Basic Instructions

1. Enter your Name and UID in the provided space.
2. Do the assignment in the notebook itself.
3. you are free to use Google Colab.
4. Upload to Google Drive.
5. Now enter the Google Drive link in the provided space. (you can do this by opening the iPython notebook uploaded using Google Collab).
6. Submit the assignment to Gradescope.

Note - You are NOT supposed to use Pytorch library for this assignment. You will receive no credit anywhere that Pytorch (or TF, caffe, etc.) is used. Additionally, we don't use cuda here, so you can use the CPU runtime in Colab -- no need for GPU.

↳ *1 cell hidden*

Part 1: Building a 2-layer Neural Network

In the first part, you will implement all the functions required to build a two layer neural network. In the next part, you will use these functions for image and text classification. Provide your code at the appropriate placeholders.

Packages

[ ] ↳ *1 cell hidden*

1. Layer Initialization

Exercise: Create and initialize the parameters of the 2-layer neural network. Use random initialization for the weight matrices (use `0.01*np.random.randn()`) and zero initialization for the biases.

```
def initialize_parameters(n_x, n_h, n_y):
    """
    Argument:
    n_x -- size of the input layer
```

```
    n_y -- size of the output layer

    Returns:
    parameters -- python dictionary containing your parameters:
                    W1 -- weight matrix of shape (n_h, n_x)
                    b1 -- bias vector of shape (n_h, 1)
                    W2 -- weight matrix of shape (n_y, n_h)
                    b2 -- bias vector of shape (n_y, 1)
    """

    np.random.seed(1)

    ### START CODE HERE ### (≈ 4 lines of code)

    W1 = 0.01*np.random.randn(n_h, n_x)
    b1 = np.zeros((n_h, 1))
    W2 = 0.01*np.random.randn(n_y, n_h)
    b2 = np.zeros((n_y, 1))

    ### END CODE HERE ###

    assert(W1.shape == (n_h, n_x))
    assert(b1.shape == (n_h, 1))
    assert(W2.shape == (n_y, n_h))
    assert(b2.shape == (n_y, 1))

    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}

    return parameters


parameters = initialize_parameters(3,2,1)
print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
```

```
W1 = [[ 0.01624345 -0.00611756 -0.00528172]
 [-0.01072969  0.00865408 -0.02301539]]
b1 = [[0.]
 [0.]]
W2 = [[ 0.01744812 -0.00761207]]
b2 = [[0.]]
```

Expected output:

W1 [[0.01624345 -0.00611756 -0.00528172] [-0.01072969 0.00865408 -0.02301539]]

b1 [[0.] [0.]]

W2 [[0.01744812 -0.00761207]]

b2 [[0.]]

2. Forward Propagation

Now that you have initialized your parameters, you will do the forward propagation module. You will start by implementing some basic functions that you will use later when implementing the model. You will complete three functions in this order:

- LINEAR
- LINEAR -> ACTIVATION where ACTIVATION will be either ReLU or Sigmoid.

The linear module computes the following equation:

$$Z = WA + b \tag{4}$$

2.1 Exercise - Build the linear part of forward propagation.

```
def linear_forward(A, W, b):
    """
    Implement the linear part of a layer's forward propagation.

    Arguments:
    A -- activations from previous layer (or input data): (size of previous layer, number c
    W -- weights matrix: numpy array of shape (size of current layer, size of previous laye
    b -- bias vector, numpy array of shape (size of the current layer, 1)

    Returns:
    Z -- the input of the activation function, also called pre-activation parameter
    cache -- a python dictionary containing "A", "W" and "b" ; stored for computing the bac
    """

    ### START CODE HERE ### (≈ 1 line of code)

    Z = W @ A + b

    ### END CODE HERE ###

    assert(Z.shape == (W.shape[0], A.shape[1]))
    cache = (A, W, b)

    return Z, cache
```

```
np.random.seed(1)
```

```
A = np.random.randn(3,2)
W = np.random.randn(1,3)
b = np.random.randn(1,1)

Z, linear_cache = linear_forward(A, W, b)
print("Z = " + str(Z))
```

```
Z = [[ 3.26295337 -1.23429987]]
```

Expected output:

Z	[[3.26295337 -1.23429987]]

2.2 - Linear-Activation Forward

In this notebook, you will use two activation functions:

- **Sigmoid**: $\sigma(Z) = \sigma(WA + b) = \frac{1}{1 + e^{-(WA+b)}}$. Write the code for the `sigmoid` function. This function returns **two** items: the activation value " `a` " and a " `cache` " that contains " `Z` " (it's what we will feed in to the corresponding backward function). To use it you could just call:

```
A, activation_cache = sigmoid(Z)
```

- **ReLU**: The mathematical formula for ReLu is $A = RELU(Z) = max(0, Z)$. Write the code for the `relu` function. This function returns **two** items: the activation value " `A` " and a " `cache` " that contains " `Z` " (it's what we will feed in to the corresponding backward function). To use it you could just call:

```
A, activation_cache = relu(Z)
```

Exercise:

- Implement the activation functions
- Build the linear activation part of forward propagation. Mathematical relation is: $A = g(Z) = g(WA_{prev} + b)$

[ ] ↳ 4 cells hidden

3. Loss function

Now you will implement forward and backward propagation. You need to compute the loss

Now you will implement forward and backward propagation. You need to compute the loss, because you want to check if your model is actually learning.

Exercise: Compute the cross-entropy loss J, using the following formula:

$$-\frac{1}{m} \sum_{i=1}^{m} (y^{(i)} \log\left(a^{(i)}\right) + (1-y^{(i)}) \log\left(1-a^{(i)}\right)) \tag{7}$$

[ ] ↳ 3 cells hidden

4. Backward propagation module

Just like with forward propagation, you will implement helper functions for backpropagation. Remember that back propagation is used to calculate the gradient of the loss function with respect to the parameters.

Now, similar to forward propagation, you are going to build the backward propagation in two steps:

- LINEAR backward
- LINEAR -> ACTIVATION backward where ACTIVATION computes the derivative of either the ReLU or sigmoid activation

Following are the relationships -

$$\begin{aligned} dA_{prev} &= W^T dZ \\ dW &= dZ A_{prev}^T \\ db &= \sum_{i=1}^{m} dZ^{(i)} \ \text{ where m is the number of samples} \\ &= dZ\hat{I} \ \text{ where } \hat{I} \text{ is a column vector of size(m,1) with all entries 1} \end{aligned}$$

4.1 - Linear backward

[ ] ↳ 9 cells hidden

5. Update Parameters

In this section you will update the parameters of the model, using gradient descent:

$$W^{[1]} = W^{[1]} - \alpha \, dW^{[1]} \tag{16}$$
$$b^{[1]} = b^{[1]} - \alpha \, db^{[1]} \tag{17}$$
$$W^{[2]} = W^{[2]} - \alpha \, dW^{[2} \tag{16}$$
$$b^{[2]} = b^{[2]} - \alpha \, db^{[2]} \tag{17}$$

where α is the learning rate. After computing the updated parameters, store them in the where α is the learning rate. After computing the updated parameters, store them in the parameters dictionary.

Exercise: Implement `update_parameters()` to update your parameters using gradient descent.

Instructions: Update parameters using gradient descent.

[ ] ↳ 3 cells hidden

Conclusion

Congrats on implementing all the functions required for building a deep neural network!

If this was challenging, that means you're learning. Also, the next part of the assignment is easier.

Part 2: Image Classification with a 2-layer Neural Network

In the next part you will put all these together to build a two-layer neural networks for image classification.

```
%matplotlib inline
plt.rcParams['figure.figsize'] = (5.0, 4.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

%load_ext autoreload
%autoreload 2

np.random.seed(1)
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

Dataset

Problem Statement: You are given a dataset ("data/train_catvnoncat.h5", "data/test_catvnoncat.h5") containing: - a training set of m_train images labelled as cat (1) or non-cat (0) - a test set of m_test images labelled as cat and non-cat - each image is of shape (num_px, num_px, 3) where 3 is for the 3 channels (RGB).

Let's get more familiar with the dataset. Load the data by completing the function and run the cell below.

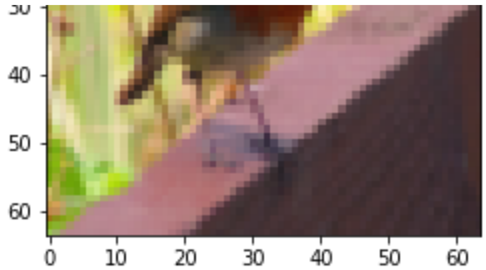
cell below.

```
def load_data(train_file, test_file):
    # Load the training data
    train_dataset = h5py.File(train_file, 'r')

    # Separate features(x) and labels(y) for training set
    train_set_x_orig = train_dataset['train_set_x'][()]
    train_set_y_orig = train_dataset['train_set_y'][()]

    # Load the test data
    test_dataset = h5py.File(test_file, 'r')

    # Separate features(x) and labels(y) for training set
    test_set_x_orig = test_dataset['test_set_x'][()]
    test_set_y_orig = test_dataset['test_set_y'][()]

    classes = np.array(test_dataset["list_classes"][:]) # the list of classes

    train_set_y_orig = train_set_y_orig.reshape((1, train_set_y_orig.shape[0]))
    test_set_y_orig = test_set_y_orig.reshape((1, test_set_y_orig.shape[0]))

    return train_set_x_orig, train_set_y_orig, test_set_x_orig, test_set_y_orig, classes


train_file="/data/train_catvnoncat.h5"
test_file="/data/test_catvnoncat.h5"

#train_file="data/train_catvnoncat.h5"
#test_file="data/test_catvnoncat.h5"

train_x_orig, train_y, test_x_orig, test_y, classes = load_data(train_file, test_file)
```

The following code will show you an image in the dataset. Feel free to change the index and re-run the cell multiple times to see other images.

```
# Example of a picture
index = 10
plt.imshow(train_x_orig[index])
print ("y = " + str(train_y[0,index]) + ". It's a " + classes[train_y[0,index]].decode("ut
```

```
y = 0. It's a non-cat picture.
```




```
# Explore your dataset
m_train = train_x_orig.shape[0]
num_px = train_x_orig.shape[1]
m_test = test_x_orig.shape[0]

print ("Number of training examples: " + str(m_train))
print ("Number of testing examples: " + str(m_test))
print ("Each image is of size: (" + str(num_px) + ", " + str(num_px) + ", 3)")
print ("train_x_orig shape: " + str(train_x_orig.shape))
print ("train_y shape: " + str(train_y.shape))
print ("test_x_orig shape: " + str(test_x_orig.shape))
print ("test_y shape: " + str(test_y.shape))
```

```
Number of training examples: 209
Number of testing examples: 50
Each image is of size: (64, 64, 3)
train_x_orig shape: (209, 64, 64, 3)
train_y shape: (1, 209)
test_x_orig shape: (50, 64, 64, 3)
test_y shape: (1, 50)
```

As usual, you reshape and standardize the images before feeding them to the network.

Figure 1: Image to vector conversion.

```
# Reshape the training and test examples
train_x_flatten = train_x_orig.reshape(train_x_orig.shape[0], -1).T   # The "-1" makes resh
test_x_flatten = test_x_orig.reshape(test_x_orig.shape[0], -1).T

# Standardize data to have feature values between 0 and 1.
train_x = train_x_flatten/255.
test_x = test_x_flatten/255.

print ("train_x's shape: " + str(train_x.shape))
print ("test_x's shape: " + str(test_x.shape))
```

```
train_x's shape: (12288, 209)
test_x's shape: (12288, 50)
```

Architecture

Now that you are familiar with the dataset, it is time to build a deep neural network to distinguish cat images from non-cat images.

2-layer neural network

Figure 2: 2-layer neural network.
The model can be summarized as: ***INPUT -> LINEAR -> RELU -> LINEAR -> SIGMOID -> OUTPUT***.

Detailed Architecture of figure 2:

- The input is a (64,64,3) image which is flattened to a vector of size $(12288, 1)$.
- The corresponding vector: $[x_0, x_1, \ldots, x_{12287}]^T$ is then multiplied by the weight matrix $W^{[1]}$ of size $(n^{[1]}, 12288)$.
- You then add a bias term and take its relu to get the following vector: $[a_0^{[1]}, a_1^{[1]}, \ldots, a_{n^{[1]}-1}^{[1]}]^T$.
- You multiply the resulting vector by $W^{[2]}$ and add your intercept (bias).
- Finally, you take the sigmoid of the result. If it is greater than 0.5, you classify it to be a cat.

General methodology

As usual you will follow the Deep Learning methodology to build the model: 1. Initialize parameters / Define hyperparameters 2. Loop for num_iterations: a. Forward propagation b. Compute loss function c. Backward propagation d. Update parameters (using parameters, and grads from backprop) 4. Use trained parameters to predict labels

Let's now implement those the model!

```
### CONSTANTS DEFINING THE MODEL ####
n_x = 12288     # num_px * num_px * 3
n_h = 7
n_y = 1
layers_dims = (n_x, n_h, n_y)
```

6. Training a Neural Network

Exercise: Use the helper functions you have implemented in the previous assignment to build a 2-layer neural network with the following structure: *LINEAR -> RELU -> LINEAR -> SIGMOID*. The functions you may need and their inputs are:

```
def initialize_parameters(n_x, n_h, n_y):
    ...
    return parameters
def linear_activation_forward(A_prev, W, b, activation):
    ...
    return A, cache
def compute_loss(AL, Y):
    ...
    return loss
def linear_activation_backward(dA, cache, activation):
    ...
    return dA_prev, dW, db
def update_parameters(parameters, grads, learning_rate):
    ...
    return parameters
```

```
def two_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_iterations = 3000,
                    print_loss=False, return_loss=False, params=None):
    """
    Implements a two-layer neural network: LINEAR->RELU->LINEAR->SIGMOID.

    Arguments:
    X -- input data, of shape (n_x, number of examples)
    Y -- true "label" vector (containing 0 if cat, 1 if non-cat), of shape (1, number of ex
    layers_dims -- dimensions of the layers (n_x, n_h, n_y)
    num_iterations -- number of iterations of the optimization loop
    learning_rate -- learning rate of the gradient descent update rule
    print_loss -- If set to True, this will print the loss every 100 iterations

    Returns:
    parameters -- a dictionary containing W1, W2, b1, and b2
    """

    np.random.seed(1)
    grads = {}
    losses = []                              # to keep track of the loss
    m = X.shape[1]                           # number of examples
    (n_x, n_h, n_y) = layers_dims

    # Initialize parameters dictionary, by calling one of the functions you'd previously in
    ### START CODE HERE ### (≈ 1 line of code)

    parameters = params or initialize_parameters(n_x, n_h, n_y)

    ### END CODE HERE ###
```

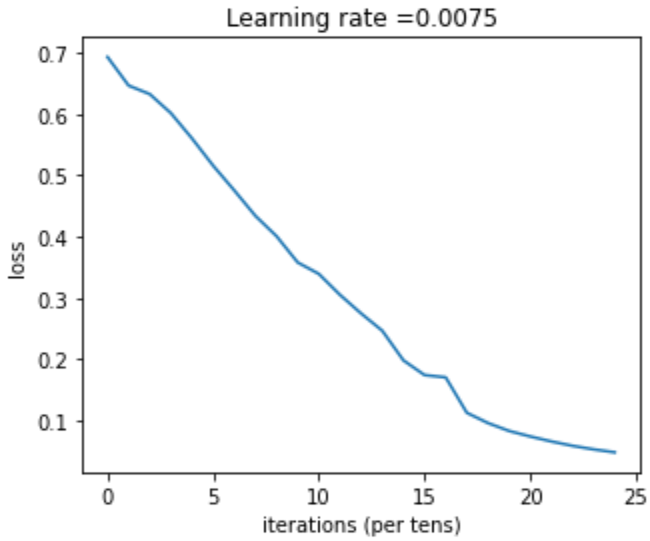
```
# Get W1, b1, W2 and b2 from the dictionary parameters.
W1 = parameters["W1"]
b1 = parameters["b1"]
W2 = parameters["W2"]
b2 = parameters["b2"]

# Loop (gradient descent)

for i in range(0, num_iterations):

    # Forward propagation: LINEAR -> RELU -> LINEAR -> SIGMOID. Inputs: "X, W1, b1, W2,
    ### START CODE HERE ### (≈ 2 lines of code)

    A1, cache1 = linear_activation_forward(X, W1, b1, "relu")
    A2, cache2 = linear_activation_forward(A1, W2, b2, "sigmoid")

    ### END CODE HERE ###

    # Compute loss
    ### START CODE HERE ### (≈ 1 line of code)

    loss = compute_loss(A2, Y)

    ### END CODE HERE ###

    # Initializing backward propagation
    dA2 = - (np.divide(Y, A2) - np.divide(1 - Y, 1 - A2)) / m

    # Backward propagation. Inputs: "dA2, cache2, cache1". Outputs: "dA1, dW2, db2; als
    ### START CODE HERE ### (≈ 2 lines of code)

    dA1, dW2, db2 = linear_activation_backward(dA2, cache2, "sigmoid")
    dA0, dW1, db1 = linear_activation_backward(dA1, cache1, "relu")

    ### END CODE HERE ###

    # Set grads['dWl'] to dW1, grads['db1'] to db1, grads['dW2'] to dW2, grads['db2'] t
    ### START CODE HERE ### (≈ 4 lines of code)

    grads['dW1'] = dW1
    grads['dW2'] = dW2
    grads['db1'] = db1
    grads['db2'] = db2

    ### END CODE HERE ###


    # Update parameters.
    ### START CODE HERE ### (approx. 1 line of code)
```

```
        parameters = update_parameters(parameters, grads, learning_rate)

        ### END CODE HERE ###

        # Retrieve W1, b1, W2, b2 from parameters
        W1 = parameters["W1"]
        b1 = parameters["b1"]
        W2 = parameters["W2"]
        b2 = parameters["b2"]

        # Print the loss every 100 training example
        if i % 100 == 0 and print_loss:
            print("Loss after iteration {}: {}".format(i, np.squeeze(loss)))
        if i % 10 == 0:
            losses.append(loss)

    # plot the loss
    if print_loss:
        plt.plot(np.squeeze(losses))
        plt.ylabel('loss')
        plt.xlabel('iterations (per tens)')
        plt.title("Learning rate =" + str(learning_rate))
        plt.show()

    # I've added some code to make it easier to see the outputs
    if return_loss:
        return parameters, np.squeeze(losses)
    else:
        return parameters
```

```
parameters = two_layer_model(train_x, train_y, layers_dims = (n_x, n_h, n_y), num_iteratio
```

```
Loss after iteration 0: 0.693049735659989
Loss after iteration 100: 0.6464320953428849
Loss after iteration 200: 0.6325140647912677
Loss after iteration 300: 0.6015024920354665
Loss after iteration 400: 0.5601966311605747
Loss after iteration 500: 0.5158304772764729
Loss after iteration 600: 0.4754901313943325
Loss after iteration 700: 0.4339163151225749
Loss after iteration 800: 0.4007977536203888
Loss after iteration 900: 0.3580705011323798
Loss after iteration 1000: 0.33942815383664116
Loss after iteration 1100: 0.30527536361962637
Loss after iteration 1200: 0.2749137728213017
Loss after iteration 1300: 0.24681768210614824
Loss after iteration 1400: 0.19850735037466094
Loss after iteration 1500: 0.17448318112556632
Loss after iteration 1600: 0.17080762978097142
Loss after iteration 1700: 0.11306524562164692
Loss after iteration 1800: 0.09629426845937143
```

```
Loss after iteration 1900: 0.08342617959726858
Loss after iteration 2000: 0.0743907870431908
Loss after iteration 2100: 0.0663074813226793
Loss after iteration 2200: 0.05919329501038164
Loss after iteration 2300: 0.053361403485605544
Loss after iteration 2400: 0.04855478562877018
```



Expected Output:

Loss after iteration 0	0.6930497356599888
Loss after iteration 100	0.6464320953428849
...	...
Loss after iteration 2400	0.048554785628770206

Good thing you built a vectorized implementation! Otherwise it might have taken 10 times longer to train this.

Now, you can use the trained parameters to classify images from the dataset.

7. Inference for Your Neural Network

Exercise:

- Implement the forward function
- Implement the predict function below to make prediction on test_images

```
def two_layer_forward(X, parameters):
    """
    Implement forward propagation for the [LINEAR->RELU]*(L-1)->LINEAR->SIGMOID computatior

    Arguments:
    X -- data, numpy array of shape (input size, number of examples)
    parameters -- output of initialize_parameters_deep()
```

```
    Returns:
    AL -- last post-activation value
    caches -- list of caches containing:
                every cache of linear_relu_forward() (there are L-1 of them, indexed from 0
                the cache of linear_sigmoid_forward() (there is one, indexed L-1)
    """

    caches = []
    A = X

    # Implement LINEAR -> RELU. Add "cache" to the "caches" list.
    ### START CODE HERE ### (approx. 3 line of code)

    A1, cache1 = linear_activation_forward(X,  parameters["W1"], parameters["b1"], "relu")
    caches.append(cache1)

    ### END CODE HERE ###

    # Implement LINEAR -> SIGMOID. Add "cache" to the "caches" list.
    ### START CODE HERE ### (approx. 3 line of code)

    A2, cache2 = linear_activation_forward(A1, parameters["W2"], parameters["b2"], "sigmoid
    caches.append(cache2)

    ### END CODE HERE ###

    assert(A2.shape == (1,X.shape[1]))

    return A2, caches


def predict(X, y, parameters, print_accuracy=True):
    """
    This function is used to predict the results of a  L-layer neural network.

    Arguments:
    X -- data set of examples you would like to label
    parameters -- parameters of the trained model

    Returns:
    p -- predictions for the given dataset X
    """

    m = X.shape[1]
    n = len(parameters) // 2 # number of layers in the neural network
    p = np.zeros((1,m))

    # Forward propagation
    ### START CODE HERE ### (≈ 1 lines of code)
```

```
    probas, caches =  two_layer_forward(X, parameters)

    ### END CODE HERE ###

    # convert probas to 0/1 predictions
    for i in range(0, probas.shape[1]):
        ### START CODE HERE ### (≈ 4 lines of code)

        p[0,i] = int(probas[0,i] > 0.5)

        ### END CODE HERE ###

    accuracy = str(np.sum((p == y)/m))
    if print_accuracy:
      print("Accuracy:", accuracy)
      return p
    else:
      return p, accuracy
```

```
predictions_train = predict(train_x, train_y, parameters)
```

```
Accuracy: 0.9999999999999998
```

```
predictions_test = predict(test_x, test_y, parameters)
```

```
Accuracy: 0.72
```

8. Explore and Explain Hyperparameters

Exercise: Identify the hyperparameters in the model and for each hyperparameter

- Briefly explain its role
- Explore a range of values and describe their impact on (a) training loss and (b) test accuracy
- Report the best hyperparameter value found.

Note: Provide your results and explanations in the answer for this question.

```
# Functions used in hyperparameter exploration
import matplotlib.pylab as pl

def training_loss_test_accuracy(learning_rate = 0.0075,
                                num_iterations = 2500,
                                num_hidden = n_h):
    parameters, losses = two_layer_model(train_x, train_y,
                                         layers_dims = (n_x, num_hidden, n_y),
                                         learning_rate = learning_rate,
```

```
                                        learning_rate = learning_rate,
                                        num_iterations = num_iterations,
                                        print_loss=False, return_loss=True)
    _, test_accuracy = predict(test_x, test_y, parameters, print_accuracy=False)
    return losses, test_accuracy

def plot_losses(hyperparam_losses, hyperparam_name, format_func, figsize=(8,8)):
  fig = plt.figure(figsize=figsize)
  ax = plt.subplot(111)

  # Plot different losses over iterations with different lines
  colors = pl.cm.jet(np.linspace(0,1,len(hyperparam_losses)))
  for color, (hyperparam, losses) in zip(colors, hyperparam_losses.items()):
      ax.plot(losses, label=format_func(hyperparam), color=color)

  plt.ylabel('loss')
  plt.xlabel('iterations (per tens)')
  plt.title(f"Losses over various {hyperparam_name}s")

  # Shrink current axis by 20%
  box = ax.get_position()
  ax.set_position([box.x0, box.y0, box.width * 0.8, box.height])

  # Put a legend to the right of the current axis
  ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))

def plot_accuracy(accuracy_dict, hyperparam_name):
  xa, ya = zip(*accuracy_dict.items())
  ya = list(map(float,ya))

  plt.plot(xa, ya)
  plt.xlabel(hyperparam_name); plt.ylabel("accuracy after 2500 iterations")
  plt.title(f"Test data accuracy over various {hyperparam_name}s")
```

NOTE: Because it would be too computationally intensive to do a search through the whole 3-dimensional space for the optimal parameters, I will optimize each of them invidually, fixing the other parameters to the values used above.

1. Learning rate

Determines how much the model changes the parameters in each iteration (in response to a non-zero gradient).

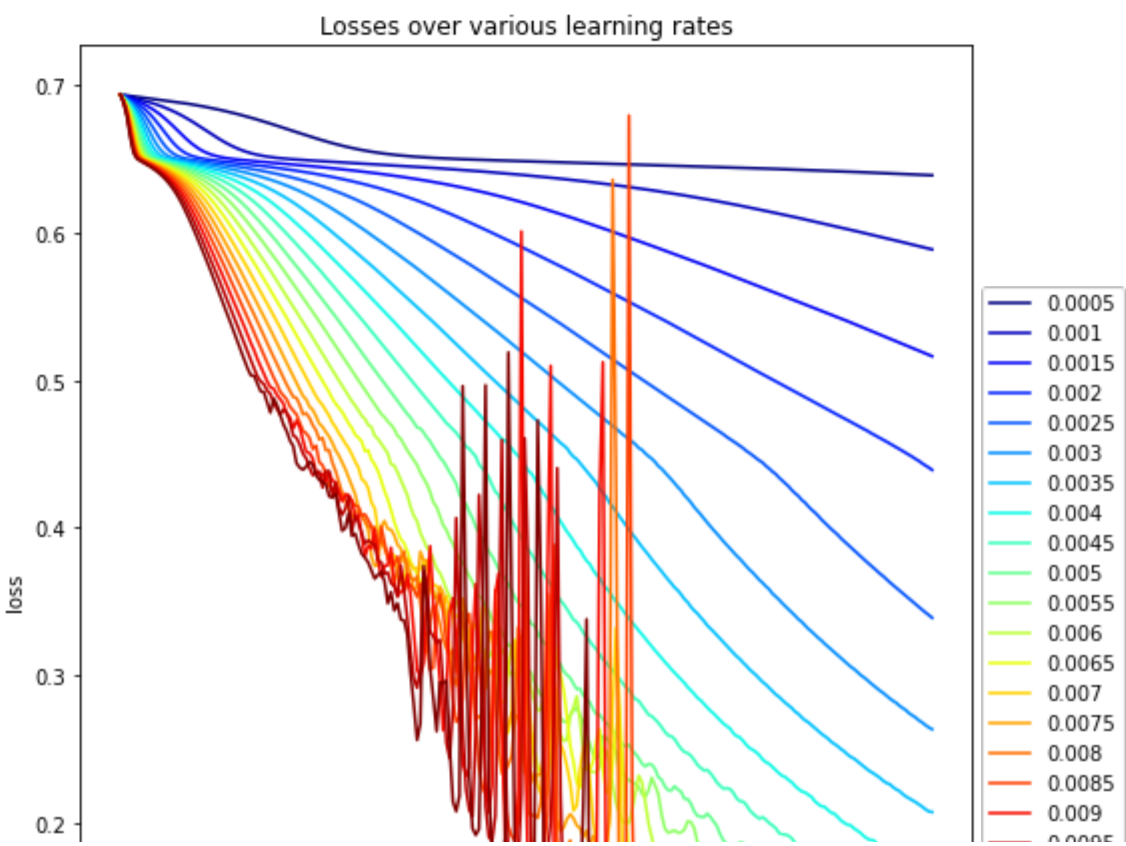
```
learning_rate_losses = {}
learning_rate_accuracy = {}

for learning_rate in np.linspace(0.0005, 0.0100, 20):
```

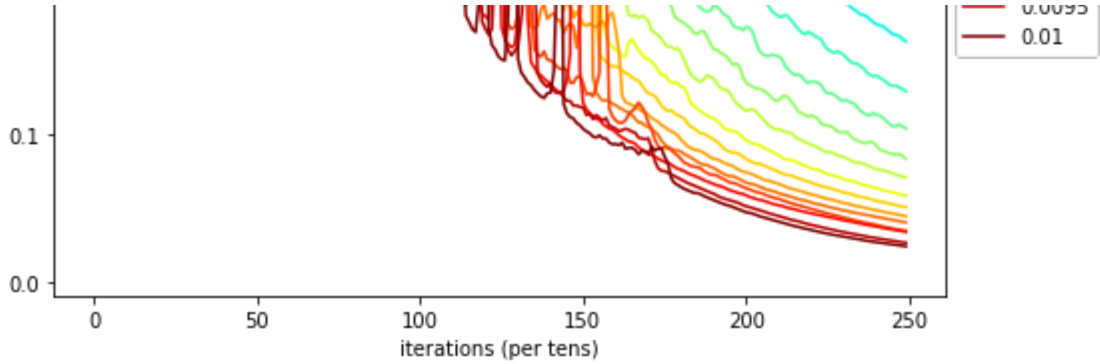
```
    losses, test_accuracy = training_loss_test_accuracy(
        learning_rate=learning_rate)
    print(f"Trained with learning rate {learning_rate}")
    learning_rate_losses[learning_rate] = losses
    learning_rate_accuracy[learning_rate] = test_accuracy
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-265-4a31e7610b62> in <module>
      3 
      4 for learning_rate in np.linspace(0.0005, 0.0100, 20):
----> 5     losses, test_accuracy = training_loss_test_accuracy(
      6         learning_rate=learning_rate)
      7     print(f"Trained with learning rate {learning_rate}")

                                 3 frames

<ipython-input-251-8f729b79c37c> in linear_backward(dZ, cache)
     19     ### START CODE HERE ### (≈ 3 lines of code)
     20 
---> 21     dA_prev = W.T @ dZ
     22     dW = dZ @ A_prev.T
     23     db = dZ @ np.ones((m,1))

KeyboardInterrupt: 
```

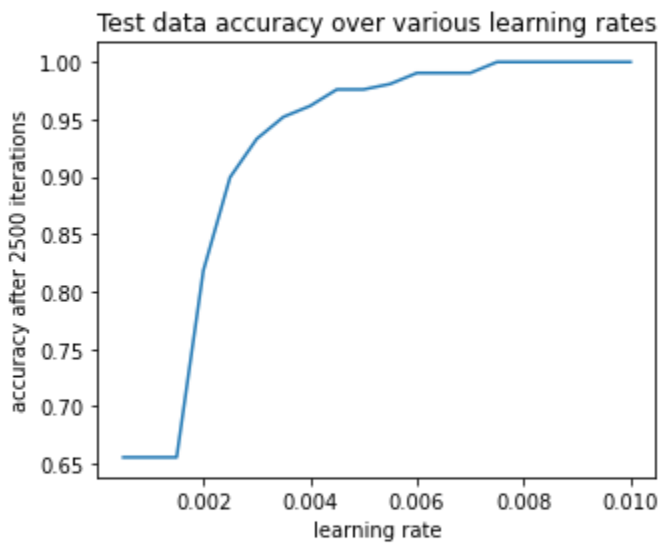
SEARCH STACK OVERFLOW

```
plot_losses(learning_rate_losses, "learning rate", format_func = lambda learning_rate: str(
```






```
plot_accuracy(learning_rate_accuracy, hyperparam_name="learning rate")
```

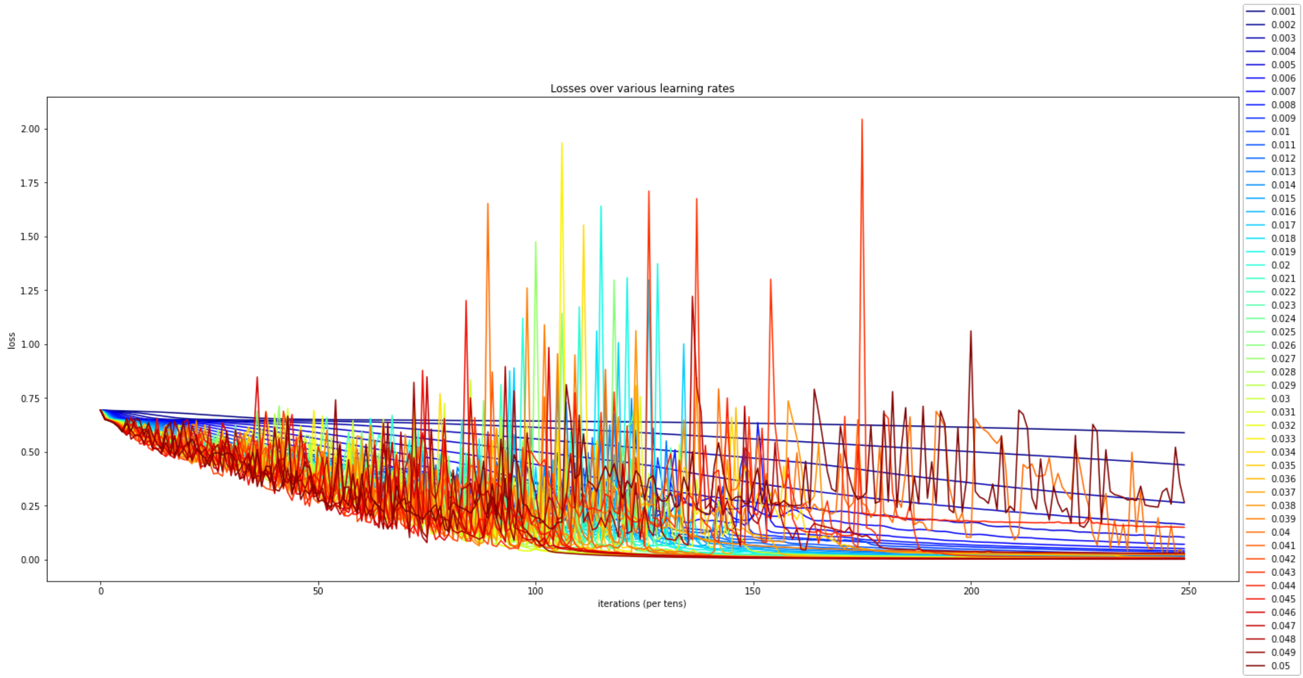


The second graph shows that the accuracy is increasing as the learning rate increase (because the weights can converge within the limited time frame), but the first graph shows that the loss is becoming increasingly unstable as the learning rate increases.

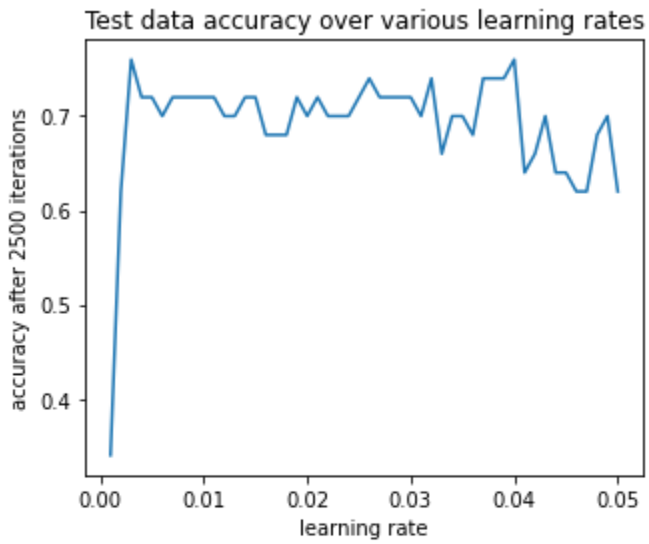
Let's see what happens if we look at larger learning rates:

```
learning_rate_losses2 = {}
learning_rate_accuracy2 = {}

learning_rate_step = 0.001
num_learning_rate_steps = 50
learning_rate_set = np.linspace(learning_rate_step,
                                learning_rate_step * num_learning_rate_steps,
                                num_learning_rate_steps)

for learning_rate in learning_rate_set:
    losses, test_accuracy = training_loss_test_accuracy(
        learning_rate=learning_rate)
    #print(f"Trained with learning rate {learning_rate}")
    learning_rate_losses2[learning_rate] = losses
```

```
    learning_rate_losses2[learning_rate] = losses
    learning_rate_accuracy2[learning_rate] = test_accuracy


plot_losses(learning_rate_losses2, "learning rate",
            format_func = lambda learning_rate: str(round(learning_rate,4)),
            figsize=(30,10))
```




```
plot_accuracy(learning_rate_accuracy2, "learning rate")
```



When the learning rate starts to get near 0.01, instability starts to occur over iterations, resulting in *lower* testing accuracy. Based on these two charts, I conjecture the best learning rate is around the given rate, 0.0075.

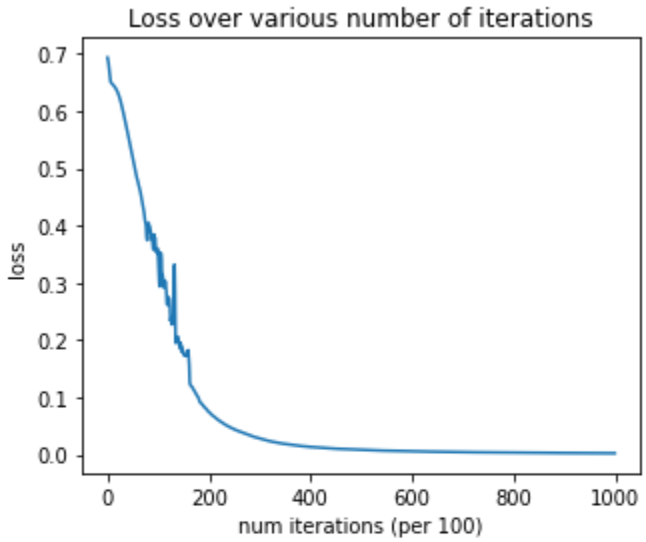
2. Number of iterations

How many cycles the parameters are updated for.

```
num_iterations_losses = {}
num_iterations_accuracy = {}

iteration_step = 100
num_iteration_steps = 100
iterations_set = np.linspace(iteration_step, iteration_step * num_iteration_steps, iteratio

# Repeatedly do {iteration_step} iterations and store the loss / accuracy
params = initialize_parameters(n_x, n_h, n_y)
for num_iterations in iterations_set:
    params, losses = two_layer_model(train_x, train_y,
                                    layers_dims = (n_x, n_h, n_y),
                                    num_iterations = iteration_step,
                                    print_loss=False, return_loss=True,
                                    params=params)
    test_accuracy = predict(test_x, test_y, params, print_accuracy=False)
    #print(f"Trained with num iterations {num_iterations}")
    num_iterations_losses[num_iterations] = losses
    num_iterations_accuracy[num_iterations] = test_accuracy


all_iteration_losses = np.concatenate(list(num_iterations_losses.values()))
plt.plot(all_iteration_losses)
```

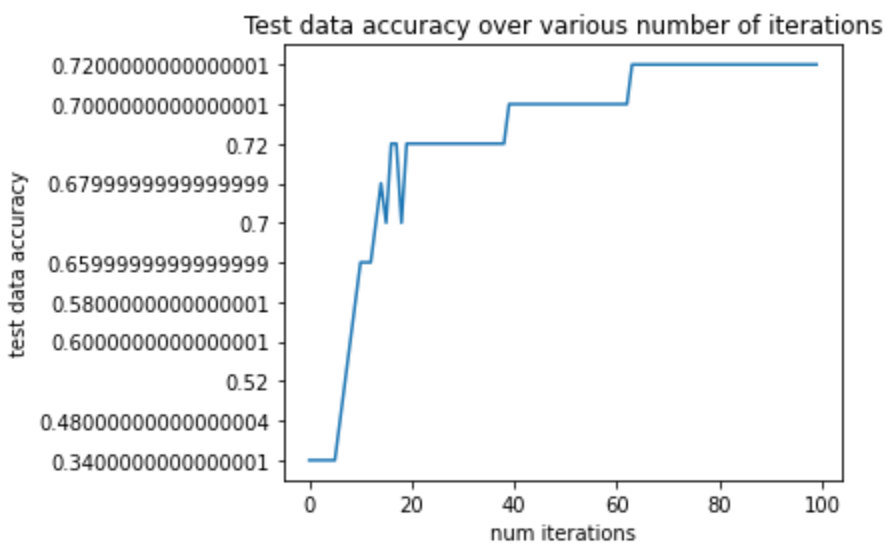
```
plt.xlabel("num iterations (per 100)"); plt.ylabel("loss")
plt.title("Loss over various number of iterations")
```

```
Text(0.5, 1.0, 'Loss over various number of iterations')
```




```
all_iteration_accuracy = [a[1] for a in num_iterations_accuracy.values()]
print(all_iteration_accuracy)
plt.plot(all_iteration_accuracy)
plt.xlabel("num iterations (per 100)"); plt.ylabel("test data accuracy")
plt.title("Test data accuracy over various number of iterations")
```

```
['0.3400000000000001', '0.3400000000000001', '0.3400000000000001', '0.340000000000000
Text(0.5, 1.0, 'Test data accuracy over various number of iterations')
```



From the limited sample available, it seems that a greater number of iterations never *decreases* the test data accuracy (I think that the model is too simple to overfit the training data after too many iterations). However, in this set the accuracy stopped increasing after 75,000 iterations, so that's what I would use as the number of iterations.

(I think it's bad practice to make decisions about your model based on the results from the testing data, since you're then optimizing to the specific test data as opposed to the real data, but whatever.)

3. Hidden Layer Size

```
num_hidden_losses = {}
num_hidden_accuracy = {}

for learning_rate in np.linspace(0.0005, 0.0100, 20):
    losses, test_accuracy = training_loss_test_accuracy(
        learning_rate=learning_rate)
    print(f"Trained with learning rate {learning_rate}")
    learning_rate_losses[learning_rate] = losses
    learning_rate_accuracy[learning_rate] = test_accuracy
```

9. Analyze Image Classification Results

First, let's take a look at some images the 2-layer model labeled incorrectly. This will show a few mislabeled images.

[ ] ↳ *3 cells hidden*

Part 3: Predict Movie Review Sentiment

Now, lets use the same architecture to predict sentiment of movie reviews. In this section, most of the implementation is already provided. The exercises are mainly to understand what the workflow is when handling the text data.

Datatset

Problem Statement: You are given a dataset ("train_imdb.txt", "test_imdb.txt") containing: - a training set of m_train reviews - a test set of m_test reviews - the labels for the training examples are such that the first 50% belong to class 1 (positive) and the rest 50% of the data belong to class 0 (negative)

Let's get more familiar with the dataset. Load the data by completing the function and run the cell below.

[ ] ↳ *6 cells hidden*

[ ]

10. Pre-Processing

From the example review, you can see that the raw data is really noisy! This is generally the case with the text data. Hence, Preprocessing the raw input and cleaning the text is essential. Please run the code snippet provided below.

Exercise: Explain what pattern the model is trying to capture using re.compile.

[ ] ↳ *3 cells hidden*

Vectorization

[ ] ↳ *4 cells hidden*

Model

[ ] ↳ *5 cells hidden*

Predict the review for our movies!

[ ] ↳ *2 cells hidden*

11. Analyze Sentiment Results

Let's take a look at some examples the 2-layer model labeled incorrectly

```
def print_mislabeled_reviews(X, y, p):
    """
    Plots images where predictions and truth were different.
    X -- dataset
    y -- true labels
    p -- predictions
    """
    a = p + y
    mislabeled_indices = np.asarray(np.where(a == 1))
    plt.rcParams['figure.figsize'] = (40.0, 40.0) # set default size of plots
    num_reviews = len(mislabeled_indices[0])
    for i in range(num_reviews):
        index = mislabeled_indices[1][i]
```

```
    print((" ").join(cv.inverse_transform(X[index].reshape(1, -1))[0]))
    print("Prediction: " + str(int(p[0,index])) + " \n Class: " + str(y[0,index]))


print_mislabeled_reviews(X_val.T, y_val, predictions_val)
```

```
actors attempt beauty believable big bit charismatic claims definitely delivery did d
Prediction: 0
 Class: 1
abuse actress beginning bottle breaking brutal budget camera chair convincing decide
Prediction: 0
 Class: 1
ago art ask ass better blood character child cinema classic come confusing cult damn
Prediction: 1
 Class: 0
anna appearance away bad better bible big black blue book boys build capture cat catc
Prediction: 0
 Class: 1
actually ahead better box budget burning character charlie come damn development did
Prediction: 0
 Class: 1
cause early effort government heavy past people problems production propaganda short
Prediction: 1
 Class: 0
act acts apart bad believe cinematography did directing direction director family giv
Prediction: 1
 Class: 0
ability acting actor ages anti better cheap cinema complete confused day decent direc
Prediction: 1
 Class: 0
action adventure adventures bad camp character characters check crew decided doc elem
Prediction: 0
 Class: 1
better body cast central cinema come coming comments company computer decides die ent
Prediction: 1
 Class: 0
admit almighty attempt big bruce carrey cast cheesy comedy dont end enjoyable fan fee
Prediction: 0
 Class: 1
action age body brain building certainly computer crazy damme daughter dead entertain
Prediction: 1
 Class: 0
acting animals best better die dont entire episode episodes funny good horrible ice j
Prediction: 0
 Class: 1
bunch doesnt feel got laugh laughed left like loud make masterpiece movie ok purpose
Prediction: 0
 Class: 1
acting adds agree camera care character characters common completely cop cruise decen
Prediction: 1
 Class: 0
alexander annoying away best changing characters christian crying does effect endless
Prediction: 1
 Class: 0
arts blood burning check complex cut director doesnt effects failed fantasy favorite
```

```
Prediction: 0
 Class: 1
acting away beautifully biggest burt came character drinking fact failure fast fell g
Prediction: 1
 Class: 0
action believe changes director episode film hard humanity humor likes making mindles
Prediction: 0
 Class: 1
charlie dont eye fake film final harder hot im know like look looks real said say sce
```

Exercise: Provide explanation as to why these examples were misclassified below.

Type your answer here

Ψ

```
(Note to self:  class 1 is (positive), class

Generally, the review which were misclassifie
words and the reviews misclassified as positi
be that these review contain words which desc
not align with the reviewer's perspective on
```

(Note to self: class 1 is (positive), class 0 is (negative))

Generally, the review which were misclassified as negative contain negative words and the reviews misclassified as positive contain positive words. It may be that these review contain words which describe the content of the movie do not align with the reviewer's perspective on the movie.