Experiment 4 - Artificial Neural Network

April 30, 2023

1 Experiment Details

1.1 Submitted By

Desh Iyer, 500081889, Year III, AI/ML(H), B5

1.2 Problem Statement

Build an Artificial Neural Network (ANN) by implementing the Backpropagation algorithm and test the same using appropriate data sets.

1.3 Theory

Artificial Neural Networks (ANNs) are a set of algorithms that are designed to recognize patterns. They are inspired by the way the human brain works and are used to solve complex problems that cannot be easily solved using traditional methods. The Backpropagation algorithm is a supervised learning method that is commonly used to train ANNs. It is used to update the weights of the network so that it can better predict the output for a given input.

The Backpropagation algorithm consists of two phases: the forward phase and the backward phase. In the forward phase, the input is propagated through the network to produce an output. In the backward phase, the error between the predicted output and the actual output is calculated and used to update the weights of the network. This process is repeated until the error is minimized.

1.4 Mathematics of ANNs

Forward propagation

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$
$$A^{[1]} = g_{\text{ReLU}}(Z^{[1]}))$$
$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$
$$A^{[2]} = g_{\text{softmax}}(Z^{[2]})$$

Backward propagation

$$dZ^{[2]} = A^{[2]} - Y$$
$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$$

$$dB^{[2]} = \frac{1}{m}\Sigma dZ^{[2]}$$

$$dZ^{[1]} = W^{[2]T} dZ^{[2]} .* g^{[1]\prime}(z^{[1]})$$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} A^{[0]T}$$

$$dB^{[1]} = \frac{1}{m}\Sigma dZ^{[1]}$$

Parameter updates

$$W^{[2]} := W^{[2]} - \alpha dW^{[2]}$$

$$b^{[2]} := b^{[2]} - \alpha db^{[2]}$$

$$W^{[1]} := W^{[1]} - \alpha dW^{[1]}$$

$$b^{[1]} := b^{[1]} - \alpha db^{[1]}$$

Vars and shapes

Forward prop

- $A^{[0]} = X$: 784 x m
- $Z^{[1]} \sim A^{[1]}$: 10 x m
- $W^{[1]}$: 10 x 784 (as $W^{[1]} A^{[0]} \sim Z^{[1]}$)
- $B^{[1]}$: 10 x 1
- $Z^{[2]} \sim A^{[2]}$: 10 x m
- $W^{[1]}$: 10 x 10 (as $W^{[2]} A^{[1]} \sim Z^{[2]}$)
- $B^{[2]}$: 10 x 1

Backprop

- $dZ^{[2]}$: 10 x m ($A^{[2]}$)
- $dW^{[2]}$: 10 x 10
- $dB^{[2]}$: 10 x 1
- $dZ^{[1]}$: 10 x m ($A^{[1]}$)
- $dW^{[1]}$: 10 x 10
- $dB^{[1]}$: 10 x 1

1.5 Steps

Here are the steps to implement the Backpropagation algorithm for an ANN:

1. Initialize the weights of the network randomly.
2. Feed the input through the network and calculate the output.
3. Calculate the error between the predicted output and the actual output.
4. Update the weights of the network using the error calculated in step 3.
5. Repeat steps 2-4 until the error is minimized.

1.6 Advantages

- ANNs can be used to solve complex problems that cannot be easily solved using traditional methods.
- Backpropagation is a powerful algorithm that can learn complex relationships between inputs and outputs.
- ANNs are highly parallelizable, which makes them suitable for use in large-scale applications.

1.7 Limitations

- ANNs can be difficult to train, and require a large amount of data and computational resources.
- ANNs can suffer from overfitting, where the network performs well on the training data but poorly on new data.
- ANNs can be difficult to interpret, and it can be hard to understand how they arrive at their predictions.

1.8 Pseudocode

Here's the pseudocode for the Backpropagation algorithm:

1. Initialize the weights randomly.
2. Repeat until the error is minimized:
    1. Feed the input through the network and calculate the output.
    2. Calculate the error between the predicted output and the actual output.
    3. Calculate the gradient of the error with respect to the weights.
    4. Update the weights using the gradient.
3. Return the trained network.

2 Check if GPU is Active

[ ]:
```
!nvidia-smi
```

```
Sun Apr 30 22:59:19 2023
+-----------------------------------------------------------------------------
--------+
| NVIDIA-SMI 530.30.02              Driver Version: 530.30.02    CUDA Version:
12.1     |
|-----------------------------------------+----------------------+-------------
--------+
| GPU  Name                  Persistence-M| Bus-Id        Disp.A | Volatile
Uncorr. ECC |
| Fan  Temp  Perf            Pwr:Usage/Cap|         Memory-Usage | GPU-Util
Compute M. |
|                                         |                      |
MIG M. |
|=========================================+======================+=============
========|
|   0  NVIDIA GeForce GTX 1650 Ti      On | 00000000:01:00.0 Off |
```

```
N/A |
| N/A   49C    P0               18W /  50W|    574MiB /  4096MiB |     33%
Default |
|                                         |                      |
N/A |
+-----------------------------------------+----------------------+-------------
--------+

+---------------------------------------------------------------------------------
--------+
| Processes:
|
|  GPU   GI   CI        PID   Type   Process name                            GPU
Memory |
|        ID   ID
Usage      |
|=================================================================================
========|
|    0   N/A  N/A      2389      G   /usr/lib/xorg/Xorg
248MiB |
|    0   N/A  N/A      3016      G   /usr/bin/gnome-shell
32MiB |
|    0   N/A  N/A      4210      G   …5354413,16873341058258144221,131072
50MiB |
|    0   N/A  N/A      5857      G   …sion,SpareRendererForSitePerProcess
182MiB |
|    0   N/A  N/A     47450      G   …,WinRetrieveSuggestionsOnlyOnDemand
57MiB |
+---------------------------------------------------------------------------------
--------+
```

3 Importing Libraries and Data

[ ]:
```python
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

# Using a relative path instead of an absolute
path = r'./data/ann/train.csv'

data = pd.read_csv(path)
```

Loaded the data into a pandas dataframe.

[ ]:
```python
data.head()
```

[ ]:

	label	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	\
0	1	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	0	
2	1	0	0	0	0	0	0	0	0	
3	4	0	0	0	0	0	0	0	0	
4	0	0	0	0	0	0	0	0	0	

	pixel8	…	pixel774	pixel775	pixel776	pixel777	pixel778	pixel779	\
0	0	…	0	0	0	0	0	0	
1	0	…	0	0	0	0	0	0	
2	0	…	0	0	0	0	0	0	
3	0	…	0	0	0	0	0	0	
4	0	…	0	0	0	0	0	0	

	pixel780	pixel781	pixel782	pixel783
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0

[5 rows x 785 columns]

Converting the data from a dataframe to a numpy array.

[ ]:
```
data = np.array(data)
```

Making sure that the model isn't overfitted, i.e., the model makes fairly accurate predictions for the training data but isn't generalised for the data it's supposed to have a high accuracy for. Setting aside a portion of the training data to perform cross-validation on to avoid overfitting.

Shuffling the data before we split the data into dev and training data. Note, `np.random.shuffle()` permutes the sequence in place.

[ ]:
```
np.random.shuffle(data)
```

[ ]:
```
data
```

[ ]:
```
array([[9, 0, 0, …, 0, 0, 0],
       [6, 0, 0, …, 0, 0, 0],
       [5, 0, 0, …, 0, 0, 0],
       …,
       [4, 0, 0, …, 0, 0, 0],
       [1, 0, 0, …, 0, 0, 0],
       [3, 0, 0, …, 0, 0, 0]])
```

[ ]:
```
# Storing the dimensions
m, n = data.shape
```

```
# m - Number of images; n - label + pixels for each image
m, n
```

[ ]: (42000, 785)

Splitting the data into dev and training. We're using dev to cross validate and we're setting aside only 1000 images to do so.

[ ]:
```
# Transposing the data using only 1000 images
data_dev = data[:1000].T

# Storing the labels in YDev
Y_dev = data_dev[0]

# Storing the pixels
X_dev = data_dev[1:]
X_dev = X_dev / 255
```

[ ]:
```
# Storing the rest of the images
data_train = data[1000:m].T

# Extract labels
Y_train = data_train[0]

# Get the rest
X_train = data_train[1:]
X_train = X_train / 255
```

Printing details of all arrays implemented so far.

[ ]:
```
print(
    f'Printing dimensions of all existing arrays:\n(i) X - pixels\nX_dev:␣
 ↪{X_dev.shape}\nX_train: {X_train.shape}\n\n(ii) Y - labels\nY_dev: {Y_dev.
 ↪shape}\nY_train: {Y_train.shape}')
```

```
Printing dimensions of all existing arrays:
(i) X - pixels
X_dev: (784, 1000)
X_train: (784, 41000)

(ii) Y - labels
Y_dev: (1000,)
Y_train: (41000,)
```

4 Defining Initial Parameters

Defining a function to initialise the neural network by creating random weights. We use `rand()` to obtain a random value between 0 and 1 and then we subtract from those values to make sure the range in which our random values lie is `[-0.5, 0.5]`.

[ ]:
```
def init_params():
    # There's 10 connections for each of the 784 nodes
    W1 = np.random.rand(10, n - 1) - 0.5

    # There's 10 biases
    b1 = np.random.rand(10, 1) - 0.5

    # Similarly,
    # There's 10 connections to 10 output nodes
    W2 = np.random.rand(10, 10) - 0.5
    b2 = np.random.rand(10, 1) - 0.5

    return W1, b1, W2, b2
```

Function implementing the ReLU (rectified linear unit) activation function.

[ ]:
```
def ReLU(Z):
    # Taking the maximum element-wise using numpy
    return np.maximum(0, Z)
```

Function implementing the softmax activation function

[ ]:
```
def softmax(Z):
    A = np.exp(Z) / sum(np.exp(Z))

    # Returning the probability
    return A
```

5 Forward Propagation

Defining a function to implement forward propagation through the neural net.

[ ]:
```
def forward_propagation(W1, b1, W2, b2, X):
    # Deactivated first layer
    Z1 = W1.dot(X) + b1

    # Activating Z1
    A1 = ReLU(Z1)

    # Creating the next layer's deactivated input
    Z2 = W2.dot(A1) + b2
```

```
    # Since the next layer is the output layer, we apply softmax
    A2 = softmax(Z2)

    return Z1, A1, Z2, A2
```

Function to implement one-hot encoding of Y. This is to represent the target classes as an array instead of a label.

[ ]:
```
def one_hot_encode(Y):
    # Encoding
    one_hot_encoded_df = pd.get_dummies(Y)

    # Taking the transpose so the columns represent images
    one_hot_encoded_array = np.array(one_hot_encoded_df).T

    return one_hot_encoded_array
```

Test to illustrate the working of `one_hot_encode(Y)`.

[ ]:
```
test = Y_train[:20]
test
```

[ ]:
```
array([1, 3, 9, 3, 9, 9, 3, 9, 5, 1, 3, 0, 6, 8, 4, 5, 2, 4, 2, 5])
```

[ ]:
```
df = pd.get_dummies(test).T
df

ls = np.array(df)
ls
```

[ ]:
```
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0],
       [0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]],
      dtype=uint8)
```

Comparing it to our function.

[ ]:
```
one_hot_encode(test)
```

[ ]:
```
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0],
```

```
       [0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]],
      dtype=uint8)
```

Alternative one-hot encoding function.

[ ]:
```
def one_hot(Y):
    one_hot_Y = np.zeros((Y.size, Y.max() + 1))

    one_hot_Y[np.arange(Y.size), Y] = 1

    one_hot_Y = one_hot_Y.T

    return one_hot_Y
```

Function to implement the derivative of ReLU.

[ ]:
```
def derivative_ReLU(Z):
    # Returning 1 if the value is greater than 0. This is because the slope of␣
 ↪the `linear` thing is 1.
    return Z > 0
```

6 Back Propagation

Function to back propagate through the neural network to calculate the differences in the weights and biases.

[ ]:
```
def back_propagation(Z1, A1, Z2, A2, W2, X, Y):
    m = Y.size
    # one_hot_encoded_Y = one_hot_encode(Y)

    # dZ2 = A2 - one_hot_encoded_Y

    one_hot_Y = one_hot(Y)
    dZ2 = A2 - one_hot_Y

    dW2 = (1 / m) * dZ2.dot(A1.T)
    db2 = (1 / m) * np.sum(dZ2)

    dZ1 = W2.T.dot(dZ2) * derivative_ReLU(Z1)
    dW1 = (1 / m) * dZ1.dot(X.T)
    db1 = (1 / m) * np.sum(dZ1)
```

```
    return dW1, db1, dW2, db2
```

7 Updating Parameters

Function to update the parameters using learning rate `alpha`.

[ ]:
```
def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1

    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2

    return W1, b1, W2, b2
```

8 Defining Gradient Descent

[ ]:
```
def get_predictions(A):
    # Returns the indices of the max values
    return np.argmax(A, 0)
```

Testing the `get_predictions()` function.

[ ]:
```
test, get_predictions(test)
```

[ ]:
```
(array([1, 3, 9, 3, 9, 9, 3, 9, 5, 1, 3, 0, 6, 8, 4, 5, 2, 4, 2, 5]), 2)
```

[ ]:
```
def get_accuracy(predictions, Y):
    # print(predictions, Y)

    return np.sum(predictions == Y) / Y.size
```

Gradient descent is an optimization algorithm which is commonly-used to train machine learning models and neural networks. Training data helps these models learn over time, and the cost function within gradient descent specifically acts as a barometer, gauging its accuracy with each iteration of parameter updates. Until the function is close to or equal to zero, the model will continue to adjust its parameters to yield the smallest possible error. Once machine learning models are optimized for accuracy, they can be powerful tools for artificial intelligence (AI) and computer science applications.

[ ]:
```
def gradient_descent(X, Y, epochs, alpha):
    # Defining weights and biases
    W1, b1, W2, b2 = init_params()

    for i in range(epochs):
        # Step 1
```

```
        Z1, A1, Z2, A2 = forward_propagation(W1, b1, W2, b2, X)

        # Step 2
        dW1, db1, dW2, db2 = back_propagation(Z1, A1, Z2, A2, W2, X, Y)

        # Step 3
        W1, b1, W2, b2 = update_params(
            W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)

        # Every 50th iteration
        if i % 50 == 0:
            # A2 is the output from the forward propagation
            predictions = get_predictions(A2)

            print("Epoch: {} | Accuracy: {:.2}".format(i,␣
↪get_accuracy(predictions, Y)))

    return W1, b1, W2, b2
```

9 Running the Neural Network

Running it on `X_train` and `Y_train`. We run it for 500 epochs with a learning rate of 1. As we can clearly tell, the output displays for each 10 iterations, a rapidly increasing accuracy settling at a nice 91.7%.

[ ]:
```
EPOCHS = 500
LEARNING_RATE = 1
```

[ ]:
```
W1, b1, W2, b2 = gradient_descent(X_train, Y_train, EPOCHS, LEARNING_RATE)
```

```
Epoch: 0 | Accuracy: 0.057
Epoch: 50 | Accuracy: 0.45
Epoch: 100 | Accuracy: 0.74
Epoch: 150 | Accuracy: 0.78
Epoch: 200 | Accuracy: 0.85
Epoch: 250 | Accuracy: 0.87
Epoch: 300 | Accuracy: 0.84
Epoch: 350 | Accuracy: 0.89
Epoch: 400 | Accuracy: 0.9
Epoch: 450 | Accuracy: 0.86
```

Running it on `X_dev` and `Y_dev` to cross-validate the model to overcome overfitting.

[ ]:
```
W1, b1, W2, b2 = gradient_descent(X_dev, Y_dev, EPOCHS, LEARNING_RATE)
```

```
Epoch: 0 | Accuracy: 0.11
Epoch: 50 | Accuracy: 0.6
Epoch: 100 | Accuracy: 0.72
```

```
Epoch: 150 | Accuracy: 0.94
Epoch: 200 | Accuracy: 0.97
Epoch: 250 | Accuracy: 0.98
Epoch: 300 | Accuracy: 0.98
Epoch: 350 | Accuracy: 0.98
Epoch: 400 | Accuracy: 0.99
Epoch: 450 | Accuracy: 0.99
```

10 Conclusion

Implementing the Backpropagation algorithm is a powerful technique for building an Artificial Neural Network. With the right data set and parameters, ANNs can be used to solve complex problems and provide insights that would be difficult or impossible to obtain using traditional methods.

Our NN will have a simple two-layer architecture. Input layer $a^{[0]}$ will have 784 units corresponding to the 784 pixels in each 28x28 input image. A hidden layer $a^{[1]}$ will have 10 units with ReLU activation, and finally our output layer $a^{[2]}$ will have 10 units corresponding to the ten digit classes with softmax activation.
