Machine Learning Final Exam Report
Haekal Rizky Yulianto - 17/415900/PA/18169

Architecture of MLP model

Input layer with 76800 neurons and using sigmoid activation function

Hidden layer with 64 neurons and using sigmoid activation function

Output layer with 3 neurons (3 class) and using softmax activation function

Architecture of CNN model

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 240, 320, 32)      2432
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 120, 160, 32)      0
_________________________________________________________________
flatten (Flatten)            (None, 614400)            0
_________________________________________________________________
dense (Dense)                (None, 32)                19660832
_________________________________________________________________
dense_1 (Dense)              (None, 3)                 99
=================================================================
Total params: 19,663,363
Trainable params: 19,663,363
Non-trainable params: 0
_________________________________________________________________
```

Steps for MLP model from scratch

1. Prepare the data and determine daisy, rose, and tulip path

Data preparation

Check directory

In [2]:
```python
import os
print(os.listdir("/Users/Haekal/Documents/ML/input"))
print(os.listdir("/Users/Haekal/Documents/ML/input/flowers/flowers"))
```

```
['.DS_Store', 'flowers']
['.DS_Store', 'daisy', 'rose', 'tulip', 'dandelion', 'sunflower']
```

Read Image

In [3]:
```python
trainLabels = [] # Image label array
data = [] # Image data array

size = 320, 240 # Resize image

def readImages(flowerPath, folder):

    imagePaths = []
    for file in os.listdir(flowerPath):
        if file.endswith("jpg"):
            imagePaths.append(flowerPath + file)
            trainLabels.append(folder)
            img = cv2.imread((flowerPath + file), 0)
            im = cv2.resize(img, size)
            data.append(im)

    return imagePaths
```

In [4]:
```python
daisy_path = "/Users/Haekal/Documents/ML/input/flowers/flowers/daisy/"
rose_path = "/Users/Haekal/Documents/ML/input/flowers/flowers/rose/"
tulip_path = "/Users/Haekal/Documents/ML/input/flowers/flowers/tulip/"
```

2. Preprocess the data

- Normalize image data

Data preprocessing

Normalize image data

In [7]:
```
X = np.array(data)
X = X.astype('float32') / 255.0
```

- One hot encoding for target/label variable

In [10]:
```
def to_numeric(x):
    if x == 'daisy':
        return '0'
    if x == 'rose':
        return '1'
    if x == 'tulip':
        return '2'
```

In [11]:
```
df['cat'] = df['category'].apply(to_numeric)
```

In [12]:
```
y = pd.get_dummies(df.cat, prefix='C')
y
```

Out[12]:

	C_0	C_1	C_2
0	1	0	0
1	1	0	0
2	1	0	0
3	1	0	0
4	1	0	0
...	...	...	...
295	0	0	1
296	0	0	1
297	0	0	1
298	0	0	1
299	0	0	1

300 rows × 3 columns

- Transform image data into 2D shape

Transform into 2D shape

In [13]:
```
print(X.shape)
print(y.shape)
```

```
(300, 240, 320)
(300, 3)
```

In [14]:
```
y = np.array(y)
```

In [15]:
```
X = X.reshape(X.shape[0], X.shape[1] * X.shape[2])
Y = y
```

In [16]:
```
print(X.shape)
print(Y.shape)
```

```
(300, 76800)
(300, 3)
```

- Split into training and testing, and transpose the data

Split the data for training and testing

In [17]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.2, random_state = 42)
numberOfTrain = X_train.shape[0]
numberOfTest = X_test.shape[0]

print('Train size : ', X_train.shape[0], 'Test size : ', X_test.shape[0])
```

```
Train size :  240 Test size :  60
```

Transpose the data

In [18]:
```
X_train = X_train.T
X_test = X_test.T
Y_train = Y_train.T
Y_test = Y_test.T
```

In [19]:
```
print(X_train.shape)
print(Y_train.shape)
```

```
(76800, 240)
(3, 240)
```

3. Model training

- Declare required function for MLP model

In [20]:
```
def sigmoid(z):
    s = 1. / (1. + np.exp(-z))
    return s

def compute_loss(Y, Y_hat):

    L_sum = np.sum(np.multiply(Y, np.log(Y_hat)))
    m = Y.shape[1]
    L = -(1./m) * L_sum

    return L

def feed_forward(X, params):

    cache = {}

    cache["Z1"] = np.matmul(params["W1"], X) + params["b1"]
    cache["A1"] = sigmoid(cache["Z1"]) # Sigmoid activation function
    cache["Z2"] = np.matmul(params["W2"], cache["A1"]) + params["b2"]
    cache["A2"] = np.exp(cache["Z2"]) / np.sum(np.exp(cache["Z2"]), axis=0) # Softmax activation function

    return cache

def back_propagate(X, Y, params, cache):

    dZ2 = cache["A2"] - Y
    dW2 = (1./m_batch) * np.matmul(dZ2, cache["A1"].T)
    db2 = (1./m_batch) * np.sum(dZ2, axis=1, keepdims=True)

    dA1 = np.matmul(params["W2"].T, dZ2)
    dZ1 = dA1 * sigmoid(cache["Z1"]) * (1 - sigmoid(cache["Z1"]))
    dW1 = (1./m_batch) * np.matmul(dZ1, X.T)
    db1 = (1./m_batch) * np.sum(dZ1, axis=1, keepdims=True)

    grads = {"dW1": dW1, "db1": db1, "dW2": dW2, "db2": db2}

    return grads
```

- Declare function to calculate accuracy

Declare function to calculate accuracy

In [21]:
```
def accuracy_metric(actual, predicted):
    correct = 0
    for i in range(len(actual)):
        if actual[i] == predicted[i]:
            correct += 1
    return correct / float(len(actual)) * 100.0
```

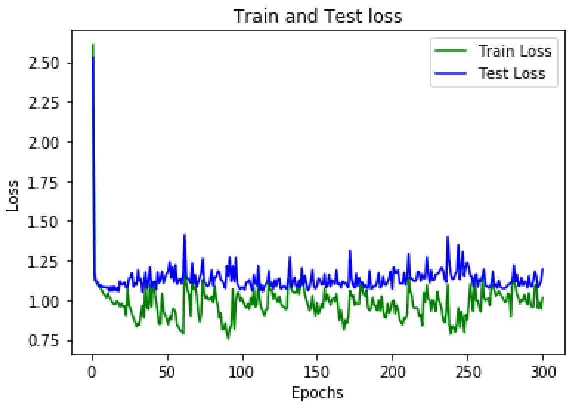
- Weight and hyperparameter initialization (Uses 128 batch size for fast computing)

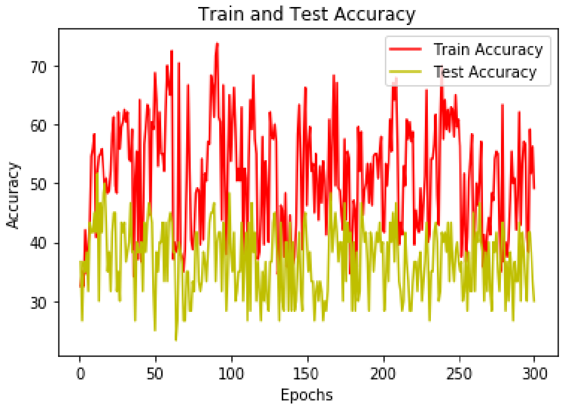
```
In [22]: np.random.seed(42) # Random seed for reproducibility
         m = numberOfTrain # Number of training data

         # Hyperparameters
         n_x = X_train.shape[0] # Number of neuron in input layer = 320 X 240 = 76800
         n_h = 64 # Number of neuron in hidden layer
         num_class = 3 # Number of class
         learning_rate = 0.1
         beta = 0.5
         batch_size = 128
         batches = -(-m // batch_size)

         # Initialization
         params = { "W1": np.random.randn(n_h, n_x) * np.sqrt(1. / n_x),
                    "b1": np.zeros((n_h, 1)) * np.sqrt(1. / n_x),
                    "W2": np.random.randn(num_class, n_h) * np.sqrt(1. / n_h),
                    "b2": np.zeros((num_class, 1)) * np.sqrt(1. / n_h) }

         V_dW1 = np.zeros(params["W1"].shape)
         V_db1 = np.zeros(params["b1"].shape)
         V_dW2 = np.zeros(params["W2"].shape)
         V_db2 = np.zeros(params["b2"].shape)

         ep = 300 # Number of epoch
         epoch = list(range(1,ep+1))
         train_history = []
         test_history = []
         train_accuracy = []
         test_accuracy = []
```

- Train for ep = 300 and test the model

```
# Training
for i in range(ep):

    for j in range (int(batches)):

        begin = j * batch_size
        end = min(begin + batch_size, X_train.shape[1] - 1)
        m_batch = end - begin

        cache = feed_forward(X_train, params)
        grads = back_propagate(X_train, Y_train, params, cache)

        # Update parameters
        V_dW1 = (beta * V_dW1 + (1. - beta) * grads["dW1"])
        V_db1 = (beta * V_db1 + (1. - beta) * grads["db1"])
        V_dW2 = (beta * V_dW2 + (1. - beta) * grads["dW2"])
        V_db2 = (beta * V_db2 + (1. - beta) * grads["db2"])

        params["W1"] = params["W1"] - learning_rate * V_dW1
        params["b1"] = params["b1"] - learning_rate * V_db1
        params["W2"] = params["W2"] - learning_rate * V_dW2
        params["b2"] = params["b2"] - learning_rate * V_db2

    # Train accuracy
    cache = feed_forward(X_train, params)
    trainpredictions = np.argmax(cache["A2"], axis=0)
    trainlabels = np.argmax(Y_train, axis=0)
    train_acc = accuracy_metric(trainlabels, trainpredictions)
    train_accuracy.append(train_acc)

    # Train loss
    train_cost = compute_loss(Y_train, cache["A2"])
    train_history.append(train_cost)

    # Test accuracy
    cache = feed_forward(X_test, params)
    testpredictions = np.argmax(cache["A2"], axis=0)
```

4. Model Evaluation





Average train accuracy = 51.29
Average test accuracy = 36.87

Average train loss = 0.98
Average test loss = 1.13

Steps for CNN model

1. Prepare the data and only include daisy, rose tulip path

In [29]:
```
# Set the path of the input folder
data_path = "/Users/Haekal/Documents/ML/input/flowers/flowers/"

# List out the directories inside the main input folder
folders = os.listdir(data_path)
folders.remove('.DS_Store')
folders.remove('dandelion')
folders.remove('sunflower')

print(folders)
```
```
['daisy', 'rose', 'tulip']
```

In [30]:
```
# Resize image
size1 = 320
size2 = 240

# lists to store data
data = []
label = []

for folder in folders:
    for file in os.listdir(os.path.join(data_path,folder)):
        if file.endswith("jpg"):
            img = cv2.imread(os.path.join(data_path,folder,file))
            img = cv2.resize(img, (size1,size2))
            data.append(img)
            label.append(folder)
        else:
            continue
```

2. Preprocess the data

- Split, one hot encoding, rescale, and assign train and validation datagen

Split the data for training and testing

In [31]:
```
# Transforming into numpy array
data = np.array(data)
label = np.array(label)

# Split dataset into train and test sets
train_data, test_data, train_label, test_label = train_test_split(data, label, test_size=0.2, random_state=42)

# Get the categories/classes
label_categories = np.unique(label)
test_label_names = test_label
```

One hot encoding for target/label variable

In [32]:
```
# Transforming object categories into numerical
encoder = LabelEncoder()

train_label = encoder.fit_transform(train_label).astype(int)
test_label = encoder.fit_transform(test_label).astype(int)
```

Normalize image data

In [33]:
```
datagen_train = ImageDataGenerator(rescale=1./255)
datagen_valid = ImageDataGenerator(rescale=1./255)

datagen_train.fit(train_data)
datagen_valid.fit(test_data)
```

3. Model training

- Build model

Build model

In [36]:
```
# CNN architecture
classifier = Sequential([
    Conv2D(32, (5,5), padding='same', activation='relu', input_shape=(size2, size1,3)),
    MaxPooling2D(2,2),
    Flatten(),
    Dense(32, activation='relu'),
    Dense(3, activation='softmax')
])

# Visualize model summary
classifier.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 240, 320, 32)      2432
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 120, 160, 32)      0
_________________________________________________________________
flatten (Flatten)            (None, 614400)            0
_________________________________________________________________
dense (Dense)                (None, 32)                19660832
_________________________________________________________________
dense_1 (Dense)              (None, 3)                 99
=================================================================
Total params: 19,663,363
Trainable params: 19,663,363
Non-trainable params: 0
_________________________________________________________________
```

- Compile, train, and validate model (Uses 128 batch size for fast computing)

Compile model

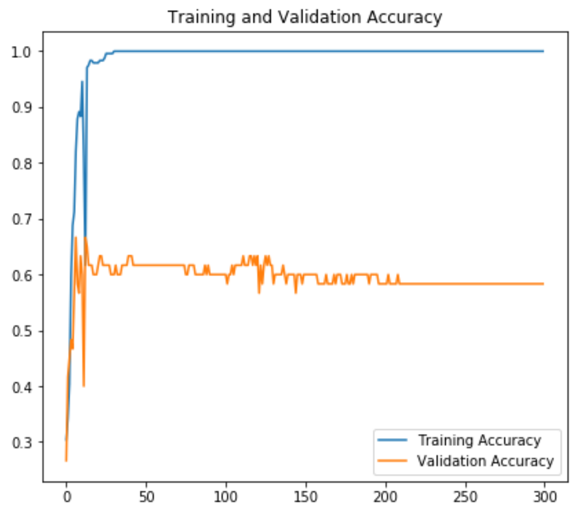
```
In [37]: classifier.compile(optimizer='Nadam', loss='sparse_categorical_crossentropy', metrics=["accuracy"])
```

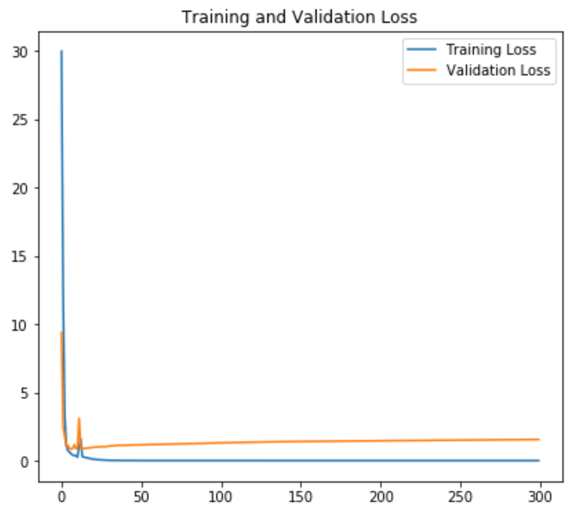
Train and validate model

```
In [38]: history = classifier.fit(datagen_train.flow(train_data, train_label, batch_size = 128),
                                  validation_data = datagen_valid.flow(test_data, test_label, batch_size = 128),
                                  epochs=300)
```

```
Train for 2 steps, validate for 1 steps
Epoch 1/300
2/2 [==============================] - 12s 6s/step - loss: 28.7655 - accuracy: 0.3042 - val_loss: 9.3779 - val_accura
cy: 0.2667
Epoch 2/300
2/2 [==============================] - 10s 5s/step - loss: 12.0539 - accuracy: 0.3417 - val_loss: 2.5022 - val_accura
cy: 0.4167
Epoch 3/300
2/2 [==============================] - 9s 4s/step - loss: 3.3815 - accuracy: 0.4042 - val_loss: 1.7734 - val_accurac
y: 0.4500
Epoch 4/300
2/2 [==============================] - 9s 4s/step - loss: 1.0990 - accuracy: 0.5958 - val_loss: 1.0663 - val_accurac
y: 0.4833
Epoch 5/300
2/2 [==============================] - 9s 4s/step - loss: 0.7181 - accuracy: 0.6875 - val_loss: 1.1209 - val_accurac
y: 0.4667
Epoch 6/300
2/2 [==============================] - 9s 4s/step - loss: 0.6384 - accuracy: 0.7125 - val_loss: 0.8903 - val_accurac
y: 0.5667
Epoch 7/300
```

4. Model Evaluation





Average training accuracy : 0.9854583
Average validation accuracy : 0.5948333

Average training loss : 0.18273056003539062
Average validation loss : 1.3833704330523808

Conclusion

A Multilayer Perceptron (MLP) model with 1 hidden layer architecture produces a faster classification result with ~51% accuracy for training and ~37% accuracy for testing, ~1.0 loss for both training and ~1.1 loss for testing.

Convolutional Neural Network (CNN) model with 1 pair of Conv-Pool layers and 1 hidden layer architecture produces a longer classification result with ~98% accuracy for training and ~59% accuracy for validation, ~0.2 loss for training and ~1.4 loss for validation.

MLPs use one perceptron for each input (e.g pixel) and the amount of weights rapidly becomes unmanageable for large images. One of the common problems is that MLPs react differently to an input (images) and its shifted version. For example in this data, the flower is not always in the same position on every image and MLPs will try to correct itself and assume that the objects will always appear in certain sections of the image. In this MLPs model architecture, both loss and accuracy are fluctuating as the model seems to be confused on what makes which images belong to which flower class, since MLP doesn't have feature extraction layer(s).

This CNN architecture successfully distinguishes the training images well (too well), but cannot generalize well in the validation images. Overfitting happens in the CNN architecture because the amount of data is too small (only 300 data in total), there are no image augmentation and dropout layer(s) or regularization(L1/L2) applied in this CNN architecture model.

Configuring a deeper layer in CNN model architecture with more than one Conv-Pool layers, applying image augmentation, using dropout layer(s), and feeding more training data will probably increase the accuracy and also avoid overfitting.

Notes

For taking 100 images from each daisy, tulip, and rose folder, I manually select the first 100 images for each folder of daisy, tulip, and rose.