

1. I spent time on tensorflow manipulating different ANNs.

3. Perceptrons do not output a class probability. They also do not use any regularization by default, and training stops as soon as there are no more prediction errors on the training set, so the model typically does not generalize as well as logistic regression. This is one reason to prefer logistic regression over perceptrons. However, if you change the activation function as sigmoid and train it with gradient descent then it becomes like the LogisticRegressionClassifier.

4. The first MLPs used just a step function, which contains only flat segments that gradient descent can't work with. The sigmoid function has well defined non-zero derivatives, which allows gradient descent to progress through each step.

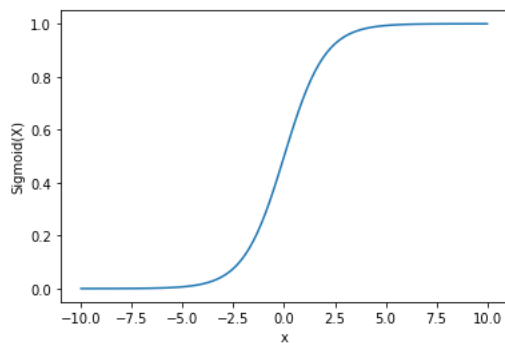
5. Three popular activation functions are sigmoid, Tanh, ReLU.

```
# Sigmoid
import matplotlib.pyplot as plt
import numpy as np
import math
```

```
x = np.linspace(-10, 10, 100)
z = 1/(1 + np.exp(-x))
```

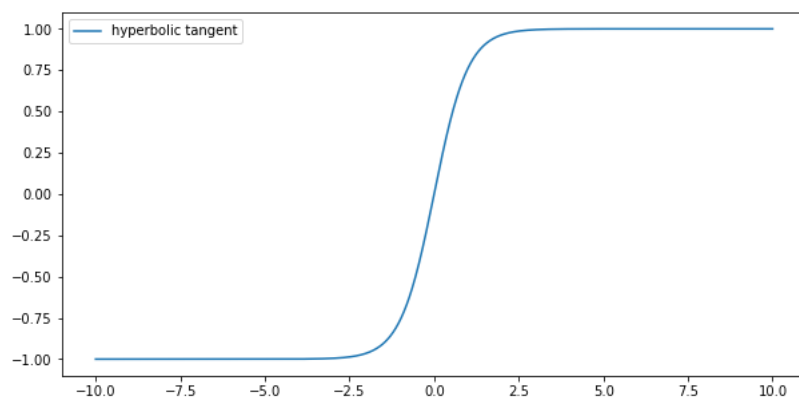
```
plt.plot(x, z)
plt.xlabel("x")
plt.ylabel("Sigmoid(X)")
```

```
plt.show()
```



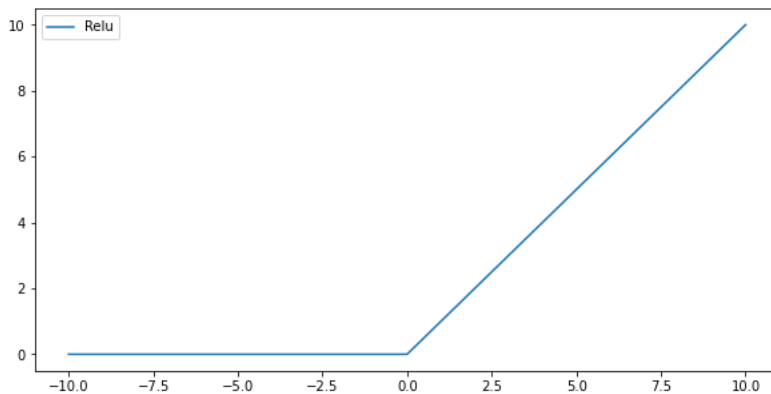
```
#Tanh
x = np.linspace(-10, 10, 1000)
y = ( 2 / ( 1 + np.exp(-2*x) ) ) - 1
```

```
plt.figure(figsize=(10, 5))
plt.plot(x, y)
plt.legend(['hyperbolic tangent'])
plt.show()
```



```
#Relu
x = np.linspace(-10, 10, 1000)
y = np.maximum(0, x)

plt.figure(figsize=(10, 5))
plt.plot(x, y)
plt.legend(['Relu'])
plt.show()
```



6. a. The input matrix shape is  $m \times 10$  where  $m$  is the batch size b. The shape of the hidden layer's weight matrix is  $10 \times 50$ , and the length of its bias vector is 50. c. The shape of the output layer's weight matrix is  $50 \times 3$ , and the length of its bias vector is 3. d. The shape of the network's output matrix is  $m \times 3$ . e.  $Y = \text{ReLU}(\text{ReLU}(XW_h + b_h)W_o + b_o)$

9. The hyperparameters to tweak in an MLP are the number of hidden layers, the number of neurons in each hidden layer, and the activation function used in each hidden layer and in the output layer.

If the MLP overfits the training data, you can try reducing the number of hidden layers and reducing the number of neurons per hidden layer.

```
import tensorflow as tf
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.model_selection import GridSearchCV
```

7.

##PART A

```
fire= pd.read_csv('https://raw.githubusercontent.com/esnt/Data/main/Fires/utah_fires.csv')
#Turning the y into binary classification
fire['NWCG_CAUSE_CLASSIFICATION']=(fire['NWCG_CAUSE_CLASSIFICATION']=='Human').astype(int)

#setting x and y
y=fire['NWCG_CAUSE_CLASSIFICATION']
X=fire.loc[:, 'FIRE_YEAR': 'FIRE_SIZE']

#Splitting the data
Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, test_size=.25, random_state= 307)

#Filling in missing values and scaling data
pipe= Pipeline([('imputer', SimpleImputer(strategy='mean')), ('scaler', StandardScaler())])
#Fitting and transforming x sets
xtrain=pipe.fit_transform(Xtrain)
xtest=pipe.transform(Xtest)

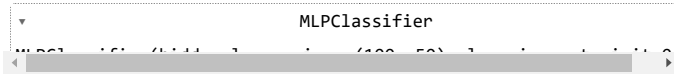
#splitting the training data into the validation sets and training
Xtrain2, Xvalid, ytrain2, yvalid = train_test_split(xtrain, ytrain, test_size=.1, random_state= 307)
```

B. In an ANN with 2 hidden layers with one with 100 nodes and another with 50 nodes, the hyperparameters and number of nodes in each layer.

```
from sklearn.neural_network import MLPClassifier
import sklearn.metrics as skm
```

```
## PART C
```

```
#fitting the mlp classier with layers and learning rate
mlp = MLPClassifier(hidden_layer_sizes=(100,50),learning_rate_init=.1)
mlp.fit(xtrain, ytrain)
```



```
#predicting the test and training accuracy
p1=mlp.predict(xtrain)
p2= mlp.predict(xtest)
print(skm.accuracy_score(ytrain,p1))
print(skm.accuracy_score(ytest,p2))
```

```
0.679900084738581
0.6844178952719878
```

```
tf.keras.backend.clear_session()
tf.random.set_seed(42)
```

```
#creating the model with two hidden layers
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[6,]),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(50, activation="relu"),
    tf.keras.layers.Dense(1, activation="sigmoid")
])
```

```
#compiling the model with adam and learning rate of .1
model.compile(loss="binary_crossentropy",
              optimizer=tf.keras.optimizers.Adam(learning_rate=.1),
              metrics=["accuracy"])
```

```
#early stopping function
early_stopping_cb = tf.keras.callbacks.EarlyStopping(patience=10,
                                                    restore_best_weights=True)
```

```
#fitting the model
history = model.fit(Xtrain2, ytrain2, epochs=100,
                   validation_data=(Xvalid, yvalid),
                   callbacks=[ early_stopping_cb])
```

```
Epoch 1/100
664/664 [=====] - 3s 2ms/step - loss: 0.5820 - accuracy: 0.7246 - val_loss: 0.5747 - val_accuracy: 0.7145
Epoch 2/100
664/664 [=====] - 1s 2ms/step - loss: 0.5658 - accuracy: 0.7297 - val_loss: 0.5521 - val_accuracy: 0.7344
Epoch 3/100
664/664 [=====] - 1s 2ms/step - loss: 0.5772 - accuracy: 0.7243 - val_loss: 0.5637 - val_accuracy: 0.7277
Epoch 4/100
664/664 [=====] - 2s 2ms/step - loss: 0.5836 - accuracy: 0.7170 - val_loss: 0.5526 - val_accuracy: 0.7209
Epoch 5/100
664/664 [=====] - 1s 2ms/step - loss: 0.5578 - accuracy: 0.7362 - val_loss: 0.5673 - val_accuracy: 0.7357
Epoch 6/100
664/664 [=====] - 1s 2ms/step - loss: 0.5538 - accuracy: 0.7365 - val_loss: 0.5357 - val_accuracy: 0.7421
Epoch 7/100
664/664 [=====] - 1s 2ms/step - loss: 0.5489 - accuracy: 0.7401 - val_loss: 0.5341 - val_accuracy: 0.7370
Epoch 8/100
664/664 [=====] - 2s 3ms/step - loss: 0.5496 - accuracy: 0.7401 - val_loss: 0.5521 - val_accuracy: 0.7213
Epoch 9/100
664/664 [=====] - 2s 2ms/step - loss: 0.5402 - accuracy: 0.7462 - val_loss: 0.5521 - val_accuracy: 0.7179
Epoch 10/100
664/664 [=====] - 1s 2ms/step - loss: 0.5362 - accuracy: 0.7454 - val_loss: 0.5361 - val_accuracy: 0.7399
Epoch 11/100
664/664 [=====] - 1s 2ms/step - loss: 0.5331 - accuracy: 0.7430 - val_loss: 0.6153 - val_accuracy: 0.7412
Epoch 12/100
664/664 [=====] - 1s 2ms/step - loss: 0.5446 - accuracy: 0.7437 - val_loss: 0.5612 - val_accuracy: 0.7399
Epoch 13/100
664/664 [=====] - 1s 2ms/step - loss: 0.5341 - accuracy: 0.7451 - val_loss: 0.5355 - val_accuracy: 0.7289
Epoch 14/100
664/664 [=====] - 1s 2ms/step - loss: 0.5264 - accuracy: 0.7448 - val_loss: 0.5601 - val_accuracy: 0.7103
Epoch 15/100
```

```

664/664 [=====] - 1s 2ms/step - loss: 0.5350 - accuracy: 0.7486 - val_loss: 0.5268 - val_accuracy: 0.7526
Epoch 16/100
664/664 [=====] - 2s 3ms/step - loss: 0.5331 - accuracy: 0.7458 - val_loss: 0.5292 - val_accuracy: 0.7497
Epoch 17/100
664/664 [=====] - 2s 3ms/step - loss: 0.5326 - accuracy: 0.7455 - val_loss: 0.5588 - val_accuracy: 0.7298
Epoch 18/100
664/664 [=====] - 1s 2ms/step - loss: 0.5331 - accuracy: 0.7446 - val_loss: 0.5367 - val_accuracy: 0.7370
Epoch 19/100
664/664 [=====] - 1s 2ms/step - loss: 0.5344 - accuracy: 0.7441 - val_loss: 0.5827 - val_accuracy: 0.7078
Epoch 20/100
664/664 [=====] - 1s 2ms/step - loss: 0.5530 - accuracy: 0.7411 - val_loss: 0.5327 - val_accuracy: 0.7488
Epoch 21/100
664/664 [=====] - 1s 2ms/step - loss: 0.5422 - accuracy: 0.7408 - val_loss: 0.5644 - val_accuracy: 0.7073
Epoch 22/100
664/664 [=====] - 1s 2ms/step - loss: 0.5304 - accuracy: 0.7456 - val_loss: 0.5318 - val_accuracy: 0.7310
Epoch 23/100
664/664 [=====] - 1s 2ms/step - loss: 0.5338 - accuracy: 0.7426 - val_loss: 0.5278 - val_accuracy: 0.7463
Epoch 24/100
664/664 [=====] - 1s 2ms/step - loss: 0.5293 - accuracy: 0.7487 - val_loss: 0.5275 - val_accuracy: 0.7463
Epoch 25/100
664/664 [=====] - 4s 7ms/step - loss: 0.5271 - accuracy: 0.7490 - val_loss: 0.5230 - val_accuracy: 0.7404
Epoch 26/100
664/664 [=====] - 2s 2ms/step - loss: 0.5324 - accuracy: 0.7420 - val_loss: 0.5317 - val_accuracy: 0.7336
Epoch 27/100
664/664 [=====] - 1s 2ms/step - loss: 0.5285 - accuracy: 0.7455 - val_loss: 0.5258 - val_accuracy: 0.7476
Epoch 28/100
664/664 [=====] - 1s 2ms/step - loss: 0.5256 - accuracy: 0.7472 - val_loss: 0.5247 - val_accuracy: 0.7454
Epoch 29/100
664/664 [=====] - 1s 2ms/step - loss: 0.5421 - accuracy: 0.7307 - val_loss: 0.5521 - val_accuracy: 0.7400

#evaluating the accuracy of the test and training
model.evaluate(xtrain, ytrain)
model.evaluate(xtest, ytest)

738/738 [=====] - 1s 2ms/step - loss: 0.5151 - accuracy: 0.7517
246/246 [=====] - 0s 2ms/step - loss: 0.5118 - accuracy: 0.7564
[0.51178377866745, 0.7563548684120178]

```

D. For the MLP Classifier, the training and test accuracy were 0.7769680535547835 and 0.7774529740721912. For the Keras TensorFlow, the training and test accuracy were .75 and .76.

E. My models were not quite as good as the best model from my HW4, which was .8, and these mlp and keras models are off by about .1.

## PART F

```

#Adding a node to the keras model
tf.keras.backend.clear_session()
tf.random.set_seed(42)

#creating the model with three hidden layers
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[6,]),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(50, activation="relu"),
    tf.keras.layers.Dense(75, activation="relu"),
    tf.keras.layers.Dense(1, activation="sigmoid")
])

#compiling the model with adam optimizer with a learning rate of .1
model.compile(loss="binary_crossentropy",
              optimizer=tf.keras.optimizers.Adam(learning_rate=.1),
              metrics=["accuracy"])

#creating early stopping
early_stopping_cb = tf.keras.callbacks.EarlyStopping(patience=10,
                                                    restore_best_weights=True)

#fitting the model
history = model.fit(Xtrain2, ytrain2, epochs=100,
                  validation_data=(Xvalid, yvalid),
                  callbacks=[early_stopping_cb])

model.evaluate(xtrain, ytrain)
model.evaluate(xtest, ytest)

Epoch 1/100
664/664 [=====] - 3s 3ms/step - loss: 0.6576 - accuracy: 0.6627 - val_loss: 0.6430 - val_accuracy: 0.6235
Epoch 2/100

```

```
664/664 [=====] - 2s 3ms/step - loss: 0.6335 - accuracy: 0.6796 - val_loss: 0.6494 - val_accuracy: 0.6493
Epoch 3/100
664/664 [=====] - 1s 2ms/step - loss: 0.6545 - accuracy: 0.6449 - val_loss: 0.6689 - val_accuracy: 0.6235
Epoch 4/100
664/664 [=====] - 1s 2ms/step - loss: 0.6612 - accuracy: 0.6311 - val_loss: 0.6625 - val_accuracy: 0.6235
Epoch 5/100
664/664 [=====] - 1s 2ms/step - loss: 0.6599 - accuracy: 0.6311 - val_loss: 0.6624 - val_accuracy: 0.6235
Epoch 6/100
664/664 [=====] - 2s 2ms/step - loss: 0.6605 - accuracy: 0.6311 - val_loss: 0.6629 - val_accuracy: 0.6235
Epoch 7/100
664/664 [=====] - 2s 2ms/step - loss: 0.6603 - accuracy: 0.6311 - val_loss: 0.6626 - val_accuracy: 0.6235
Epoch 8/100
664/664 [=====] - 2s 2ms/step - loss: 0.6609 - accuracy: 0.6311 - val_loss: 0.6649 - val_accuracy: 0.6235
Epoch 9/100
664/664 [=====] - 2s 3ms/step - loss: 0.6605 - accuracy: 0.6311 - val_loss: 0.6629 - val_accuracy: 0.6235
Epoch 10/100
664/664 [=====] - 2s 3ms/step - loss: 0.6600 - accuracy: 0.6311 - val_loss: 0.6714 - val_accuracy: 0.6235
Epoch 11/100
664/664 [=====] - 2s 2ms/step - loss: 0.6611 - accuracy: 0.6311 - val_loss: 0.6635 - val_accuracy: 0.6235
738/738 [=====] - 2s 3ms/step - loss: 0.6426 - accuracy: 0.6303
246/246 [=====] - 1s 2ms/step - loss: 0.6371 - accuracy: 0.6347
[0.6371209025382996, 0.6347229480743408]
```

It looks like adding a layer made the accuracy go down, which is interesting. I wonder how I can know the best number of hidden layers and nodes to use like if there is a gridsearch type of function or something.

✓ 22s completed at 6:20 PM

