IN3063 Mathematics and Programming of AI

Resit Coursework

Team Members:
James Ly (acvt729)
Tadas Vaisvila (acnh582)
Alexander Downs (acvt702)

Contents

James Ly (acvt729)
Tadas Vaisvila (acnh582)
Alexander Downs (acvt702)

Intro:

This program is a simple Neural Network that evaluates the accuracy of a training set and testing set, with the implementation of three kinds of activation methods (Sigmoid, ReLu, LeakyReLu), Softmax and Inverted Dropout.

This program is made using Python 3 through Jupyter Notebook IDE.

Instructions:

Simple Neural Network with Dropout.ipynb -

Best way to run this program is to restart the whole kernel, as the cells follow a sequential routine.

For re-running the program without restarting the whole kernel, after the initial run, run the cells from Loading Data and Training NN and below.

You'll require both the fashion-mnist_test.zip and fashion-mnist_train.zip file for this to run, this can be downloaded from the Google Drive linked below.

Neural Network with SGD Momentum.ipynb -

Run the program by pressing the restart kernel and run button. After which you can change the arguments for the neural network and train function and run from the In [5] code block.

Make sure you have the pkl_quickdraw.pkl file beforehand. You can create it by running Repackaging Quickdraw MNIST.ipynb, which requires the .npy files from numpy_quickdraw_indv.zip or by downloading from the Google Drive.

Repackaging Quickdraw MNIST.ipynb -

Run the program from start to finish. Ensure that you have the .npy files from numpy_quickdraw_indv.zip which can be downloaded from the Google Drive.

Google Drive Dataset

https://drive.google.com/drive/folders/1s2Ni2n0RjN2gfEagrqMHpTigm511qxLO?usp=sharing

Task 1: Linear layers with ReLU, leaky ReLU and Sigmoid activations

Linear layers – During the forward pass, the linear activation is applied to the entry and exit points of both the Input-Hidden layer as well as the Hidden-Output layer. This is performed by a simple dot multiplication where the input, at that point in the layer, is multiplied by its respective weights.

i.e. Input-Hidden; take the input_vector (Input value) and apply dot multiplication with self.wih (The Input-Hidden weight).

```
def train_single(self, input_vector, target_vector):
    # Forward Propagation
    # Input Layer to Hidden Layer
    if self.bias:
        # adding bias node to the end of the input_vector
        input_vector = np.concatenate( (input_vector, [self.bias]) )

    input_vector = np.array(input_vector, ndmin=2).T
    target_vector = np.array(target_vector, ndmin=2).T

    output_vector1 = np.dot(self.wih, input_vector) # linear
    output_vector_hidden = self.activation_ih(output_vector1) # non-linear

    # Hidden Layer to Output Layer
    if self.bias:
        output_vector_hidden = np.concatenate( (output_vector_hidden, [[self.bias]]) )

    output_vector2 = np.dot(self.who, output_vector_hidden) # linear
    output_vector_network = self.activation_ho(output_vector2) # non-linear

    # Backward Propagation
    # Output Layer to Hidden Layer
    output_errors = target_vector - output_vector_network

    # if using softmax
    if self.dactivation_oh == dsoftmax:
        ovn = output_vector_network.reshape(output_vector_network.size,)
        dsoftmax_output = dsoftmax(ovn, self.no_of_out_nodes)
        tmp = np.dot(dsoftmax_output, output_errors)

    else:
        tmp = output_errors * self.dactivation_oh(output_vector_network) # derivative

    if use_dropout:
        tmp = tmp / self.active_hidden_percentage # Inverted Dropout

    tmp = self.learning_rate  * np.dot(tmp, output_vector_hidden.T)
    self.who += tmp # update hidden-output weights

    # Hidden Layer to Input Layer
    # calculate hidden errors:
    hidden_errors = np.dot(self.who.T, output_errors)
    # update the weights:
    tmp = hidden_errors * self.dactivation_hi(output_vector_hidden) # derivative

    if use_dropout:
        tmp = tmp / self.active_input_percentage # Inverted Dropout

    if self.bias:
        x = np.dot(tmp, input_vector.T)[:-1,:]
    else:
        x = np.dot(tmp, input_vector.T)
    self.wih += self.learning_rate * x # update input-hidden weights
```

Figure 1: Snippet of the Training function showing the layers

Activation functions – Following the linear activation, an additional activation is applied to the output of the linear activation.

These are the activation functions (ReLU, leaky ReLU, and Sigmoid). These functions will be used to transform the linear activation output to the final output value in each layer.

To change the activation, pass the activation_function variable during the initialisation of the class as: ‘sigmoid’ for Sigmoid activation, ‘relu’ for ReLU activation, ‘leakyrelu’ for Leaky ReLU activation.

```
@np.vectorize
# Activation Functions
def sigmoid(x):
    return 1 / (1 + np.e ** -x)

def dsigmoid(x):
    return x * (1 - x)
```

```
def relu(x):
    return np.maximum(0, x)

def drelu(x):
    return np.where(x <= 0, 0, 1)

def leaky_relu(x):
    return np.where(x >= 0, x, x * 0.01)

def dleaky_relu(x):
    out = np.ones_like(x)
    out[x < 0] *= 0.01
    return out
```

Figure 2: Snippet of Non-linear Activation Functions

After which we calculate the errors and apply the derivative to discover the updated weights for each layer and then apply it. This only applies in training.

Task 2: Inverted Dropout

We apply dropout by changing the variables passed when calling the train function and the initialising the neural network class. These variables are: active_input_percentage, active_hidden_percentage, no_of_dropout_tests, and use_dropout.

active_input_percentage and active_hidden_percentage affects the number of nodes active at a current layer by a certain percentage. i.e. 0.7; refers to 70% active nodes and 30% deactivated. The active_input_percentage is for Input layer and active_hidden_percentage for the Hidden layer.

no_of_dropout_tests, instead, is used to separate the data into partitions for training to be run.

use_dropout, is necessary, effectively a Boolean that decides whether dropout is being used or not

Inverted dropout is applied to the weights during training, here the percentages are applied divisibly rather than multiplicatively during testing.

```
# Backward Propagation
# Output Layer to Hidden Layer
output_errors = target_vector - output_vector_network

# if using softmax
if self.dactivation_oh == dsoftmax:
    ovn = output_vector_network.reshape(output_vector_network.size,)
    dsoftmax_output = dsoftmax(ovn, self.no_of_out_nodes)
    tmp = np.dot(dsoftmax_output, output_errors)

else:
    tmp = output_errors * self.dactivation_oh(output_vector_network) # derivative

if use_dropout:
    tmp = tmp / self.active_hidden_percentage # Inverted Dropout

tmp = self.learning_rate  * np.dot(tmp, output_vector_hidden.T)
self.who += tmp # update hidden-output weights
```

Figure 3: Inverted Dropout used to Calculate weights

Task 3: Softmax classifier

The softmax function takes in the input and normalizes it into a probability distribution. The result is proportional to the input.

We apply this function during Hidden layer where instead of a non-linear activation softmax is applied instead. The non-linear activation during the Input layer is Sigmoid.

```
def softmax(x):
    e_x = np.exp(x)
    return e_x / e_x.sum()

def dsoftmax(x, y):
    si_sj = -x * x.reshape(y, 1)
    s_der = np.diag(x) + si_sj
    return s_der
```

Figure 4: Softmax and derivative function

Task 4: Fully connected NN

The neural network was created with all the implementations made in the previous three tasks (tasks 1 to 3). Parameters used can be seen below, epochs were set to ten (run through network ten times) and the accuracy test (train and test) were close to 0.0 meaning high accuracy. The learning rate was set to 0.1, The Fashion-MNIST dataset was used.

Loading data and Training NN -

Load the data from the pickle file, initialise the Neural Network with the desired arguments and then train them.

Notable arguments:

no_of_in_nodes - original image data size (28x28) being passed through equals in put sizeno_of_out_nodes - desired amount of output nodes for the data
no_of_hidden_nodes - desired number of hidden nodes to process the data
learning_rate - the desired learning rate
activation_function - the desired activation functions;
includes: Sigmoid 'sigmoid', ReLU 'relu', and Leaky ReLU 'leakyrelu'

To use different activation functions for the Neural Network, initialise a new NeuralNetwork class and set the value of the activation_function variable as 'sigmoid' for Sigmoid, 'relu' for ReLU, and 'leakyrelu' for LeakyReLU.

For Softmax pass the activation_function variable, when initialising the Neural Network, to 'softmax'.

To use dropout, when running the Train set the use_dropout variable to True and change the active_input_percentage, active_hidden_percentage, and no_of_dropout_tests accordingly.

```
class NeuralNetwork:

    def __init__(self,
                 no_of_in_nodes,
                 no_of_out_nodes,
                 no_of_hidden_nodes,
                 activation_function,
                 learning_rate,
                 bias=None,
                ):
        self.no_of_in_nodes = no_of_in_nodes
        self.no_of_out_nodes = no_of_out_nodes
        self.no_of_hidden_nodes = no_of_hidden_nodes
        self.learning_rate = learning_rate
        self.bias = bias

        if activation_function == 'sigmoid':
            self.activation_ih = sigmoid
            self.activation_ho = sigmoid
            self.dactivation_oh = dsigmoid
            self.dactivation_hi = dsigmoid

        if activation_function == 'softmax':
            self.activation_ih = sigmoid
            self.activation_ho = softmax
            self.dactivation_oh = dsoftmax
            self.dactivation_hi = dsigmoid

        if activation_function == 'relu':
            self.activation_ih = relu
            self.activation_ho = relu
            self.dactivation_oh = drelu
            self.dactivation_hi = drelu

        if activation_function == 'leakyrelu':
            self.activation_ih = leaky_relu
            self.activation_ho = leaky_relu
            self.dactivation_oh = dleaky_relu
            self.dactivation_hi = dleaky_relu

        self.use_dropout = False
        self.active_input_percentage = 1.0
        self.active_hidden_percentage = 1.0

        self.create_weight_matrices()
```

Figure 5: Neural Network Class

```
simple_NN = NeuralNetwork(no_of_in_nodes = image_pixels,
                          no_of_out_nodes = 10,
                          no_of_hidden_nodes = 100,
                          learning_rate = 0.0001,
                          activation_function = 'leakyrelu')

simple_NN.train(train_imgs,
                train_labels_one_hot,
                active_input_percentage=1,
                active_hidden_percentage=1,
                no_of_dropout_tests = 1,
                use_dropout = False,
                epochs=epochs)
```

Figure 6: Initialising the Neural Network and Training

```
def mean_squared_error(self, data, labels):
    actual = labels
    predicted = [None] * len(actual)

    corrects, wrongs = 0, 0
    for i in range(len(data)):
        res = self.run(data[i])
        res_max = res.argmax()
        predicted[i] = res_max

    sum_square_error = 0.0
    for i in range(len(actual)):
        sum_square_error += (actual[i] - predicted[i])**2.0
    mean_square_error = 1.0 / len(actual) * sum_square_error
    return mean_square_error
```

Figure 7: Mean Square Error Function

Task 5: Hyper-parameters

We apply SGD Momentum to our fully connected Neural Network, created for task 4. This is performed by calculating momentum changes near the end of every train_single function call. The resulting momentum changes are saved to an array (self.v0 and self.v1), one for each weight, and is applied to the weight afterwards. Updating the weight values by the momentum.

```
# Hidden Layer to Input Layer
# calculate hidden errors:
hidden_errors = np.dot(self.who.T, output_errors)
# update the weights:
tmp = hidden_errors * self.dactivation_hi(output_vector_hidden) # derivative

if self.use_dropout:
    tmp = tmp / self.active_input_percentage # Inverted Dropout

if self.bias:
    delta_w0 = np.dot(tmp, input_vector.T)[:-1,:]
else:
    delta_w0 = np.dot(tmp, input_vector.T)

# Momentum
if not self.use_momentum:
    self.wih += self.learning_rate * delta_w0
    self.who += self.learning_rate * delta_w1
else:
    self.v0 = self.momentum * self.v0 + (1 - self.momentum) * delta_w0
    self.v1 = self.momentum * self.v1 + (1 - self.momentum) * delta_w1
    self.wih += self.learning_rate * self.v0
    self.who += self.learning_rate * self.v1
```

Figure 8: SGD Momentum applied in Training

This momentum functionality wasn't applied with batching.

External Source:
https://github.com/ngailapdi/MNIST_numpy/blob/master/MNIST_SGD_Numpy.ipynb

Testing Results:

Training and Testing Accuracy

	TRAINING ACCURACY (3 S.F.)	TESTING ACCURACY (3 S.F.)
SIGMOID (0.15)	0.757	0.756
RELU (0.15)	0.1	0.1
RELU (0.00001)	0.2094	0.2074
LEAKY RELU (0.0001)	0.452	0.454
SOFTMAX (0.15)	0.7033	0.698
SIGMOID W/ DROPOUT (0.15)	0.8051	0.8057

Label Precision

		ACTIVATION FUNCTIONS (LR)											
		Sigmoid (0.15)		**ReLU (0.15)**		**ReLU (0.00001)**		**Leaky ReLU (0.0001)**		**Softmax (0.15)**		**Sigmoid w/ Dropout (0.15)**	
		Train	Test	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test
LABELS (1-10)	1	0.464	0.449	0.1	0.1	0.573	0.55	0.53	0.546	0.726	0.708	0.801	0.793
	2	0.945	0.955	0.0*	0.0*	0.942	0.937	0.919	0.821	0.951	0.956	0.963	0.967
	3	0.828	0.823	0.0*	0.0*	0.0006	0.001	0.205	0.198	0.929	0.979	0.527	0.545
	4	0.837	0.837	0.0*	0.0*	0.007	0.007	0.165	0.168	0.857	0.863	0.859	0.859
	5	0.532	0.575	0.0*	0.0*	0.005	0.011	0.082	0.1	0.001	0.002	0.798	0.814
	6	0.846	0.824	0.0*	0.0*	0.421	0.422	0.059	0.053	0.77	0.745	0.932	0.812
	7	0.354	0.366	0.0*	0.0*	0.135	0.132	0.125	0.113	0.0*	0.0*	0.44	0.459
	8	0.940	0.921	0.0*	0.0*	0.0*	0.0*	0.856	0.868	0.937	0.917	0.908	0.891
	9	0.896	0.883	0.0*	0.0*	0.0*	0.0*	0.718	0.729	0.919	0.912	0.958	0.955
	10	0.927	0.927	0.0*	0.08	0.009	0.009	0.865	0.853	0.89	0.905	0.962	0.962

* Values of 0.0 are the result of NaN

Conclusion:

Out of all the tests that were performed, Sigmoid showed to be the most accurate, at 0.756 accuracy against testing data, but also the most stable. With the other tests, you would quickly encounter Errors that causes all the results to give off NaN values. NaN values stand for Not a Number, this data type is used to describe a value that has no definition and occurs commonly when performing operations against numbers with a value of zero and below and more specific to Neural Networks, occurs due to data being missing, the gradient blowing up and bad learning rate definition.

We can rule out NaN occurring due to missing data since Sigmoid had run perfectly fine, then let's test the learning rate. Applying the same learning rate used in Sigmoid to ReLU (figure 8), which is 0.15, it's evident that somewhere along the passthrough of each node the values became too large for the program to handle. Part of the Confusion Matrix and Label Precisions were able to output a single line of results shows that initially the program could cope with the numbers however started a quick descent afterwards.

Fine tuning the learning rate, we finally came across a value that didn't simply produce NaN values, of which is 0.00001 (figure 9). At this miniscule learning rate, we achieved an accuracy of 20% (0.2074) for the testing set, and although the values were simply NaN, there were still NaN values present as shown in the Confusion Matrix and Label Precision; regarding the 8^{th} and 9^{th} label (1-10).

With this mindset in play, testing Leaky ReLU seemed to perform far better than ReLU and didn't produce any NaN values. This is probably due to Leaky ReLU being able to avoid saturation where the signal lost due to zero/vanishing gradient and chaotic noise during digital rounding.

For Softmax, we apply Sigmoid during the Input to Hidden Layer and then Softmax for Hidden to Output Layer. Here we have another activation function that can compete with the Sigmoid results, this may be due to use using the Sigmoid function for the first layer. The result also drew a NaN value which is mostly due to an overfitting issue.

Surprisingly, Sigmoid when applying dropout (0.7 active nodes) with exactly the same learning rate managed to achieve a 0.8057 accuracy during testing. This result might be an outlier however, since after running a few more times the test accuracy we've achieved had a low of 0.741 and the most was 0.8057; the initial run. These varying results are probably due to the nature of dropout where a random set of nodes are ignored, in the case of the highest accuracy result would be the nodes that were turned off were ones with values closer to the target label, and the lowest result with node values lower than the target label.