loss for this batch
                #how far are we from the correct labels?
                loss_value = loss_object(y_batch, probs)

            #store loss value
            loss_history.append(loss_value.numpy().mean())
            #use the tape to automatically retrieve the gradients of the trainable variables
            #with respect to the loss
            gradients = tape.gradient(loss_value, mlp.trainable_weights)
            #running one step of gradient descent by updating (going backwards now)
            #the value of the trainable variables to minimize the loss
            optimizer.apply_gradients(zip(gradients, mlp.trainable_weights))
            # Update training metric.
            train_metric(y_batch, probs)

            #log every n batches
            if step%200 == 0:
                print('Step %s. Loss Value = %s; Mean Loss = %s' %(step, str(loss_value.numpy()), np.mean(loss_history)))

        #display metrics at the end of each epoch
        train_accuracy = train_metric.result()
        print('Training accuracy for epoch %d: %s' %(epoch+1, float(train_accuracy)))
        #reset training metrics (at the end of each epoch)
        train_metric.reset_states()
```

MLP from scratch with TF

Controlling the fit and predict


```
'''
Define a low level fit and predict making use of the tape.gradient
'''
def low_level_fit_and_predict():
    .............
            gradients = tape.gradient(loss_value, mlp.trainable_weights)
            #running one step of gradient descent by updating (going backwards now)
            #the value of the trainable variables to minimize the loss
            optimizer.apply_gradients(zip(gradients, mlp.trainable_weights))
            # Update training metric.
            train_metric(y_batch, probs)

            #log every n batches
            if step%200 == 0:
                print('Step %s. Loss Value = %s; Mean Loss = %s' %(step, str(loss_value.numpy()), np.mean(loss_history)))

        #display metrics at the end of each epoch
        train_accuracy = train_metric.result()
        print('Training accuracy for epoch %d: %s' %(epoch+1, float(train_accuracy)))
        #reset training metrics (at the end of each epoch)
        train_metric.reset_states()
```

MLP from scratch with TF
Controlling the fit and predict


```
'''
Define a low level fit and predict making use of the tape.gradient
'''
def low_level_fit_and_predict():
    ............
            gradients = tape.gradient(loss_value, mlp.trainable_weights)
            #running one step of gradient descent by updating (going backwards now)
            #the value of the trainable variables to minimize the loss
            optimizer.apply_gradients(zip(gradients, mlp.trainable_weights))
            # Update training metric.
            train_metric(y_batch, probs)

            #log every n batches
            if step%200 == 0:
                print('Step %s. Loss Value = %s; Mean loss = %s' %(step, str(loss_value.numpy()), np.mean(loss_history)))

        #display metrics at the end of each epoch
        train_accuracy = train_metric.result()
        print('Training accuracy for epoch %d: %s' %(epoch+1, float(train_accuracy)))
        #reset training metrics (at the end of each epoch)
        train_metric.reset_states()

        #run a validation loop at the end of each epoch
        for x_batch_val, y_batch_val in validation_dataset:
            val_probs = mlp(x_batch_val)
            #update val metrics
            val_metric(y_batch_val, val_probs)
        val_acc = val_metric.result()
        val_metric.reset_states()
        print('Validation accuracy for epoch %d: %s' % (epoch+1, float(val_acc)))
```

MLP from scratch with TF

Controlling the fit and predict


```
'''
Define a low level fit and predict making use of the tape.gradient
'''
def low_level_fit_and_predict():
    ............
            gradients = tape.gradient(loss_value, mlp.trainable_weights)
            #running one step of gradient descent by updating (going backwards now)
            #the value of the trainable variables to minimize the loss
            optimizer.apply_gradients(zip(gradients, mlp.trainable_weights))
            # Update training metric.
            train_metric(y_batch, probs)

            #log every n batches
            if step%200 == 0:
                print('Step %s. Loss Value = %s; Mean loss = %s' %(step, str(loss_value.numpy()), np.mean(loss_history)))

        #display metrics at the end of each epoch
        train_accuracy = train_metric.result()
        print('Training accuracy for epoch %d: %s' %(epoch+1, float(train_accuracy)))
        #reset training metrics (at the end of each epoch)
        train_metric.reset_states()

        #run a validation loop at the end of each epoch
        for x_batch_val, y_batch_val in validation_dataset:
            val_probs = mlp(x_batch_val)
            #update val metrics
            val_metric(y_batch_val, val_probs)
        val_acc = val_metric.result()
        val_metric.reset_states()
        print('Validation accuracy for epoch %d: %s' % (epoch+1, float(val_acc)))

    #now predict
    print('\nGenerating predictions for ten samples...')
    predictions = mlp(x_test[:10])
    print('Predictions shape:', predictions.shape)

    for i, prediction in enumerate(predictions):
        #tf.argmax returns the INDEX with the largest value across axes of a tensor
        predicted_value = tf.argmax(prediction)
        label = y_test[i]
        print('Predicted a %d. Real value is %d.' %(predicted_value, label))
```

MLP from scratch with TF

High-level fit and predict


```
'''
Define a high level fit and predict making use tf.Keras APIs
'''
def high_level_fit_and_predict():
```

MLP from scratch with TF
High-level fit and predict


```
'''
Define a high level fit and predict making use tf.Keras APIs
'''
def high_level_fit_and_predict():
    #shortcut to compile and fit a model!
    #able to do this because our model subclasses tf.keras.Model
    mlp.compile(optimizer, loss=loss_object, metrics=[train_metric])
```

MLP from scratch with TF

High-level fit and predict


```
'''
Define a high level fit and predict making use tf.Keras APIs
'''
def high_level_fit_and_predict():
    #shortcut to compile and fit a model!
    #able to do this because our model subclasses tf.keras.Model
    mlp.compile(optimizer, loss=loss_object, metrics=[train_metric])
    #since the train_dataset already takes care of batching, we don't pass a batch_size argument
    #passing validation data for monitoring validation loss and metrics at the end of each epoch
    history = mlp.fit(train_dataset, validation_data=validation_dataset, epochs=epochs)
    #print('\nHistory values per epoch:', history.history)
```

MLP from scratch with TF

High-level fit and predict


```
'''
Define a high level fit and predict making use tf.Keras APIs
'''
def high_level_fit_and_predict():
    #shortcut to compile and fit a model!
    #able to do this because our model subclasses tf.keras.Model
    mlp.compile(optimizer, loss=loss_object, metrics=[train_metric])
    #since the train_dataset already takes care of batching, we don't pass a batch_size argument
    #passing validation data for monitoring validation loss and metrics at the end of each epoch
    history = mlp.fit(train_dataset, validation_data=validation_dataset, epochs=epochs)
    #print('\nHistory values per epoch:', history.history)

    #evaluating the model on the test data
    print('\nEvaluating model on test data...')
    scores = mlp.evaluate(x_test, y_test, batch_size=batch_size, verbose=0)
    print('Evaluation %s: %s' %(mlp.metrics_names, str(scores)))
```

MLP from scratch with TF
High-level fit and predict


```python
'''
Define a high level fit and predict making use tf.Keras APIs
'''
def high_level_fit_and_predict():
    #shortcut to compile and fit a model!
    #able to do this because our model subclasses tf.keras.Model
    mlp.compile(optimizer, loss=loss_object, metrics=[train_metric])
    #since the train_dataset already takes care of batching, we don't pass a batch_size argument
    #passing validation data for monitoring validation loss and metrics at the end of each epoch
    history = mlp.fit(train_dataset, validation_data=validation_dataset, epochs=epochs)
    #print('\nHistory values per epoch:', history.history)

    #evaluating the model on the test data
    print('\nEvaluating model on test data...')
    scores = mlp.evaluate(x_test, y_test, batch_size=batch_size, verbose=0)
    print('Evaluation %s: %s' %(mlp.metrics_names, str(scores)))

    #finally, generating predictions (the output of the last layer)
    print('\nGenerating predictions for ten samples...')
    predictions = mlp.predict(x_test[:10])
    #now, for each prediction in predictions, get the value with higher "probability"
    #look at the shape, it is as (3, 10). For each prediction, we have the prob of beeing 0, beeing 1, etc...
    #we now choose the index of the list with higher "probability"
    #if pos=3 is the one with higher probability it means it predicts a 3
    print('Predictions shape:', predictions.shape)
    for i, prediction in enumerate(predictions):
        #tf.argmax returns the INDEX with the largest value across axes of a tensor
        predicted_value = tf.argmax(prediction)
        label = y_test[i]
        print('Predicted a %d. Real value is %d.' %(predicted_value, label))
```

MLP from scratch with TF

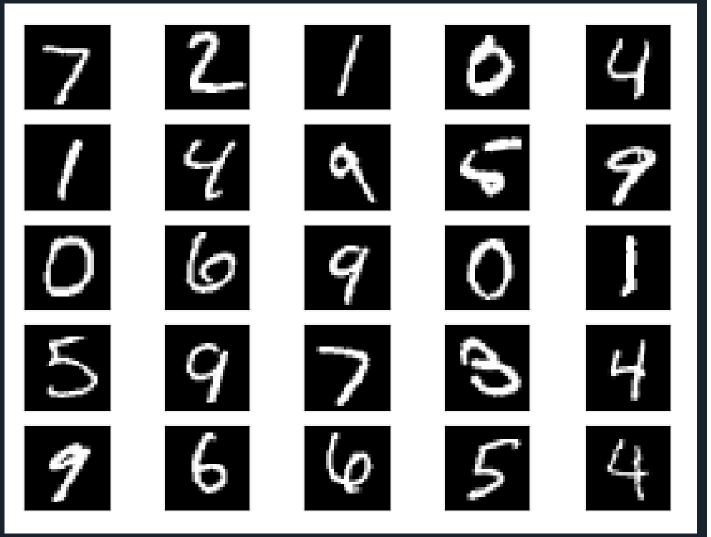
Running it: Low-level results


```
'''
Run
'''
#hyperparameters
epochs = 1
batch_size = 32
learning_rate = 1e-3
output_neurons = 10

#load data
train_dataset, validation_dataset, x_test, y_test = import_data()
#init our model
mlp, optimizer, loss_object, train_metric, val_metric = prepare_model()
#use low-level or high-level fit and predict
low_level_fit_and_predict()
#high_level_fit_and_predict()
```

MLP from scratch with TF

Running it: Low-level results


```
***** Log import_data *****
Train data shape (60000, 28, 28)
Test data shape (10000, 28, 28)
Number of training samples 60000
Number of testing samples 10000
***** Log import_data *****
Epoch 1/1
Step 0. Loss Value = 2.303298; Mean loss = 2.303298
Step 200. Loss Value = 0.8385514; Mean loss = 1.3295121
Step 400. Loss Value = 0.62667996; Mean loss = 0.98740995
Step 600. Loss Value = 0.3768852; Mean loss = 0.8498528
Step 800. Loss Value = 0.7029902; Mean loss = 0.7556675
Step 1000. Loss Value = 0.32704055; Mean loss = 0.70003974
Step 1200. Loss Value = 0.5054481; Mean loss = 0.6572727
Step 1400. Loss Value = 0.48230478; Mean loss = 0.6243781
Step 1600. Loss Value = 0.42824277; Mean loss = 0.5991417
Training accuracy for epoch 1: 0.8305454254150391
Validation accuracy for epoch 1: 0.9196000099182129

Generating predictions for ten samples...
Predictions shape: (10, 10)
Predicted a 7. Real value is 7.
Predicted a 2. Real value is 2.
Predicted a 1. Real value is 1.
Predicted a 0. Real value is 0.
Predicted a 4. Real value is 4.
Predicted a 1. Real value is 1.
Predicted a 5. Real value is 4.
Predicted a 3. Real value is 9.
Predicted a 6. Real value is 5.
Predicted a 9. Real value is 9.
```



MLP from scratch with TF

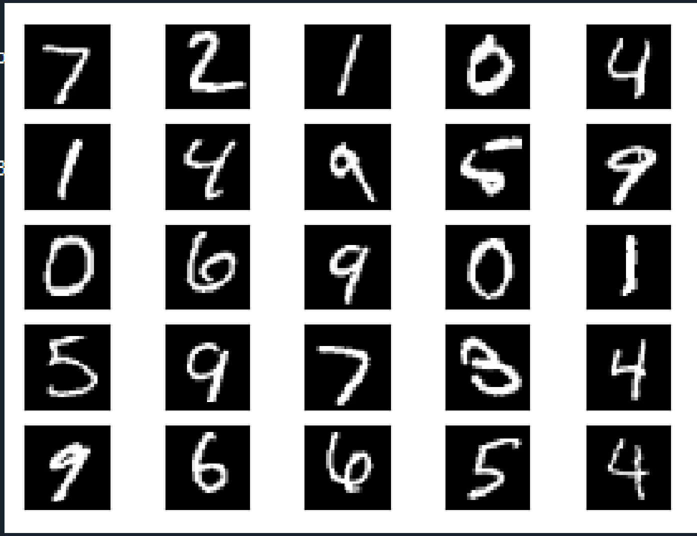
Running it: High-level results


```
'''
Run
'''
#hyperparameters
epochs = 1
batch_size = 32
learning_rate = 1e-3
output_neurons = 10

#load data
train_dataset, validation_dataset, x_test, y_test = import_data()
#init our model
mlp, optimizer, loss_object, train_metric, val_metric = prepare_model()
#use low-level or high-level fit and predict
#low_level_fit_and_predict()
high_level_fit_and_predict()
```

MLP from scratch with TF

Running it: High-level results


```
***** Log import_data *****
Train data shape (60000, 28, 28)
Test data shape (10000, 28, 28)
Number of training samples 60000
Number of testing samples 10000
***** Log import_data *****
Epoch 1/5
1719/1719 [==============================] - 8s 5ms/step - loss: 0.5888 - sparse_categorical_accuracy: 0.8279 -
val_loss: 0.0000e+00 - val_sparse_categorical_accuracy: 0.0000e+00
Epoch 2/5
1719/1719 [==============================] - 4s 3ms/step - loss: 0.3468 - sparse_categorical_accuracy: 0.8957 -
val_loss: 0.2417 - val_sparse_categorical_accuracy: 0.9306
Epoch 3/5
1719/1719 [==============================] - 4s 2ms/step - loss: 0.2881 - sparse_categorical_accuracy: 0.9115 -
val_loss: 0.2155 - val_sparse_categorical_accuracy: 0.9358
Epoch 4/5
1719/1719 [==============================] - 4s 2ms/step - loss: 0.2541 - sparse_categorical_accuracy: 0.9210 -
val_loss: 0.1922 - val_sparse_categorical_accuracy: 0.9444
Epoch 5/5
1719/1719 [==============================] - 4s 3ms/step - loss: 0.2327 - sp
val_loss: 0.1828 - val_sparse_categorical_accuracy: 0.9442

Evaluating model on test data...
Evaluation ['loss', 'sparse_categorical_accuracy']: [0.214388842856884, 0.93

Generating predictions for ten samples...
Predictions shape: (10, 10)
Predicted a 7. Real value is 7.
Predicted a 2. Real value is 2.
Predicted a 1. Real value is 1.
Predicted a 0. Real value is 0.
Predicted a 4. Real value is 4.
Predicted a 1. Real value is 1.
Predicted a 4. Real value is 4.
Predicted a 3. Real value is 9.
Predicted a 6. Real value is 5.
Predicted a 9. Real value is 9.
```



A Summary!

So, what did we see?

1. Weights and bias (and how they related between them)
2. Activation functions
   - In particular, relu and softmax
3. Logits and probs
4. Epochs and Batch processing
5. Loss, Gradients and the Gradient Tape
   - Going forward and backwards
6. Optimizers and Metrics

Summary
Weights and Bias

Weights

- A coefficient for a feature in a linear model, or an edge in a deep network. The goal of training a linear model is to determine the ideal weight for each feature. If a weight is 0, then its corresponding feature does not contribute to the model.
- Weights are the coefficients of the equation which we are trying to solve!

Bias

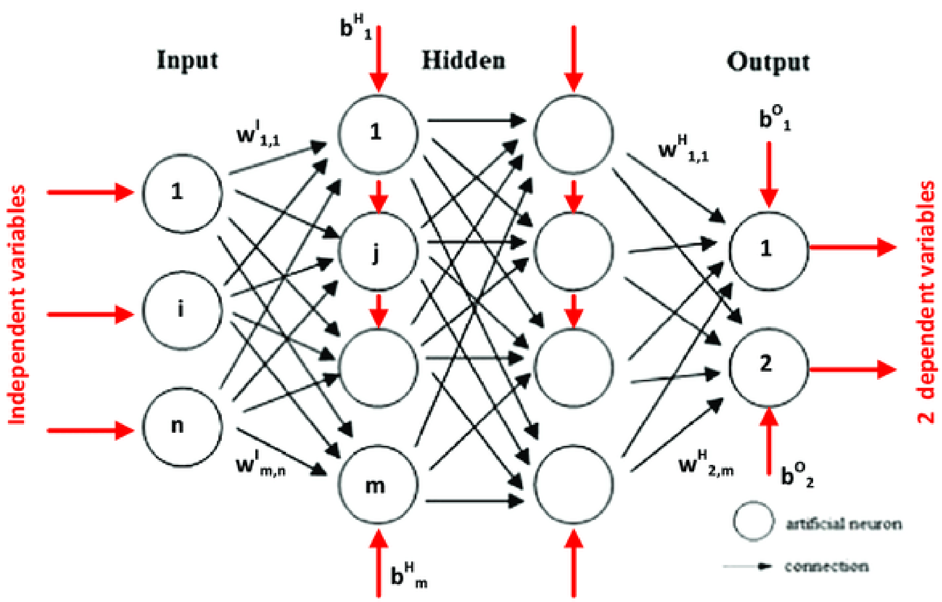
- An intercept or offset from an origin. Bias is referred to as b in machine learning models and used to offset the result. It helps the model in a way that it can fit better for the given data

If we have, as inputs: $x_1, x_2, \ldots, x_n$

And, as weights: $w_1, w_2, \ldots, w_n$

The weighted sum is: $x_1 w_1 + x_2 w_2 + \cdots + x_n w_n + b$

Summary
Weights and Bias



If we have, as inputs: $x_1, x_2, \ldots, x_n$

And, as weights: $w_1, w_2, \ldots, w_n$

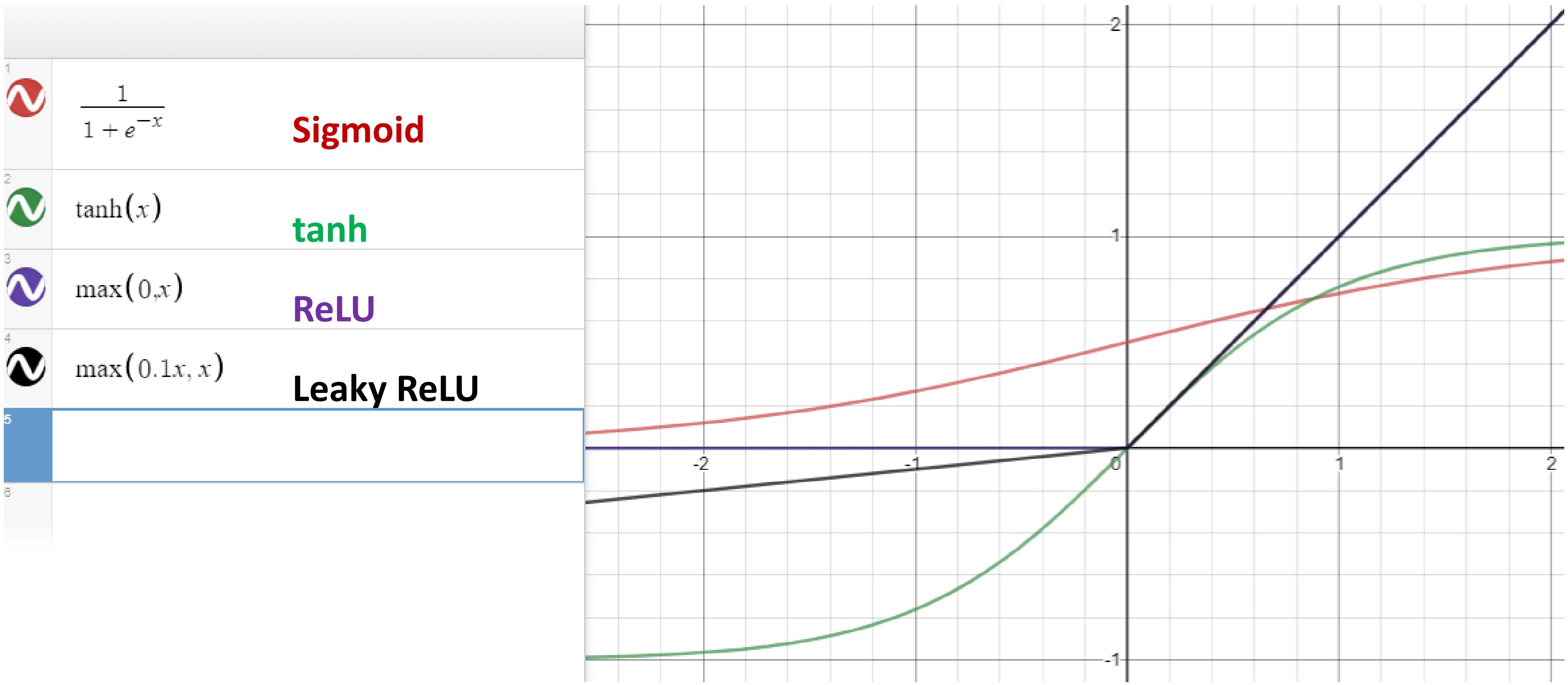
The weighted sum is: $x_1w_1 + x_2w_2 + \cdots + x_nw_n + b$

Summary
Activation Functions

- A function that takes in the weighted sum of all of the inputs from the previous layer and then generates a non-linear transformation over such inputs to pass as output value to the next layer
- The non-linear transformation to the input makes the network capable of learning more complex patterns, which is essential for learning and modeling complex data, such as video, images, sequences or audio, just to name a few
- Indeed, a linear equation would be simple to solve but is limited in its capacity to solve complex problems. On the one hand, with a linear activation function it would not be possible to use backpropagation because the derivative of the function is a constant and, on the other hand, all layers of the neural network would collapse into one (the last layer would be a linear function of the first layer - a linear combination of linear functions remains a linear function)

$$activation_{function}(x_1 w_1 + x_2 w_2 + \cdots + x_n w_n + b)$$

Summary
Activation Functions



$$activation_{function}(x_1w_1 + x_2w_2 + \cdots + x_nw_n + b)$$

Summary
Logits and Probs (DL terms!)

Logits

- In deep learning, logits is popularly used to describe the non-normalized output (can go from [-∞, +∞]) of the last layer when solving a multi-class classification problem. The logits typically become an input to the softmax function. The softmax function then generates a vector of (normalized) probabilities with one value for each class.
- In other words, a vector of raw (non-normalized) predictions that a classification model generates, which is then passed to a normalization function

Probs

- A vector of (normalized) probabilities with one value for each existing class. In other words, the output of the softmax function over the logits

Softmax

- Yet another activation function that maps [-∞, +∞] to [0, 1] (similar as Sigmoid). Softmax normalizes the sum of the values (output vector) to be 1. It outputs a vector that represents the probability distributions

Summary

Logits and Probs (in DL terms!)

Logits

2

1

0.1

Softmax

$$\sigma(z)_i = \frac{e^{Z_i}}{\sum_{j=1}^{K} e^{Z_j}}$$

Probs

p = 0.7

p = 0.2

p = 0.1

Adds up to 1

Summary
Epochs and Batch Processing

Epochs

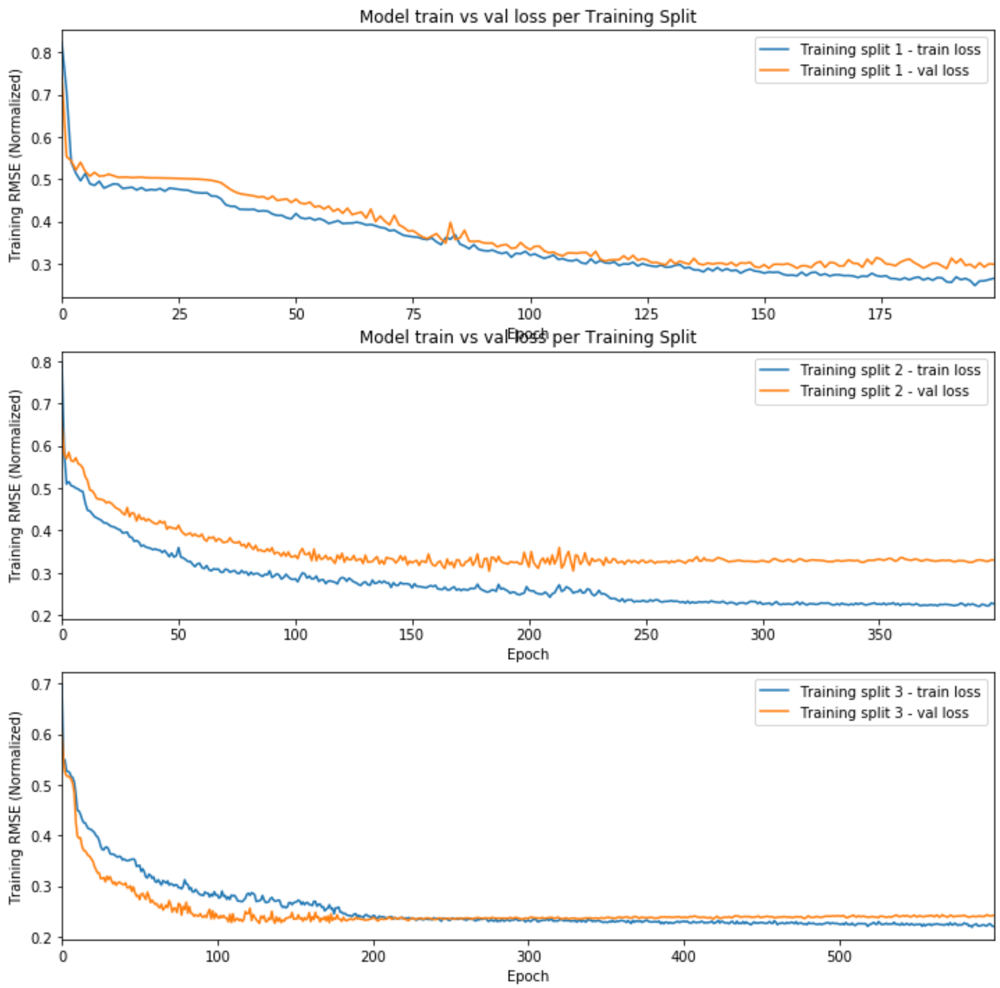
- A full training pass over the entire dataset such that each example has been seen once. Thus, an epoch represents N/batch_size training iterations, where N is the total number of examples. For instance, when using a batch size of 32, there will be 64 training iterations over a dataset with 2048 examples, per epoch
- Controls the number of complete passes through the training dataset
- The number of epochs may be large (10, 100, 500, 1000, …), in order for the learning algorithm to run until the error has been sufficiently minimized
- It is common to create line plots, sometimes called learning curves, that plot epochs (along the x-axis) by the error of the model (on the y-axis)

```
'''
Define a low level fit and predict making use of the tape.gradient
'''
def low_level_fit_and_predict():
    #manually, let's iterate over the epochs and fit ourselves
    for epoch in range(epochs):
        print('Epoch %d/%d' %(epoch+1, epochs))

        #to store loss values
        loss_history = []

        #iterate over all batchs
        for step, (x_batch, y_batch) in enumerate(train_dataset):
```

Summary
Epochs and Batch Processing

Epochs

Learning curves



Summary

Epochs and Batch Processing

Batch processing

- The batch size defines the number of samples to work through before updating the trainable parameters, i.e., it is a hyperparameter (of gradient descent) that controls the number of training samples to work through before updating the model’s internal parameters
- At the end of the batch, the predictions are compared with the expected labels. The error is then calculated. From this error, backpropagation is able to update the trainable parameters
- Batch size is usually fixed during training and inference

Summary
Epochs and Batch Processing

Batch processing

- Batch Gradient Descent
  - Batch Size = Size of the Training Set
  - 1 step of gradient descent in 1 epoch
  - All the examples for every step of Gradient Descent
- Stochastic Gradient Descent
  - Batch Size = 1
  - N steps of gradient descent per epoch (where N is the size of the Training Set)
  - SGD converges faster but is computationally slower
- Mini-Batch Gradient Descent
  - 1 < Batch Size < Size of the Training Set
  - A batch with size of 32 means we will split the entire dataset in batches of 32 elements. For a dataset with 2048 examples we will have 64 batches (iterations) of 32 elements each
  - Neither batch or stochastic gradient descent - between the both

Loss

- What we want is to tune the weights and bias in order to minimize a specific loss/cost function
- The loss tells us how far the model's predictions are from the actual labels
- Or a measure of how bad the model is
- Used to estimate the loss of the model so that the weights can be updated to reduce the loss on the next evaluation

```python
'''
Preparing the model, the optimizers, the loss function and some metrics
'''
def prepare_model():
    mlp = MultilayerPerceptron(output_neurons=output_neurons)
    #instantiate an optimizer
    optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
    #instantiate a loss object (from_logits=False as we are applying a softmax activation over the last layer)
    loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False)
```

Summary
Loss, Gradients and the Gradient Tape

Loss

- Regression models may use MAE (L1 loss), MSE (L2 loss) or RMSE, for example (there are others)

$$MAE = \frac{1}{n}\sum_{j=1}^{n}|y_j - \hat{y}_j| \qquad MSE = \frac{1}{n}\sum_{j=1}^{n}\left(y_j - \hat{y}_j\right)^2 \qquad RMSE = \sqrt{\frac{1}{n}\sum_{j=1}^{n}\left(y_j - \hat{y}_j\right)^2}$$

#	Error	|Error|	Error^2
1	1	1	1
2	1	1	1
3	3	3	9
4	3	3	9

MAE	MSE	RMSE
2	5	2.24

#	Error	|Error|	Error^2
1	0	0	0
2	0	0	0
3	0	0	0
4	10	10	100

MAE	MSE	RMSE
2.5	25	5

(https://www.tensorflow.org/api_docs/python/tf/keras/losses/)

Summary

Loss, Gradients and the Gradient Tape

Loss

- Classification models may use, among others (attention to the from_logits argument):
  - Binary Crossentropy - use this cross-entropy loss when there are only two label classes. Remember entropy (events with equal probability lead to a larger entropy)?;
  - Sparse Categorical Crossentropy - use when there are two or more label classes. Labels are expected to be integers. If you want to provide labels using one-hot representation, use Categorical Crossentropy.
  - Categorical Crossentropy - use when there are two or more label classes. Labels are expected to be one-hot encoded.

$$CE = -\sum_{j}^{C} y_j \log(\hat{y}_j)$$

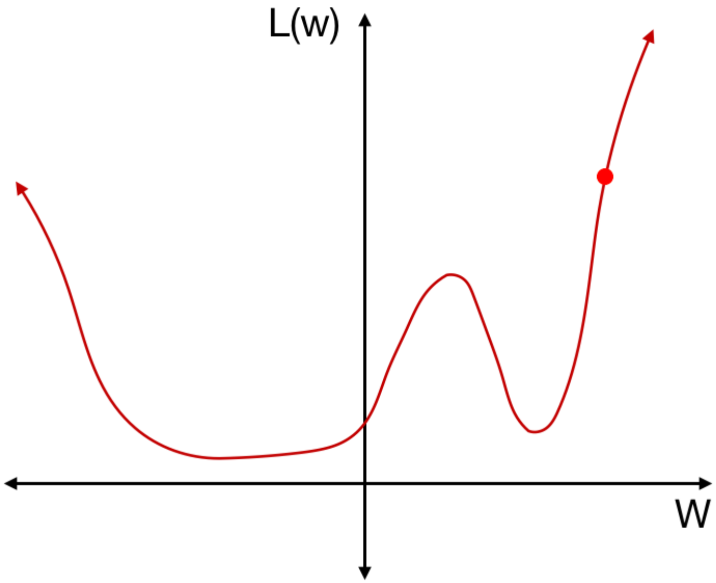
(https://www.tensorflow.org/api_docs/python/tf/keras/losses/)

Summary
Loss, Gradients and the Gradient Tape

Gradient

- What we want is to tune the weights and bias in order to minimize a specific loss/cost function
- The gradient tells us the direction in which to update the weights in regard to the obtained loss value
- We will backpropagate the error to update the weights and bias

L(w) sets the loss value (dependent on the weights).
W sets the weights. It is easier to think 2D.



(https://towardsdatascience.com/the-beginners-guide-to-gradient-descent-c23534f808fd)

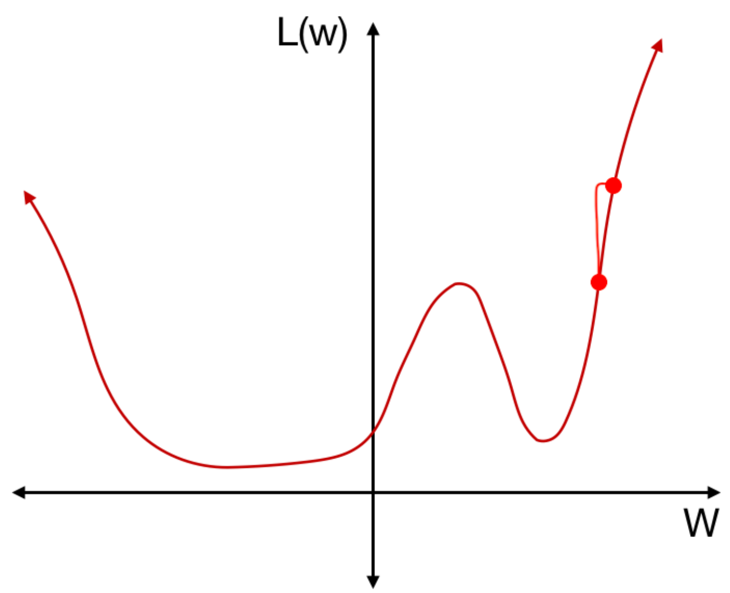
Gradient

- What we want is to tune the weights and bias in order to minimize a specific loss/cost function
- The gradient tells us the direction in which to update the weights in regard to the obtained loss value
- We will backpropagate the error to update the weights and bias

L(w) sets the loss value (dependent on the weights).
W sets the weights. It is easier to think 2D.

Updating the weights (*lr* stands for learning rate):

$$W_{t+1} = W_t - \frac{\partial L(w)}{\partial W_x} \cdot lr$$



(https://towardsdatascience.com/the-beginners-guide-to-gradient-descent-c23534f808fd)

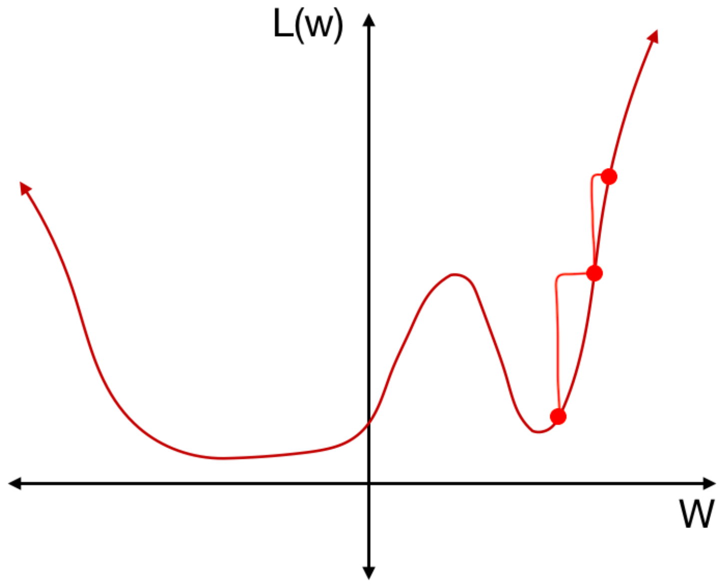
Gradient

- What we want is to tune the weights and bias in order to minimize a specific loss/cost function
- The gradient tells us the direction in which to update the weights in regard to the obtained loss value
- We will backpropagate the error to update the weights and bias

L(w) sets the loss value (dependent on the weights).
W sets the weights. It is easier to think 2D.

Updating the weights (*lr* stands for learning rate):

$$W_{t+1} = W_t - \frac{\partial L(w)}{\partial W_x} \cdot lr$$

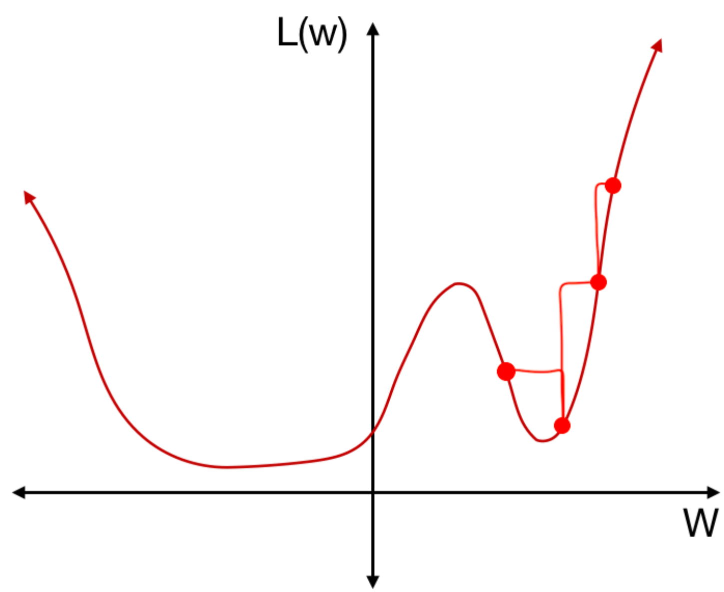


(https://towardsdatascience.com/the-beginners-guide-to-gradient-descent-c23534f808fd)

Gradient

- What we want is to tune the weights and bias in order to minimize a specific loss/cost function
- The gradient tells us the direction in which to update the weights in regard to the obtained loss value
- We will backpropagate the error to update the weights and bias

L(w) sets the loss value (dependent on the weights).
W sets the weights. It is easier to think 2D.



Updating the weights (*lr* stands for learning rate):

$$W_{t+1} = W_t - \frac{\partial L(w)}{\partial W_x} \cdot lr$$

(https://towardsdatascience.com/the-beginners-guide-to-gradient-descent-c23534f808fd)

Summary
Loss, Gradients and the Gradient Tape

Gradient Tape

- The gradient tape allows us to implement the gradient descent optimization algorithm (aka learning - find the best weights and biases to improve the performance of the neural network)
- TensorFlow provides the tf.GradientTape API for computing the gradient of a computation with respect to its input variables
- All operations executed inside the context of a tf.GradientTape are recorded into a "tape“
- We then use the tape and the gradients associated with each recorded operation to compute the gradients of a "recorded" computation
- In other words, all forward-pass operations get recorded to a "tape". To compute the gradient, we play the tape backwards

```
with tf.GradientTape() as tape:
    #running the forward pass of all layers
    #operations being recorded into the tape
    probs = mlp(x_batch)
    #computing the loss for this batch
    #how far are we from the correct labels?
    loss_value = loss_object(y_batch, probs)

#store loss value
loss_history.append(loss_value.numpy().mean())
#use the tape to automatically retrieve the gradients of the trainable variables
#with respect to the loss
gradients = tape.gradient(loss_value, mlp.trainable_weights)
#running one step of gradient descent by updating (going backwards now)
#the value of the trainable variables to minimize the loss
optimizer.apply_gradients(zip(gradients, mlp.trainable_weights))
```

Summary
Loss, Gradients and the Gradient Tape

Gradient Tape

```
x = tf.ones((2, 2))
print(x)

with tf.GradientTape() as t:
    t.watch(x) #watching the input tensor
    y = tf.reduce_sum(x)
    print(y)
    z = tf.multiply(y, y)
    print(z)

#derivative of z with respect to x (input tensor)
dz_dx = t.gradient(z, x)
print(dz_dx)
for i in [0, 1]:
    for j in [0, 1]:
        print('dz_dx = %d' %dz_dx[i][j].numpy())
```

»

```
#our x: the input tensor
tf.Tensor(
[[1. 1.]
 [1. 1.]], shape=(2, 2), dtype=float32)
#y: 4x
tf.Tensor(4.0, shape=(), dtype=float32)
#z=16x^2
tf.Tensor(16.0, shape=(), dtype=float32)
#dz_dx=32x (at x=[[1  1][1  1]])
tf.Tensor(
[[8. 8.]
 [8. 8.]], shape=(2, 2), dtype=float32)
dz_dx = 8
dz_dx = 8
dz_dx = 8
dz_dx = 8
```

Persistent Gradient Tape

```
#to compute multiple gradients, create a
#persistent tape. Resources are released only
#when the tape is garbage collected.
x = tf.constant(5.0)
with tf.GradientTape(persistent=True) as t:
    t.watch(x) #variable to watch - x
    y = x * x #x^2
    z = y * y #x^4

dy_dx = t.gradient(y, x) #10 (2x at x=5)
print('dy_dx = %d' %dy_dx.numpy())

dz_dx = t.gradient(z, x) #500 (4*x^3 at x=5)
print('dz_dx = %d' %dz_dx.numpy())

del t #drop the reference to the persistent tape
```

```
dy_dx = 10
dz_dx = 500
```

Summary
Loss, Gradients and the Gradient Tape

Higher-Order Derivatives with Gradient Tape

```
#trainable variables (created by tf.Variable
#where trainable=True - default) are
#automatically watched
x = tf.Variable(1.0)
print(x)

#GradientTapes can compute higher-order der.
with tf.GradientTape() as t:
    with tf.GradientTape() as t2:
        y = x * x * x #x^3
        dy_dx = t2.gradient(y, x) #3x^2
    #the gradient is differentiable as well
    d2y_dx2 = t.gradient(dy_dx, x) #6x

print('dy_dx = %d' %dy_dx.numpy())
print('d2y_dx2 = %d' %d2y_dx2.numpy())
```

```
<tf.Variable 'Variable:0' shape=()
                dtype=float32, numpy=1.0>

dy_dx = 3

d2y_dx2 = 6
```

Summary

Loss, Gradients and the Gradient Tape

Higher-Order Derivatives with Gradient Tape

```
#trainable variables (created by tf.Variable
#where trainable=True - default) are
#automatically watched
x = tf.Variable(1.0, trainable=False)
print(x)

#GradientTapes can compute higher-order der.
with tf.GradientTape() as t:
    with tf.GradientTape() as t2:
        y = x * x * x #x^3
        dy_dx = t2.gradient(y, x) #3x^2
    #the gradient is differentiable as well
    d2y_dx2 = t.gradient(dy_dx, x) #6x

print('dy_dx = %d' %dy_dx.numpy())
print('d2y_dx2 = %d' %d2y_dx2.numpy())
```

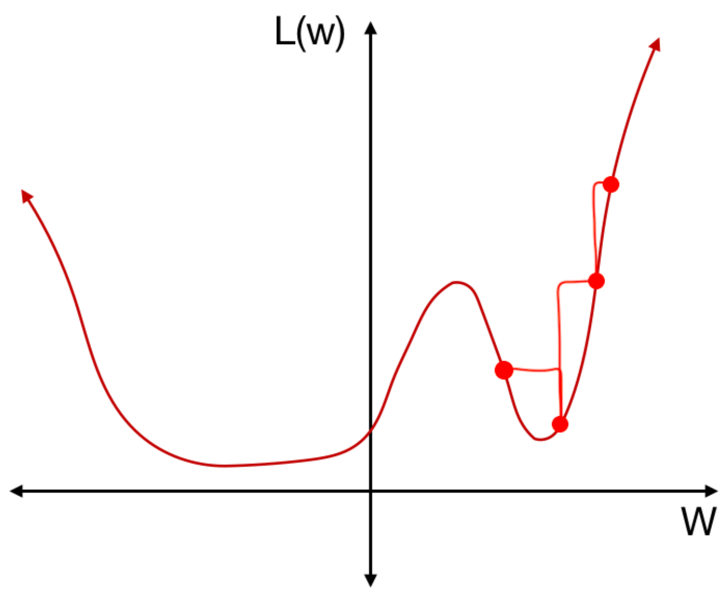
```
File "tensorflow_core\python\eager\
backprop.py", line 984, in gradient:
    AttributeError: 'NoneType' object has no
    attribute 'dtype'
```

Summary

Optimizers and Metrics

Optimizers

- An optimizer applies the computed gradients to the trainable variables to minimize the loss
- Implements backprop for you (minimize loss)



Summary
Optimizers and Metrics

Optimizers

- In TensorFlow there are plenty of optimizers (TensorFlow infers the backprop path from the forward one)! Be thankful!
- Examples of optimizers:
  - tf.train.GradientDescentOptimizer
  - tf.train.AdagradOptimizer
  - tf.train.MomentumOptimizer
  - tf.train.AdamOptimizer
  - ...

```
'''
Preparing the model, the optimizers, the loss function and some metrics
'''
def prepare_model():
    mlp = MultilayerPerceptron(output_neurons=output_neurons)
    #instantiate an optimizer
    optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
    #instantiate a loss object (from_logits=False as we are applying a softmax activation over the last layer)
    loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False)
```

Summary
Optimizers and Metrics

Metrics

- A function that is used to judge the performance of the model
- Metric functions are to be supplied in the metrics parameter when a model is compiled or to be manually used on low-level fit (important to call reset_states())
- A metric function is similar to a loss function, except that the results from evaluating a metric are not used when training the model (indeed, you may use any of the loss functions as a metric function)

```python
'''
Preparing the model, the optimizers, the loss function and some metrics
'''
def prepare_model():
    mlp = MultilayerPerceptron(output_neurons=output_neurons)
    #instantiate an optimizer
    optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
    #instantiate a loss object (from_logits=False as we are applying a softmax activation over the last layer)
    loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False)
    #using a metric too
    train_metric = tf.keras.metrics.SparseCategoricalAccuracy()
    val_metric = tf.keras.metrics.SparseCategoricalAccuracy()
    return mlp, optimizer, loss_object, train_metric, val_metric
```

Summary
Optimizers and Metrics

Metrics

- Supplied in the metrics parameter when a model is compiled

```
'''
Define a high level fit and predict making use tf.Keras APIs
'''
def high_level_fit_and_predict():
    #shortcut to compile and fit a model!
    #able to do this because our model subclasses tf.keras.Model
    mlp.compile(optimizer, loss=loss_object, metrics=[train_metric])
    #since the train_dataset already takes care of batching, we don't pass a batch_size argument
    #passing validation data for monitoring validation loss and metrics at the end of each epoch
    history = mlp.fit(train_dataset, validation_data=validation_dataset, epochs=epochs)
    #print('\nHistory values per epoch:', history.history)
```

Summary
Optimizers and Metrics

Metrics

- Manually used on low-level fit (important to call reset_states())

```
'''
Define a low level fit and predict making use of the tape.gradient
'''
def low_level_fit_and_predict():
    ............
            gradients = tape.gradient(loss_value, mlp.trainable_weights)
            #running one step of gradient descent by updating (going backwards now)
            #the value of the trainable variables to minimize the loss
            optimizer.apply_gradients(zip(gradients, mlp.trainable_weights))
            # Update training metric.
            train_metric(y_batch, probs)

            #log every n batches
            if step%200 == 0:
                print('Step %s. Loss Value = %s; Mean loss = %s' %(step, str(loss_value.numpy()), np.mean(loss_history)))

        #display metrics at the end of each epoch
        train_accuracy = train_metric.result()
        print('Training accuracy for epoch %d: %s' %(epoch+1, float(train_accuracy)))
        #reset training metrics (at the end of each epoch)
        train_metric.reset_states()

        #run a validation loop at the end of each epoch
        for x_batch_val, y_batch_val in validation_dataset:
            val_probs = mlp(x_batch_val)
            #update val metrics
            val_metric(y_batch_val, val_probs)
        val_acc = val_metric.result()
        val_metric.reset_states()
        print('Validation accuracy for epoch %d: %s' % (epoch+1, float(val_acc)))
```

Summary
Vectorization

It is also worth mentioning the importance of Vectorization!

```
import numpy as np
import time

#create two large random arrays
a = np.random.rand(10000000)
b = np.random.rand(10000000)

start = time.time()
c = np.dot(a, b) #vectorized version of dot multiplication
print('Result = %d' %c) #print the result
print('Vectorized version: %f ms' %(1000*(time.time()-start)))

d = 0
start = time.time()
for i in range(10000000):
    d += a[i] * b[i] #let us do it ourselves
print('Result = %d' %d) #print the result
print('With loops: %f ms' %(1000*(time.time()-start)))
```

```
#first run
Result = 2500272
Vectorized version: 15.637398 ms
Result = 2500272
With loops: 4724.763393 ms

#second run
Result = 2499909
Vectorized version: 14.266253 ms
Result = 2499909
With loops: 4825.137854 ms

#you get it!
#it is taking more than 300 times
#the time it took the vectorized version
```

Glossary

- Activation Functions, Batch processing, Bias, Epochs, Logits, Loss, MLP, Softmax, Weights, and so on…

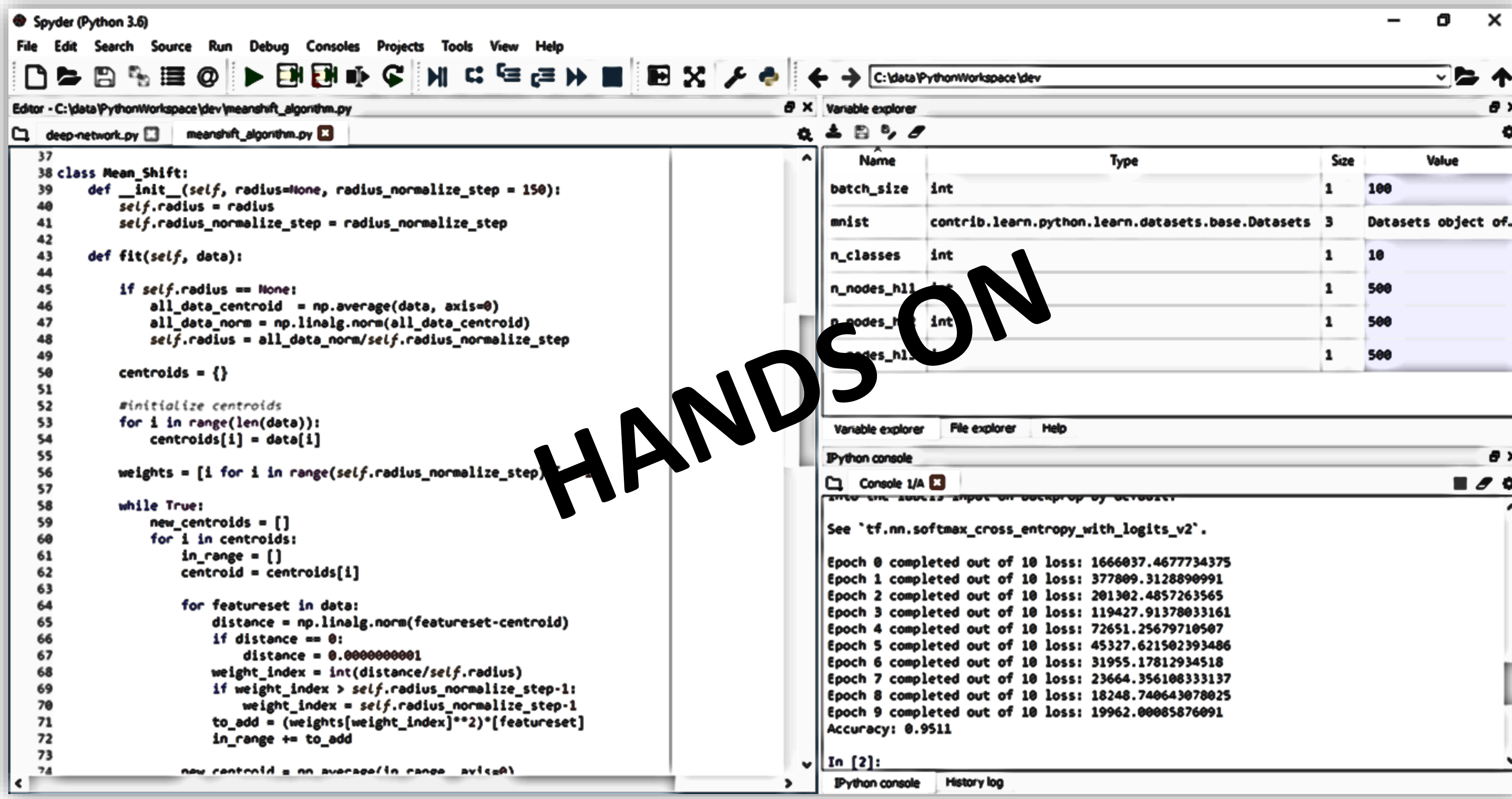
Check the previous slides for the definition of each and every one of the terms we saw today.

Resources

- Official Documentation
  - https://www.tensorflow.org/api_docs/
  - https://www.tensorflow.org/tutorials/customization/autodiff?hl=pt
  - https://www.tensorflow.org/tutorials/customization/custom_training?hl=pt (nice tutorial)
  - …

- Papers, Books, online courses, tutorials…
  - (Book) Deep Learning With Python by Jason Brownlee
  - (Book) Machine Learning Algorithms From Scratch by Jason Brownlee
  - (Book) Deep Learning with TensorFlow 2 and Keras by Antonio Gulli, Amita Kapoor & Sujit Pal

Hands On


Spyder (Python 3.6)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\data\PythonWorkspace\dev
Editor - C:\data\PythonWorkspace\dev\meanshift_algorithm.py
deep-network.py
meanshift_algorithm.py
class Mean_Shift:
def __init__(self, radius=None, radius_normalize_step = 150):
self.radius = radius
self.radius_normalize_step = radius_normalize_step
def fit(self, data):
if self.radius == None:
all_data_centroid = np.average(data, axis=0)
all_data_norm = np.linalg.norm(all_data_centroid)
self.radius = all_data_norm/self.radius_normalize_step
centroids = {}
#initialize centroids
for i in range(len(data)):
centroids[i] = data[i]
weights = [i for i in range(self.radius_normalize_step)
while True:
new_centroids = []
for i in centroids:
in_range = []
centroid = centroids[i]
for featureset in data:
distance = np.linalg.norm(featureset-centroid)
if distance == 0:
distance = 0.0000000001
weight_index = int(distance/self.radius)
if weight_index > self.radius_normalize_step-1:
weight_index = self.radius_normalize_step-1
to_add = (weights[weight_index]**2)*[featureset]
in_range += to_add
Variable explorer
Name | Type | Size | Value
batch_size | int | 1 | 100
mnist | contrib.learn.python.learn.datasets.base.Datasets | 3 | Datasets object of…
n_classes | int | 1 | 10
1 | 500
1 | 500
1 | 500
Variable explorer | File explorer | Help
IPython console
Console 1/A
See `tf.nn.softmax_cross_entropy_with_logits_v2`.
Epoch 0 completed out of 10 loss: 1666037.4677734375
Epoch 1 completed out of 10 loss: 377809.3128890991
Epoch 2 completed out of 10 loss: 201302.4857263565
Epoch 3 completed out of 10 loss: 119427.91378033161
Epoch 4 completed out of 10 loss: 72651.25679710507
Epoch 5 completed out of 10 loss: 45327.621502393486
Epoch 6 completed out of 10 loss: 31955.17812934518
Epoch 7 completed out of 10 loss: 23664.356108333137
Epoch 8 completed out of 10 loss: 18248.740643078025
Epoch 9 completed out of 10 loss: 19962.00085876091
Accuracy: 0.9511
In [2]:
IPython console | History log
HANDS ON