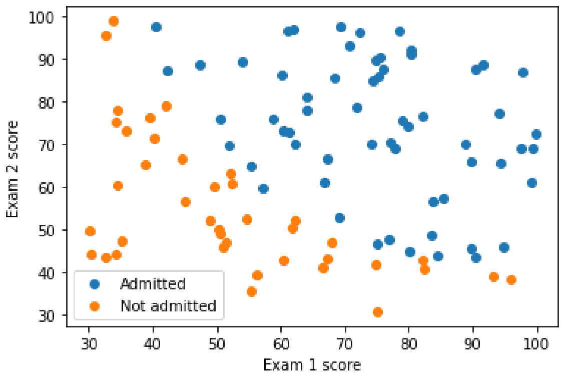
<u>Build a classification model that estimates the probability of admission based on the exam scores using logistic regression</u>

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import scipy.optimize as opt    # more on this later
In [2]:
data = pd.read_csv('Exp2.txt', header = None)
X = data.iloc[:,:-1]
y = data.iloc[:,2]
data.head(10)
Out[2]:
```

	0	1	2
0	34.623660	78.024693	0
1	30.286711	43.894998	0
2	35.847409	72.902198	0
3	60.182599	86.308552	1
4	79.032736	75.344376	1
5	45.083277	56.316372	0
6	61.106665	96.511426	1
7	75.024746	46.554014	1
8	76.098787	87.420570	1
9	84.432820	43.533393	1

```
In [3]:
mask = y == 1
adm = plt.scatter(X[mask][0].values, X[mask][1].values)
not_adm = plt.scatter(X[~mask][0].values, X[~mask][1].values)
plt.xlabel('Exam 1 score')
plt.ylabel('Exam 2 score')
plt.legend((adm, not_adm), ('Admitted', 'Not admitted'))
plt.show()
```



In [4]:

```
def sigmoid(x):
    return 1/(1+np.exp(-x))
```

In [5]:

```
def costFunction(theta, X, y):
   J = (-1/m) * np.sum(np.multiply(y, np.log(sigmoid(X @ theta)))
       + np.multiply((1-y), np.log(1 - sigmoid(X @ theta))))
   return J
```

In [6]:

```
def gradient(theta, X, y):
   return ((1/m) * X.T @ (sigmoid(X @ theta) - y))
```

In [7]:

```
(m, n) = X.shape
X = np.hstack((np.ones((m,1)), X))
y = y[:, np.newaxis]
theta = np.zeros((n+1,1)) # intializing theta with all zeros
J = costFunction(theta, X, y)
print(J)
```

0.6931471805599453

C:\Users\dell\AppData\Local\Temp\ipykernel_11536\2622592648.py:3: FutureWarning: Support for multi-dimensional indexing (e.g. `obj[:, None]`) is deprecated and will be removed in a future version. Convert to a numpy array before indexing instead.

  y = y[:, np.newaxis]

In [8]:

```
temp = opt.fmin_tnc(func = costFunction,
                    x0 = theta.flatten(),fprime = gradient,
                    args = (X, y.flatten()))
#the output of above function is a tuple whose first element #contains the optimized values of theta
theta_optimized = temp[0]
print(theta_optimized)
```

[-25.16131856   0.20623159   0.20147149]

In [9]:

```
J = costFunction(theta_optimized[:,np.newaxis], X, y)
print(J)
```

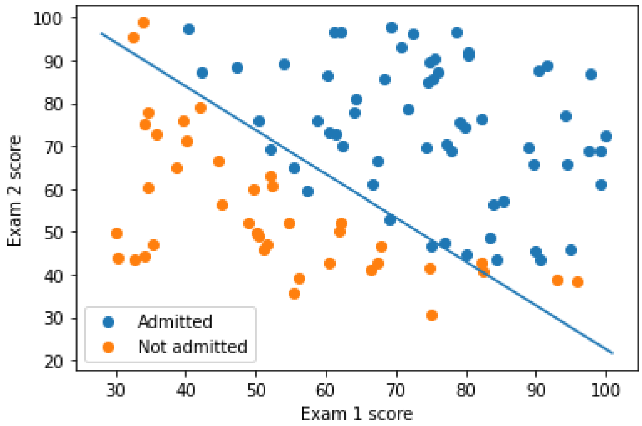
0.20349770158947483

In [10]:

```
plot_x = [np.min(X[:,1]-2), np.max(X[:,2]+2)]
```

In [11]:

```
plot_y = -1/theta_optimized[2]*(theta_optimized[0]
        + np.dot(theta_optimized[1],plot_x))
In [12]:
mask = y.flatten() == 1
adm = plt.scatter(X[mask][:,1], X[mask][:,2])
not_adm = plt.scatter(X[~mask][:,1], X[~mask][:,2])
decision_boun = plt.plot(plot_x, plot_y)
plt.xlabel('Exam 1 score')
plt.ylabel('Exam 2 score')
plt.legend((adm, not_adm), ('Admitted', 'Not admitted'))
plt.show()
```




```
In [13]:
def accuracy(X, y, theta, cutoff):
   pred = [sigmoid(np.dot(X, theta)) >= cutoff]
   acc = np.mean(pred == y)
   print("The accuracy of the work is",acc * 100)
accuracy(X, y.flatten(), theta_optimized, 0.5)
output:
The accuracy of the work is 89.0
```

Implement un-regularized and regularized versions of the neural network cost function and compute gradients via the backpropagation algorithm

```
import sys
assert sys.version_info >= (3, 5)
import numpy as np
import scipy.io as sio
%matplotlib inline
import matplotlib.pyplot as plt
import matplotlib as mpl
%load_ext autoreload
%autoreload 2
```

In [2]:

```
df_path = 'ex4data1.mat'
data = sio.loadmat(df_path)
X_train = data['X']
y_train = data['y']
print('\ni. Training features[X_train]:\n\tRows: %d, columns: %d' % (X_train.shape[0],
X_train.shape[1]))
print('ii. Training labels[y_train]:\n\tRows: %d, columns: %d' % (y_train.shape[0], y_train.shape[1]))
```

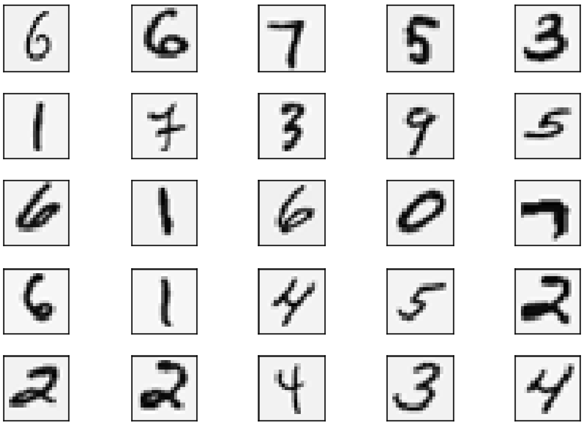
```
i. Training features[X_train]:
	Rows: 5000, columns: 400
ii. Training labels[y_train]:
	Rows: 5000, columns: 1
```

In [3]:

```
def visualizeData(x):
  fig, ax = plt.subplots(nrows =5, ncols=5,sharex=True, sharey=True)
  ax = ax.flatten()
  m = x.shape[0]
  for i in range(25):
    img = x[np.random.randint(0,m),:].reshape(20,20,order="F")
    ax[i].imshow(img, cmap=mpl.cm.binary, interpolation="nearest")
    ax[i].axis("on")
  ax[0].set_xticks([])
  ax[0].set_yticks([])
  plt.tight_layout()
  plt.show()
```

In [4]:

```
visualizeData(X_train)
```



In [5]:

```
# Load the weights into variables W1 and W2
df = 'ex4weights.mat'
weights = sio.loadmat(df)
W1 = weights['Theta1']
W2 = weights['Theta2']
W1_flat = W1.flatten()
W2_flat = W2.flatten()
nn_weights = np.concatenate((W1_flat,W2_flat),axis=0).reshape(-1,1) #Alternatively unrolling
print('\nNeural Network Parameters Successfully Loaded ...\n')
```

Neural Network Parameters Successfully Loaded .

In [6]:

```
print('W1: ', W1.shape)
print('W2: ', W2.shape)
print('NN_WEIGHTS: ', nn_weights.shape)
```

W1: (25, 401)
W2: (10, 26)
NN_WEIGHTS: (10285, 1)

In [7]:

```
input_layer_size  = 400;  # 20x20 Input Images of Digits
hidden_layer_size = 25;   # 25 hidden units
num_labels = 10;          # 10 labels, from 1 to 10 (note that we have mapped "0" to label 10)
```

In [8]:

```
def initialize_parameters(nn_weights, input_layer_size, hidden_layer_size, num_labels):
   W1 = np.reshape(nn_weights[0:hidden_layer_size * (input_layer_size + 1)], (hidden_layer_size,
(input_layer_size + 1)));
   W2 = np.reshape(nn_weights[(hidden_layer_size * (input_layer_size + 1)):,], (num_labels,
(hidden_layer_size + 1)));
   assert (W1.shape == (hidden_layer_size, input_layer_size + 1))
   assert (W2.shape == (num_labels, hidden_layer_size + 1))
   parameters = {"W1":W1,
              "W2":W2}
   return parameters
```

In [9]:

```
parameters = initialize_parameters(nn_weights, input_layer_size, hidden_layer_size, num_labels);
W1 = parameters["W1"]
```

```
W2 = parameters["W2"]
print('The shape of X is: ' + str(X_train.shape))
print('The shape of Y is: ' + str(y_train.shape))
print('The shape of W1 is: ' + str(W1.shape))
print('The shape of W2 is: ' + str(W2.shape))
The shape of X is: (5000, 400)
The shape of Y is: (5000, 1)
The shape of W1 is: (25, 401)
The shape of W2 is: (10, 26)
In [10]:
def sigmoid(z):
   return (1/(1+ np.exp(-z)))
In [11]:
def sigmoidGradient(z):
   return np.multiply(sigmoid(z),(1 - sigmoid(z)))
In [12]:
print('\nEvaluating sigmoid gradient...\n')
z = np.array([[-1, -0.5, 0, 0.5, 1]]);
g = sigmoidGradient(z);
print('\nComputed gradients for Z:',str(g)+'\n');
print('\nGradient at [z = 0]:',str(g[0,2])+'\n');
Evaluating sigmoid gradient...
Computed gradients for Z: [[0.19661193 0.23500371 0.25       0.23500371 0.19661193]]
Gradient at [z = 0]: 0.25
In [13]:
def feedForward(X, parameters):
   W1 = parameters["W1"]
   W2 = parameters["W2"]
   m = X.shape[0];
   A1 = np.insert(X,0,1,axis=1);          # X.shape = (5000 x 401)
   Z2 = np.dot(A1,W1.T);                  # Z1.shape = (5000 x 401).(401 x 25)= (5000 x 25)
   A2=np.insert(sigmoid(Z2),0,1,axis=1);   # A1.shape = (5000x26)
   Z3= np.dot(A2,W2.T);                   # Z2.shape = (5000 x26) . (26x10) =  (5000 x 10)
   A3=sigmoid(Z3);                        # A2.shape = (5000 x 10)
   cache = {"A1":A1,
          "Z2":Z2,
          "A2":A2,
          "Z3":Z3,
          "A3":A3}
   return A3, cache
In [14]:
def nnCostFunction(A3, Y, parameters,Lambda):
   W1 = parameters["W1"]
   W2 = parameters["W2"]
   m = Y.shape[0]
   y_matrix = np.zeros((m, num_labels)) #y_matrix.shape = (5000x10)
   for i in range(m):
      y_matrix[i, Y[i] - 1] = 1
```

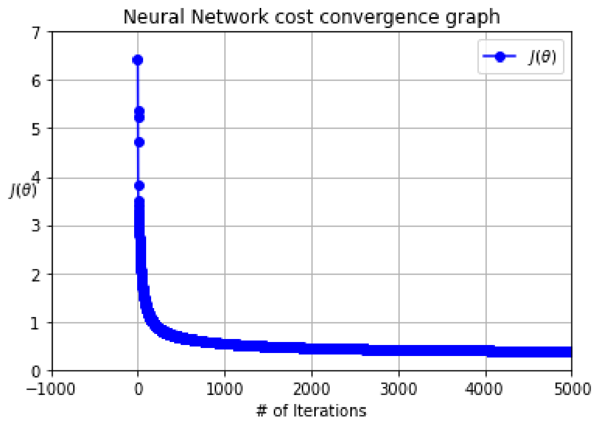
```
    reg_term = (Lambda/(2*m)) * (np.sum(np.sum(np.square(W1[:,1:]))) +
np.sum(np.sum(np.square(W2[:,1:]))));
    cost = ((1/m * np.sum(np.sum((np.multiply(-y_matrix,np.log(A3))- np.multiply((1-
y_matrix),np.log(1-A3)))))) + reg_term);
    return cost
In [15]:
Lambda = 0 #No regularisation
A3, cache = feedForward(X_train,parameters)
cost = nnCostFunction(A3, y_train, parameters, Lambda)
print('\nCost at parameters (loaded from ex4weights): %.6f\n' % (cost));
Cost at parameters (loaded from ex4weights): 0.287629

In [16]:
Lambda = 1 #With regularisation
A3, cache = feedForward(X_train,parameters)
cost = nnCostFunction(A3, y_train, parameters, Lambda)
print('\nCost at parameters (loaded from ex4weights): %.6f\n' % (cost));
Cost at parameters (loaded from ex4weights): 0.383770
In [17]:
def randInitializeWeights(L_in, L_out):
    W = np.zeros((L_out, 1 + L_in));
    epsilon_init = np.sqrt(6)/np.sqrt(L_in + L_out);
    W = np.random.randn(L_out, 1 + L_in) * 2 * epsilon_init - epsilon_init;
    return W
In [18]:
print('\nInitializing Neural Network Parameters ...\n')
initial_W1 = randInitializeWeights(input_layer_size, hidden_layer_size);
initial_W2 = randInitializeWeights(hidden_layer_size, num_labels);
initial_nn_params = np.concatenate((initial_W1.flatten(),initial_W2.flatten()),axis=0).reshape(-1,1)
print('Shape of initial_W1 is: ', initial_W1.shape)
print('Shape of initial_W2 is: ', initial_W2.shape)
print('Shape of nn_params is: ', initial_nn_params.shape)
Initializing Neural Network Parameters ..
Shape of initial_W1 is:  (25, 401)
Shape of initial_W2 is:  (10, 26)
Shape of nn_params is:  (10285, 1)
In [19]:
def backward_propagation(parameters, cache, x, y, Lambda, learning_rate):
    A1 = cache["A1"]
    Z2 = cache["Z2"]
    A2 = cache["A2"]
    Z3 = cache["Z3"]
    A3 = cache["A3"]
    m = x.shape[0];
    y_matrix = np.zeros((m, num_labels))
    for i in range(m):
        y_matrix[i, y[i] - 1] = 1
    d3 = A3-y_matrix;                   #dZ2.shape = (5000x10)
```

```
    u = sigmoid(Z2);                         #u.shape = (5000 x 25)
    sig_grad = np.multiply(u,(1-u))              #(5000 x 25)
    d2 = np.multiply(np.dot(d3, W2[:,1:]),sig_grad) # d2.shape = (5000x25)
    delta1 = np.dot(d2.T,A1)                 # delta1.shape = (25x5000).(5000 x 401)
    delta2 = np.dot(d3.T,A2)                 # delta2.shape = (10x5000).(5000 x 26)
    temp1=W1; #
    temp2=W2;
    temp1[:,0]=0;
    temp2[:,0]=0;
    dW1 = (1/m * delta1) + ((Lambda/m) * temp1)
    dW2 = (1/m * delta2) + ((Lambda/m) * temp2)

    W1 = W1 - learning_rate * dW1
    W2 = W2 - learning_rate * dW2
    parameters = {"W1":W1,
              "W2":W2}
    return parameters
```
In [20]:
```
def nn_model(X,Y, initial_nn_params, input_layer_size, hidden_layer_size, num_labels, Lambda,
learning_rate, print_cost=False):
    parameters = initialize_parameters(initial_nn_params, input_layer_size, hidden_layer_size,
num_labels)
    W1 = parameters["W1"]
    W2 = parameters["W2"]
    m = X.shape[0]
    cost_vec = [] # To store cost values per iterations
    for i in range((m)):
        A3, cache = feedForward(X,parameters)
        cost = nnCostFunction(A3, Y, parameters, Lambda)
        cost_vec.append(cost)
        parameters = backward_propagation(parameters, cache, X, Y, Lambda, learning_rate)
        if print_cost and i % 1000 ==0:
            print("Iteration %i: Cost:%f"%(i,cost))
    return parameters, np.array(cost_vec)
```
In [21]:
```
parameters, cost_vec = nn_model(X_train,y_train,
                initial_nn_params,
                input_layer_size,
                hidden_layer_size,
                num_labels,
                Lambda=1,
                learning_rate=1, print_cost=True)
```
```
Iteration 0: Cost:6.448021
Iteration 1000: Cost:0.543185
Iteration 2000: Cost:0.453916
Iteration 3000: Cost:0.416889
Iteration 4000: Cost:0.396834
```
In [22]:

```
def nnLearningCurve(cost_vec):
    plt.plot(range(len(cost_vec)),cost_vec,'b-o', label= r'${J{(\theta)}}$')
    plt.grid(True)
    plt.title("Neural Network cost convergence graph")
    plt.xlabel('# of Iterations')
    plt.ylabel(r'${J{(\theta)}}$', rotation=1)
    plt.xlim([-1000,len(cost_vec)])
    plt.ylim([0,7])
    plt.legend()
```

In [23]:

```
nnLearningCurve(cost_vec) # calling the function to plot the learning curve
```



In [24]:

```
def predict(X, parameters):

    A3, cache = feedForward(X,parameters)
    prediction = np.argmax(A3, axis=1);
    prediction = prediction + 1;
    return prediction
```

In [25]:

```
prediction = predict(X_train, parameters)
correct = [1 if a == b else 0 for (a, b) in zip(prediction, y_train)]
accuracy = (sum(map(int, correct)) / float(len(correct)))
print('\nTraining Accuracy = %.2f' % (accuracy * 100)+'%\n')
```

Training Accuracy = 98.42%

In [26]:

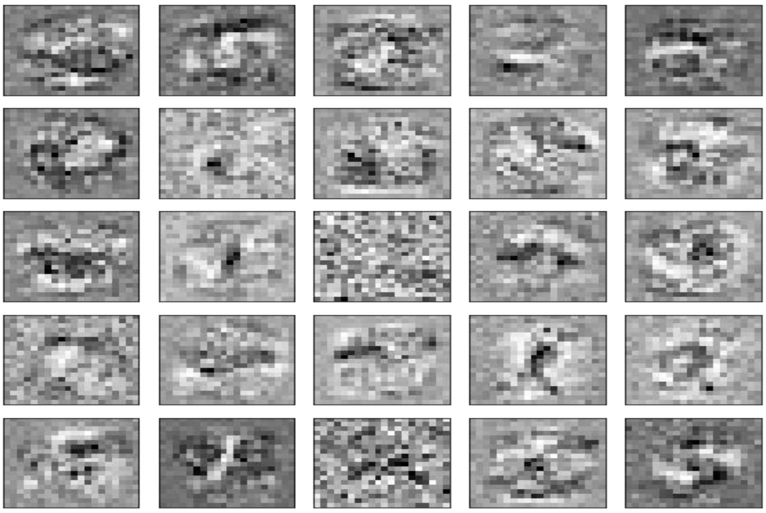
```
W1 = parameters["W1"] # The updated weight from backpropagation algorithm
hidden_W1 = W1[:,1:]  # Creating a temp variable to remove the 0th entry for the bias
print('\ni. Theta1[with the bias]:\n      Rows: %d, columns: %d' % (W1.shape[0], W1.shape[1])+'\n')
print('ii. Theta1[without the bias]:\n      Rows: %d, columns: %d' % (hidden_W1.shape[0], hidden_W1.shape[1]))
```

i. Theta1[with the bias]:
    Rows: 25, columns: 401
ii. Theta1[without the bias]:
    Rows: 25, columns: 400

In [27]:

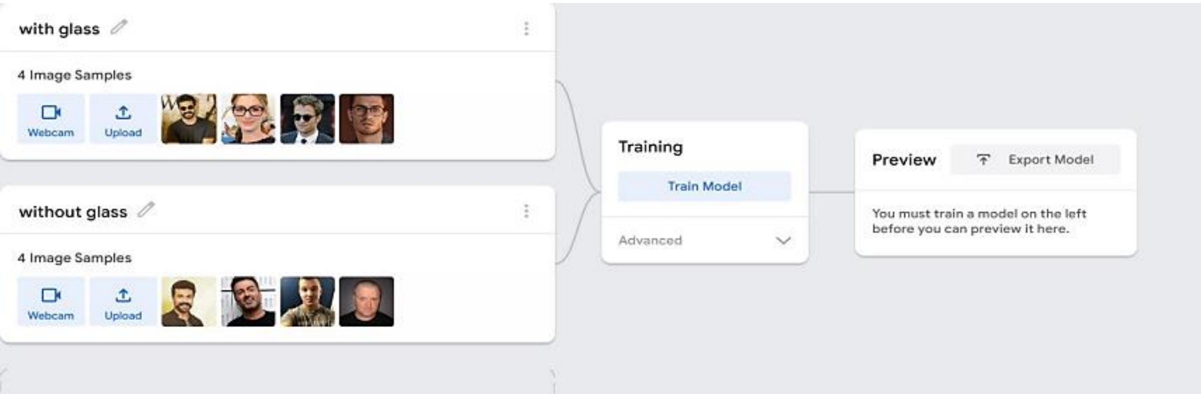
```
def visualizeHiddenLayer(x):
```

```
    fig, ax = plt.subplots(nrows =5, ncols=5,figsize=(8,8),sharex=True, sharey=True)
    ax = ax.flatten()
    (m,n) = x.shape
    print("\nVisualizing hidden layer  ...\n")
    for i in range(m):
        img = x[i,:].reshape(20,20,order="F")
        ax[i].imshow(img, cmap='Greys')
    ax[0].set_xticks([])
    ax[0].set_yticks([])
    plt.tight_layout()
    plt.show()
visualizeHiddenLayer(hidden_W1)
Visualizing hidden layer  ...
```



Teachable Machine - In Browser Object Recognition through Brain.JS
Description :
Teachable Machine is a web-based tool developed by Google's Creative Lab that allows users to create machine learning models without the need for coding or prior experience in machine learning. It is designed to make machine learning more accessible and easier to understand for a wide range of users, including educators, students, and hobbyists.
Process:

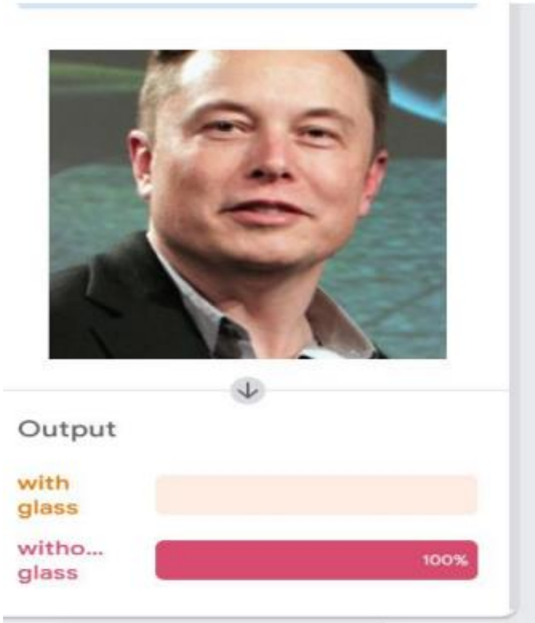
1. Collect and prepare training data: Gather a diverse set of images that include people with and without glasses. Ensure that the images are properly labeled and organized into separate folders for each class (e.g., "With Glasses" and "Without Glasses").
2. Launch Teachable Machine: Open the Teachable Machine website (https://teachablemachine.withgoogle.com/) in your web browser.
3. Choose image input: Since you are working with images, select the "Image Project" option.
4. Collect training examples: Click the "Upload" button and select the appropriate folders containing the labeled images of people with and without glasses. Make sure to capture various angles, poses, and lighting conditions.
5. Label and train the model: Teachable Machine provides an interface for labeling the collected training examples. Assign the "With Glasses" or "Without Glasses" label to each example accordingly. Once labeled, click the "Train Model" button to initiate the training process.
6. Test and refine the model: After training is complete, use the "Test" tab to evaluate the model's performance. Upload new images of people with or without glasses to see how accurately the model classifies them. If the results are not satisfactory, consider collecting more diverse training data or refining the labeling process.
7. Export and use the model: Once you are satisfied with the model's performance, you can export it by clicking the "Export Model" button.



Export your model to use it in projects.

Tensorflow.js Tensorflow Tensorflow Lite

```
// the link to your model provided by Teachable Machine export panel
const URL = "./my_model/";

let model, webcam, labelContainer, maxPredictions;

// Load the image model and setup the webcam
async function init() {
    const modelURL = URL + "model.json";
    const metadataURL = URL + "metadata.json";

    // load the model and metadata
    // Refer to tmImage.loadFromFiles() in the API to support files from a file picker
    // or files from your local hard drive
    // Note: the pose library adds "tmImage" object to your window (window.tmImage)
    model = await tmImage.load(modelURL, metadataURL);
    maxPredictions = model.getTotalClasses();

    // Convenience function to setup a webcam
    const flip = true; // whether to flip the webcam
    webcam = new tmImage.Webcam(200, 200, flip); // width, height, flip
    await webcam.setup(); // request access to the webcam
    await webcam.play();
    window.requestAnimationFrame(loop);

    // append elements to the DOM
    document.getElementById("webcam-container").appendChild(webcam.canvas);
    labelContainer = document.getElementById("label-container");
    for (let i = 0; i < maxPredictions; i++) { // and class labels
        labelContainer.appendChild(document.createElement("div"));
    }
}

async function loop() {
```



Liv.ai - App for Speech recognition and Synthesis through APIs

Audio to text:

Install the required libraries:

```
pip install SpeechRecognition
pip install pydub
pip install ffmpeg-python
```

Import the necessary modules in your Python script:

```
import speech_recognition as sr
from pydub import AudioSegment
import ffmpeg
```

Load and preprocess the audio file:

```
# Provide the path to your audio file
audio_path = "path/to/your/audio/file.wav"
# Load the audio file
audio = AudioSegment.from_wav(audio_path)
# Export the audio as a temporary WAV file
temp_wav_file = "temp.wav"
audio.export(temp_wav_file, format="wav")
```

Perform audio-to-text conversion:

```
# Create a recognizer object
recognizer = sr.Recognizer()
with sr.AudioFile(temp_wav_file) as source:
    audio_data = recognizer.record(source)
    text = recognizer.recognize_google(audio_data)
    print("Transcription: ", text)
```

text to audio conversation:

install the required library:

```
pip install pyttsx3
```

Import the necessary module in your Python script:

```
import pyttsx3
```

Initialize the text-to-speech engine:

```
engine = pyttsx3.init()
```

Set the properties for the speech output (optional):

```
# Set the voice
voices = engine.getProperty('voices')
engine.setProperty('voice', voices[0].id)  # Change the index to select a different voice
rate = engine.getProperty('rate')
engine.setProperty('rate', 150)  # Adjust the value as desired
```

Convert text to speech:

```
text = "Hello, how are you?"
engine.say(text)
engine.runAndWait()
```

Save speech to an audio file (optional):

```
output_file = "output.wav"
engine.save_to_file(text, output_file)
engine.runAndWait()
```

output:

audio is successfully generated

Build a Convolutional Neural Network for Cat vs Dog Image Classification

```
import numpy as np
import matplotlib.pyplot as plt
import os
import cv2
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
```

Next, we'll define some constants for our model:

```
IMG_SIZE = 100
NUM_CLASSES = 2
```

Now, let's load and preprocess the data:

```
def load_data():
  data = []
  labels = []
  cat_path = "path/to/cat/images"  # Update with the actual path to your cat images
  dog_path = "path/to/dog/images"  # Update with the actual path to your dog images
  # Load cat images
  for img in os.listdir(cat_path):
    img_array = cv2.imread(os.path.join(cat_path, img))
    img_array = cv2.resize(img_array, (IMG_SIZE, IMG_SIZE))
    data.append(img_array)
    labels.append("cat")
  # Load dog images
  for img in os.listdir(dog_path):
    img_array = cv2.imread(os.path.join(dog_path, img))
    img_array = cv2.resize(img_array, (IMG_SIZE, IMG_SIZE))
    data.append(img_array)
    labels.append("dog")
  # Convert the labels to categorical
  le = LabelEncoder()
  labels = le.fit_transform(labels)
  labels = to_categorical(labels, num_classes=NUM_CLASSES)

  return np.array(data), np.array(labels)

# Load the data
data, labels = load_data()

# Split the data into training and testing sets
train_data, test_data, train_labels, test_labels = train_test_split(data, labels, test_size=0.2,
random_state=42)
```

Now, let's build the CNN model:

```
model = Sequential()
# Convolutional layers
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(IMG_SIZE, IMG_SIZE, 3)))
```

```
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(128, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))

# Flatten the output from convolutional layers
model.add(Flatten())

# Fully connected layers
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(NUM_CLASSES, activation='softmax'))

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

Finally, let's train and evaluate the model:

```
# Train the model
model.fit(train_data, train_labels, epochs=10, batch_size=32)

# Evaluate the model
loss, accuracy = model.evaluate(test_data, test_labels)
print("Test Loss:", loss)
print("Test Accuracy:", accuracy)
```

Building a Chatbot using AWS Lex, Pandora bots

Process:

1. **Create a Pandora bots account:** Visit the Pandora bots website (https://www.pandorabots.com/) and create an account if you don't have one already.
2. **Create a bot:** Log in to your Pandora bots account and create a new bot. Give it a name and configure any desired settings.
3. **Design your bot's conversational flow:** Use the Pandora bots platform to design your bot's conversational flow. Define the intents (user inputs) and corresponding responses that your bot should handle.
4. **Deploy your bot:** Once you've designed your bot, deploy it on the Pandora bots platform. This will make your bot accessible via a unique URL.
5. **Create an AWS Lex bot:** Go to the AWS Management Console and navigate to AWS Lex. Create a new bot by providing a name, description, and selecting the desired language.
6. **Define the intents in AWS Lex:** In AWS Lex, define the intents that align with the conversational flow you created in Pandora bots. Define sample utterances for each intent and map them to the corresponding responses from your Pandora bots bot.
7. **Configure the AWS Lex bot:** Configure the AWS Lex bot by setting up channels, prompts, and other parameters as needed.
8. **Test and integrate your chatbot:** Test your chatbot in the AWS Lex console to ensure it behaves as expected. Once you're satisfied, you can integrate your chatbot into your desired application or platform by using the AWS Lex SDKs or APIs.

Output: