Notebook Assignment 3 - Due Date TBD:

Deep Learning Software Frameworks (updated: November 14, 2024)

Python Classes for Neural Networks: Deep learning frameworks (TensorFlow) allow us to easily train models with millions of parameters because of they use object oriented programming (OOP) principles. OOPs methods provide a way to structure programs in a modular manner by bundling related properties and behaviors in **objects**. An *object* is like a "software brick" with internal variables and methods (functions) that are available to other "bricks" through a well defined published interface. This means that these "software objects (bricks)" can be modularly stacked on top of each other in a way that allows one to easily build large programs.

The main concept in OOP is the **class**. A *class* may be seen as a template for creating an object. It defines what variables are held within the object, provides initialization functions to create an *instance* of the class, and provides the functions (a.k.a. methods) that allow one to change the object's internal variables. Python has long used OOPs principles and this is one of the reasons it has become the de facto language used by deep learning engineers. This course will be using TensorFlow and the effective use of TensorFlow is made easier if we understand how its objects are created.

In Python, one defines the "class" blueprint using the `class` keyword followed by a name and a colon. You then declare the `__init__()` method which is called when we create an instance of the class. The `__init__()` method is called a class *constructor*. We will now walk through a concrete example of two class definitions used in creating a sequential neural network object.

The first class of interest is the `SequentialModel` class. The following Python script defines the class template.

```
class SequentialModel:

     #class constructor
     def __init__(self,layers):
          self.layers = layers

     def __call__(self,inputs):
```

```
            x = inputs
            for layer in self.layers:
                 x = layer(x)
            return x

      @property
      def weights(self):
            weights = []
            for layer in self.layers:
                 weights += layer.weights
            return weights
```

The SequentialModel class has a single internal variable, `layers`, which is simply a list of *layer* objects that we will define below. Our template has three functions (methods); `__init__`, `__call__`, and `weights`. The `__init__` method is the constructor initializing the internal variable `self.layers` with a list of layer objects that are passed as inputs in the constructor's call. The second method, `__call__`, is called on an existing instance of the class and returns the sequential models output for a given input. Note that this call simply calls the model's *layer* objects in a recursive manner, using the output of one layer as the input to the next layer. The last method `weights` is defined using a Python `@property` decorator. This function provides access to the internal variables of the model layer's weights. We will need to use it when updating the weights of the model.

The second class we will have to define is a `DenseLayer` class. In particular, our sequential model will be built by concatenating several DenseLayer objects together. The following Python script defines the `DenseLayer` class.

```
import tensorflow as tf
import numpy as np

class DenseLayer:

      #class constructor
      def __init__(self, input_size, output_size, activation):
            self.activation = activation
```

```
        w_shape = (input_size, output_size)
        w_initial_value = tf.random.uniform(w_shape, minval = 0, maxval = 1e-1)
        self.W = tf.Variable(w_initial_value)

        b_shape = (output_size,)
        b_initial_value = tf.zeros(b_shape)
        self.b = tf.Variable(b_initial_value)

    def __call__(self, inputs):
        return self.activation(tf.matmul(inputs, self.W)+self.b)

    @property
    def weights(self):
        return [self.W, self.b]
```

The `DenseLayer` object has three internal variables that are initialized by the class constructor using three arguments passed to the constructor. One of these variables, `self.activation` is a pointer to an *activation function*. The second variable `self.W` is a matrix of layer weights with the shape `(input_size, output_size)`. In this code we initial `self.W` randomly. The third variable `self.b` is a vector of biases with the shape `(input_size,)`. Note that both of these variables, `self.W` and `self.b`, have been cast as TensorFlow variables (`tf.Variable`). We do this because we will later use TensorFlow's `GradientTape` object to automate the computation of the model loss' gradient with respect to these two variables. The DenseLayer class uses the `property` decorator to declare a `weights` method that returns the layer's weights when called. Recall from our definition of the `SequentialModel` class that we used this layer method when implement the SequentialModel's weights method.

Problem 1:

(1) Create the input, `X`, and target, `Y`, data matrices from the ring data file, `data`

```
data = np.load("data/ring_data_-5.npy")
```

in the same way you did for a similar ring data file in assignment 1. Be sure to normalize the input samples `X` so they are `float32` numbers between -1 and 1. Your final input data matrix `X` should have the shape $(2000, 2)$. Be sure to convert

the target data, `Y`, so its components are `int8` numbers 0 or 1. The original data set has targets of -1 and 1, so you will need to convert the -1 targets to 0. Do a scatter plot of the data samples with different colors for the 0 and 1 class (similar to what you did in assignment 1). Your plot should show two half rings of points that are not separable by a linear discriminant.

(2) Use the preceding class definitions to create an instance of a neural network using the script,

```
layer1 = DenseLayer(2, 2, tf.nn.relu)
layer2 = DenseLayer(2, 2, tf.nn.softmax)
model = SequentialModel([layer1, layer2])
```

The two outputs of this model are the probability of the input being in class 0 or class 1, so you are solving a logistic regression problem with this model. Write a Python function

```
def evaluate(model, X,Y):
```

that takes the instantiated model, the input data `X`, and output targets `Y` and returns the *accuracy* of the model (i.e. the percentage of correct model predictions). You should get something between 50-75%

Problem 2:. In assignment 1 you analytically computed the gradient of your model's empirical risk function with respect to the model weights. That analytical approach is impractical when we have extremely deep neural network models with tens or hundreds of thousands of weights. In practice, we use a method known as *automatic differentiation* to evaluate the empirical risk's gradient for a specified model. This approach constructs a *computation graph* that records the order of numerical operations performed in computing the model's output. This computation graph can then be used to automate the computation of the empirical risk function's gradient.

TensorFlow provides a *GradientTape* object to perform automatic differentiation. The following code shows how the GradientTape object is used to compute empirical risk's gradient and then update the weights in the model.

```
#record the operations used to compute model's loss
with tf.GradientTape() as tape:
     Yhat = model(X)
```

```
     loss = loss_function(Y, Yhat)

#compute the gradient
gradients = tape.gradient(loss, model.weights)

#use the gradient to update weights
learning_rate = 0.1
for g,w in zip(gradients, model.weights)
     w.assign_sub(g * learning_rate)
```

In the preceding script `loss_function` returns the loss for a given set of model predictions, `Yhat`, and the data targets, `Y`. The loss function for your logistic regression problem is the sparse categorical crossentropy function. This is one of the functions in the Keras library. In particular, you could write this function as

```
def loss_function(Y,Yhat):
     scce = tf.keras.losses.sparse_categorical_crossentropy
     per_sample_losses = scce(Y, Yhat)
     loss = tf.reduce_mean(per_sample_losses)
     return loss
```

All of the operations within the scope of the statement

```
with tf.GradientTape() as tape:
```

are recorded on the `tape` object, so these are the operations used to compute the model's prediction and the loss function. The next statement outside of the scope of the `tape` object actually computes the gradients

```
gradients = tape.gradient( loss, model.weights)
```

with respect to all of the model's weights and biases. The last part of the above code then updates the weights.

(1) Write a Python script that partitions the dataset, (`X`,`Y`), into training (`X_train`,`Y_train`) and testing (`X_test`, `Y_test`) datasets. Assume a 80/20 split with 80% of the data in the training set.
(2) Write a Python function

`def fit(model, X_train, Y_train, X_test, Y_test, N, lr):`

This function takes the training and testing datasets and trains the model using gradient descent for `N` epochs and a learning rate of `lr` on the training data, `(X_train,Y_train)`. Have your function evaluate the training loss on `(X_train,Y_train)` after every epoch. Have your function evaluate the testing loss on `X_test,Y_test)` after every epoch. Your function should print the training and testing loss after every 10th training epoch. Have your function return a numpy array, `history`, with shape `(N+1,2)` where `N` is the number of epochs we trained for and `history[k,:]` is the training and testing loss for the model after the kth training epoch.

(3) Use your `fit` function to train the original `model` (2 layers with the first layer having 2 outputs) for 1000 epochs assuming a learning rate `lr=0.1`. You should see the training and testing loss decrease from about 0.69 to a level of 0.20 (approximately).

- Use the `evaluate` function you wrote above to evaluate the accuracy of your final model on the testing dataset.
- Use the `history` array returned by your function to plot the training and testing loss as a function of training epoch.
- Scatter plot all of the input samples, `X`, and color these points with your final model's predicted classification.

(4) Repeat the preceding problem using a model formed from 3 dense layers with the middle layer having 512 inputs and 512 outputs using the ReLu activation function.