feed-forward-neural-networks

March 9, 2024

1 Implementing Feed Forward Neural Network for Multi-Class Image Classification using Fashion-MNIST Dataset with Pure Mathematical Gradient Descent

1.0.1 Author:

- `Sachin M Sabariram`
- Github
- Linkedin

1.1 Introduction

In the previous notebook, we used a single perceptron for multiclass classification on the classical MNIST dataset. However, it became evident that the model struggled to capture the non-linear relationships between the features (individual pixels). To address this limitation, we'll now implement a feed-forward neural network (FFNN) with multiple perceptrons and introduce a non-linear ReLU activation function between the layers.

Here, we are using the Fashion-MNIST dataset, which consists of images of clothing items instead of handwritten digits. Additionally, instead of relying on external libraries like PyTorch, we will implement the neural network using only NumPy. This approach will involve manually calculating derivatives for backpropagation, providing a deeper understanding of the underlying principles.

1.1.1 Notebook Contents

1. **Data Exploration and Preprocessing**
   - Load and explore the Fashion-MNIST dataset.
   - Preprocess the data for training and validation.
2. **Feed Forward Neural Network Architecture**
   - Define the architecture of the feed-forward neural network.
   - Introduce the concept of hidden layers and ReLU activation function.
3. **Model Training**
   - Implement forward propagation to compute predictions.
   - Implement backpropagation to compute gradients and update weights.
   - Train the model using gradient descent optimization.
4. **Evaluation and Validation**
   - Evaluate the trained model on the validation set.
   - Visualize the performance metrics such as accuracy and loss.
5. **Hyperparameter Tuning**

    - Explore the impact of different hyperparameters on model performance.
    - Fine-tune hyperparameters for optimal results.
6. **Conclusion**
    - Summarize key findings and insights from the implementation.
    - Discuss potential improvements and future directions.

1.2 Acknowledgments

This notebook is created for educational purposes, assuming a basic understanding of neural networks and machine learning concepts. It emphasizes implementing the neural network from scratch using NumPy for better comprehension of the underlying principles.

Note: Ensure that you have the required libraries installed by running the necessary `pip install` commands.

```
[356]: import numpy as np
```

1.2.1 Downloading and Preprocessing the dataset

We are gonna use the fashion MNIST dataset from the torchvision datasets library as it eases the process of manually downloading

```
[357]: from torchvision.datasets import FashionMNIST
```

```
[3]: dataset = FashionMNIST(root="./data", train=True, download=True)
```

```
[4]: len(dataset)
```

```
[4]: 60000
```

Creating train and validation data

```
[5]: from torch.utils.data import random_split
```

```
[6]: val_size = 10000
     train_size = len(dataset)-val_size
     train_ds, val_ds = random_split(dataset, [train_size, val_size])
     len(train_ds),len(val_ds)
```

```
[6]: (50000, 10000)
```

Seperating images and labels

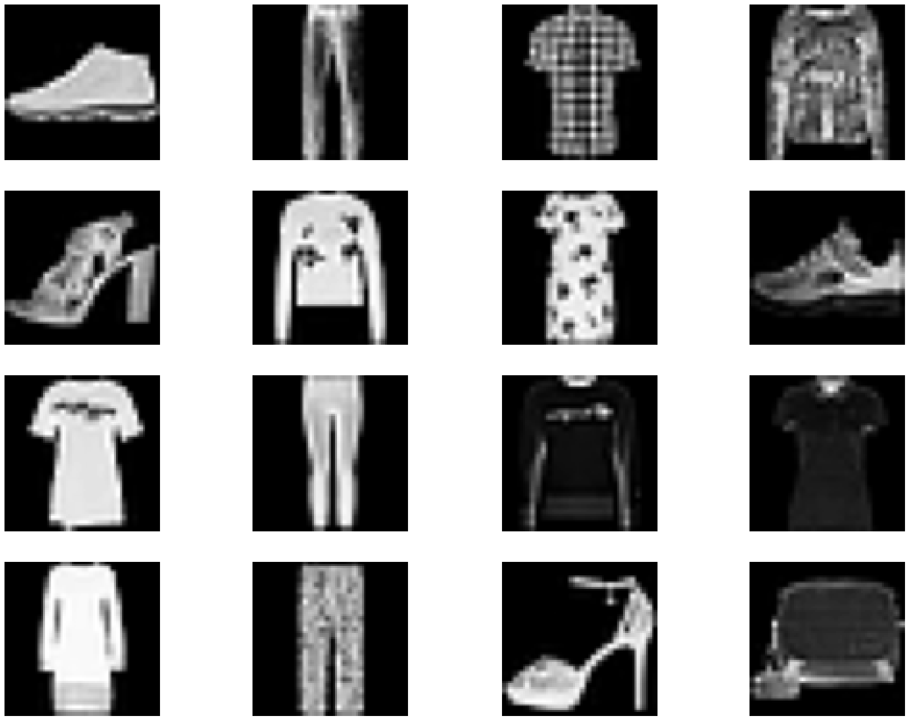
```
[7]: train_images = np.array([data[0] for data in train_ds])
     train_labels = np.array([data[1] for data in train_ds]).reshape(50000,1)
```

```
[8]: val_images = np.array([data[0] for data in val_ds])
     val_labels = np.array([data[1] for data in val_ds]).reshape(10000,1)
```

Let's visualise some data

```
[9]: from torchvision.utils import make_grid
     import matplotlib.pyplot as plt
```

```
[10]: images = train_images[:16]
      fig, axes = plt.subplots(4, 4)
      for i in range(4):
          for j in range(4):
              axes[i, j].imshow(images[i * 4 + j], cmap='gray')
              axes[i, j].axis('off')
      plt.show()
```



As it's a simple feed forward neural network so we are gonna flatten got the dimensions (meaning which each pixel will be treated as an individual feature), ofcourse it has it's drawbacks let's discuss it later in the results part

```
[11]: train_images_flatten = np.array([image.flatten() for image in train_images])
```

```
[221]: val_images_flatten = np.array([image.flatten() for image in val_images])
```

```
[12]: train_images_flatten.shape
```

```
[12]: (50000, 784)
```

1.2.2 Let's create helper functions

1.2.3 Functions Overview

- `linear_layer()`: This function helps us create the weights and biases based on the input and output shape. Typically, the weight matrix (w) should be compatible with the input (no. of inputs x no. of features), resulting in the shape (no. of inputs x no. of outputs). Therefore, w should be initialized to the shape (no. of outputs x no. of features), and the bias should be of shape (no. of outputs).
- `linear_function()`: Implements the basic linear equation of the perceptron, which is y = X.wT + b, where X is the input, w is the weight matrix, b is the bias vector, and y is the output.
- `leaky_relu()`: Activation function that introduces non-linearity. We use leaky ReLU to address the vanishing gradient problem.
- `softmax()`: This function converts the outputs from the final layers to logits. It is essential for multi-class classification tasks. The choice of softmax is discussed in detail in our previous notebook.
- `hot_encode()`: Converts numerical labels into one-hot embeddings to match with the softmax output. This function is crucial for training multi-class classifiers.
- `cross_entropy()`: Loss function used for multi-class classification. It calculates the difference between the predicted probabilities (after softmax) and the true labels.
- `accuracy()`: Calculates the accuracy by comparing predictions with actual labels.

[360]:
```
def linear_layer(input_size, output_size):
    w = np.random.randn(output_size, input_size) * np.sqrt(2 / input_size)  #␣
↪Xavier initialization
    b = np.zeros(output_size)
    return [w, b]
```

[361]:
```
def linear_function(x, w, b):
    return x @ w.T + b
```

[362]:
```
def leaky_relu(x, alpha=0.01):  # Leaky ReLU activation
    return np.where(x > 0, x, x * alpha)
```

[363]:
```
test_param = linear_layer(784,10)
linear_function(train_images_flatten, test_param[0], test_param[1]).shape
```

[363]:
```
(50000, 10)
```

[364]:
```
leaky_relu(np.array([1,2,-3,5,-4]))
```

[364]:
```
array([ 1.  ,  2.  , -0.03,  5.  , -0.04])
```

[365]:
```
def softmax(preds):
    Ei = np.exp(preds - np.max(preds, axis=1, keepdims=True))
    return Ei / np.sum(Ei, axis=1, keepdims=True)
```

[22]:
```
def hot_encode(y, n_class):
    encoded = np.zeros((y.shape[0], n_class))
    positions = y[:, 0].astype(int) - 1
    encoded[np.arange(y.shape[0]), positions] = 1
    return encoded
```

[23]:
```
outputs = hot_encode(train_labels, 10)
```

[222]:
```
val_output = hot_encode(val_labels, 10)
```

[24]:
```
def cross_entropy_loss(y_true, y_pred):
    """
    Calculates the cross-entropy loss between true labels and predicted labels.

    Args:
        y_true: A numpy array of true labels (one-hot encoded).
        y_pred: A numpy array of predicted probabilities (output of softmax).

    Returns:
        The cross-entropy loss value.
    """

    # Clip predictions to avoid log(0) errors
    y_pred = np.clip(y_pred, 1e-9, 1 - 1e-9)

    # Calculate cross-entropy for each sample and element
    loss = -np.sum(y_true * np.log(y_pred), axis=1)

    # Return average loss across all samples
    return np.mean(loss)
```

[367]:
```
def batch_norm(x):
    mean = np.mean(x, axis=0)
    variance = np.var(x, axis=0)
    x_normalized = (x - mean) / np.sqrt(variance + 1e-9)
    return x_normalized
```

[368]:
```
def accuracy(y, pred):
    return np.sum(np.equal(np.argmax(y, axis=1), np.argmax(pred, axis=1))) /␣
 ↪len(y)
```

1.2.4 Let's define the base model architecture !

[372]:
```python
class forward_model:
    def __init__(self, x, outputs):
        self.layer1 = linear_layer(784, 32)
        self.layer2 = linear_layer(32, 64)
        self.layer3 = linear_layer(64, 10)
        self.x = x
        self.y = outputs

    def forward(self,input):
        self.input = input
        self.out1 = linear_function(self.input, self.layer1[0], self.layer1[1])
        self.out1_bn = batch_norm(self.out1)
        self.out1_relu = leaky_relu(self.out1_bn)

        self.out2 = linear_function(self.out1_relu, self.layer2[0], self.
 ↪layer2[1])
        self.out2_bn = batch_norm(self.out2)
        self.out2_relu = leaky_relu(self.out2_bn)

        self.out3 = linear_function(self.out2_relu, self.layer3[0], self.
 ↪layer3[1])
        self.final_out = softmax(self.out3)
        self.loss = cross_entropy_loss(self.outputs, self.final_out)
```

1.2.5 Computing gradients for the backprop (Gradient descent)

1.2.6 Gradients of Loss Function with Respect to Parameters

In the context of neural networks, computing gradients is essential for updating model parameters during the training process. Let's delve into the gradients of the loss function with respect to the parameters.

The loss function (L) is defined as the cross-entropy loss:

$$Loss(L) = -\sum y_{\text{true}} \cdot \log(y_{\text{pred}})$$

And the softmax activation (S) is given by:

$$Softmax(S) = \frac{e^{Z_i - \max(Z)}}{\sum_k e^{Z_i - \max(Z)}}$$

Where

$$Z = (X \cdot W + b)$$

, representing the logits obtained from the linear transformation of input (X) with weights (W) and bias (b).

To compute the gradients, we use the chain rule:

1. **Partial derivative of Loss with respect to Predicted Probabilities** (dL/dS)**:** This is explained in detail in this article, and for our purposes, we can directly use:

$$\frac{dL}{dS} = -\sum y_{\text{true}} \cdot \frac{1}{y_{\text{pred}}}$$

2. **Partial derivative of Softmax Output with respect to Logits ((dL/dZ)):** The combined derivative of the loss function with respect to logits can be computed as detailed in the above article reference.

$$\frac{dL}{dZ_3} = y_{\text{pred}} - y_{\text{true}}$$

   For the subsequent chain rule, we need to find:

$$\frac{dL}{dW_3} = \frac{dL}{dZ_3} \cdot \frac{dZ_3}{dW_3}$$

$$\frac{dL}{db_3} = \frac{dL}{dZ_3} \cdot \frac{dZ_3}{db_3}$$

3. **Partial derivative of Logits with respect to Weight W3:**

$$\frac{dZ_3}{dW_3} = out2_{\text{relu}}$$

4. **Partial derivative of Logits with respect to Bias b3:**

$$\frac{dZ_3}{db_3} = 1$$

Therefore, the final expressions for the gradients at final layer are:

$$\frac{dL}{dW_3} = (y_{\text{pred}} - y_{\text{true}})T \cdot out2_{\text{relu}}$$

$$\frac{dL}{db_3} = \sum(y_{\text{pred}} - y_{\text{true}})$$

Likewise the gradients for the subsequent parameters can you computed using the chain rule by propagating the loss,

$$\frac{dL}{dZ_2} = \frac{dL}{dZ_3} \cdot \frac{dZ_3}{dA_2} \cdot \frac{dA2}{dZ_2}$$

$$\frac{dL}{dZ_1} = \frac{dL}{dZ_3} \cdot \frac{dZ_3}{dA_2} \cdot \frac{dA2}{dZ_2} \cdot \frac{dZ_2}{dA_1} \cdot \frac{dA_1}{dZ_1}$$

Equating and solving the equations, we get

$$\frac{dL}{dW_2} = (((y_{\text{pred}} - y_{\text{true}})T \cdot W_3) \cdot dReLu_{\text{out2}}) \cdot out1_{\text{relu}}$$

$$\frac{dL}{db_3} = \sum((y_{\text{pred}} - y_{\text{true}})T \cdot W_3) \cdot dReLu_{\text{out2}})$$

And finally,

$$\frac{dL}{dW_2} = ((((y_{\text{pred}} - y_{\text{true}})T \cdot W_3) \cdot dReLu_{\text{out2}}) \cdot out1_{\text{relu}}) \cdot dReLu_{\text{out1}}) \cdot X$$

$$\frac{dL}{db_3} = \sum((((y_{\text{pred}} - y_{\text{true}})T \cdot W_3) \cdot dReLu_{\text{out2}}) \cdot out1_{\text{relu}}) \cdot dReLu_{\text{out1}})$$

These gradients guide the parameter updates during the optimization process, enabling the model to learn and improve its predictions.

[366]:
```
def relu_derivative(x):
    return np.where(x > 0, 1, 0.01)  # Derivative of Leaky ReLU
```

Putting everything together!

[370]:
```
class forward_model:
    def __init__(self, x, outputs):
        self.layer1 = linear_layer(784, 32)
        self.layer2 = linear_layer(32, 64)
        self.layer3 = linear_layer(64, 10)
        self.x = x
        self.y = outputs

    def forward(self,input):
        self.input = input
        self.out1 = linear_function(self.input, self.layer1[0], self.layer1[1])
        self.out1_bn = batch_norm(self.out1)
        self.out1_relu = leaky_relu(self.out1_bn)

        self.out2 = linear_function(self.out1_relu, self.layer2[0], self.
 ↪layer2[1])
        self.out2_bn = batch_norm(self.out2)
        self.out2_relu = leaky_relu(self.out2_bn)

        self.out3 = linear_function(self.out2_relu, self.layer3[0], self.
 ↪layer3[1])
        self.final_out = softmax(self.out3)
        self.loss = cross_entropy_loss(self.outputs, self.final_out)

    def predict(self,image):
        self.out1 = linear_function(image, self.layer1[0], self.layer1[1])
        self.out1_bn = batch_norm(self.out1)
        self.out1_relu = leaky_relu(self.out1_bn)

        self.out2 = linear_function(self.out1_relu, self.layer2[0], self.
↪layer2[1])
        self.out2_bn = batch_norm(self.out2)
        self.out2_relu = leaky_relu(self.out2_bn)

        self.out3 = linear_function(self.out2_relu, self.layer3[0], self.
↪layer3[1])
        self.final_out = softmax(self.out3)

        return np.argmax(self.final_out, axis=1)

    def backward(self, lr):
        def gradient3(out2_relu=self.out2_relu,y=self.outputs,pred=self.
↪final_out):
            """
                out2 = (50000 x 64)
                (y - pred) = (50000 x 10)
                out - (10, 64)
            """
            dw3 =  (pred-y).T @ out2_relu
            db3 = np.sum((pred-y), axis=0)

            return (dw3,db3)

        def gradient2(out1_relu=self.out1_relu, out2=self.out2, y=self.outputs,␣
↪pred=self.final_out, W3=self.layer3[0]):
            """
                out1_relu = (50000 x 32)
                out2 = (50000 x 64)
                (y-pred) = (50000 x 10)
                w3 = (10,64)
                out = 64 x 32
            """
            dw2 =  (((pred-y) @ W3) * relu_derivative(out2)).T @ out1_relu
            db2 = np.sum(((pred-y) @ W3) * relu_derivative(out2), axis=0)

            return (dw2, db2)

        def gradient1(x=self.input, out1=self.out1, out2=self.out2, y=self.
↪outputs, pred=self.final_out, W2=self.layer2[0], W3=self.layer3[0]):
            """
                x = (50000 x 784)
                out1 = (50000 x 32)
                out2 = (50000 x 64)
                (y - pred) = (50000 x 10)
                W2 = (64 x 32)
                W3 = (10 x 64)
                out = 32 x 784
            """
            dw1 = (((((pred-y) @ W3) * relu_derivative(out2)) @ W2) *␣
↪relu_derivative(out1)).T @ x
            db1 = np.sum(((((pred-y) @ W3) * relu_derivative(out2)) @ W2) *␣
↪relu_derivative(out1), axis=0)

            return (dw1, db1)


        dw3, db3 = gradient3()
        dw2, db2 = gradient2()
        dw1, db1 = gradient1()

        # Update parameters of layer 3
        self.layer3[0] -= dw3 * lr
        self.layer3[1] -= db3 * lr

        # Update parameters of layer 2
        self.layer2[0] -= dw2 * lr
        self.layer2[1] -= db2 * lr

        # Update parameters of layer 1
        self.layer1[0] -= dw1 * lr
        self.layer1[1] -= db1 * lr

    def validation_step(self):
        self.outputs = val_output
        self.forward(val_images_flatten)
        print("val Acc:", accuracy(self.final_out, val_output))

    def train(self, epochs, lr):
        for _ in range(epochs):
            losses = []
            for x,y in zip(np.array_split(self.x,250),np.array_split(self.
↪y,250)):
                self.outputs = y
                self.forward(x)
                self.backward(lr)
                losses.append(self.loss)
            print("Train_loss", np.mean(losses))
            self.validation_step()
```

[322]:
```
model4 = forward_model1(train_images_flatten, outputs)
```

Training the Model

[325]:
```
model4.train(10,1e-2)
```

```
Train_loss 0.7063300300725631
val Acc: 0.8138
Train_loss 0.6662926364944667
val Acc: 0.8144
Train_loss 0.6771065247123004
val Acc: 0.8209
Train_loss 0.639487538812707
val Acc: 0.8207
Train_loss 0.6479291617816061
val Acc: 0.8247
Train_loss 0.6443258565726947
val Acc: 0.8281
Train_loss 0.6266126193949508
val Acc: 0.8259
Train_loss 0.643697757571862
val Acc: 0.8295
Train_loss 0.6236391890414471
val Acc: 0.8255
Train_loss 0.6136685391446344
val Acc: 0.8316
```

[326]:
```
preds = model4.predict(train_images_flatten)
```

[327]:
```
accuracy(model4.final_out,outputs)
```

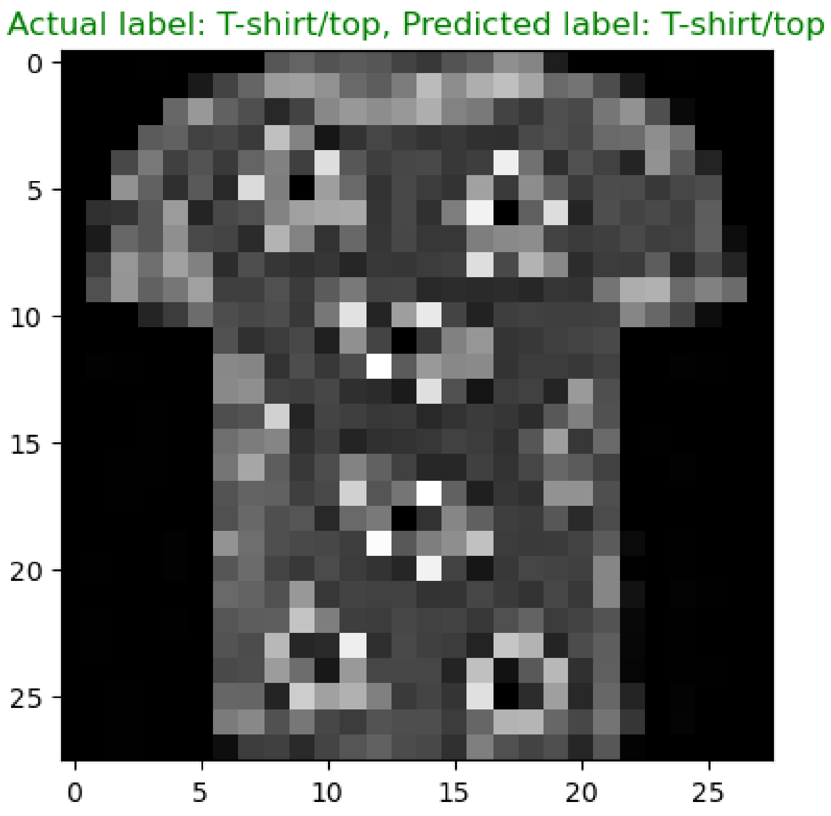
[327]:
```
0.83662
```

The model was trianed using variable hyperameters of learning rates (which turned out to be very crucial in this implementation), start with a higher learning rate and gradually lower the learning rate as the loss starts going up

Let's validate the model

[375]:
```
from IPython.display import display, clear_output
import time

def test(number, display_time=1):
    clear_output(wait=True)
    plt.imshow(val_images[number], cmap='gray')

    actual_label = dataset.classes[val_labels[number][0]]
    preds = (model4.predict(np.
 ↪array([val_images_flatten[number],val_images_flatten[2]]))[0]+1)%10
    print(val_labels[number][0],preds)
    predicted_label = dataset.classes[preds]
    if actual_label == predicted_label:
        title_color = 'green'
        correct_prediction = 1
    else:
        title_color = 'red'
        correct_prediction = 0

    plt.title(f"Actual label: {actual_label}, Predicted label:␣
 ↪{predicted_label}", color=title_color)
    plt.show()
    time.sleep(display_time)

    return correct_prediction

# Initialize counters
total_correct = 0
total_samples = 10

# Test and count correct predictions
for i in range(total_samples):
    random_sample = int(np.random.uniform(1, 10000))
    total_correct += test(random_sample)

# Display count of correct predictions
print(f"\nTotal Correct Predictions: {total_correct}/{total_samples}")
```

```
0 0
```



```
Total Correct Predictions: 7/10
```

[376]:
```
test_dataset = FashionMNIST(root="./data", train=False, download=True)
```

[382]:
```
test_images = np.array([data[0] for data in test_dataset])
test_labels = np.array([data[1] for data in test_dataset]).reshape(10000,1)
```

[384]:
```
test_images_flatten = np.array([image.flatten() for image in test_images])
test_images_flatten.shape
```

[384]:
```
(10000, 784)
```

[402]:
```
test_preds = (model4.predict(test_images_flatten).reshape(10000,1)+1)%10
```

[407]:
```
(np.sum(np.equal(test_preds,test_labels))/len(test_preds))*100
```

[407]:
```
82.32000000000001
```

Phew! We got 82% accuracy on the test dataset

1.2.7 Summary

- Started from a simple perceptron, now rocking a multi-layer neural network with cool activation functions.
- Explored batch normalization and gradients, boosting accuracy over 80% - pretty neat, huh?
- But hey, can we kick it up a notch, especially with real-world images?
- Found a snag: our pixel-centric approach falls short for objects not centered in images.
- Enter Convolutional Neural Networks (CNNs) - the heroes of image understanding!
- Every image model's BFF? You guessed it, CNNs!
- Ready for the CNN adventure? Let's dive in!
- But wait, what about data augmentation? It can supercharge our training data!
- Oh, and let's not forget transfer learning - it's like getting a head start on training.
- Happy coding, and let's keep the image mastery vibes going!