Koszalin University of Technology

Applications of artificial intelligence
project report

Handwritten text symbol recognition with deep neural networks

Paweł Frankowski *Kacper Ochnik*

supervised by
Dr. Adam Słowik

February 2, 2024

Contents

1 Introduction - Kacper

Handwritten text symbol recognition with deep neural networks. Feed Forward Networks (FFN) In feedforward neural network input moves from input to output layer. Backpropagation is propagatin the error backward from the output to the input layer.

2 Our goal - Pawel [Done]

The primary objective of our project is to develop a handwritten text symbol recognition system using deep neural networks. We aimed to create a model capable of accurately identifying and classifying handwritten digits ranging from 0 to 9 on a matrix of 28x28 pixels.

2.1 Specific objectives

In pursuit of our overarching goal, we have identified the following specific objectives:

1. **Project setup** - Set up the project environment and install the necessary libraries and packages.
2. **Code implementation** - Write Python code to implement the deep neural network architecture. This includes developing modules for data preprocessing, model training, and evaluation.
3. **Data Collection** - Gather a comprehensive dataset of handwritten digits (0 to 9) in a 28x28 pixel matrix format from MNIST.
4. **Learning** - Train the model using the collected dataset.
5. **Optimization** - Improve the model's accuracy through optimization techniques. Accelerate computational efficiency for faster calculations.
6. **Testing** - Create testing GUI for the trained model. Evaluate the model's performance metrics.
7. **Documentation** - Write a comprehensive report documenting the project's objectives, methodology, and outcomes.

2.2 Expected Outcomes

Upon successful completion of our project, we anticipate achieving the following outcomes:

- Develop a robust deep neural network model capable of recognizing and classifying handwritten digits from 0 to 9.
- Train the model to achieve a acceptable level of accuracy.

3 Decision boundary - Kacper

...

4 Weights and Biases - Pawel [Done]

Weights are indicators of the importance of the input in predicting the final output. Biases are essential as they allow the network to have some flexibility in fitting the data. In the training process of our neural network, the weights and biases play a crucial role in determining the model's performance. Here, we discuss the initialization, calculation, and adjustment of weights and biases in the network.

4.1 Initialization

The `Layer` class represents a single layer in the neural network. It contains information about the number of nodes in and out, weights, biases, and gradients. The neural network is initialized with random weights using the Xavier/Glorot initialization technique. This method helps in achieving better convergence during training.

```python
class Layer:
    def __init__(self, num_nodes_in, num_nodes_out):
        self.num_nodes_in = num_nodes_in
        self.num_nodes_out = num_nodes_out
        self.weights = np.random.randn(self.num_nodes_in, self.num_nodes_out) * np.sqrt(2.0 / (self.num_nodes_in + self.num_nodes_out))
        self.biases = np.random.randn(self.num_nodes_out)
        self.cost_gradient_w = np.zeros((self.num_nodes_in, self.num_nodes_out))
        self.cost_gradient_b = np.zeros(self.num_nodes_out)
        self.weight_constant = 0.9
```

Figure 1: Initialization of random biases and weights using Xavier/Glorot technique

When a neural network is first trained, it is first fed with input. Since it isn't trained yet, we don't know which weights should we use for each input. And so, each input is randomly assigned a weight. It will very likely give incorrect output.

4.2 Weighted sum of inputs

In the `Layer` class, the calculation of outputs begins with the computation of weighted inputs. For each node in the layer, the weighted inputs are calculated using the formula.:

```python
def calculate_outputs(self, inputs):
    weighted_inputs = np.dot(inputs, self.weights) + self.biases
    outputs = af.sigmoid(weighted_inputs)
    return outputs
```

Figure 2: The dot product of inputs and weights, representing the weighted sum of inputs

The `calculate outputs` method calculates the dot product of inputs and weights, representing the weighted sum of inputs. The biases are then added to this sum to introduce flexibility in fitting the data. Finally, the result is passed through the sigmoid activation function to introduce non-linearity to the model, and the outputs are obtained.

5 Hidden layers - Kacper

...

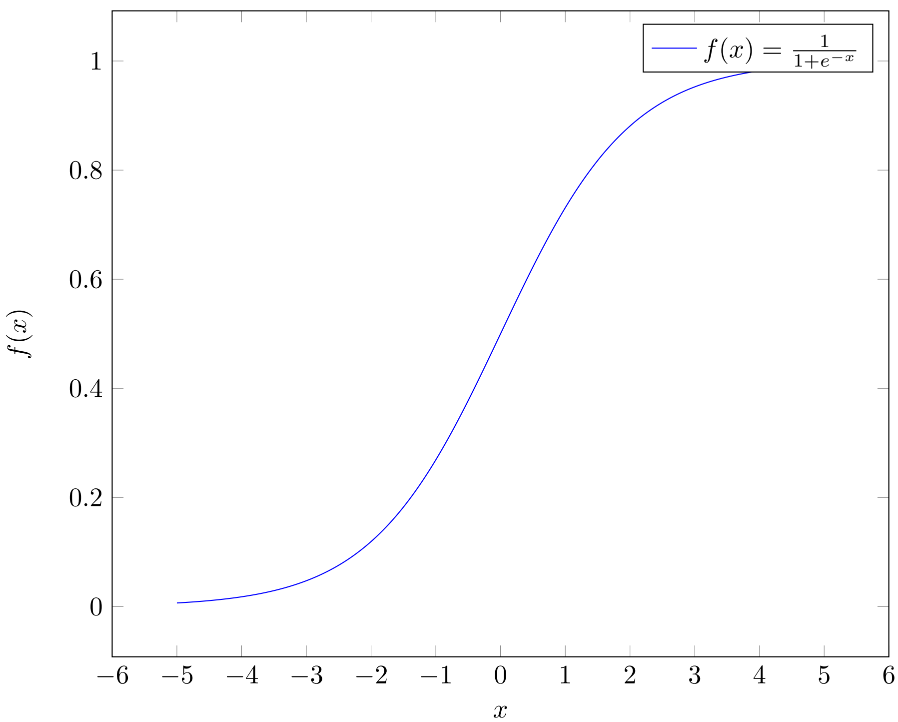
6 Activation functions - Pawel [Done]

Activation functions play a crucial role in neural networks, introducing non-linearity to the model and allowing it to learn complex relationships in the data. Here, we describe various activation functions implemented in the `activation functions` module.

6.1 Sigmoid function

The sigmoid function squashes its input to the range (0, 1), making it suitable for binary classification problems.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{1}$$



(2)

6.2 The derivative of the sigmoid function

The derivative of the sigmoid function is used in the backpropagation algorithm to calculate the gradients of the cost function with respect to the weights and biases. The derivative of the sigmoid function is given by the formula:

```python
def sigmoid(x):
    x = np.clip(x, -500, 500)
    return 1 / (1 + np.exp(-x))
```

Figure 3: Sigmoid function

Limiting the values of x to prevent overflow. Overflow refers to a situation where the exponential function in the sigmoid formula produces very large positive values, causing numerical instability in calculations. The exponential function, especially when dealing with large positive inputs, can lead to floating-point overflow, which means the result becomes too large to be represented within the numerical precision of the system.

$$f'(x) = f(x) \cdot (1 - f(x)) \tag{3}$$

```python
def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))
```

Figure 4: The derivative of the sigmoid function

7 Cost function - Kacper

...

8 Gradient descent and backpropagation - Pawel [Done]

8.1 Gradients of the cost function

Gradient Descent - Is an optimization algorithm that is used to find the weights that minimize the cost function. We need two things to do so. Direction in which to navigate and the size of the steps for navigating. To know which direction to navigate the cost function, gradient descent uses backpropagation. More specifically, it uses the gradients calculated through backpropagation. These gradients are used for determining the direction to navigate to find the minimum point. Descresing slope will lead us to the minimum point. The cost gradients for weights and biases are initialized with zeros, and during training, these gradients will be updated based on the backpropagation algorithm.

```python
class Layer:
    def __init__(self, num_nodes_in, num_nodes_out):
        self.num_nodes_in = num_nodes_in
        self.num_nodes_out = num_nodes_out
        self.weights = np.random.randn(self.num_nodes_in, self.num_nodes_out) * np.sqrt(2.0 / (self.num_nodes_in + self.num_nodes_out))
        self.biases = np.random.randn(self.num_nodes_out)
        self.cost_gradient_w = np.zeros((self.num_nodes_in, self.num_nodes_out))
        self.cost_gradient_b = np.zeros(self.num_nodes_out)
        self.weight_constant = 0.9
```

Figure 5: Initialization a vector of zeros with the same length as the bias vector

This matrix will be used to accumulate the gradients of the cost function with respect to the weights during the backpropagation process.

```python
def apply_gradients(self, learning_rate):
    for layer in self.layers:
        if self.layers.index(layer) == 0:
            continue
        else:
            layer.apply_gradients(learning_rate)
```

Figure 6: The `apply_gradients` method in `Neural network` class updates the weights and biases for each layer.

```python
class Layer:
    def apply_gradients(self, learning_rate):
        self.weights -= learning_rate * self.cost_gradient_w
        self.biases -= learning_rate * self.cost_gradient_b

```

Figure 7: The `apply_gradients` method in `Layer` class updates the weights and biases for a layer.

The `apply gradients` method is responsible for updating the weights and biases of a layer based on the calculated gradients.

8.2 Data loading and learning

```python
mnist = MnistDataloader(
    "./input/train-images-idx3-ubyte/train-images-idx3-ubyte",
    "./input/train-labels-idx1-ubyte/train-labels-idx1-ubyte",
    "./input/t10k-images-idx3-ubyte/t10k-images-idx3-ubyte",
    "./input/t10k-labels-idx1-ubyte/t10k-labels-idx1-ubyte",
)
mnist_data, mnist_test = mnist.load_data()
images, labels = mnist_data

nn = NeuralNetwork(784, 89, 10)
if os.path.exists('model.pkl'):
    nn = NeuralNetwork.load_model('model.pkl')
```

Figure 8: Data loading from mnist and creating a neural network

A neural network is instantiated with the specified architecture: 784 input nodes, one hidden layer with 89 nodes, and 10 output nodes. Checks if a pre-trained model file `model.pkl` exists.If it does, the model is loaded from the file. Otherwise, the model is trained using the MNIST dataset.

8.3 Backpropagation

Backpropagation - is a training algorithm for feedforward neural networks that propagates the error backward from the output to the input layer. It plays a crucial role in gradient descent, a process of finding the minimum of the cost function. Gradient descent relies on backpropagation to calculate gradients by moving backward in the neural network. Together, backpropagation and gradient descent is used for the purpose of improving the prediction accuracy of neural networks. They help improve prediction accuracy by reducing the output error in neural networks.

9 Cost landscape - Kacper

...

10 Learning algorithm - naive approach, calculus approach, digit recognition - Kacper

Kacper you should check other sections because there could be duplicates! ...

11 Chain rule - Kacper

...

12 Testing the network - Kacper

...

13 Conclusion - Pawel

This is the conclusion of the document.

```python
learning_rate = 0.8
batch_size = 100
numberOfSteps = 1000

training_data = []
for j in range(1000):
    flattened_image = [pixel for sublist in images[j] for pixel in sublist]
    training_data.append(DataPoint(flattened_image, labels[j], 10))
batches = []
for j in range(0, len(training_data), batch_size):
    batches.append(training_data[j:j + batch_size])

print("Data loaded. Starting training...")

for i in range(numberOfSteps):
    for j in range(len(batches)):
        nn.learn(batches[j], learning_rate)
        print("Step: ", i, "Batch: ", j, " Cost: ", nn.total_cost(batches[j]))
        plt.pause(0.01)
    nn.save_model('model.pkl')
    print("Step: ", i, " Cost: ", nn.total_cost(training_data))
    learning_rate *= 0.95
plt.show()
```

Figure 9: Training a neural network using a simple form of gradient descent on the MNIST dataset.

Learning rate - determines the step size at each iteration of gradient descent and the speed at which we move down the slope. Learning rate plays important role in between optimization time and accuracy. Step size is measured by a parameter alpha α. Small α means small step size, large α means large step size. If its too large then it can jump through minimum of the function. This parameter needs to be optimized. High learning rate results in a higher step value and opposite.
Batch size refers to the number of training examples utilized in one iteration.
The number of steps, represents the total number of times the entire training dataset is passed forward and backward through the neural network. Too few steps may lead to underfitting, while too many steps may result in overfitting on the training data. The optimal number of steps depends on the complexity of the task and the dataset.

```python
def backpropagate(self, data_point):
    outputs = self.calculate_outputs(data_point.inputs)
    deltas = [output - expected_output for output, expected_output in zip(outputs, data_point.expected_outputs)]

    for i in reversed(range(len(self.layers))):
        layer = self.layers[i]
        inputs = data_point.inputs if i == 0 else self.layers[i - 1].calculate_outputs(data_point.inputs)
        layer.calculate_gradients(inputs, deltas)
        if i > 0 and i < len(self.layers) - 1:
            deltas = np.dot(layer.weights, deltas) * af.sigmoid_derivative(layer.calculate_outputs(data_point.inputs))

```

Figure 10: The `backpropagate` function is part of the neural network training process.

```python
def calculate_gradients(self, inputs, deltas):
    self.cost_gradient_w = np.outer(inputs, deltas)
    self.cost_gradient_b = np.array(deltas)
```

Figure 11: The `calculate_gradients` method computes the gradients needed for the weight and bias adjustments.

It represents how much the weights and biases should be changed to minimize the difference between the predicted and expected outputs. It's like figuring out the direction and magnitude of the correction needed to improve the network's performance.