# New Section

from google.colab import drive

drive.mount('/content/drive')

from IPython.display import Image

Image("image.png")

A diagram of a network

Description automatically generated

import numpy as np

np.set\_printoptions(precision=4)

def initialize():

X = np.array([[0.05, 0.10]]) # Inputs

W1 = np.array([[0.15,0.20], [0.25,0.30]]) # Weights to calculate outputs for hidden layer 1

b1 = 0.35 # Bias for hidden layer 1

W2 = np.array([[0.40,0.45], [0.50,0.55]]) # Weights to calculate outputs for output layer

b2 = 0.60 # Bias for output layer

Y = np.array([[0.01, 0.99]]) # Desired output

learning\_rate = 0.5

no\_of\_iter = int(100)

return (X, W1, b1, W2, b2, Y, learning\_rate, no\_of\_iter)

def forward\_pass (X, W1, b1, W2, b2, Y):

### Forward pass: Calculate hidden layer 1 (there is only 1 hidden layer in this example)

Z1 = np.dot(X,W1.T) + b1 # WtX + b

A1 = 1/(1 + np.exp(-Z1)) # Sigmoid(z) = 1 / (1 + e^(-z))

### Forward pass: Calculate output layer

Z2 = np.dot(A1,W2.T) + b2 # WtX + b

A2 = 1/(1 + np.exp(-Z2)) # Sigmoid(z) = 1 / (1 + e^(-z))

### Calculate error/cost function

E = np.sum(1/2\*np.square(Y - A2)) # squared error function

return (A1, A2, E)

def back\_propagation(X, W1, W2, Y, A1, A2, learning\_rate):

### Back propogation

### Adjust W2

dEdA2 = A2 - Y

dA2dZ2 = np.multiply (A2,1-A2)

dZ2dW2 = A1

dEdW2 = dEdA2 \* dA2dZ2 \* dZ2dW2

W2\_adj = W2 - learning\_rate \* dEdW2.T

W2 = W2\_adj

### Adjust W1

dZ2dA1 = W2.T

dA1dZ1 = np.multiply(A1,1-A1)

dZ1dW1 = X

dEdW1 = dEdA2 \* dA2dZ2 \* dZ2dA1 \* dA1dZ1 \* dZ1dW1

W1\_adj = W1 - learning\_rate \* dEdW1.T

W1 = W1\_adj

return (W1, W2)

def main():

(X, W1, b1, W2, b2, Y, learning\_rate, no\_of\_iter) = initialize()

for iter in range (0,no\_of\_iter):

(A1, A2, E) = forward\_pass(X, W1, b1, W2, b2, Y)

(W1, W2) = back\_propagation(X, W1, W2, Y, A1, A2, learning\_rate)

print ('W1 = {} \n\n W2 = {} \n\n Output = {} \n Desired output = {} \n Error = {}'.format(W1, W2, A2, Y, E))

main()

question 2

import numpy as np

import matplotlib.pyplot as plt

from testCases import \*

import sklearn

import sklearn.datasets

import sklearn.linear\_model

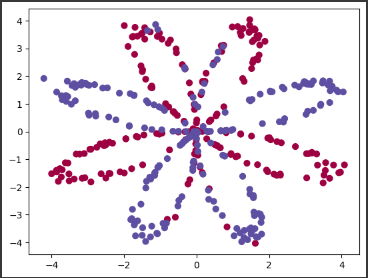
from planar\_utils import plot\_decision\_boundary, sigmoid, load\_planar\_dataset, load\_extra\_datasets

%matplotlib inline

np.random.seed(1) # set a seed so that the results are consistent

X,Y = load\_planar\_dataset()

plt.scatter(X[0, :], X[1, :], c=Y[0], s=40, cmap=plt.cm.Spectral);



### START CODE HERE ### (≈ 3 lines of code)

shape\_X = X.shape

shape\_Y = Y.shape

m = Y.shape[1] # training set size

### END CODE HERE ###

print ('The shape of X is: ' + str(shape\_X))

print ('The shape of Y is: ' + str(shape\_Y))

print ('I have m = %d training examples!' % (m))

def layer\_sizes(X, Y):

"""

Arguments:

X -- input dataset of shape (input size, number of examples)

Y -- labels of shape (output size, number of examples)

Returns:

n\_x -- the size of the input layer

n\_h -- the size of the hidden layer

n\_y -- the size of the output layer

"""

### START CODE HERE ### (≈ 3 lines of code)

n\_x = X.shape[0] # size of input layer

n\_h = 4

n\_y = Y.shape[0] # size of output layer

### END CODE HERE ###

return (n\_x, n\_h, n\_y)

X\_assess, Y\_assess = layer\_sizes\_test\_case()

(n\_x, n\_h, n\_y) = layer\_sizes(X\_assess, Y\_assess)

print("The size of the input layer is: n\_x = " + str(n\_x))

print("The size of the hidden layer is: n\_h = " + str(n\_h))

print("The size of the output layer is: n\_y = " + str(n\_y))

def initialize\_parameters(n\_x, n\_h, n\_y):

"""

Argument:

n\_x -- size of the input layer

n\_h -- size of the hidden layer

n\_y -- size of the output layer

Returns:

params -- 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(2) # we set up a seed so that your output matches ours although the initialization is random.

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

W1 = np.random.randn(n\_h,n\_x)\*0.01

b1 = np.zeros((n\_h,1))

W2 = np.random.randn(n\_y,n\_h)\*0.01

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

n\_x, n\_h, n\_y = initialize\_parameters\_test\_case()

parameters = initialize\_parameters(n\_x, n\_h, n\_y)

print("W1 = " + str(parameters["W1"]))

print("b1 = " + str(parameters["b1"]))

print("W2 = " + str(parameters["W2"]))

print("b2 = " + str(parameters["b2"]))

def forward\_propagation(X, parameters):

"""

Argument:

X -- input data of size (n\_x, m)

parameters -- python dictionary containing your parameters (output of initialization function)

Returns:

A2 -- The sigmoid output of the second activation

cache -- a dictionary containing "Z1", "A1", "Z2" and "A2"

"""

# Retrieve each parameter from the dictionary "parameters"

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

W1 = parameters["W1"]

b1 = parameters["b1"]

W2 = parameters["W2"]

b2 = parameters["b2"]

### END CODE HERE ###

# Implement Forward Propagation to calculate A2 (probabilities)

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

Z1 = np.dot(W1,X)+b1

A1 = np.tanh(Z1)

Z2 = np.dot(W2,A1)+b2

A2 = sigmoid(Z2)

### END CODE HERE ###

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

cache = {"Z1": Z1,

"A1": A1,

"Z2": Z2,

"A2": A2}

return A2, cache

X\_assess, parameters = forward\_propagation\_test\_case()

A2, cache = forward\_propagation(X\_assess, parameters)

# Note: we use the mean here just to make sure that your output matches ours.

print(np.mean(cache['Z1']) ,np.mean(cache['A1']),np.mean(cache['Z2']),np.mean(cache['A2']))

def compute\_cost(A2, Y, parameters):

"""

Computes the cross-entropy cost given in equation (13)

Arguments:

A2 -- The sigmoid output of the second activation, of shape (1, number of examples)

Y -- "true" labels vector of shape (1, number of examples)

parameters -- python dictionary containing your parameters W1, b1, W2 and b2

Returns:

cost -- cross-entropy cost given equation (13)

"""

m = Y.shape[1] # number of example

# Compute the cross-entropy cost

### START CODE HERE ### (≈ 2 lines of code)

logprobs =np.multiply(np.log(A2),Y)+np.multiply(np.log(1-A2),1-Y)

cost = - np.sum(logprobs) /m

### END CODE HERE ###

cost = np.squeeze(cost) # makes sure cost is the dimension we expect.

# E.g., turns [[17]] into 17

assert(isinstance(cost, float))

return cost

A2, Y\_assess, parameters = compute\_cost\_test\_case()

print("cost = " + str(compute\_cost(A2, Y\_assess, parameters)))

def backward\_propagation(parameters, cache, X, Y):

"""

Implement the backward propagation using the instructions above.

Arguments:

parameters -- python dictionary containing our parameters

cache -- a dictionary containing "Z1", "A1", "Z2" and "A2".

X -- input data of shape (2, number of examples)

Y -- "true" labels vector of shape (1, number of examples)

Returns:

grads -- python dictionary containing your gradients with respect to different parameters

"""

m = X.shape[1]

# First, retrieve W1 and W2 from the dictionary "parameters".

### START CODE HERE ### (≈ 2 lines of code)

W1 = parameters["W1"]

W2 = parameters["W2"]

### END CODE HERE ###

# Retrieve also A1 and A2 from dictionary "cache".

### START CODE HERE ### (≈ 2 lines of code)

A1 = cache["A1"]

A2 = cache["A2"]

### END CODE HERE ###

# Backward propagation: calculate dW1, db1, dW2, db2.

### START CODE HERE ### (≈ 6 lines of code, corresponding to 6 equations on slide above)

dZ2 = A2-Y

dW2 = np.dot(dZ2,A1.T)/m

db2 = np.sum(dZ2,axis=1,keepdims=True)/m

dZ1 = np.dot(W2.T,dZ2)\*(1 - np.power(A1, 2))

dW1 = np.dot(dZ1,X.T)/m

db1 = np.sum(dZ1,axis=1,keepdims=True)/m

### END CODE HERE ###

grads = {"dW1": dW1,

"db1": db1,

"dW2": dW2,

"db2": db2}

return grads

parameters, cache, X\_assess, Y\_assess = backward\_propagation\_test\_case()

grads = backward\_propagation(parameters, cache, X\_assess, Y\_assess)

print ("dW1 = "+ str(grads["dW1"]))

print ("db1 = "+ str(grads["db1"]))

print ("dW2 = "+ str(grads["dW2"]))

print ("db2 = "+ str(grads["db2"]))

def update\_parameters(parameters, grads, learning\_rate = 1.2):

"""

Updates parameters using the gradient descent update rule given above

Arguments:

parameters -- python dictionary containing your parameters

grads -- python dictionary containing your gradients

Returns:

parameters -- python dictionary containing your updated parameters

"""

# Retrieve each parameter from the dictionary "parameters"

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

W1 = parameters["W1"]

b1 = parameters["b1"]

W2 = parameters["W2"]

b2 = parameters["b2"]

### END CODE HERE ###

# Retrieve each gradient from the dictionary "grads"

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

dW1 = grads["dW1"]

db1 = grads["db1"]

dW2 = grads["dW2"]

db2 = grads["db2"]

## END CODE HERE ###

# Update rule for each parameter

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

W1 = W1-learning\_rate\*dW1

b1 = b1-learning\_rate\*db1

W2 = W2-learning\_rate\*dW2

b2 = b2-learning\_rate\*db2

### END CODE HERE ###

parameters = {"W1": W1,

"b1": b1,

"W2": W2,

"b2": b2}

return parameters

parameters, grads = update\_parameters\_test\_case()

parameters = update\_parameters(parameters, grads)

print("W1 = " + str(parameters["W1"]))

print("b1 = " + str(parameters["b1"]))

print("W2 = " + str(parameters["W2"]))

print("b2 = " + str(parameters["b2"]))

def nn\_model(X, Y, n\_h, num\_iterations = 10000, print\_cost=False):

np.random.seed(3)

n\_x = layer\_sizes(X, Y)[0]

n\_y = layer\_sizes(X, Y)[2]

parameters = initialize\_parameters(n\_x,n\_h,n\_y)

W1 = parameters["W1"]

b1 = parameters["b1"]

W2 = parameters["W2"]

b2 = parameters["b2"]

for i in range(0, num\_iterations):

A2, cache = forward\_propagation(X,parameters)

cost = compute\_cost(A2, Y, parameters)

grads = backward\_propagation(parameters, cache, X, Y)

parameters = update\_parameters(parameters, grads)

if print\_cost and i % 1000 == 0:

print ("Cost after iteration %i: %f" %(i, cost))

return parameters

def predict(parameters, X):

"""

Using the learned parameters, predicts a class for each example in X

Arguments:

parameters -- python dictionary containing your parameters

X -- input data of size (n\_x, m)

Returns

predictions -- vector of predictions of our model (red: 0 / blue: 1)

"""

# Computes probabilities using forward propagation, and classifies to 0/1 using 0.5 as the threshold.

### START CODE HERE ### (≈ 2 lines of code)

A2, cache = forward\_propagation(X, parameters)

predictions = (A2>0.5)

### END CODE HERE ###

return predictions

parameters, X\_assess = predict\_test\_case()

predictions = predict(parameters, X\_assess)

print("predictions mean = " + str(np.mean(predictions)))

parameters = nn\_model(X, Y, n\_h = 4, num\_iterations = 10000, print\_cost=True)

predictions = predict(parameters, X)

print ('Accuracy: %d' % float((np.dot(Y,predictions.T) + np.dot(1-Y,1-predictions.T))/float(Y.size)\*100) + '%')

4.6 - Tuning hidden layer size (optional/ungraded exercise)

# This may take about 2 minutes to run

hidden\_layer\_sizes = [1, 2, 3, 4, 5, 20, 50]

for i, n\_h in enumerate(hidden\_layer\_sizes):

parameters = nn\_model(X, Y, n\_h, num\_iterations = 5000)

# plot\_decision\_boundary(lambda x: predict(parameters, x.T), X, Y)

predictions = predict(parameters, X)

accuracy = float((np.dot(Y, predictions.T) + np.dot(1 - Y, 1 - predictions.T)) / float(Y.size) \* 100)

print("Accuracy for {} hidden units: {} %".format(n\_h, accuracy))

question 3

from tensorflow.keras.datasets import mnist

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Flatten

from tensorflow.keras.utils import to\_categorical

from matplotlib import pyplot as plt

import numpy as np

# Load the MNIST dataset

(train\_data, train\_target), (test\_data, test\_target) = mnist.load\_data()

fig, axs = plt.subplots(2, 5, figsize=(16,8))

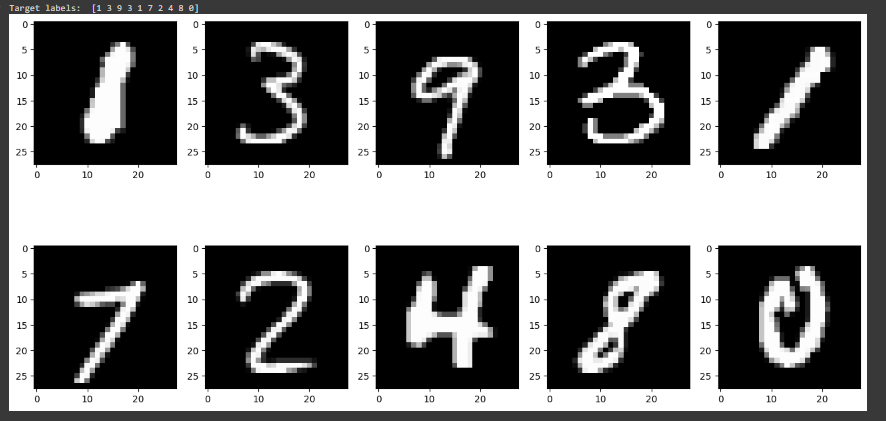
random\_numbers = np.random.randint(0, 60000, 10)

print('Target labels: ', train\_target[random\_numbers])

for idx, ax in enumerate(axs.ravel()):

ax.imshow(train\_data[random\_numbers[idx]], cmap='gray')

plt.show()



new\_train\_data = train\_data / 255.0

new\_test\_data = test\_data / 255.0

new\_train\_target = to\_categorical(train\_target)

new\_test\_target = to\_categorical(test\_target)

from tensorflow.keras.regularizers import l1, l2

# Modify the Model architecture with regularization

model = Sequential()

model.add(Flatten(input\_shape=(28, 28)))

# Add regularized layers

model.add(Dense(128, activation='relu', kernel\_regularizer=l2(0.001))) # L2 regularization

model.add(Dense(64, activation='relu', kernel\_regularizer=l1(0.001))) # L1 regularization

model.add(Dense(64, activation='relu', kernel\_regularizer=l2(0.001)))

model.add(Dense(32, activation='relu', kernel\_regularizer=l1(0.001)))

model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical\_crossentropy', optimizer='adam', metrics=['accuracy'])

train\_info = model.fit(new\_train\_data, new\_train\_target, epochs=20, batch\_size=128)

plt.plot(train\_info.history['loss'])

plt.xlabel('epochs')

plt.ylabel('loss')

plt.title('Model loss')

plt.show()

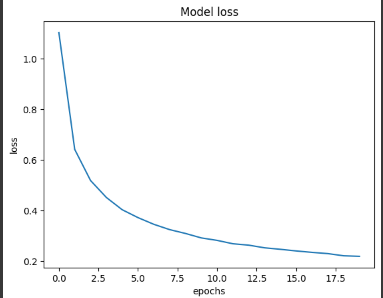
plt.plot(train\_info.history['accuracy'])

plt.xlabel('epochs')

plt.ylabel('accuracy')

plt.title('Model accuracy')

plt.show()



A graph with a line

Description automatically generated

loss, acc = model.evaluate(new\_test\_data, new\_test\_target)

print(f'Loss of the Test dataset is: {loss}\nAccuracy of the test dataset is: {acc}')

from sklearn.metrics import confusion\_matrix

import seaborn as sns

# Predict the classes of the test set

test\_predictions = model.predict(new\_test\_data)

test\_pred\_classes = np.argmax(test\_predictions, axis=1)

test\_true\_classes = np.argmax(new\_test\_target, axis=1)

# Create the confusion matrix

cm = confusion\_matrix(test\_true\_classes, test\_pred\_classes)

# Plot the confusion matrix

plt.figure(figsize=(10, 8))

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=range(10), yticklabels=range(10))

plt.xlabel('Predicted')

plt.ylabel('True')

plt.title('Confusion Matrix')

plt.show()

# Modify the Model architecture

model = Sequential()

model.add(Flatten(input\_shape=(28, 28)))

# Add more layers and neurons

model.add(Dense(128, activation='relu'))

model.add(Dense(64, activation='relu'))

model.add(Dense(64, activation='relu'))

model.add(Dense(32, activation='relu'))

model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical\_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train the model with more epochs

train\_info = model.fit(new\_train\_data, new\_train\_target, epochs=20, batch\_size=128)

from tensorflow.keras.regularizers import l1, l2

# Modify the Model architecture with regularization

model = Sequential()

model.add(Flatten(input\_shape=(28, 28)))

# Add regularized layers

model.add(Dense(128, activation='relu', kernel\_regularizer=l2(0.001))) # L2 regularization

model.add(Dense(64, activation='relu', kernel\_regularizer=l1(0.001))) # L1 regularization

model.add(Dense(64, activation='relu', kernel\_regularizer=l2(0.001)))

model.add(Dense(32, activation='relu', kernel\_regularizer=l1(0.001)))

model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical\_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train the model

train\_info = model.fit(new\_train\_data, new\_train\_target, epochs=20, batch\_size=128)

from sklearn.metrics import confusion\_matrix

import seaborn as sns

# Predict the classes of the test set

test\_predictions = model.predict(new\_test\_data)

test\_pred\_classes = np.argmax(test\_predictions, axis=1)

test\_true\_classes = np.argmax(new\_test\_target, axis=1)

# Create the confusion matrix

cm = confusion\_matrix(test\_true\_classes, test\_pred\_classes)

# Plot the confusion matrix

plt.figure(figsize=(10, 8))

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=range(10), yticklabels=range(10))

plt.xlabel('Predicted')

plt.ylabel('True')

plt.title('Confusion Matrix')

plt.show()

