multi-class-dl

March 9, 2024

1 Mathematical Approach to Multi-Class Image Classification using Gradient Descent with MNIST Dataset

1.0.1 Author:

- `Sachin M Sabariram`
- Github
- Linkedin

1.1 Introduction

This Jupyter Notebook demonstrates a mathematical approach to solving the problem of multi-class image classification using the MNIST dataset. The primary focus is on implementing a gradient descent-based solution to train a model for recognizing handwritten digits, in the later part of the notebook we implemented a simplified pytorch version of the same.

1.1.1 Notebook Contents

1. **Data Exploration and Preprocessing**
   - Load and explore the MNIST dataset.
   - Preprocess the data for training and validation.
2. **Linear Regression for Image Classification**
   - Implement linear regression for initial understanding.
   - Extend linear regression to multi-class classification.
3. **Softmax activation and Cross-entropy Loss**
   - Implement Softmax activation to convert logits to probs.
   - Implemnt Categorical Cross-entropy to compute the loss.
4. **Gradient Descent Optimization**
   - Compute the partial deverivatives of the loss function w.r.t params.
   - Apply gradient descent to optimize the model parameters.
5. **Training and Validation**
   - Training the model
   - Performing validation and Visualizing results.
   - Extracting weights from the model & Visualizing.
6. **Pytorch Implementation**
   - Implement the same using Pytorch
   - Formalising as a model
7. **Conclusion**
   - Summarize key findings and way ahead.

1.2 Acknowledgments

This notebook is created for educational purposes, and it assumes a basic understanding of mathematics and machine learning concepts.

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

1.2.1 Data loading and Preprocessing (MNIST)

[32]:
```
from math import exp
import numpy as np
```

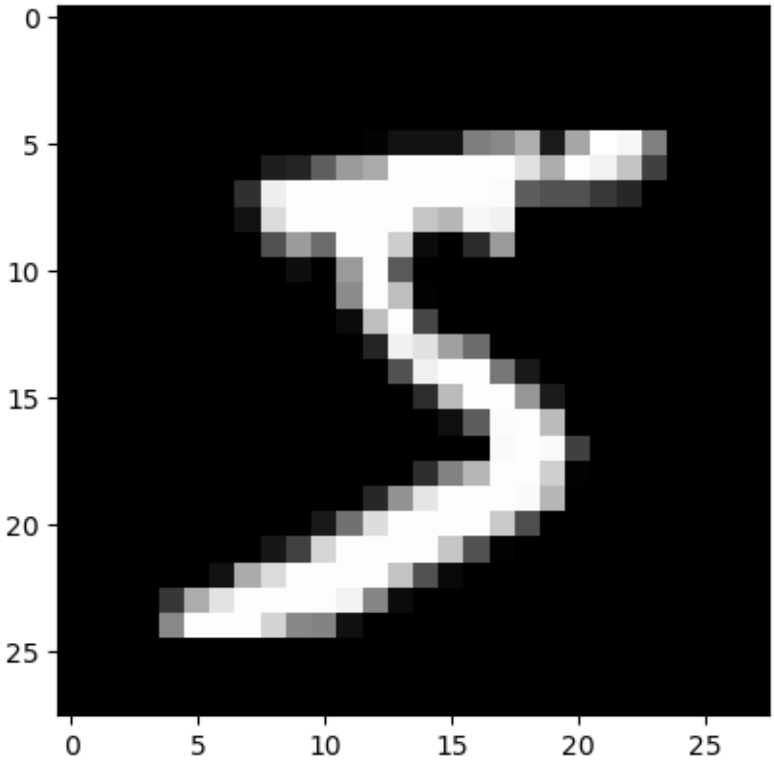
[54]:
```
from torchvision.datasets import MNIST
```

[65]:
```
#dataset = MNIST(root="./data", train=True, download=True)
```

[60]:
```
import matplotlib.pyplot as plt
%matplotlib inline
```

[64]:
```
plt.imshow(dataset[0][0], cmap='gray')
```

[64]:
```
<matplotlib.image.AxesImage at 0x28eedbd90>
```



[69]:
```
from torch.utils.data import random_split
```

[71]:
```
train_ds, val_ds = random_split(dataset, [50000,10000])
print(len(train_ds), len(val_ds))
```

```
50000 10000
```

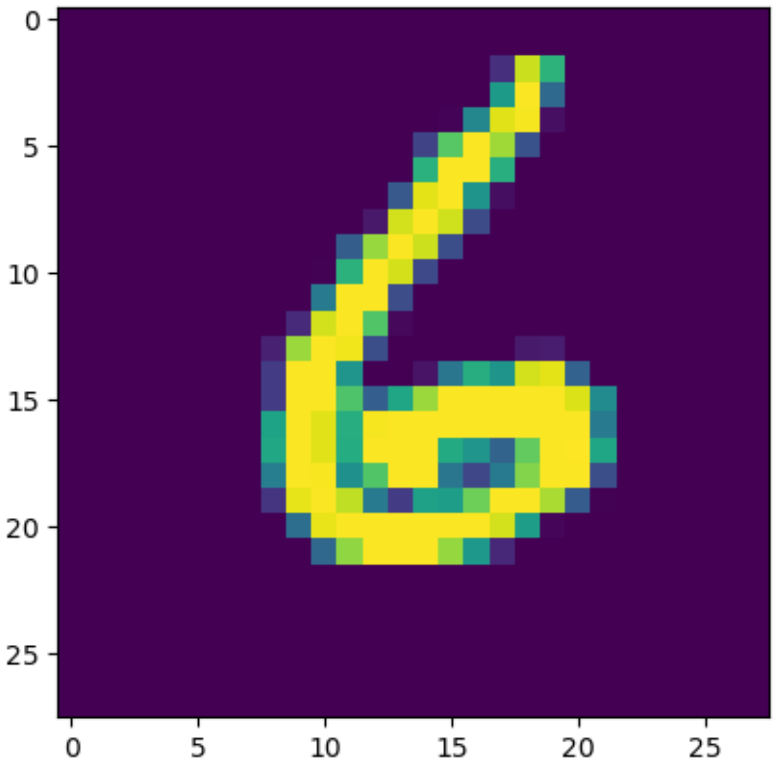
[93]:
```
train_images = np.array([data[0] for data in train_ds])
train_labels = np.array([data[1] for data in train_ds]).reshape(50000,1)
val_images = np.array([data[0] for data in val_ds])
val_labels = np.array([data[1] for data in val_ds]).reshape(10000,1)
```

[94]:
```
train_images.shape, train_labels.shape
```

[94]:
```
((50000, 28, 28), (50000, 1))
```

[95]:
```
plt.imshow(train_images[10])
print(train_labels[10])
```

```
[6]
```



[99]:
```
train_image_flatten = np.array([image.flatten() for image in train_images])
train_image_flatten.shape
```

[99]:
```
(50000, 784)
```

[107]:
```
inputs = train_image_flatten
```

1.2.2 Linear Layer for Classification

In the realm of neural networks, the linear layer serves as a fundamental building block, and its structure is reminiscent of that in Linear Regression. However, in the context of classification tasks, there are notable distinctions. The linear layer for classification retains the same structural foundation, but there are key differences: it produces 'n' outputs, each corresponding to a class label, and introduces an activation function along with a different loss function to compute the loss.

Structure: The structure of the linear layer remains consistent with the one employed in Linear Regression. However, for classification, the key modification lies in the number of outputs. Specifically, there are 'n' outputs, where 'n' represents the number of class labels. The parameters of this layer include:

- **Weights:** The weight matrix has dimensions No. of Output Labels (n) x No. of Features. Each row of the weight matrix corresponds to the set of weights associated with a particular class.
- **Bias:** The bias vector has dimensions No. of Output Labels (n). Each element of the bias vector corresponds to the bias term for a specific class label.

In summary, the linear layer transforms the input features into 'n' outputs, providing the raw scores or logits for each class.

Activation Function and Loss Function: To introduce non-linearity into the model and prepare the outputs for classification, an activation function is applied to the raw scores produced by the linear layer. Common activation functions include the softmax function, which converts the logits into probabilities, ensuring they are in the range [0, 1] and sum to 1 across all classes.

Parameters:

- **Weights:** The weight matrix `W` with dimensions $n \times$ No. of Features
- **Bias:** The bias vector `b` with dimensions n

This enhanced structure not only transforms the linear layer for classification but also emphasizes the significance of the activation function and its role in making the network suitable for multi-class classification tasks.

[256]:
```
w = np.random.randn(10,784)
b = np.random.randn(10)
```

[257]:
```
w.shape, b.shape
```

[257]:
```
((10, 784), (10,))
```

[258]:
```
def linear_function(x,w,b):
    return x@w.T + b
```

[259]:
```
preds = linear_function(inputs,w,b)
```

[260]:
```
preds.shape
```

[260]:
```
(50000, 10)
```

[261]:
```
preds[0]
```

[261]:
```
array([-1566.64369145, -1956.93471031, -3323.32965968,  3007.39365989,
        1440.39707145, -3816.11489615,  1669.36085159,   -83.41115368,
         931.3081048 ,  -933.28037131])
```

1.2.3 Implementing Softmax

In the context of the linear layer, the outputs $Z = (X \cdot w + b)$ can be large numbers, encompassing both positive and negative values. To transform these values into probabilities, a scaling operation is necessary.

Although the Sigmoid function

$$Sigmoid(Z) = \frac{1}{1 + e^{-z}}$$

can be employed to scale the values between 0 and 1, it does not ensure that these values sum up to one. This summation is crucial in multiclass classification scenarios, where we require probabilities for individual classes that collectively sum to 1.

To address this requirement, the Softmax function comes into play. It scales the outputs to the range of [0, 1], ensuring that they also sum to 1.

The Softmax function is defined as:

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

Here, The term `max(z)` helps with numerical stability during the exponentiation process.

[262]:
```
def softmax(preds):
    Ei = np.exp(preds - np.max(preds, axis=1, keepdims=True))
    return Ei / np.sum(Ei, axis=1, keepdims=True)
```

[263]:
```
preds_softmax = softmax(preds)
```

[264]:
```
preds_softmax.shape
```

[264]:
```
(50000, 10)
```

[265]:
```
preds_softmax[0],sum(preds_softmax[0])
```

[265]:
```
(array([0., 0., 0., 1., 0., 0., 0., 0., 0., 0.]), 1.0)
```

1.2.4 Implementing Cross entropy Loss

After applying the softmax function, we obtain a list of probabilities for each sample, corresponding to the number of classes. This implies that the class with the highest probability becomes the predicted output class.

To align these predicted probabilities with the actual labels, it is crucial to represent the output labels using one-hot encodings. This ensures a matching structure between the output of the softmax function and the true labels.

When we have two lists – one containing the predicted probabilities for each class and the other being a one-hot vector with a 1 for the true label and 0 otherwise – we can employ cross entropy as a differentiable loss function to measure the dissimilarity between the predicted and true distributions.

The cross-entropy loss is given by the formula:

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

Here, y_{true} represents the one-hot encoded true labels, and y_{pred} denotes the predicted probabilities for each class. The negative sign is included to ensure that the loss increases as the dissimilarity between the predicted and true distributions grows.

[266]:
```
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)
```

[267]:
```
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
```

[268]:
```
outputs = hot_encode(train_labels,10)
```

[269]:
```
outputs.shape
```

[269]:
```
(50000, 10)
```

[270]:
```
cross_entropy_loss(outputs, preds_softmax)
```

[270]:
```
19.954854304381726
```

1.2.5 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 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.

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

For the subsequent chain rule, we need to find:

$$\frac{dL}{dW} = \frac{dL}{dZ} \cdot \frac{dZ}{dW}$$

$$\frac{dL}{db} = \frac{dL}{dZ} \cdot \frac{dZ}{db}$$

3. **Partial derivative of Logits with respect to Weights ((dZ/dW)):**

$$\frac{dZ}{dW} = X^T$$

4. **Partial derivative of Logits with respect to Bias ((dZ/db)):**

$$\frac{dZ}{db} = 1$$

Therefore, the final expressions for the gradients are:

$$\frac{dL}{dW} = X^T \cdot (y_{\text{true}} - y_{\text{pred}})$$

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

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

[299]:
```
def gradient(x,y,pred):
    """
        X = (5000 x 784)
        (y - pred) = (5000 x 10)
        out - (10, 784)
    """
    def dLdW():
        return (y-pred).T @ x
    def dLdb():
        return np.sum((y-pred), axis=0)

    return (dLdW(),dLdb())
```

[288]:
```
dw,db = gradient(inputs,outputs,softmax(linear_function(inputs,w,b)))
```

[289]:
```
w1 = w + (dw * 1e-3)
b1 = b + (db * 1e-3)
```

[290]:
```
cross_entropy_loss(outputs,softmax(linear_function(inputs,w,b)))
```

[290]:
```
18.68699773590632
```

[291]:
```
cross_entropy_loss(outputs,softmax(linear_function(inputs,w1,b1)))
```

[291]:
```
7.701594516271124
```

[282]:
```
def accuracy(y,pred):
    return np.sum(np.equal(np.argmax(y, axis=1), np.argmax(pred, axis=1)))/
 ↪len(y)
```

[411]:
```
for i in range(20):
    preds = softmax(linear_function(inputs,w,b))
    loss = cross_entropy_loss(outputs, preds)
    dw, db = gradient(inputs, outputs, preds)
    w += dw * 1e-4
    b += db * 1e-5

    if i%5==0:
        print(f'\nLoss: {loss}')
        print(f'Accuracy: {accuracy(outputs,preds)*100:.3f}')
```

```
Loss: 1.6160002708871004
Accuracy: 92.202

Loss: 1.6155167119878502
Accuracy: 92.204

Loss: 1.6126845483533494
Accuracy: 92.218

Loss: 1.613513478986787
Accuracy: 92.214
```

[412]:
```
accuracy(outputs,preds)*len(outputs)
```

[412]:
```
46108.0
```

1.2.6 Validation

[320]:
```
val_images.shape
```

[320]:
```
(10000, 28, 28)
```

[323]:
```
val_inputs = np.array([image.flatten() for image in val_images])
```

[324]:
```
val_inputs.shape
```

[324]:
```
(10000, 784)
```

[325]:
```
val_labels.shape
```

[325]:
```
(10000, 1)
```

[326]:
```
val_outputs = hot_encode(val_labels,10)
```

[327]:
```
val_outputs.shape
```

[327]:
```
(10000, 10)
```

[413]:
```
val_preds = softmax(linear_function(val_inputs,w,b))
```

[414]:
```
cross_entropy_loss(val_outputs, val_preds)
```

[414]:
```
1.9169020908250427
```

[415]:
```
accuracy(val_outputs, val_preds)
```

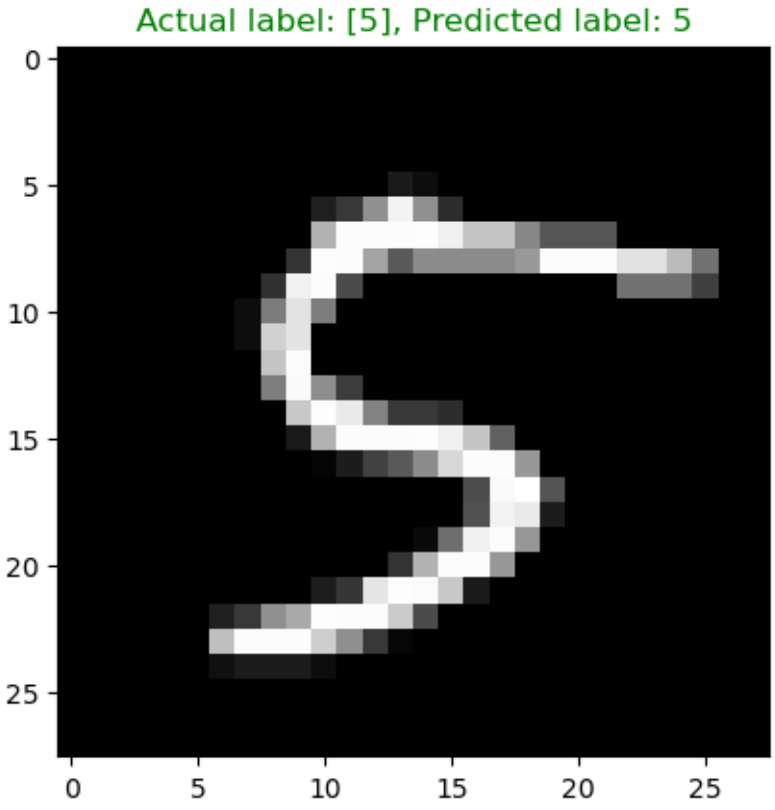
[415]:
```
0.9075
```

[420]:
```
np.random.uniform(1,10000)
```

[420]:
```
7218.846456798443
```

[490]:
```
from IPython.display import display, clear_output
import time

def test(number, display_time=2):
    clear_output(wait=True)
    plt.imshow(val_images[number], cmap='gray')

    actual_label = val_labels[number]
    predicted_label = np.argmax(val_preds[number]) + 1
    predicted_label =  0 if predicted_label==10 else predicted_label

    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}")
```




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

1.2.7 Visulalising the weights

[458]:
```
w.shape, b.shape
```

[458]:
```
((10, 784), (10,))
```

[459]:
```
w_comb = w + b.reshape(10,1)
```

[460]:
```
w_split = np.array([w_splits.reshape(28,28) for w_splits in w_comb])
```

[461]:
```
w_split.shape
```

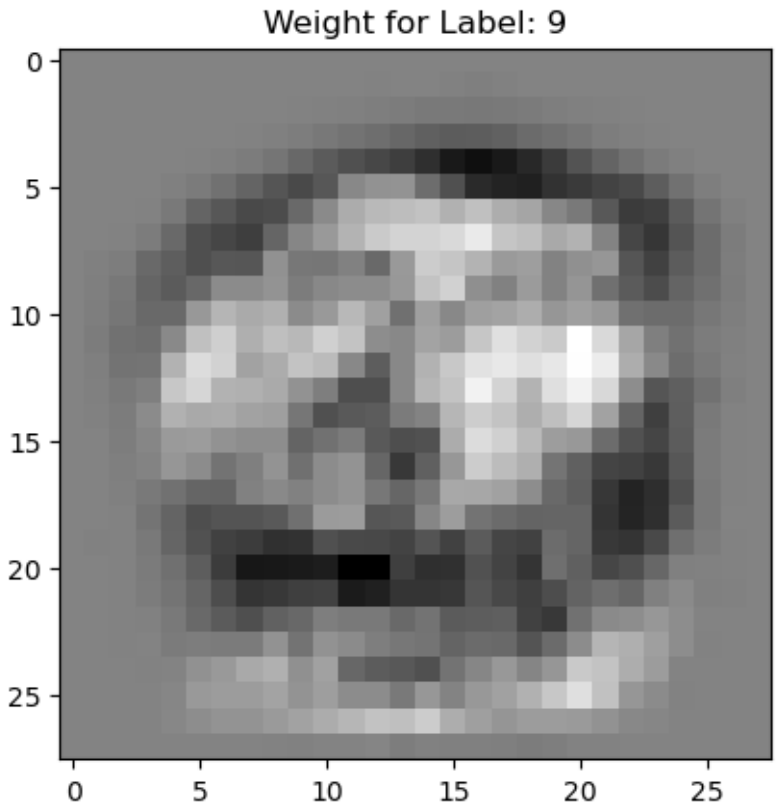
[461]:
```
(10, 28, 28)
```

[462]:
```
w_split = (w_split - np.min(w_split))/(np.max(w_split) - np.min(w_split))
```

[464]:
```
np.min(w_split), np.max(w_split)
```

[464]:
```
(0.0, 1.0)
```

[626]:
```
def w_show(img, label):
    clear_output(wait=True)
    plt.imshow(img, cmap='gray')
    plt.title(f'Weight for Label: {label}')
    plt.show()
    time.sleep(1)
for i in range(9):
    w_show(w_split[i], i+1)
```



By visualising the weights/bias we can see that our model had applied pixel level weightage for each labels

1.3 Pytorch Implementation

Importing the dataset & converting it to tensors

[ ]:
```
from torchvision.transforms import transforms
```

[505]:
```
torch_dataset = MNIST(root="./data", train=True, transform=transforms.
↪ToTensor())
```

[506]:
```
torch_dataset
```

[506]:
```
Dataset MNIST
    Number of datapoints: 60000
    Root location: ./data
    Split: Train
    StandardTransform
Transform: ToTensor()
```

[507]:
```
train_ds_tensor, val_ds_tensor = random_split(torch_dataset, [50000,10000])
```

[508]:
```
len(train_ds_tensor), len(val_ds_tensor)
```

[508]:
```
(50000, 10000)
```

Creating dataloader

[509]:
```
from torch.utils.data import DataLoader
```

[510]:
```
batch_size = 128
train_dl = DataLoader(train_ds_tensor, batch_size, shuffle=True)
val_dl = DataLoader(val_ds_tensor, batch_size)
```

[514]:
```
for image,label in train_dl:
    print(image.shape)
    print(label.shape)
    break
```

```
torch.Size([128, 1, 28, 28])
torch.Size([128])
```

Note We need to flatten the image to 28*28 as before before passing it to the model as we treat individual pixels as a feature

Creating the Model class

[516]:
```
import torch.nn as nn
```

[520]:
```
input_size = 28*28
no_classes = 10
```

[583]:
```
class MNIST(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer = nn.Linear(input_size, no_classes)

    def forward(self, image):
        #reshapes the image before passing it to the linear model
        image = image.reshape(-1, 784)
        out = self.layer(image)
        return out
```

[584]:
```
model = MNIST()
```

[585]:
```
print(model.layer.weight.shape)
print(model.layer.bias.shape)
print(list(model.parameters()))
```

```
torch.Size([10, 784])
torch.Size([10])
```

```
[Parameter containing:
tensor([[-0.0323,  0.0226,  0.0227,  …,  0.0047,  0.0277,  0.0306],
        [ 0.0270,  0.0352, -0.0019,  …, -0.0042,  0.0069, -0.0271],
        [-0.0335,  0.0242,  0.0101,  …,  0.0011, -0.0147, -0.0220],
        …,
        [ 0.0200, -0.0337,  0.0061,  …,  0.0189,  0.0174, -0.0106],
        [-0.0068,  0.0173,  0.0021,  …, -0.0017, -0.0112, -0.0086],
        [ 0.0152,  0.0205, -0.0069,  …,  0.0343,  0.0282,  0.0080]],
       requires_grad=True), Parameter containing:
tensor([ 0.0040, -0.0325, -0.0104, -0.0221, -0.0141, -0.0150,  0.0246, -0.0216,
         0.0334, -0.0065], requires_grad=True)]
```

[586]:
```
for images,labels in train_dl:
    print(images.shape)
    outputs = model(images)
    break
```

```
torch.Size([128, 1, 28, 28])
```

[587]:
```
outputs.shape
```

[587]:
```
torch.Size([128, 10])
```

[588]:
```
outputs[:2]
```

[588]:
```
tensor([[ 0.2037, -0.1295, -0.2655, -0.2487,  0.0830,  0.2305, -0.0959, -0.2870,
          0.2175, -0.0692],
        [-0.0066, -0.2114,  0.3139, -0.0906,  0.0068, -0.0864, -0.1630,  0.1087,
          0.0977,  0.1598]], grad_fn=<SliceBackward0>)
```

Let's convert to probs using softmax

[589]:
```
import torch
import torch.nn.functional as F
```

[590]:
```
probs = F.softmax(outputs, dim=1)
```

[591]:
```
probs[:2]
```

[591]:
```
tensor([[0.1247, 0.0894, 0.0780, 0.0793, 0.1105, 0.1281, 0.0924, 0.0763, 0.1264,
         0.0949],
        [0.0969, 0.0790, 0.1335, 0.0891, 0.0982, 0.0895, 0.0829, 0.1088, 0.1076,
         0.1145]], grad_fn=<SliceBackward0>)
```

[592]:
```
# Probabilities to Predictions
```

[593]:
```
_, preds = torch.max(probs, dim=1)
```

[594]:
```
preds[:2]
```

[594]:
```
tensor([5, 2])
```

[595]:
```
torch.sum(preds==labels)
```

[595]:
```
tensor(13)
```

Accuracy Function

[596]:
```
def accuracy(outputs, labels):
    _, preds = torch.max(outputs, dim=1)
    return torch.tensor(torch.sum(preds==labels).item() / len(preds))
```

[597]:
```
accuracy(outputs,labels)
```

[597]:
```
tensor(0.1016)
```

Loss Function and Optimiser

[600]:
```
loss = F.cross_entropy(outputs, labels)
```

[601]:
```
loss
```

[601]:
```
tensor(2.3095, grad_fn=<NllLossBackward0>)
```

Let's build a layout for the fit function

[609]:
```
def fit(model, train_loader, val_loader, lr, epochs):

    optimiser = torch.optim.SGD(model.parameters(), lr)
    history = [] #logging epoch-wise

    for epoch in range(1,epochs+1):

        # Training phase
        for batch in train_dl:
            loss = model.train_step(batch)
            loss.backward()
            optimiser.step()
            optimiser.zero_grad()

        # Validation phase
        result = evaluate(model, val_loader) # runs validation batches and␣
↪returns val_loss & val_acc

        model.epoch_end(epoch, result) #displays current epoch results
        history.append(result)
```

```
    return history
```

[603]:
```
### Evaluate in fit
def evaluate(model, val_loader):

    """
    validation_step: returns dictionary of 'val_acc' & 'val_loss' for each batch
    validation_end: returns average of 'val_acc' & 'val_loss' for all batches
    """

    outputs = [model.validation_step(batch) for batch in val_loader]
    return model.validation_end(outputs)
```

To-do: Missing functions in Model - train_step - validation_step - validation_end - epoch_end

[613]:
```
class MNIST(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer = nn.Linear(input_size, no_classes)

    def forward(self, image):
        #reshapes the image before passing it to the linear model
        image = image.reshape(-1, 784)
        out = self.layer(image)
        return out

    def train_step(self, batch):
        images, labels = batch
        out = self(images)
        loss = F.cross_entropy(out, labels)
        return loss

    def validation_step(self, batch):
        images, labels = batch
        out = self(images)
        loss = F.cross_entropy(out, labels)
        acc = accuracy(out, labels)
        return {'val_loss':loss, 'val_acc':acc}

    def validation_end(self, outputs):
        batch_loss = [batch['val_loss'] for batch in outputs]
        overall_loss = torch.stack(batch_loss).mean()
        batch_acc = [batch['val_acc'] for batch in outputs]
        overall_acc = torch.stack(batch_acc).mean()
        return {'val_loss':overall_loss.item(), 'val_acc':overall_acc.item()}
```

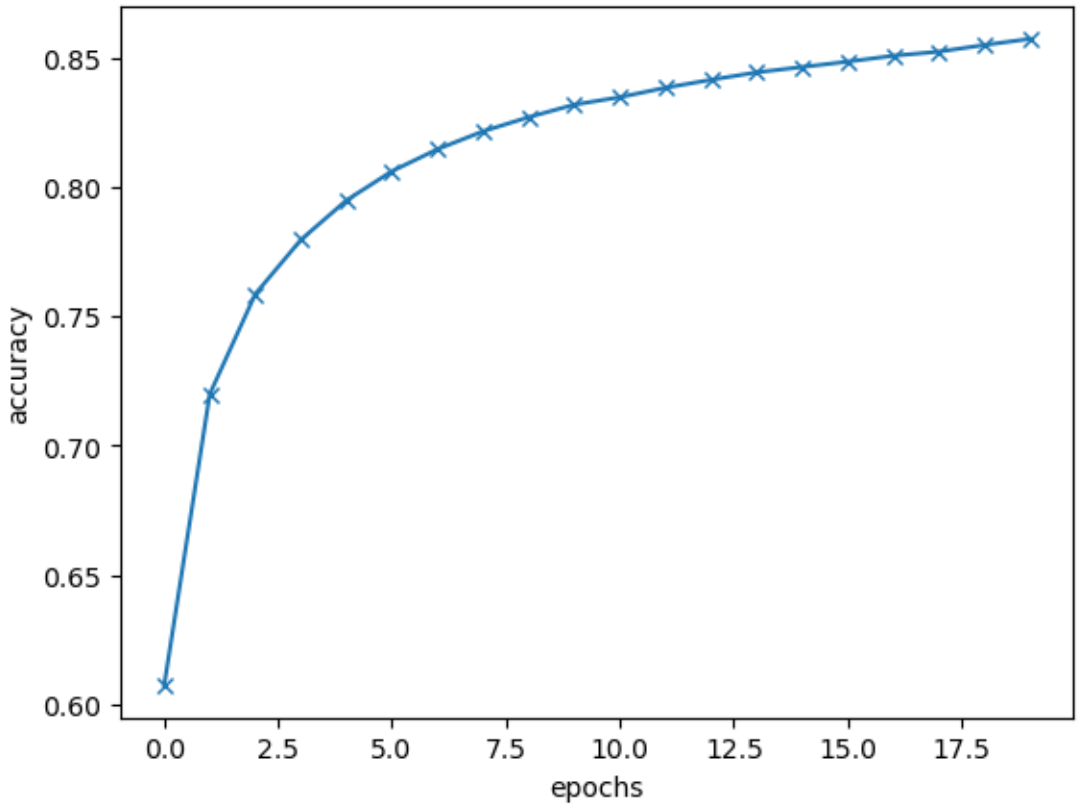
```
    def epoch_end(self, epoch,result):
        print(f"Epoch: {epoch} val_loss: {result['val_loss']} val_acc:␣
↪{result['val_acc']}")
```

[623]:
```
model = MNIST()
```

[624]:
```
history = fit(model, train_dl, val_dl, 1e-3, 20)
```

```
Epoch: 1 val_loss: 1.9808387756347656 val_acc: 0.6073971390724182
Epoch: 2 val_loss: 1.7033159732818604 val_acc: 0.719936728477478
Epoch: 3 val_loss: 1.4966230392456055 val_acc: 0.7586036324501038
Epoch: 4 val_loss: 1.3408232927322388 val_acc: 0.7795688509941101
Epoch: 5 val_loss: 1.2212696075439453 val_acc: 0.7947982549667358
Epoch: 6 val_loss: 1.127517580986023 val_acc: 0.8060719966888428
Epoch: 7 val_loss: 1.0524122714996338 val_acc: 0.8145767450332642
Epoch: 8 val_loss: 0.9911119341850281 val_acc: 0.821499228477478
Epoch: 9 val_loss: 0.940024197101593 val_acc: 0.826938271522522
Epoch: 10 val_loss: 0.896916925907135 val_acc: 0.8317840099334717
Epoch: 11 val_loss: 0.8600155115127563 val_acc: 0.8346518874168396
Epoch: 12 val_loss: 0.8280544877052307 val_acc: 0.8383108973503113
Epoch: 13 val_loss: 0.8001006841659546 val_acc: 0.8413766026496887
Epoch: 14 val_loss: 0.775393545627594 val_acc: 0.8442444801330566
Epoch: 15 val_loss: 0.7534260749816895 val_acc: 0.8463212251663208
Epoch: 16 val_loss: 0.7337090969085693 val_acc: 0.848397970199585
Epoch: 17 val_loss: 0.7159491181373596 val_acc: 0.8506724834442139
Epoch: 18 val_loss: 0.6998531818389893 val_acc: 0.8522547483444214
Epoch: 19 val_loss: 0.6851527690887451 val_acc: 0.8548259735107422
Epoch: 20 val_loss: 0.6716356873512268 val_acc: 0.8571993708610535
```

[625]:
```
acc = [x['val_acc'] for x in history]
plt.plot(acc,'-x')
plt.xlabel('epochs')
plt.ylabel('accuracy')
plt.show()
```



1.3.1 Summary

- In this notebook, we implemented a single-layer artificial neural network (ANN) for multiclass classification on the MNIST dataset. The demonstration showcased the efficacy of deep learning using mathematical concepts and its subsequent implementation in PyTorch.
- Despite achieving a validation accuracy exceeding 89%, the simplicity of the model limits its real-world applicability due to assumptions of linear relations and individual pixel weighting. Notably, our model assigns weights based on pixel-level information, a strategy that might not generalize well to more complex real-world scenarios.
- An inherent limitation arises from the nature of the MNIST dataset, which is highly preprocessed. Visualizing the model parameters reveals that our current approach assumes certain pixel-level relationships, such as assigning greater weight to middle pixels for a specific label, a notion inconsistent with real-world scenarios.
- To address these limitations, upcoming weeks will focus on the development of more complex artificial neural networks that can overcome these drawbacks and offer improved performance in diverse and challenging contexts.