Working with Images & Logistic Regression in PyTorch

Part 3 of "Deep Learning with Pytorch: Zero to GANs"

This tutorial series is a hands-on beginner-friendly introduction to deep learning using PyTorch, an open-source neural networks library. These tutorials take a practical and coding-focused approach. The best way to learn the material is to execute the code and experiment with it yourself. Check out the full series here:

1. PyTorch Basics: Tensors & Gradients
2. Gradient Descent & Linear Regression
3. Working with Images & Logistic Regression
4. Training Deep Neural Networks on a GPU
5. Image Classification using Convolutional Neural Networks
6. Data Augmentation, Regularization and ResNets
7. Generating Images using Generative Adversarial Networks

This tutorial covers the following topics:

- Working with images in PyTorch (using the MNIST dataset)
- Splitting a dataset into training, validation, and test sets
- Creating PyTorch models with custom logic by extending the `nn.Module` class
- Interpreting model outputs as probabilities using Softmax and picking predicted labels
- Picking a useful evaluation metric (accuracy) and loss function (cross-entropy) for classification problems
- Setting up a training loop that also evaluates the model using the validation set
- Testing the model manually on randomly picked examples
- Saving and loading model checkpoints to avoid retraining from scratch

How to run the code

This tutorial is an executable Jupyter notebook hosted on Jovian. You can *run* this tutorial and experiment with the code examples in a couple of ways: *using free online resources* (recommended) or *on your computer*.

Option 1: Running using free online resources (1-click, recommended)

The easiest way to start executing the code is to click the **Run** button at the top of this page and select **Run on Colab**. Google Colab is a free online platform for running Jupyter notebooks using

Google's cloud infrastructure. You can also select "Run on Binder" or "Run on Kaggle" if you face issues running the notebook on Google Colab.

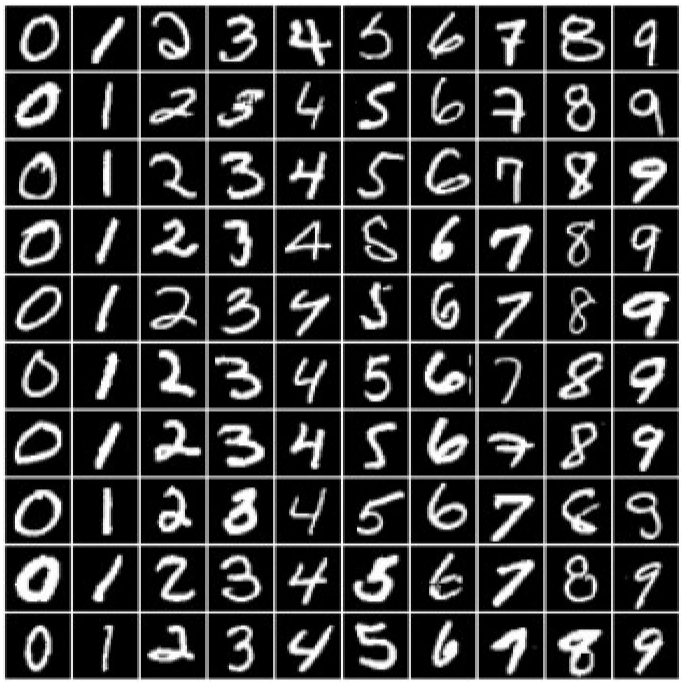
Option 2: Running on your computer locally

To run the code on your computer locally, you'll need to set up Python, download the notebook and install the required libraries. We recommend using the Conda distribution of Python. Click the **Run** button at the top of this page, select the **Run Locally** option, and follow the instructions.

> **Jupyter Notebooks**: This tutorial is a Jupyter notebook - a document made of *cells*. Each cell can contain code written in Python or explanations in plain English. You can execute code cells and view the results, e.g., numbers, messages, graphs, tables, files, etc., instantly within the notebook. Jupyter is a powerful platform for experimentation and analysis. Don't be afraid to mess around with the code & break things - you'll learn a lot by encountering and fixing errors. You can use the "Kernel > Restart & Clear Output" or "Edit > Clear Outputs" menu option to clear all outputs and start again from the top.

Working with Images

In this tutorial, we'll use our existing knowledge of PyTorch and linear regression to solve a very different kind of problem: *image classification*. We'll use the famous *MNIST Handwritten Digits Database* as our training dataset. It consists of 28px by 28px grayscale images of handwritten digits (0 to 9) and labels for each image indicating which digit it represents. Here are some sample images from the dataset:



We begin by installing and importing `torch` and `torchvision`. `torchvision` contains some utilities for working with image data. It also provides helper classes to download and import popular datasets like MNIST automatically

```
# Uncomment and run the appropriate command for your operating system, if required

# Linux / Binder
# !pip install numpy matplotlib torch==1.7.0+cpu torchvision==0.8.1+cpu torchaudio==

# Windows
# !pip install numpy matplotlib torch==1.7.0+cpu torchvision==0.8.1+cpu torchaudio==

# MacOS
# !pip install numpy matplotlib torch torchvision torchaudio
```

```
# Imports
import torch
import torchvision
from torchvision.datasets import MNIST
```

```
# Download training dataset
dataset = MNIST(root='data/', download=True)
```

```
100%|██████████| 9.91M/9.91M [00:00<00:00, 16.8MB/s]
100%|██████████| 28.9k/28.9k [00:00<00:00, 458kB/s]
100%|██████████| 1.65M/1.65M [00:00<00:00, 4.17MB/s]
100%|██████████| 4.54k/4.54k [00:00<00:00, 6.11MB/s]
```

When this statement is executed for the first time, it downloads the data to the `data/` directory next to the notebook and creates a PyTorch `Dataset`. On subsequent executions, the download is skipped as the data is already downloaded. Let's check the size of the dataset.

```
len(dataset)
```

```
60000
```

The dataset has 60,000 images that we'll use to train the model. There is also an additional test set of 10,000 images used for evaluating models and reporting metrics in papers and reports. We can create the test dataset using the `MNIST` class by passing `train=False` to the constructor.

```
test_dataset = MNIST(root='data/', train=False)
len(test_dataset)
```

```
10000
```

Let's look at a sample element from the training dataset.

```
dataset[0]
```

```
(<PIL.Image.Image image mode=L size=28x28>, 5)
```

It's a pair, consisting of a 28x28px image and a label. The image is an object of the class `PIL.Image.Image`, which is a part of the Python imaging library Pillow. We can view the image within Jupyter using `matplotlib`, the de-facto plotting and graphing library for data science in Python.

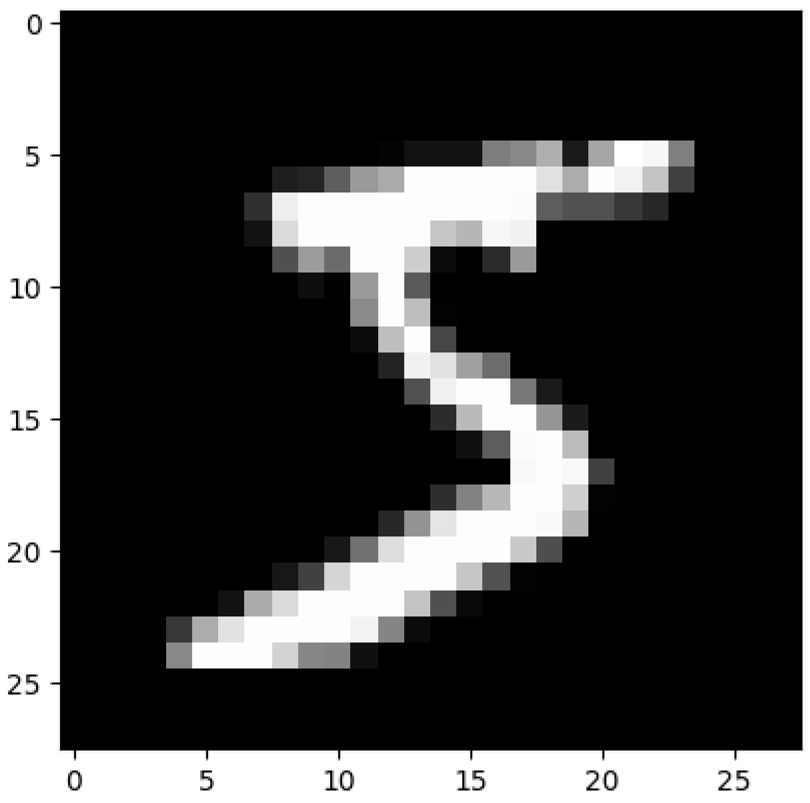
```
import matplotlib.pyplot as plt
%matplotlib inline
```

The statement `%matplotlib inline` indicates to Jupyter that we want to plot the graphs within the notebook. Without this line, Jupyter will show the image in a popup. Statements starting with `%` are called magic commands and are used to configure the behavior of Jupyter itself. You can find a full list of magic commands here: https://ipython.readthedocs.io/en/stable/interactive/magics.html .

Let's look at a couple of images from the dataset.

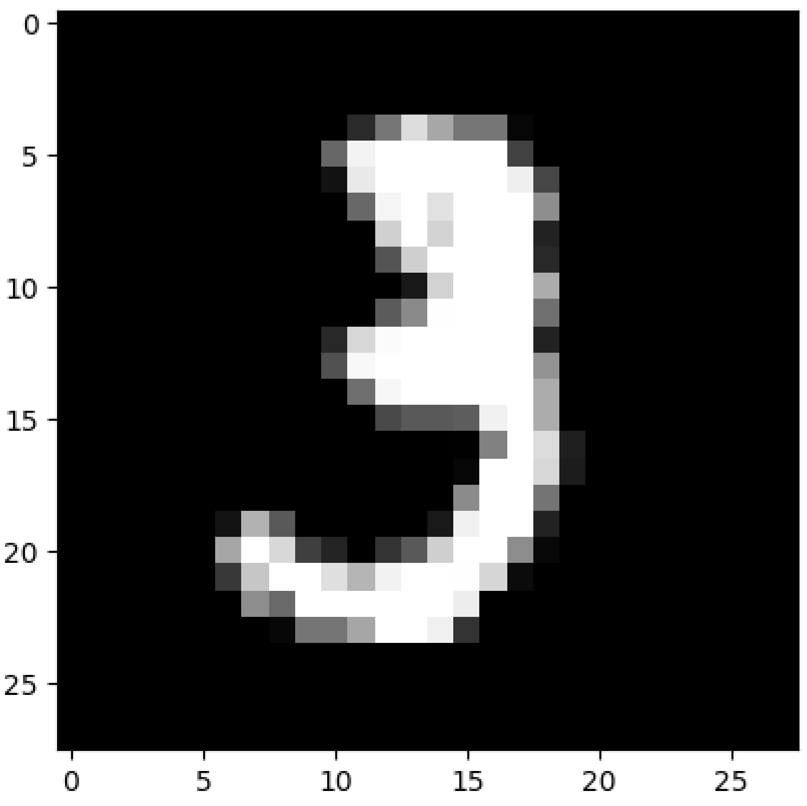
```
image, label = dataset[0]
plt.imshow(image, cmap='gray')
print('Label:', label)
```

```
Label: 5
```



```
image, label = dataset[10]
plt.imshow(image, cmap='gray')
print('Label:', label)
```

```
Label: 3
```



It's evident that these images are relatively small in size, and recognizing the digits can sometimes be challenging even for the human eye. While it's useful to look at these images, there's just one problem here: PyTorch doesn't know how to work with images. We need to convert the images into tensors. We can do this by specifying a transform while creating our dataset.

```
import torchvision.transforms as transforms
```

PyTorch datasets allow us to specify one or more transformation functions that are applied to the images as they are loaded. The `torchvision.transforms` module contains many such predefined functions. We'll use the `ToTensor` transform to convert images into PyTorch tensors.

```
# MNIST dataset (images and labels)
dataset = MNIST(root='data/',
                train=True,
                transform=transforms.ToTensor())
```

```
img_tensor, label = dataset[0]
print(img_tensor.shape, label)
```

```
torch.Size([1, 28, 28]) 5
```

The image is now converted to a 1x28x28 tensor. The first dimension tracks color channels. The second and third dimensions represent pixels along the height and width of the image, respectively. Since images in the MNIST dataset are grayscale, there's just one channel. Other datasets have images with color, in which case there are three channels: red, green, and blue (RGB).

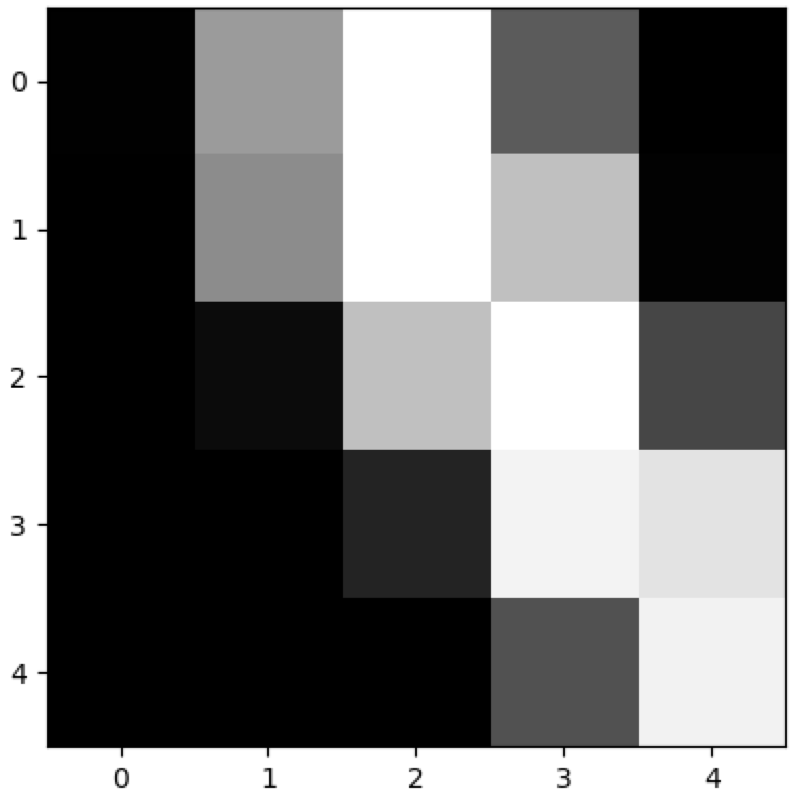
Let's look at some sample values inside the tensor.

```
print(img_tensor[:,10:15,10:15])
print(torch.max(img_tensor), torch.min(img_tensor))
```

```
tensor([[[0.0039, 0.6039, 0.9922, 0.3529, 0.0000],
         [0.0000, 0.5451, 0.9922, 0.7451, 0.0078],
         [0.0000, 0.0431, 0.7451, 0.9922, 0.2745],
         [0.0000, 0.0000, 0.1373, 0.9451, 0.8824],
         [0.0000, 0.0000, 0.0000, 0.3176, 0.9412]]])
tensor(1.) tensor(0.)
```

The values range from 0 to 1, with `0` representing black, `1` white, and the values in between different shades of grey. We can also plot the tensor as an image using `plt.imshow`.

```
# Plot the image by passing in the 28x28 matrix
plt.imshow(img_tensor[0,10:15,10:15], cmap='gray');
```



Note that we need to pass just the 28x28 matrix to `plt.imshow`, without a channel dimension. We also pass a color map (`cmap=gray`) to indicate that we want to see a grayscale image.

Training and Validation Datasets

While building real-world machine learning models, it is quite common to split the dataset into three parts:

1. **Training set** - used to train the model, i.e., compute the loss and adjust the model's weights using gradient descent.
2. **Validation set** - used to evaluate the model during training, adjust hyperparameters (learning rate, etc.), and pick the best version of the model.
3. **Test set** - used to compare different models or approaches and report the model's final accuracy.

In the MNIST dataset, there are 60,000 training images and 10,000 test images. The test set is standardized so that different researchers can report their models' results against the same collection of images.

Since there's no predefined validation set, we must manually split the 60,000 images into training and validation datasets. Let's set aside 10,000 randomly chosen images for validation. We can do this using the `random_spilt` method from PyTorch.

```
from torch.utils.data import random_split

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

```
(50000, 10000)
```

It's essential to choose a random sample for creating a validation set. Training data is often sorted by the target labels, i.e., images of 0s, followed by 1s, followed by 2s, etc. If we create a validation set using the last 20% of images, it would only consist of 8s and 9s. In contrast, the training set would contain no 8s or 9s. Such a training-validation would make it impossible to train a useful model.

We can now create data loaders to help us load the data in batches. We'll use a batch size of 128.

```
from torch.utils.data import DataLoader

batch_size = 128
```

```
train_loader = DataLoader(train_ds, batch_size, shuffle=True)
val_loader = DataLoader(val_ds, batch_size)
```

We set `shuffle=True` for the training data loader to ensure that the batches generated in each epoch are different. This randomization helps generalize & speed up the training process. On the other hand, since the validation data loader is used only for evaluating the model, there is no need to shuffle the images.

Save and upload your notebook

Whether you're running this Jupyter notebook online or on your computer, it's essential to save your work from time to time. You can continue working on a saved notebook later or share it with friends and colleagues to let them execute your code. Jovian offers an easy way of saving and sharing your Jupyter notebooks online.

```
# Install the library
!pip install jovian --upgrade --quiet
```

```
  Preparing metadata (setup.py) ... done
  ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 68.6/68.6 kB 2.9 MB/s eta 0:00:00
  Building wheel for uuid (setup.py) ... done
```

```
import jovian
```

```
jovian.commit(project='03-logistic-regression')
```

```
[jovian] Detected Colab notebook...
[jovian] jovian.commit() is no longer required on Google Colab. If you ran this
then just save this file in Colab using Ctrl+S/Cmd+S and it will be updated on J
Also, you can also delete this cell, it's no longer necessary.
```

`jovian.commit` uploads the notebook to your Jovian account, captures the Python environment, and creates a shareable link for your notebook, as shown above. You can use this link to share your work and let anyone (including you) run your notebooks and reproduce your work.

Model

Now that we have prepared our data loaders, we can define our model.

- A **logistic regression** model is almost identical to a linear regression model. It contains weights and bias matrices, and the output is obtained using simple matrix operations (`pred`

= x @ w.t() + b).

- As we did with linear regression, we can use `nn.Linear` to create the model instead of manually creating and initializing the matrices.
- Since `nn.Linear` expects each training example to be a vector, each `1x28x28` image tensor is *flattened* into a vector of size 784 `(28*28)` before being passed into the model.
- The output for each image is a vector of size 10, with each element signifying the probability of a particular target label (i.e., 0 to 9). The predicted label for an image is simply the one with the highest probability.

```
import torch.nn as nn

input_size = 28*28
num_classes = 10

# Logistic regression model
# model = MnistModel() # Removed redundant instantiation
```

Of course, this model is a lot larger than our previous model in terms of the number of parameters. Let's take a look at the weights and biases.

```
print(model.weight.shape)
model.weight
```

```
torch.Size([10, 784])
Parameter containing:
tensor([[ 0.0138,  0.0334, -0.0016,  ...,  0.0122, -0.0297, -0.0244],
        [ 0.0170,  0.0228,  0.0121,  ..., -0.0355,  0.0074,  0.0299],
        [-0.0286,  0.0194,  0.0269,  ..., -0.0135, -0.0096, -0.0203],
        ...,
        [-0.0093, -0.0350, -0.0311,  ..., -0.0174, -0.0095, -0.0134],
        [ 0.0100,  0.0279, -0.0191,  ..., -0.0213,  0.0335,  0.0317],
        [ 0.0354, -0.0292,  0.0132,  ...,  0.0271, -0.0142, -0.0262]],
       requires_grad=True)
```

```
print(model.bias.shape)
model.bias
```

```
torch.Size([10])
Parameter containing:
tensor([ 0.0048, -0.0265,  0.0106, -0.0037, -0.0149, -0.0191, -0.0125, -0.0206,
         0.0303, -0.0164], requires_grad=True)
```

Although there are a total of 7850 parameters here, conceptually, nothing has changed so far. Let's try and generate some outputs using our model. We'll take the first batch of 100 images from our dataset and pass them into our model.

```
for images, labels in train_loader:
    outputs = model(images)
    break
```

The code above leads to an error because our input data does not have the right shape. Our images are of the shape 1x28x28, but we need them to be vectors of size 784, i.e., we need to flatten them. We'll use the `.reshape` method of a tensor, which will allow us to efficiently 'view' each image as a flat vector without really creating a copy of the underlying data. To include this additional functionality within our model, we need to define a custom model by extending the `nn.Module` class from PyTorch.

A class in Python provides a "blueprint" for creating objects. Let's look at an example of defining a new class in Python.

```
class Person:
    # Class constructor
    def __init__(self, name, age):
        # Object properties
        self.name = name
        self.age = age

    # Method
    def say_hello(self):
        print("Hello my name is " + self.name + "!")
```

Here's how we create or *instantiate* an object of the class `Person`.

```
bob = Person("Bob", 32)
```

The object `bob` is an instance of the class `Person`.

We can access the object's properties (also called attributes) or invoke its methods using the `.` notation.

```
bob.name, bob.age
```

```
('Bob', 32)
```

```
bob.say_hello()
```

```
Hello my name is Bob!
```

You can learn more about Python classes here: https://www.w3schools.com/python/python_classes.asp .

Classes can also build upon or *extend* the functionality of existing classes. Let's extend the `nn.Module` class from PyTorch to define a custom model.

```
class MnistModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(input_size, num_classes)

    def forward(self, xb):
        xb = xb.reshape(-1, 784)
        out = self.linear(xb)
        return out

model = MnistModel()
```

Inside the `__init__` constructor method, we instantiate the weights and biases using `nn.Linear`. And inside the `forward` method, which is invoked when we pass a batch of inputs to the model, we flatten the input tensor and pass it into `self.linear`.

`xb.reshape(-1, 28*28)` indicates to PyTorch that we want a *view* of the `xb` tensor with two dimensions. The length along the 2nd dimension is 28*28 (i.e., 784). One argument to `.reshape` can be set to `-1` (in this case, the first dimension) to let PyTorch figure it out automatically based on the shape of the original tensor.

Note that the model no longer has `.weight` and `.bias` attributes (as they are now inside the `.linear` attribute), but it does have a `.parameters` method that returns a list containing the weights and bias.

```
print(model.linear.weight.shape, model.linear.bias.shape)
list(model.parameters())
```

```
torch.Size([10, 784]) torch.Size([10])
[Parameter containing:
 tensor([[-0.0035, -0.0092,  0.0262,  ..., -0.0279,  0.0322,  0.0297],
         [ 0.0301, -0.0036,  0.0216,  ..., -0.0152, -0.0031, -0.0134],
         [-0.0264,  0.0206,  0.0149,  ...,  0.0161,  0.0204, -0.0135],
         ...,
         [ 0.0008, -0.0307, -0.0269,  ...,  0.0051, -0.0070,  0.0128],
         [-0.0152, -0.0293, -0.0343,  ...,  0.0326, -0.0107,  0.0179],
         [-0.0300,  0.0143,  0.0320,  ..., -0.0230, -0.0334,  0.0342]],
        requires_grad=True),
 Parameter containing:
 tensor([ 0.0138, -0.0098,  0.0150,  0.0048,  0.0290,  0.0215, -0.0330,
    -0.0320,
             0.0101, -0.0322], requires_grad=True)]
```

We can use our new custom model in the same way as before. Let's see if it works.

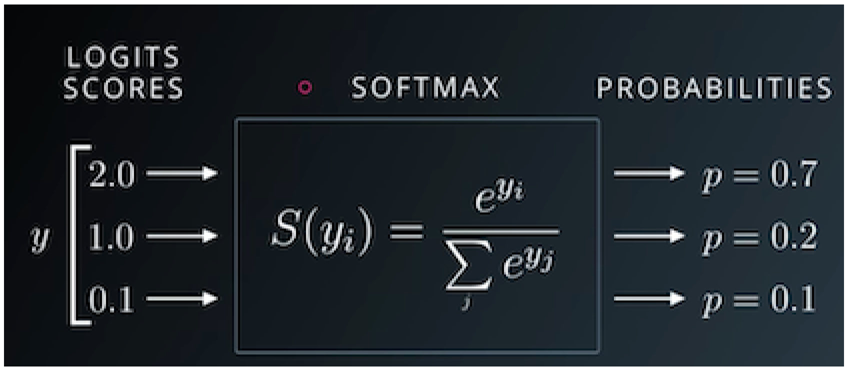
```
for images, labels in train_loader:
    outputs = model(images)
    break

print('outputs.shape : ', outputs.shape)
print('Sample outputs :\n', outputs[:2].data)
```

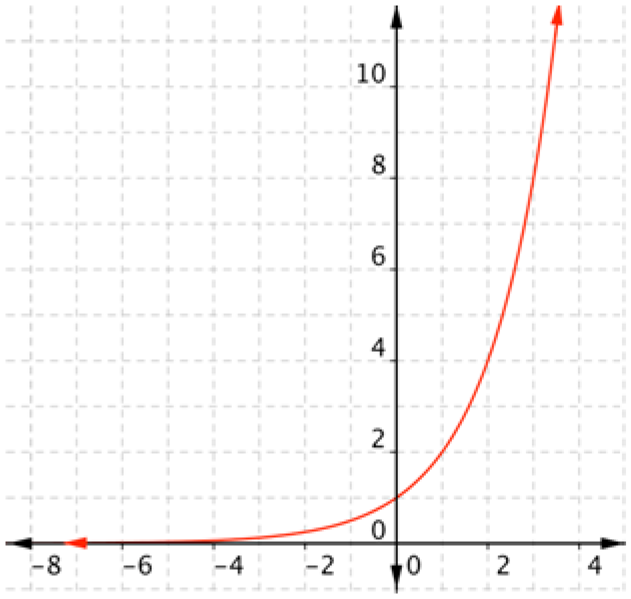
```
outputs.shape :  torch.Size([128, 10])
Sample outputs :
 tensor([[ 0.0928, -0.3581,  0.0472, -0.0446,  0.1926,  0.1925, -0.3282, -0.1472
         -0.0525, -0.1916],
        [ 0.5122,  0.0402, -0.3291,  0.2650, -0.1216,  0.2414, -0.0104, -0.0008,
          0.1551,  0.1108]])
```

For each of the 100 input images, we get 10 outputs, one for each class. As discussed earlier, we'd like these outputs to represent probabilities. Each output row's elements must lie between 0 to 1 and add up to 1, which is not the case.

To convert the output rows into probabilities, we use the softmax function, which has the following formula:



First, we replace each element `yi` in an output row by `e^yi`, making all the elements positive.



Then, we divide them by their sum to ensure that they add up to 1. The resulting vector can thus be interpreted as probabilities.

While it's easy to implement the softmax function (you should try it!), we'll use the implementation that's provided within PyTorch because it works well with multidimensional tensors (a list of output rows in our case).

```
import torch.nn.functional as F
```

The softmax function is included in the `torch.nn.functional` package and requires us to specify a dimension along which the function should be applied.

```
# Apply softmax for each output row
probs = F.softmax(outputs, dim=1)

# Look at sample probabilities
print("Sample probabilities:\n", probs[:2].data)

# Add up the probabilities of an output row
print("Sum: ", torch.sum(probs[0]).item())
```

```
Sample probabilities:
 tensor([[0.1145, 0.0729, 0.1094, 0.0998, 0.1265, 0.1265, 0.0752, 0.0901, 0.0990
          0.0862],
         [0.1495, 0.0932, 0.0644, 0.1167, 0.0793, 0.1140, 0.0886, 0.0895, 0.1046,
          0.1001]])
Sum:  1.0
```

Finally, we can determine the predicted label for each image by simply choosing the index of the element with the highest probability in each output row. We can do this using `torch.max`, which returns each row's largest element and the corresponding index.

```
max_probs, preds = torch.max(probs, dim=1)
print(preds)
print(max_probs)
```

```
tensor([4, 0, 5, 0, 3, 5, 4, 5, 3, 9, 3, 5, 5, 9, 3, 0, 0, 5, 0, 9, 5, 5, 6, 5,
        0, 5, 5, 5, 9, 1, 5, 0, 0, 0, 3, 5, 5, 0, 3, 4, 1, 4, 5, 5, 5, 9, 1, 9,
        1, 9, 1, 3, 5, 5, 6, 9, 1, 5, 1, 3, 5, 5, 4, 5, 5, 3, 5, 5, 1, 1, 3, 0,
        0, 5, 5, 5, 1, 0, 5, 4, 9, 1, 5, 6, 1, 0, 4, 5, 5, 3, 0, 4, 5, 8, 4, 5,
        0, 4, 5, 1, 5, 4, 0, 5, 5, 5, 1, 5, 5, 4, 5, 5, 9, 9, 0, 5, 5, 5, 0, 5,
        9, 5, 5, 4, 5, 9, 4, 5])
tensor([0.1265, 0.1495, 0.1238, 0.1285, 0.1257, 0.1283, 0.1343, 0.1479, 0.1238,
        0.1699, 0.1167, 0.1338, 0.1308, 0.1552, 0.1317, 0.1389, 0.1488, 0.1389,
        0.1256, 0.1335, 0.1354, 0.1392, 0.1262, 0.1416, 0.1314, 0.1298, 0.1215,
        0.1312, 0.1433, 0.1226, 0.1354, 0.1190, 0.1196, 0.1707, 0.1169, 0.1104,
        0.1274, 0.1296, 0.1177, 0.1311, 0.1134, 0.1253, 0.1280, 0.1484, 0.1254,
        0.1297, 0.1402, 0.1255, 0.1357, 0.1207, 0.1154, 0.1229, 0.1363, 0.1526,
        0.1262, 0.1304, 0.1144, 0.1254, 0.1282, 0.1326, 0.1479, 0.1381, 0.1213,
        0.1328, 0.1363, 0.1322, 0.1671, 0.1302, 0.1238, 0.1389, 0.1357, 0.1225,
        0.1304, 0.1430, 0.1682, 0.1160, 0.1385, 0.1242, 0.1133, 0.1374, 0.1250,
        0.1283, 0.1550, 0.1196, 0.1224, 0.1184, 0.1299, 0.1275, 0.1292, 0.1447,
        0.1546, 0.1161, 0.1380, 0.1205, 0.1280, 0.1350, 0.1273, 0.1227, 0.1286,
        0.1447, 0.1326, 0.1235, 0.1224, 0.1364, 0.1233, 0.1342, 0.1280, 0.1250,
        0.1419, 0.1213, 0.1154, 0.1302, 0.1340, 0.1428, 0.1263, 0.1530, 0.1381,
        0.1383, 0.1282, 0.1377, 0.1334, 0.1212, 0.1246, 0.1286, 0.1291, 0.1155,
        0.1222, 0.1219], grad_fn=<MaxBackward0>)
```

The numbers printed above are the predicted labels for the first batch of training images. Let's compare them with the actual labels.

```
labels
```

```
tensor([9, 0, 7, 8, 7, 6, 5, 3, 5, 0, 2, 2, 8, 7, 4, 5, 7, 4, 8, 3, 0, 4, 5, 9,
        4, 3, 9, 8, 3, 9, 7, 1, 3, 0, 5, 5, 6, 0, 3, 8, 1, 7, 2, 4, 8, 3, 6, 5,
        3, 5, 1, 1, 5, 4, 3, 7, 1, 5, 2, 5, 8, 9, 7, 4, 6, 7, 9, 9, 2, 5, 6, 1,
        8, 9, 5, 1, 3, 1, 5, 4, 7, 0, 8, 1, 1, 3, 8, 0, 5, 3, 0, 7, 7, 2, 4, 4,
        3, 7, 5, 8, 3, 5, 0, 4, 9, 5, 6, 8, 6, 9, 6, 6, 0, 0, 1, 4, 9, 5, 9, 4,
        2, 1, 7, 9, 7, 2, 9, 0])
```

Most of the predicted labels are different from the actual labels. That's because we have started with randomly initialized weights and biases. We need to train the model, i.e., adjust the weights using gradient descent to make better predictions.

Evaluation Metric and Loss Function

Just as with linear regression, we need a way to evaluate how well our model is performing. A natural way to do this would be to find the percentage of labels that were predicted correctly, i.e,. the **accuracy** of the predictions.

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

The `==` operator performs an element-wise comparison of two tensors with the same shape and returns a tensor of the same shape, containing `True` for unequal elements and `False` for equal elements. Passing the result to `torch.sum` returns the number of labels that were predicted correctly. Finally, we divide by the total number of images to get the accuracy.

Note that we don't need to apply softmax to the outputs since its results have the same relative order. This is because `e^x` is an increasing function, i.e., if `y1 > y2`, then `e^y1 > e^y2`. The same holds after averaging out the values to get the softmax.

Let's calculate the accuracy of the current model on the first batch of data.

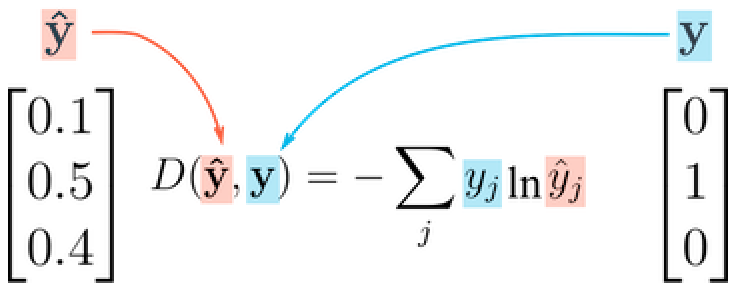
```
accuracy(outputs, labels)
```

```
tensor(0.1719)
```

Accuracy is an excellent way for us (humans) to evaluate the model. However, we can't use it as a loss function for optimizing our model using gradient descent for the following reasons:

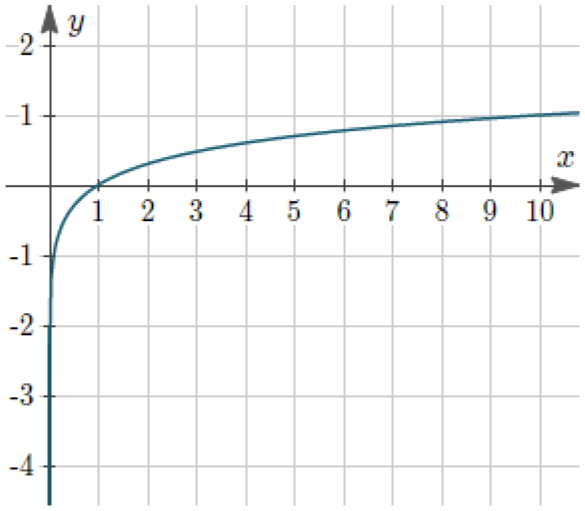
1. It's not a differentiable function. `torch.max` and `==` are both non-continuous and non-differentiable operations, so we can't use the accuracy for computing gradients w.r.t the weights and biases.
2. It doesn't take into account the actual probabilities predicted by the model, so it can't provide sufficient feedback for incremental improvements.

For these reasons, accuracy is often used as an **evaluation metric** for classification, but not as a loss function. A commonly used loss function for classification problems is the **cross-entropy**, which has the following formula:


$\hat{\mathbf{y}}$
$\mathbf{y}$
$\begin{bmatrix}0.1\\0.5\\0.4\end{bmatrix}$
$D(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_j y_j \ln \hat{y}_j$
$\begin{bmatrix}0\\1\\0\end{bmatrix}$

While it looks complicated, it's actually quite simple:

- For each output row, pick the predicted probability for the correct label. E.g., if the predicted probabilities for an image are `[0.1, 0.3, 0.2, ...]` and the correct label is `1`, we pick the corresponding element `0.3` and ignore the rest.
- Then, take the logarithm of the picked probability. If the probability is high, i.e., close to 1, then its logarithm is a very small negative value, close to 0. And if the probability is low (close to 0), then the logarithm is a very large negative value. We also multiply the result by -1, which results is a large postive value of the loss for poor predictions.



- Finally, take the average of the cross entropy across all the output rows to get the overall loss for a batch of data.

Unlike accuracy, cross-entropy is a continuous and differentiable function. It also provides useful feedback for incremental improvements in the model (a slightly higher probability for the correct label leads to a lower loss). These two factors make cross-entropy a better choice for the loss function.

As you might expect, PyTorch provides an efficient and tensor-friendly implementation of cross-entropy as part of the `torch.nn.functional` package. Moreover, it also performs softmax internally, so we can directly pass in the model's outputs without converting them into probabilities.

```
loss_fn = F.cross_entropy
```

```
# Loss for current batch of data
loss = loss_fn(outputs, labels)
print(loss)
```

```
tensor(2.3168, grad_fn=<NllLossBackward0>)
```

We know that cross-entropy is the negative logarithm of the predicted probability of the correct label averaged over all training samples. Therefore, one way to interpret the resulting number e.g. `2.23` is look at `e^-2.23` which is around `0.1` as the predicted probability of the correct label, on average. *The lower the loss, The better the model.*

Training the model

Now that we have defined the data loaders, model, loss function and optimizer, we are ready to train the model. The training process is identical to linear regression, with the addition of a "validation phase" to evaluate the model in each epoch. Here's what it looks like in pseudocode:

```
for epoch in range(num_epochs):
    # Training phase
    for batch in train_loader:
        # Generate predictions
        # Calculate loss
        # Compute gradients
        # Update weights
        # Reset gradients

    # Validation phase
    for batch in val_loader:
        # Generate predictions
        # Calculate loss
        # Calculate metrics (accuracy etc.)
    # Calculate average validation loss & metrics

    # Log epoch, loss & metrics for inspection
```

Some parts of the training loop are specific the specific problem we're solving (e.g. loss function, metrics etc.) whereas others are generic and can be applied to any deep learning problem.

We'll include the problem-independent parts within a function called `fit`, which will be used to train the model. The problem-specific parts will be implemented by adding new methods to the `nn.Module` class.

```
def fit(epochs, lr, model, train_loader, val_loader, opt_func=torch.optim.SGD):
    optimizer = opt_func(model.parameters(), lr)
    history = [] # for recording epoch-wise results

    for epoch in range(epochs):
        # Training Phase
        for batch in train_loader:
            loss = model.training_step(batch)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()

        # Validation phase
        result = evaluate(model, val_loader)
        model.epoch_end(epoch, result)
        history.append(result)

    return history
```

The `fit` function records the validation loss and metric from each epoch. It returns a history of the training, useful for debugging & visualization.

Configurations like batch size, learning rate, etc. (called hyperparameters), need to picked in advance while training machine learning models. Choosing the right hyperparameters is critical for training a reasonably accurate model within a reasonable amount of time. It is an active area of research and experimentation in machine learning. Feel free to try different learning rates and see how it affects the training process.

Let's define the `evaluate` function, used in the validation phase of `fit`.

```
def evaluate(model, val_loader):
    outputs = [model.validation_step(batch) for batch in val_loader]
    return model.validation_epoch_end(outputs)
```

Finally, let's redefine the `MnistModel` class to include additional methods `training_step`, `validation_step`, `validation_epoch_end`, and `epoch_end` used by `fit` and `evaluate`.

```
class MnistModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(input_size, num_classes)

    def forward(self, xb):
        xb = xb.reshape(-1, 784)
        out = self.linear(xb)
        return out

    def training_step(self, batch):
        images, labels = batch
        out = self(images)                  # Generate predictions
        loss = F.cross_entropy(out, labels) # Calculate loss
        return loss

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

    def validation_epoch_end(self, outputs):
        batch_losses = [x['val_loss'] for x in outputs]
        epoch_loss = torch.stack(batch_losses).mean()   # Combine losses
        batch_accs = [x['val_acc'] for x in outputs]
        epoch_acc = torch.stack(batch_accs).mean()      # Combine accuracies
        return {'val_loss': epoch_loss.item(), 'val_acc': epoch_acc.item()}

    def epoch_end(self, epoch, result):
        print("Epoch [{}], val_loss: {:.4f}, val_acc: {:.4f}".format(epoch, result['

model = MnistModel()
```

Before we train the model, let's see how the model performs on the validation set with the initial set of randomly initialized weights & biases.

```
result0 = evaluate(model, val_loader)
result0
```

```
{'val_loss': 2.2947819232940674, 'val_acc': 0.14665743708610535}
```

The initial accuracy is around 10%, which one might expect from a randomly initialized model (since it has a 1 in 10 chance of getting a label right by guessing randomly).

We are now ready to train the model. Let's train for five epochs and look at the results.

```
history1 = fit(5, 0.001, model, train_loader, val_loader)
```

```
Epoch [0], val_loss: 1.9309, val_acc: 0.6394
Epoch [1], val_loss: 1.6677, val_acc: 0.7353
Epoch [2], val_loss: 1.4711, val_acc: 0.7652
Epoch [3], val_loss: 1.3223, val_acc: 0.7868
Epoch [4], val_loss: 1.2077, val_acc: 0.7991
```

That's a great result! With just 5 epochs of training, our model has reached an accuracy of over 80% on the validation set. Let's see if we can improve that by training for a few more epochs. Try changing the learning rates and number of epochs in each of the cells below.

```
history2 = fit(5, 0.001, model, train_loader, val_loader)
```

```
Epoch [0], val_loss: 1.1173, val_acc: 0.8080
Epoch [1], val_loss: 1.0447, val_acc: 0.8147
Epoch [2], val_loss: 0.9851, val_acc: 0.8203
Epoch [3], val_loss: 0.9355, val_acc: 0.8258
Epoch [4], val_loss: 0.8935, val_acc: 0.8303
```

```
history3 = fit(5, 0.001, model, train_loader, val_loader)
```

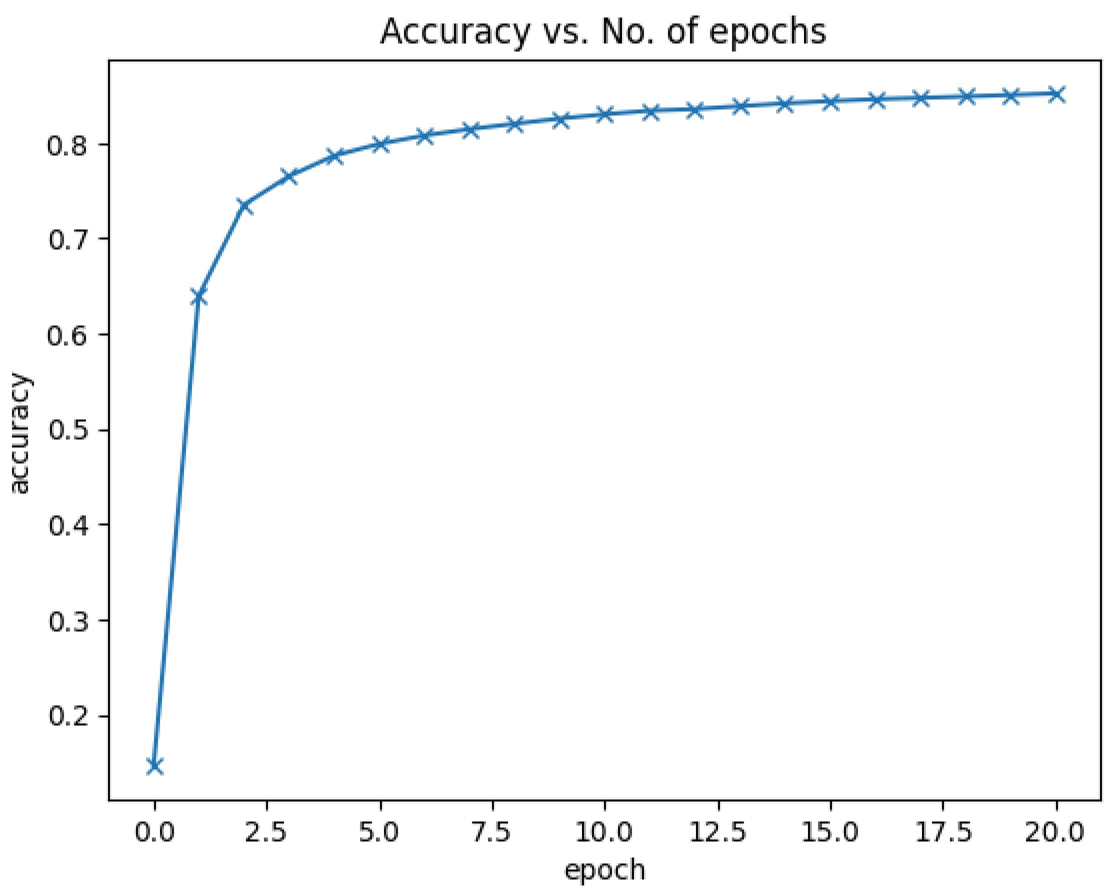
```
Epoch [0], val_loss: 0.8575, val_acc: 0.8341
Epoch [1], val_loss: 0.8264, val_acc: 0.8357
Epoch [2], val_loss: 0.7991, val_acc: 0.8388
Epoch [3], val_loss: 0.7749, val_acc: 0.8418
Epoch [4], val_loss: 0.7534, val_acc: 0.8441
```

```
history4 = fit(5, 0.001, model, train_loader, val_loader)
```

```
Epoch [0], val_loss: 0.7341, val_acc: 0.8461
Epoch [1], val_loss: 0.7167, val_acc: 0.8475
Epoch [2], val_loss: 0.7009, val_acc: 0.8490
Epoch [3], val_loss: 0.6865, val_acc: 0.8506
Epoch [4], val_loss: 0.6733, val_acc: 0.8523
```

While the accuracy does continue to increase as we train for more epochs, the improvements get smaller with every epoch. Let's visualize this using a line graph.

```
history = [result0] + history1 + history2 + history3 + history4
accuracies = [result['val_acc'] for result in history]
plt.plot(accuracies, '-x')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.title('Accuracy vs. No. of epochs');
```



It's quite clear from the above picture that the model probably won't cross the accuracy threshold of 90% even after training for a very long time. One possible reason for this is that the learning rate might be too high. The model's parameters may be "bouncing" around the optimal set of parameters for the lowest loss. You can try reducing the learning rate and training for a few more epochs to see if it helps.

The more likely reason that **the model just isn't powerful enough**. If you remember our initial hypothesis, we have assumed that the output (in this case the class probabilities) is a **linear function** of the input (pixel intensities), obtained by perfoming a matrix multiplication with the weights matrix and adding the bias. This is a fairly weak assumption, as there may not actually exist a linear relationship between the pixel intensities in an image and the digit it represents. While it works reasonably well for a simple dataset like MNIST (getting us to 85% accuracy), we need more sophisticated models that can capture non-linear relationships between image pixels and labels for complex tasks like recognizing everyday objects, animals etc.

Let's save our work using `jovian.commit`. Along with the notebook, we can also record some metrics from our training.

```
jovian.log_metrics(val_acc=history[-1]['val_acc'], val_loss=history[-1]['val_loss'])
```

```
[jovian] Please enter your API key ( from https://jovian.com/ ):
API KEY:
---------------------------------------------------------------------------
Abort                                     Traceback (most recent call last)
/tmp/ipython-input-52-2086543719.py in <cell line: 0>()
----> 1 jovian.log_metrics(val_acc=history[-1]['val_acc'], val_loss=history[-1]
['val_loss'])

                          9 frames
/usr/local/lib/python3.11/dist-packages/click/termui.py in prompt_func(text)
    149             if hide_input:
    150                 echo(None, err=err)
--> 151             raise Abort() from None
    152 
    153         if value_proc is None:

Abort:
```

```
jovian.commit(project='03-logistic-regression', environment=None)
```

Testing with individual images

While we have been tracking the overall accuracy of a model so far, it's also a good idea to look at model's results on some sample images. Let's test out our model with some images from the predefined test dataset of 10000 images. We begin by recreating the test dataset with the `ToTensor` transform.

```
# Define test dataset
test_dataset = MNIST(root='data/',
                     train=False,
                     transform=transforms.ToTensor())
```

Here's a sample image from the dataset.

```
img, label = test_dataset[0]
plt.imshow(img[0], cmap='gray')
print('Shape:', img.shape)
print('Label:', label)
```

Let's define a helper function `predict_image`, which returns the predicted label for a single image tensor.

```
def predict_image(img, model):
    xb = img.unsqueeze(0)
    yb = model(xb)
    _, preds  = torch.max(yb, dim=1)
    return preds[0].item()
```

`img.unsqueeze` simply adds another dimension at the begining of the 1x28x28 tensor, making it a 1x1x28x28 tensor, which the model views as a batch containing a single image.

Let's try it out with a few images.

```
img, label = test_dataset[0]
plt.imshow(img[0], cmap='gray')
print('Label:', label, ', Predicted:', predict_image(img, model))
```

```
img, label = test_dataset[10]
plt.imshow(img[0], cmap='gray')
print('Label:', label, ', Predicted:', predict_image(img, model))
```

```
img, label = test_dataset[193]
plt.imshow(img[0], cmap='gray')
print('Label:', label, ', Predicted:', predict_image(img, model))
```

```
img, label = test_dataset[1839]
plt.imshow(img[0], cmap='gray')
print('Label:', label, ', Predicted:', predict_image(img, model))
```

Identifying where our model performs poorly can help us improve the model, by collecting more training data, increasing/decreasing the complexity of the model, and changing the hypeparameters.

As a final step, let's also look at the overall loss and accuracy of the model on the test set.

```
test_loader = DataLoader(test_dataset, batch_size=256)
result = evaluate(model, test_loader)
result
```

We expect this to be similar to the accuracy/loss on the validation set. If not, we might need a better validation set that has similar data and distribution as the test set (which often comes from real world data).

Saving and loading the model

Since we've trained our model for a long time and achieved a resonable accuracy, it would be a good idea to save the weights and bias matrices to disk, so that we can reuse the model later and avoid retraining from scratch. Here's how you can save the model.

```
torch.save(model.state_dict(), 'mnist-logistic.pth')
```

The `.state_dict` method returns an `OrderedDict` containing all the weights and bias matrices mapped to the right attributes of the model.

```
model.state_dict()
```

To load the model weights, we can instante a new object of the class `MnistModel`, and use the `.load_state_dict` method.

```
model2 = MnistModel()
model2.load_state_dict(torch.load('mnist-logistic.pth'))
model2.state_dict()
```

Just as a sanity check, let's verify that this model has the same loss and accuracy on the test set as before.

```
test_loader = DataLoader(test_dataset, batch_size=256)
result = evaluate(model2, test_loader)
result
```

As a final step, we can save and commit our work using the `jovian` library. Along with the notebook, we can also attach the weights of our trained model, so that we can use it later.

```
jovian.commit(project='03-logistic-regression', environment=None, outputs=['mnist-lo
```