

50.039 Theory and Practice of Deep Learning

W3-S3 Introduction to Deep Learning using the PyTorch framework

Matthieu De Mari



Introduction (Week 3)

1. What is the **PyTorch library** and its **benefits**?
2. What is a **PyTorch tensor object** and its typical **attributes**?
3. How to implement some typical **tensor operations**?
4. What is **broadcasting** on tensors?
5. What are **tensor locations** in terms of computation?
6. How to **transform our original NumPy shallow Neural Network** class so it uses **PyTorch** now instead?
7. How to implement a **forward**, **loss** and **accuracy** metric in PyTorch?
8. What are some measurable **performance benefits** of using **PyTorch** over NumPy and **GPUs** over CPUs?

Introduction (Week 3)

9. What is the **autograd/backprop** module in PyTorch, and how does it use a **computational graph** to **compute all derivatives**?
10. How to use the **autograd** to implement **derivatives** and a **vanilla gradient descent**?
11. How to implement **backprop** in PyTorch for our **shallow Neural Network** class?
12. How to use **PyTorch** to implement **advanced optimizers**?
13. How to use **PyTorch** to implement **advanced initializers**?
14. How to use **PyTorch** to implement **regularization**?
15. How to finally revise our **trainer** function to obtain a minimal, yet complete Neural Network in PyTorch?

Introduction (Week 3)

16. What are the **Dataset** and **Dataloader** objects in **PyTorch**?
17. How to implement a custom **Dataloader** and **Dataset** object in PyTorch?
18. How to move from binary classification to **multi-class classification**?
19. How to adjust output probabilities using the **softmax** function?
20. How to change the **cross-entropy loss** so it works in **multi-class classification**?
21. How to implement **building blocks** in PyTorch?
22. How to implement and train our first **Deep Neural Network**?
23. What are **additional good practices** in PyTorch?

To summarize (last two sessions)

We now have a full Neural Network class, in PyTorch, with:

- 2 linear layers, sigmoid activation functions,
- Xavier uniform initialization on trainable parameters,
- Forward pass method,
- Autograd backpropagation and trainer method,
- Adam optimizer,
- Dataloader allowing for stochastic mini-batches,
- Cross entropy loss and accuracies,
- L1 regularization.

And it runs/trains at the speed of light (almost...) on GPU!

```

1 class ShallowNeuralNet_PT(torch.nn.Module):
2     def __init__(self, n_x, n_h, n_y, device):
3         super().__init__()
4         self.n_x, self.n_h, self.n_y = n_x, n_h, n_y
5         self.W1 = torch.nn.Parameter(torch.zeros(size = (n_x, n_h), requires_grad = True, \
6                                                     dtype = torch.float64, device = device))
7         torch.nn.init.xavier_uniform_(self.W1.data)
8         self.b1 = torch.nn.Parameter(torch.zeros(size = (1, n_h), requires_grad = True, \
9                                                     dtype = torch.float64, device = device))
10        torch.nn.init.xavier_uniform_(self.b1.data)
11        self.W2 = torch.nn.Parameter(torch.zeros(size = (n_h, n_y), requires_grad = True, \
12                                                    dtype = torch.float64, device = device))
13        torch.nn.init.xavier_uniform_(self.W2.data)
14        self.b2 = torch.nn.Parameter(torch.zeros(size = (1, n_y), requires_grad = True, \
15                                                    dtype = torch.float64, device = device))
16        torch.nn.init.xavier_uniform_(self.b2.data)
17        self.loss = torch.nn.BCELoss()
18        self.accuracy = BinaryAccuracy()
19    def forward(self, inputs):
20        return torch.sigmoid(torch.matmul(torch.sigmoid(torch.matmul(inputs, self.W1) + self.b1), self.W2) + self.b2)
21    def train(self, inputs, outputs, N_max = 1000, alpha = 1, beta1 = 0.9, beta2 = 0.999, \
22             batch_size = 32, lambda_val = 1e-3):
23        dataset = torch.utils.data.TensorDataset(inputs, outputs)
24        data_loader = torch.utils.data.DataLoader(dataset, batch_size = batch_size, shuffle = True)
25        optimizer = torch.optim.Adam(self.parameters(), lr = alpha, betas = (beta1, beta2), eps = 1e-08)
26        optimizer.zero_grad()
27        self.loss_history = []
28        for iteration_number in range(1, N_max + 1):
29            for batch in data_loader:
30                inputs_batch, outputs_batch = batch
31                total_loss = self.loss(self(inputs_batch), outputs_batch.to(torch.float64))\
32                    + lambda_val*sum(torch.abs(param).sum() for param in self.parameters()).item()
33                self.loss_history.append(total_loss)
34                total_loss.backward()
35                optimizer.step()
36                optimizer.zero_grad()
37            if(iteration_number % (N_max//20) == 1):
38                pred = self(inputs)
39                acc_val = self.accuracy(pred, outputs).item()
40                print("Iteration {} - Loss = {} - Accuracy = {}".format(iteration_number, total_loss, acc_val))

```

Built-in datasets

- Pytorch has a few **built-in datasets**, ready to be downloaded and used on models: typically the most common ones that have been used to demonstrate concepts, such as **MNIST** or **CIFAR-10**.
- For more details on the available **built-in datasets**, have a look at the following page:
<https://pytorch.org/vision/stable/datasets.html>.

Built-in datasets

- Let us demonstrate using the **FashionMNIST** dataset.
- This dataset consists of 28 by 28 greyscale images.
- It is typically used to design image classification models (i.e. models that receive images as inputs), and attempt to predict what is in the image in question.
- The 10 classes (bag, shirt, etc.) are indexed with 0-9 values, corresponding to the 10 types of fashion objects found in the dataset.

```
1 # Dictionary of labels and their identification
2 labels_map = {0: "T-Shirt",
3               1: "Trouser",
4               2: "Pullover",
5               3: "Dress",
6               4: "Coat",
7               5: "Sandal",
8               6: "Shirt",
9               7: "Sneaker",
10              8: "Bag",
11              9: "Ankle Boot"}
```

```
1 # Define folder path as string
2 # Dataset will be downloaded and stored there.
3 folder_path = "./data"
4
5 # Download (download = True) training data (train = True)
6 # to folder specified in root parameter.
7 # The transform parameter specifies that the image samples will
8 # be converted to Tensors, ready to be used by models.
9 training_data = datasets.FashionMNIST(root = folder_path, \
10                                     train = True, \
11                                     download = True, \
12                                     transform = ToTensor())
```


Built-in datasets

- Let us demonstrate this with the **FashionMNIST** dataset.
- This dataset consists of 28 by 28 greyscale images.
- It is typically used to design image classification models (i.e. models that receive images as inputs), and attempt to predict what is in the image in question.
- The 10 classes (bag, shirt, etc.) are indexed with 0-9 values, corresponding to the 10 types of fashion objects found in the dataset.

```
1 # Dataset contains 60000 samples, that are greyscale
2 # images with size 28 by 28 pixels.
3 print(training_data.data.shape)
```

```
torch.Size([60000, 28, 28])
```

```
1 # We can then fetch a sample using the [] notations
2 sample_index = 894
3 img, label = training_data[sample_index]
4 print("Image: ", img.shape)
5 print("Label: ", label)
```

```
Image: torch.Size([1, 28, 28])
```

```
Label: 8
```

```
1 # Display sample
2 sample_index = 894
3 img, label = training_data[sample_index]
4 plt.figure(figsize = (3, 3))
5 plt.axis("off")
6 plt.imshow(img.squeeze(), cmap="gray")
7 plt.title("Image {} - Label = {} ({})" .format(sample_index, \
8                                               label, \
9                                               labels_map[label]))
10 plt.show()
```

Image 894 - Label = 8 (Bag)



Writing a custom Dataset object

- Most of the time, when demonstrating concepts, we will rely on a "simple" **built-in dataset**, available in the PyTorch library.
- In practice, however, you will often play with a **custom dataset**, fitting your project needs.
- Most datasets will be provided by your future employers, or can be found on dataset search engines, such as **Kaggle**, **Google Dataset Search**, etc.
- Today, we will play with a simplified version of the **Ames Housing Dataset**, which can be found online, here:
<https://www.kaggle.com/datasets/prevek18/ames-housing-dataset?resource=download>

AutoSave

AmesHousing.xlsx

Last Modified: 9 January

File

Home

Insert

Page Layout

Formulas

Data

Review

View

Automate

Help

Acrobat

Table Design

Clipboard

Cut

Copy

Format Painter

Font

Calibri

11

B

I

U

Wrap Text

Merge & Center

Alignment

General

\$

%

←0

→0

Conditional Formatting

Format as Table

Styles

Normal

Bad

Good

Neutral

Calculation

Check Cell

Cells

Insert

Delete

Format

Editing

AutoSum

Fill

Clear

Sort & Filter

Find & Select

Analysis

Analyze Data

Sensitivity

Comments

Share

G10

1616

| | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q |
|----|----------|--------------|--------------|------------|----------------|---------------|------------|------------|-------------|-----------|-----------|---------------|---------------|--------------|-------------|---------|-----------|
| 1 | Lot Area | Overall Qual | Overall Cond | Year Built | Year Remod/Add | Total Bsmt SF | 1st Flr SF | 2nd Flr SF | Gr Liv Area | Full Bath | Half Bath | Bedroom AbvGr | Kitchen AbvGr | TotRms AbvGr | Garage Area | Yr Sold | SalePrice |
| 2 | 31770 | 6 | 5 | 1960 | 1960 | 1080 | 1656 | 0 | 1656 | 1 | 0 | 3 | 1 | 7 | 528 | 2010 | 215 |
| 3 | 11622 | 5 | 6 | 1961 | 1961 | 882 | 896 | 0 | 896 | 1 | 0 | 2 | 1 | 5 | 730 | 2010 | 105 |
| 4 | 14267 | 6 | 6 | 1958 | 1958 | 1329 | 1329 | 0 | 1329 | 1 | 1 | 3 | 1 | 6 | 312 | 2010 | 172 |
| 5 | 11160 | 7 | 5 | 1968 | 1968 | 2110 | 2110 | 0 | 2110 | 2 | 1 | 3 | 1 | 8 | 522 | 2010 | 244 |
| 6 | 13830 | 5 | 5 | 1997 | 1998 | 928 | 928 | 701 | 1629 | 2 | 1 | 3 | 1 | 6 | 482 | 2010 | 189 |
| 7 | 9978 | 6 | 6 | 1998 | 1998 | 926 | 926 | 678 | 1604 | 2 | 1 | 3 | 1 | 7 | 470 | 2010 | 195 |
| 8 | 4920 | 8 | 5 | 2001 | 2001 | 1338 | 1338 | 0 | 1338 | 2 | 0 | 2 | 1 | 6 | 582 | 2010 | 213 |
| 9 | 5005 | 8 | 5 | 1992 | 1992 | 1280 | 1280 | 0 | 1280 | 2 | 0 | 2 | 1 | 5 | 506 | 2010 | 193 |
| 10 | 5389 | 8 | 5 | 1995 | 1996 | 1595 | 1616 | 0 | 1616 | 2 | 0 | 2 | 1 | 5 | 608 | 2010 | 230 |
| 11 | 7500 | 7 | 5 | 1999 | 1999 | 994 | 1028 | 776 | 1804 | 2 | 1 | 3 | 1 | 7 | 442 | 2010 | 189 |
| 12 | 10000 | 6 | 5 | 1993 | 1994 | 763 | 763 | 892 | 1655 | 2 | 1 | 3 | 1 | 7 | 440 | 2010 | 175 |
| 13 | 7980 | 6 | 7 | 1992 | 2007 | 1168 | 1187 | 0 | 1187 | 2 | 0 | 3 | 1 | 6 | 420 | 2010 | 185 |
| 14 | 8402 | 6 | 5 | 1998 | 1998 | 789 | 789 | 676 | 1465 | 2 | 1 | 3 | 1 | 7 | 393 | 2010 | 180 |
| 15 | 10176 | 7 | 5 | 1990 | 1990 | 1300 | 1341 | 0 | 1341 | 1 | 1 | 2 | 1 | 5 | 506 | 2010 | 172 |
| 16 | 6820 | 8 | 5 | 1985 | 1985 | 1488 | 1502 | 0 | 1502 | 1 | 1 | 1 | 1 | 4 | 528 | 2010 | 212 |
| 17 | 53504 | 8 | 5 | 2003 | 2003 | 1650 | 1690 | 1589 | 3279 | 3 | 1 | 4 | 1 | 12 | 841 | 2010 | 538 |
| 18 | 12134 | 8 | 7 | 1988 | 2005 | 559 | 1080 | 672 | 1752 | 2 | 0 | 4 | 1 | 8 | 492 | 2010 | 164 |
| 19 | 11394 | 9 | 2 | 2010 | 2010 | 1856 | 1856 | 0 | 1856 | 1 | 1 | 1 | 1 | 8 | 834 | 2010 | 394 |
| 20 | 19138 | 4 | 5 | 1951 | 1951 | 864 | 864 | 0 | 864 | 1 | 0 | 2 | 1 | 4 | 400 | 2010 | 143 |
| 21 | 13175 | 6 | 6 | 1978 | 1988 | 1542 | 2073 | 0 | 2073 | 2 | 0 | 3 | 1 | 7 | 500 | 2010 | 210 |
| 22 | 11751 | 6 | 6 | 1977 | 1977 | 1844 | 1844 | 0 | 1844 | 2 | 0 | 3 | 1 | 7 | 546 | 2010 | 190 |
| 23 | 10625 | 7 | 6 | 1974 | 1974 | 1053 | 1173 | 0 | 1173 | 2 | 0 | 3 | 1 | 6 | 528 | 2010 | 170 |
| 24 | 7500 | 7 | 5 | 2000 | 2000 | 814 | 814 | 860 | 1674 | 2 | 1 | 3 | 1 | 7 | 663 | 2010 | 210 |
| 25 | 11241 | 6 | 7 | 1970 | 1970 | 1004 | 1004 | 0 | 1004 | 1 | 0 | 2 | 1 | 5 | 480 | 2010 | 149 |
| 26 | 12537 | 5 | 6 | 1971 | 2008 | 1078 | 1078 | 0 | 1078 | 1 | 1 | 3 | 1 | 6 | 500 | 2010 | 149 |
| 27 | 8450 | 5 | 6 | 1968 | 1968 | 1056 | 1056 | 0 | 1056 | 1 | 0 | 3 | 1 | 6 | 304 | 2010 | 142 |
| 28 | 8400 | 4 | 5 | 1970 | 1970 | 882 | 882 | 0 | 882 | 1 | 0 | 2 | 1 | 4 | 525 | 2010 | 120 |
| 29 | 10500 | 4 | 5 | 1971 | 1971 | 864 | 864 | 0 | 864 | 1 | 0 | 3 | 1 | 5 | 0 | 2010 | 115 |
| 30 | 5858 | 7 | 5 | 1999 | 1999 | 1405 | 1337 | 0 | 1337 | 2 | 0 | 2 | 1 | 5 | 511 | 2010 | 184 |
| 31 | 1680 | 6 | 5 | 1971 | 1971 | 483 | 483 | 504 | 987 | 1 | 1 | 2 | 1 | 5 | 264 | 2010 | 90 |
| 32 | 1680 | 5 | 5 | 1971 | 1971 | 525 | 525 | 567 | 1092 | 1 | 1 | 3 | 1 | 6 | 320 | 2010 | 105 |
| 33 | 1680 | 6 | 5 | 1971 | 1971 | 525 | 525 | 567 | 1092 | 1 | 1 | 3 | 1 | 6 | 264 | 2010 | 88 |
| 34 | 4043 | 6 | 5 | 1977 | 1977 | 1069 | 1069 | 0 | 1069 | 2 | 0 | 2 | 1 | 4 | 440 | 2010 | 122 |
| 35 | 2280 | 6 | 6 | 1975 | 1975 | 855 | 855 | 601 | 1456 | 2 | 1 | 3 | 1 | 6 | 440 | 2010 | 149 |
| 36 | 2280 | 7 | 6 | 1975 | 1975 | 836 | 836 | 0 | 836 | 1 | 0 | 2 | 1 | 4 | 308 | 2010 | 120 |
| 37 | 2280 | 6 | 5 | 1978 | 1978 | 855 | 855 | 601 | 1456 | 2 | 1 | 3 | 1 | 7 | 440 | 2010 | 140 |
| 38 | 12858 | 9 | 5 | 2009 | 2010 | 1590 | 1627 | 707 | 2334 | 2 | 1 | 3 | 1 | 10 | 751 | 2010 | 370 |
| 39 | 11478 | 8 | 5 | 2007 | 2008 | 1704 | 1704 | 0 | 1704 | 2 | 0 | 3 | 1 | 7 | 772 | 2010 | 300 |
| 40 | 10159 | 9 | 5 | 2009 | 2010 | 1930 | 1940 | 0 | 1940 | 2 | 1 | 3 | 1 | 8 | 606 | 2010 | 395 |
| 41 | 12883 | 8 | 5 | 2009 | 2010 | 1544 | 1544 | 0 | 1544 | 2 | 0 | 3 | 1 | 7 | 868 | 2010 | 290 |
| 42 | 12182 | 7 | 5 | 2005 | 2005 | 1541 | 1541 | 0 | 1541 | 2 | 0 | 3 | 1 | 7 | 532 | 2010 | 220 |
| 43 | 11520 | 9 | 5 | 2005 | 2005 | 1698 | 1698 | 0 | 1698 | 2 | 0 | 3 | 1 | 7 | 730 | 2010 | 275 |
| 44 | 14122 | 8 | 5 | 2005 | 2006 | 1822 | 1822 | 0 | 1822 | 2 | 0 | 3 | 1 | 8 | 678 | 2010 | 259 |

AmesHousing

Writing a custom Dataset object

- The Ames dataset includes a variety of features for approximately 2,800 houses in Ames, Iowa.
- Features include the size of the house (in square feet), the number of bedrooms and bathrooms, the year the house was built, and more. It also includes the sale price for each house.
- The Ames Housing Dataset is a popular choice for machine learning projects, and it has been used to build models for predicting house prices based on various features.
- It consists of an Excel file (AmesHousing.xlsx) stored in the ./ames/ folder. The original dataset can be found in the AmesHousing.csv file, but we have simplified it by removing some of its features.

Writing a custom Dataset object

The features we are interested in are:

- **Lot Area:** The area of the lot in square feet.
- **Overall Qual:** A rating of the overall material and finish of the house (1-10).
- **Overall Cond:** A rating of the overall condition of the house (1-10).
- **Year Built:** The year the house was built.
- **Year Remod/Add:** The year the house was remodeled or had an addition added.
- **Total Bsmt SF:** The total surface of the basement, in square feet.
- **1st Flr SF:** The first floor surface, in square feet.
- **2nd Flr SF:** The second floor surface, in square feet.
- **Gr Liv Area:** The above grade (ground) living area, in square feet.
- **Full Bath:** The number of full bathrooms.
- **Half Bath:** The number of half bathrooms.
- **Bedroom AbvGr:** The number of bedrooms.
- **Kitchen AbvGr:** The number of kitchens.
- **TotRms AbvGrd:** The total number of rooms (does not include bathrooms).
- **Garage Area:** The size of the garage, in square feet.
- **Yr Sold:** The year the property was sold.

These **16 features** will be used as **inputs**, and the **output** will consist of just **1 feature**, in the final column of the Excel file, which is:

- **SalePrice:** The sale price, in dollars.

Writing a custom Dataset object

Let us start by loading the **Excel** file into a pandas **DataFrame** first.

Note: Not familiar with the pandas library? Find 10 minutes to pick it up, it will definitely serve you in the long run!

https://pandas.pydata.org/docs/user_guide/10min.html

```
1 # Load dataset using pandas, and showing the first five entries
2 ames_dataset = pd.read_excel("./ames/AmesHousing.xlsx")
3 print(ames_dataset.head(5))
```

| | Lot Area | Overall Qual | Overall Cond | Year Built | Year Remod/Add | \ |
|---|----------|--------------|--------------|------------|----------------|---|
| 0 | 31770 | 6 | 5 | 1960 | 1960 | |
| 1 | 11622 | 5 | 6 | 1961 | 1961 | |
| 2 | 14267 | 6 | 6 | 1958 | 1958 | |
| 3 | 11160 | 7 | 5 | 1968 | 1968 | |
| 4 | 13830 | 5 | 5 | 1997 | 1998 | |

| | Total Bsmt SF | 1st Flr SF | 2nd Flr SF | Gr Liv Area | Full Bath | Half Bath | \ |
|---|---------------|------------|------------|-------------|-----------|-----------|---|
| 0 | 1080 | 1656 | 0 | 1656 | 1 | 0 | |
| 1 | 882 | 896 | 0 | 896 | 1 | 0 | |
| 2 | 1329 | 1329 | 0 | 1329 | 1 | 1 | |
| 3 | 2110 | 2110 | 0 | 2110 | 2 | 1 | |
| 4 | 928 | 928 | 701 | 1629 | 2 | 1 | |

| | Bedroom AbvGr | Kitchen AbvGr | TotRms AbvGrd | Garage Area | Yr Sold | \ |
|---|---------------|---------------|---------------|-------------|---------|---|
| 0 | 3 | 1 | 7 | 528 | 2010 | |
| 1 | 2 | 1 | 5 | 730 | 2010 | |
| 2 | 3 | 1 | 6 | 312 | 2010 | |
| 3 | 3 | 1 | 8 | 522 | 2010 | |
| 4 | 3 | 1 | 6 | 482 | 2010 | |

| | SalePrice |
|---|-----------|
| 0 | 215000 |
| 1 | 105000 |
| 2 | 172000 |
| 3 | 244000 |
| 4 | 189900 |

Writing a custom Dataset object

To write a custom **dataset class** in PyTorch, we need to:

- Create a **class** that **subclasses** **torch.utils.data.Dataset**.
- Define a **__init__** that takes in the required arguments, and stores them as member variables.
- Define a **__getitem__** that takes **index** as input, and **returns** the **data** and **label** at that index as an array. This will allow to use the square bracket notation on our dataset object.
- Define a method **__len__** that returns the **number of samples in the dataset**.

```
class AmesHousingDataset(torch.utils.data.Dataset):

    #The init method will simply initialize attributes, which consist
    # of the details related to the dataset.
    def __init__(self, file_path = "./ames/AmesHousing.xlsx"):
        # Whole data as a pandas array
        self.data = pd.read_excel(file_path)
        self.dataset_length = len(self.data) #2928

        # Extract inputs
        self.input_features_number = 16
        self.input_features = self.data.iloc[:, :16]

        # Extract outputs
        self.output_features_number = 1
        self.output_feature = self.data.iloc[:, 16]

    # The getitem method returns the sample with given index
    # x will consist of the 16 input features for the given sample,
    # whereas y will consist of the 1 output feature for the given sample.
    def __getitem__(self, index):
        # Fetch inputs
        x = self.input_features.iloc[index].values
        # Fetch outputs
        y = self.output_feature.iloc[index]
        return x, y

    # Finally, the len special method should return the number of samples,
    # in the dataset. We could use self.dataset_length, but it is more
    # modular to use len(self.data).
    def __len__(self):
        return len(self.data)
```


Writing a custom Dataset object

To write a custom **dataset class** in PyTorch, we need to:

- Create a **class** that **subclasses** **torch.utils.data.Dataset**.
- Define a **__init__** that takes in the required arguments, and stores them as member variables.
- Define a **__getitem__** that takes **index** as input, and **returns** the **data** and **label** at that index as an array. This will allow to use the square bracket notation on our dataset object.
- Define a method **__len__** that returns the **number of samples in the dataset**.

```

1  # Instantiate the dataset
2  ames_dataset = AmesHousingDataset('./ames/AmesHousing.xlsx')
3
4  # Fetch sample with index 286
5  sample_input, sample_output = ames_dataset[286]
6  # Input is a (16,) numpy array, with the following values
7  print(type(sample_input), sample_input.shape)
8  print(sample_input)
9  # Output is a single value, of type numpy int64
10 print(type(sample_output), sample_output.shape)
11 print(sample_output)

```

```

<class 'numpy.ndarray'> (16,)
[6858    6    4 1915 1950  806  841  806 1647    1    1    4    1    6
 216 2010]
<class 'numpy.int64'> ()
128000

```


Writing a custom Dataloader

- Before we can feed this dataset object to Neural Networks, we need to supplement it with a **Dataloader**.
- The Dataloader will **shuffle the samples randomly** and produce **mini-batches** of a given size.
- This Dataloader typically allows for **stochastic mini-batches**, as discussed in Week 2.
- The Dataloader will also **transform arrays into tensors**.

```
1 # Instantiate the dataset
2 ames_dataset = AmesHousingDataset('./ames/AmesHousing.xlsx')
3
4 # Create a DataLoader for the dataset
5 ames_dataloader = torch.utils.data.DataLoader(ames_dataset, \
6                                              batch_size = 32, \
7                                              shuffle = True)
```

Writing a custom Dataloader

- We can then use this custom Dataloader to generate mini-batches using a **for** loop.
- Notice how this Dataloader generates **92** ($= 2928/32$, rounded up) **batches** of **32 samples**.
- With the exception of the **last batch** (with index 91), that only contains **16 samples** ($= 2928 \% 32$).

Restricted

```
1 for batch_number, batch in enumerate(ames_dataloader):
2     inputs, outputs = batch
3     print("---")
4     print("Batch number: ", batch_number)
5     print(inputs.shape)
6     print(outputs.shape)
```

```
---
Batch number:  0
torch.Size([32, 16])
torch.Size([32])
```

```
---
Batch number:  1
torch.Size([32, 16])
torch.Size([32])
```

```
---
Batch number:  2
torch.Size([32, 16])
torch.Size([32])
```

```
---
Batch number:  3
torch.Size([32, 16])
torch.Size([32])
```

```
---
Batch number:  4
```

```
---
Batch number:  91
torch.Size([16, 16])
torch.Size([16])
```

Restricted

Writing a custom Dataloader

- We can then use this custom Dataloader to generate mini-batches using a **for** loop.
- **Good practice:** the custom Dataset and custom Dataloader definition should be repeated to generate **training, testing** and **validation sets dataloaders**.
- This (or these) Dataloader(s) will then be fed to our **trainer** function later on.

```

1  for batch_number, batch in enumerate(ames_dataloader):
2      inputs, outputs = batch
3      print("---")
4      print("Batch number: ", batch_number)
5      print(inputs.shape)
6      print(outputs.shape)

```

```

---
Batch number:  0
torch.Size([32, 16])
torch.Size([32])
---
Batch number:  1
torch.Size([32, 16])
torch.Size([32])
---
Batch number:  2
torch.Size([32, 16])
torch.Size([32])
---
Batch number:  3
torch.Size([32, 16])
torch.Size([32])
---
Batch number:  4

```

```

---
Batch number:  91
torch.Size([16, 16])
torch.Size([16])

```

Introducing the MNIST dataset

MNIST is a widely-used dataset for the benchmarking of machine learning and computer vision algorithms.

It consists of

- a **training set** of 60,000 examples,
- and a **testing set** of 10,000 examples.

All samples consist of 28x28 pixel grayscale images of handwritten digits (0 to 9).

MNIST is often used as a "Hello, World!" example, due to its simplicity and the availability of efficient implementations of various learning algorithms.



Introducing the MNIST dataset

MNIST is a widely-used dataset for the benchmarking of machine learning and computer vision algorithms.

It consists of

- a **training set** of 60,000 examples,
- and a **testing set** of 10,000 examples.

All samples consist of 28x28 pixel grayscale images of handwritten digits (0 to 9).

Careful however: the MNIST dataset is often accused of having been **overused** (which is true) and of **being too simple**.



Introducing the MNIST dataset

- It is a good dataset to use for testing and comparing the performance of different models, as well as for getting familiar with the basics of machine learning and deep learning.
- The images serve as inputs, and the task is therefore to predict which of the ten digits appears in the image.
- This is therefore a **classification task**, like before, except that it consists of **10 different classes** (corresponding to the 0-9 digits) **instead of just two like in binary classification**.



Writing the MNIST Dataset and Dataloader

```
1 # Define transform to convert images to tensors and normalize them
2 transform_data = Compose([ToTensor(),
3                             Normalize((0.1307,), (0.3081,))])
4
5 # Load the data
6 batch_size = 64
7 train_dataset = MNIST(root='./mnist/', train = True, download = True, transform = transform_data)
8 test_dataset = MNIST(root='./mnist/', train = False, download = True, transform = transform_data)
9 train_loader = torch.utils.data.DataLoader(train_dataset, batch_size = batch_size, shuffle = True)
10 test_loader = torch.utils.data.DataLoader(test_dataset, batch_size = batch_size, shuffle = False)
```

```
1 # Try the dataloader
2 for batch_number, batch in enumerate(train_loader):
3     inputs, outputs = batch
4     print("---")
5     print("Batch number: ", batch_number)
6     print(inputs.shape)
7     print(outputs.shape)
8     break
```

```
---
Batch number: 0
torch.Size([64, 1, 28, 28])
torch.Size([64])
```


Introducing the MNIST dataset

In general, it is good to do the following:

- Scale the data (pixel values) to the $[0,1]$ range.
- Normalize the data to have zero mean and unit standard deviation.

We will then subtract a mean of 0.1307 and divide by a standard deviation of 0.3081.

These values are basically the mean and the standard deviation of the dataset divided by 255 (original max pixel value).

```
# Define transform to convert images to tensors and normalize them  
transform_data = Compose([ToTensor(),  
                           Normalize((0.1307,), (0.3081,))])
```


Writing the MNIST Dataset and Dataloader

```
1 # Define transform to convert images to tensors and normalize them
2 transform_data = Compose([ToTensor(),
3                             Normalize((0.1307,), (0.3081,))])
4
5 # Load the data
6 batch_size = 64
7 train_dataset = MNIST(root='./mnist/', train = True, download = True, transform = transform_data)
8 test_dataset = MNIST(root='./mnist/', train = False, download = True, transform = transform_data)
9 train_loader = torch.utils.data.DataLoader(train_dataset, batch_size = batch_size, shuffle = True)
10 test_loader = torch.utils.data.DataLoader(test_dataset, batch_size = batch_size, shuffle = False)
```

```
1 # Try the dataloader
2 for batch_number, batch in enumerate(train_loader):
3     inputs, outputs = batch
4     print("---")
5     print("Batch number: ", batch_number)
6     print(inputs.shape)
7     print(outputs.shape)
8     break
```

```
---
Batch number: 0
torch.Size([64, 1, 28, 28])
torch.Size([64])
```

From binary to multi-class classification

In **binary classification**, we would produce a **single value p** as **output**, with p between 0 and 1.

- This value p would correspond to the **probability of being of class 1**.
- The **probability of being of class 0** would then simply be $1 - p$.
- We would then use a **threshold 0.5** to decide if the sample is predicted of class 0 or 1.

- Unfortunately, when we have more than 2 classes, **we can no longer rely on a single output value p** .

- Instead, it is often preferable to have the model output **10 values**:

$$(p_0, p_1, p_2, \dots, p_9)$$

- Where each p_i corresponds to the **probability of being of class i** .
- This could typically be done by asking for **the final layer to produce $n_y = 10$ values** instead of just a single $n_y = 1$ value.

From binary to multi-class classification

- Unfortunately, this is not good enough.
- The p_i are probabilities and **their sum should be equal to 1**, i.e.
- To normalize the outputs produced by the final fully connected layer, we will use the **softmax** operation, which is a special activation function.

$$\sum_i^9 p_i = 1$$

- A **fully connected layer** ($WX + b$) is not smart enough to do that on its own: it might produce values that may not sum up to 1.

Softmax function

Definition (the **softmax** function):

The **softmax** function transforms a vector of N values

$$Y = (y_0, y_1, y_2, \dots, y_K),$$

into another vector of N values

$$P = (p_0, p_1, p_2, \dots, p_K).$$

The new vector **P** is **guaranteed to be summing up to 1**, i.e.

$$\sum_{k=0}^K p_k = 1$$

The **softmax** operation

$$p_i = s(y_i, y_{-i})$$

is defined, $\forall i$, as:

$$p_i = s(y_i, y_{-i}) = \frac{\exp(y_i)}{\sum_k^K \exp(y_k)}$$

Note: the y_{-i} notation comes from game theory, and consists of every element in vector Y except y_i .

$$y_{-i} = (y_1, \dots, y_{i-1}, y_{i+1}, \dots, y_K)$$

Softmax function

Softmax can be manually implemented as:

```

1 def softmax(x):
2     # Subtract the maximum value from each element of the input vector x
3     # to avoid numerical instability (this is optional, but equivalent)
4     x = x - np.max(x)
5     # Compute the exponent of each element
6     exp_x = np.exp(x)
7     # Normalize the exponentiated values by their sum
8     return exp_x/np.sum(exp_x)

```

```

1 # Ten values that do not sum up to 1
2 Y = np.array([-1, 4, 10, 5, 7, 1, 4, -2, -1, 2])/20
3 print(sum(Y))
4 P = softmax(Y)
5 print(P)
6 print(np.sum(P))

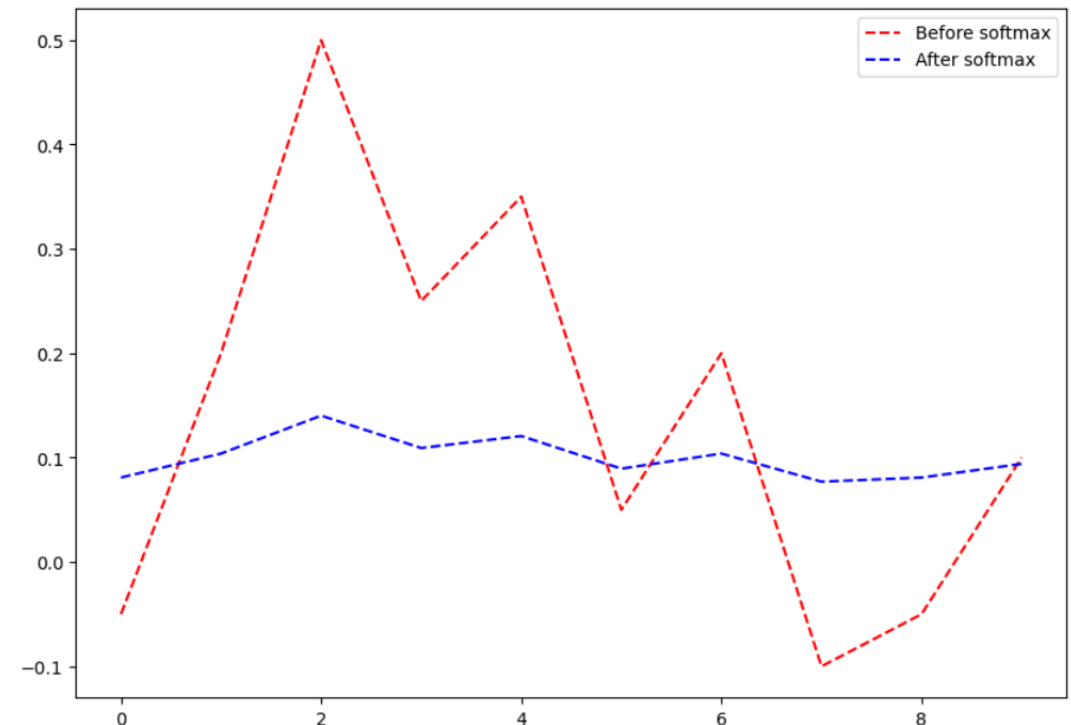
```

```

1.45
[0.08089815 0.10387528 0.14021696 0.10920108 0.12068586 0.08940628
 0.10387528 0.0769527  0.08089815 0.09399024]
0.9999999999999999

```

It will rescale values so that the trend is preserved, but the new vector sums to 1.



Softmax function

In the case of multi-label classification, we will use the **softmax** operation as the final activation after the last fully connected layer.

This will produce a vector of 10 values,

$$P = (p_0, p_1, p_2, \dots, p_9),$$

corresponding to the **probability for sample of being of each class i** .

The predicted class $pred$ will be the index i of the highest p_i value, i.e.

$$pred = \arg \max_i [p_i].$$

Softmax function

For instance, this simple neural network consists of **two fully-connected/linear layers**.

A **single ReLU activation** is used between both layers.

No final activation yet.

It also consists of a **flattening** operation, which will transform our input images (2D tensors, size 28 by 28), into a “flattened” 1D tensor with size 784 ($= 28 \times 28$).

```
1 class ShallowNeuralNet(torch.nn.Module):
2     def __init__(self, n_x, n_h, n_y):
3         super().__init__()
4         # Define two layers using the nn.Linear()
5         self.fc1 = torch.nn.Linear(n_x, n_h)
6         self.fc2 = torch.nn.Linear(n_h, n_y)
7
8     def forward(self, x):
9         # Flatten images (transform them from 28x28 2D
10        # matrices to 784 1D vectors)
11        x = x.view(x.size(0), -1)
12        # First Wx + b operation
13        out1 = self.fc1(x)
14        # Using ReLU operation as activation after first layer
15        act1 = torch.relu(out1)
16        # Second Wx + b operation and return
17        out2 = self.fc2(act1)
18        return out2
```

Softmax function

- Consider a neural network model, with **784 input size**, **128 hidden size**, and **10 output size**, eventually transferring the model to a device.
- Next, we get a **single sample** from the **train_loader** iterator and extract sample info in the variables **data** and **target**.
- This can be simply done with the **next** and **iter** functions.

```

1  # Initialize model
2  model = ShallowNeuralNet(n_x = 784, \
3                           n_h = 128, \
4                           n_y = 10).to(device)
5
6  # Get a single sample
7  sample = next(iter(train_loader))
8  data, target = sample
9  print(data.shape)
10 print(target.shape)
11 data1 = data[0].to(device)
12 target1 = target[0].to(device)
13 print(data1.shape)
14 print(target1)
15
16 # Forward pass
17 out2 = model(data1)
18 act2 = torch.nn.functional.softmax(out2, dim = 1)
19 print(out2)
20 print(act2)
21 print(act2.sum())

```

torch.Size([64, 1, 28, 28])
 torch.Size([64])
 torch.Size([1, 28, 28])
 tensor(4, device='cuda:0')
 tensor([[[-0.2145, -0.0527, -0.1197, -0.0488, 0.2503, -0.1870, -0.1401, 0.1202,
 0.1220, 0.1017]], device='cuda:0', grad_fn=<AddmmBackward0>])
 tensor([[0.0812, 0.0954, 0.0892, 0.0958, 0.1292, 0.0834, 0.0874, 0.1134, 0.1136,
 0.1113]], device='cuda:0', grad_fn=<SoftmaxBackward0>])
 tensor(1., device='cuda:0', grad_fn=<SumBackward0>)

Softmax function

- Perform a forward pass through the model, storing the output in the variable out2.
- Apply the softmax operation on out2., PyTorch offers a functional implementation of the softmax:

torch.nn.functional.softmax()

- We can then verify that softmax will adjust the output of the neural network so that it is now summing up to 1.

```

1 # Initialize model
2 model = ShallowNeuralNet(n_x = 784, \
3                           n_h = 128, \
4                           n_y = 10).to(device)
5
6 # Get a single sample
7 sample = next(iter(train_loader))
8 data, target = sample
9 print(data.shape)
10 print(target.shape)
11 data1 = data[0].to(device)
12 target1 = target[0].to(device)
13 print(data1.shape)
14 print(target1)
15
16 # Forward pass
17 out2 = model(data1)
18 act2 = torch.nn.functional.softmax(out2, dim = 1)
19 print(out2)
20 print(act2)
21 print(act2.sum())

```

```

torch.Size([64, 1, 28, 28])
torch.Size([64])
torch.Size([1, 28, 28])
tensor(4, device='cuda:0')
tensor([[[-0.2145, -0.0527, -0.1197, -0.0488,  0.2503, -0.1870, -0.1401,  0.1202,
          0.1220,  0.1017]], device='cuda:0', grad_fn=<AddmmBackward0>])
tensor([[0.0812, 0.0954, 0.0892, 0.0958, 0.1292, 0.0834, 0.0874, 0.1134, 0.1136,
          0.1113]], device='cuda:0', grad_fn=<SoftmaxBackward0>])
tensor(1., device='cuda:0', grad_fn=<SumBackward0>)

```

Forward implementation

- Our model is however the same as before.

We do not use the softmax operation in the forward.

- This is normal as **the softmax operation will be used in the loss function, cross_entropy,** which will be used in the **trainer()** function later.

```
1 class ShallowNeuralNet(torch.nn.Module):
2     def __init__(self, n_x, n_h, n_y):
3         super().__init__()
4         # Define two layers using the nn.Linear()
5         self.fc1 = torch.nn.Linear(n_x, n_h)
6         self.fc2 = torch.nn.Linear(n_h, n_y)
7
8     def forward(self, x):
9         # Flatten images (transform them from 28x28 2D
10        # matrices to 784 1D vectors)
11        x = x.view(x.size(0), -1)
12        # First Wx + b operation
13        out1 = self.fc1(x)
14        # Using ReLU operation as activation after first layer
15        act1 = torch.relu(out1)
16        # Second Wx + b operation and return
17        out2 = self.fc2(act1)
18        return out2
```

From binary cross entropy...

- Speaking of, in the case of the **binary classification**, we used the following loss function, namely the **log-likelihood function**.

$$L(x, y) = -\frac{1}{N} \sum_k^N y_k \ln(p(x_k)) + (1 - y_k) \ln(1 - p(x_k))$$

- But in the case of MNIST, **we have more than two classes...**

How does the loss function change now that we have 10 classes?

...To multi-class cross entropy!

- The adjustment is actually quite simple, and the **multi-class cross-entropy loss function** simply rewrites as shown below:

$$L(x, y) = -\frac{1}{N} \sum_k^N \sum_{i=0}^9 y_k^i \ln(p_i(x_k)).$$

In the formula above, $p_i(x_k)$ denotes the **probability for sample x_k of being of class i** . In other words, it is the i -th value of the output vector produced by the model for sample x_k after softmax has been applied.

The value y_k^i is the ground truth value for the probability of being of class i for sample x_k . For instance, if the sample x_k is of class 2:

$$Y_k = (y_k^0, y_k^1, y_k^2, y_k^3, \dots, y_k^9) = (0, 0, 1, 0, \dots, 0).$$

Setting a model in train mode

New good practice: some operations (layers, activations, etc.) in forward will have **two different behaviors** depending on whether

- the model is currently **training**,
- or if we are using its trained version for **evaluation**.

(Note: at the moment, we have not seen such operations.)

But let us keep this in mind, and accept that is good practice to set the model to either **train()** or **eval()** mode.

```
1 # Initialize the model and optimizer
2 model = ShallowNeuralNet(n_x = 784, n_h = 64, n_y = 10).to(device)
3 optimizer = torch.optim.Adam(model.parameters(), lr = 1e-3)
4 # Set model in train mode!
5 model.train()
```

Training our model with Adam GD, as before

```
1 # Training model
2 num_epochs = 5
3 for epoch in range(num_epochs):
4     # Go through all samples in train dataset
5     for i, (images, labels) in enumerate(train_loader):
6         # Get from dataloader and send to device
7         images = images.to(device)
8         labels = labels.to(device)
9         # Forward pass
10        outputs = model(images)
11        # Compute loss
12        loss = torch.nn.functional.cross_entropy(outputs, labels)
13        # Backward and optimize
14        optimizer.zero_grad()
15        loss.backward()
16        optimizer.step()
17        # Display
18        if (i+1) % 300 == 0:
19            print (f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/{len(train_loader)}], Loss: {loss.item():.4f}')
```

```
Epoch [1/5], Step [300/938], Loss: 0.2895
Epoch [1/5], Step [600/938], Loss: 0.1588
Epoch [1/5], Step [900/938], Loss: 0.1753
Epoch [2/5], Step [300/938], Loss: 0.0461
Epoch [2/5], Step [600/938], Loss: 0.1251
Epoch [2/5], Step [900/938], Loss: 0.1592
Epoch [3/5], Step [300/938], Loss: 0.1241
Epoch [3/5], Step [600/938], Loss: 0.0511
Epoch [3/5], Step [900/938], Loss: 0.0553
Epoch [4/5], Step [300/938], Loss: 0.0514
Epoch [4/5], Step [600/938], Loss: 0.0339
```

Eval mode and accuracy after training

After training, we will evaluate our trained model.

- Set the model in **eval() mode** (good practice for later).
- Predict on **test dataloader**.
- Calculate **accuracy** manually (we could have probably also used a torch function to do that).
- Good trained model!

```
1 # Evaluate model accuracy on test after training
2 # Set model in eval mode!
3 model.eval()
4 # Evaluate
5 with torch.no_grad():
6     correct = 0
7     total = 0
8     for images, labels in test_loader:
9         # Get images and labels from test loader
10        images = images.to(device)
11        labels = labels.to(device)
12        # Forward pass and predict class using max
13        outputs = model(images)
14        _, predicted = torch.max(outputs.data, 1)
15        # Check if predicted class matches label
16        # and count number of correct predictions
17        total += labels.size(0)
18        correct += (predicted == labels).sum().item()
19 # Compute final accuracy and display
20 accuracy = correct/total
21 print(f'Evaluation after training, Accuracy: {accuracy:.4f}')
```

Evaluation after training, Accuracy: 0.9712

It is now time...

It is now time for us to define and train our first **Deep Neural Network**.

By definition, a deep neural network consists of more than two layers.

Here we will create a deep neural network with four layers:

- **three linear layers** with **ReLU** activation,
- followed by **one linear layer**, finished with a **softmax activation**.

In general, **it is good practice to have the size decrease progressively by a factor of at least 2 from one layer to another.**

For instance, we decided here, to have

- the first layer receive inputs of size 784 and produce outputs of size 80,
- the second layer receive inputs of size 80 and produce outputs of size 80,
- the third layer receive inputs of size 40 and produce outputs of size 40,
- and the fourth layer receive inputs of size 20 and produce outputs of size 10.

Creating blocks

Good practice (another one): create building blocks for modularity in your neural networks.

- The **DenseReLU** class is a custom PyTorch module that consists of a **linear layer** followed by a **ReLU activation** function.
- The **DenseNoReLU** class is similar, but it applies **no activation function after the linear layer**.

Important note: no softmax, for the same reason as before.

```
1 class DenseReLU(torch.nn.Module):
2     def __init__(self, n_x, n_y):
3         super().__init__()
4         # Define Linear layer using the nn.Linear()
5         self.fc = torch.nn.Linear(n_x, n_y)
6
7     def forward(self, x):
8         # Wx + b operation
9         # Using ReLU operation as activation after
10        return torch.relu(self.fc(x))
```

```
1 class DenseNoReLU(torch.nn.Module):
2     def __init__(self, n_x, n_y):
3         super().__init__()
4         # Define Linear layer using the nn.Linear()
5         self.fc = torch.nn.Linear(n_x, n_y)
6
7     def forward(self, x):
8         # Wx + b operation
9         # No activation function
10        return self.fc(x)
```

Creating blocks

The DeepNeuralNet class will here represent the overall deep neural network.

It starts by initializing four layers:

- Three **DenseReLU** layers,
- And one **DenseNoReLU**.

It then combines them into a single PyTorch sequential model using **torch.nn.Sequential()**.

```
class DeepNeuralNet(torch.nn.Module):
    def __init__(self, n_x, n_h, n_y):
        super().__init__()
        # Define three Dense + ReLU layers,
        # followed by one Dense + Softmax layer
        self.layer1 = DenseReLU(n_x, n_h[0])
        self.layer2 = DenseReLU(n_h[0], n_h[1])
        self.layer3 = DenseReLU(n_h[1], n_h[2])
        self.layer4 = DenseNoReLU(n_h[2], n_y)

        # Combine all four layers
        self.combined_layers = torch.nn.Sequential(self.layer1,
                                                    self.layer2,
                                                    self.layer3,
                                                    self.layer4)

    def forward(self, x):
        # Flatten images (transform them from 28x28
        # 2D matrices to 784 1D vectors)
        x = x.view(x.size(0), -1)
        # Pass through all four layers
        out = self.combined_layers(x)
        return out
```

```
# Initialize the model and optimizer
model = DeepNeuralNet(n_x = 784, n_h = [80, 40, 20], n_y = 10).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr = 1e-3)
```

Creating blocks

The forward pass of the network is then simply defined:

- The input image is first flattened from a 2D image to a 1D vector,
- It is then passed through the combined layers we have assembled in Sequential().

This modular approach, defining blocks of layers and eventually assembling them in a larger network, is very common in deep neural networks, especially when the architectures are very heavy and include many layers.

```
class DeepNeuralNet(torch.nn.Module):
    def __init__(self, n_x, n_h, n_y):
        super().__init__()
        # Define three Dense + ReLU layers,
        # followed by one Dense + Softmax layer
        self.layer1 = DenseReLU(n_x, n_h[0])
        self.layer2 = DenseReLU(n_h[0], n_h[1])
        self.layer3 = DenseReLU(n_h[1], n_h[2])
        self.layer4 = DenseNoReLU(n_h[2], n_y)

        # Combine all four layers
        self.combined_layers = torch.nn.Sequential(self.layer1,
                                                    self.layer2,
                                                    self.layer3,
                                                    self.layer4)

    def forward(self, x):
        # Flatten images (transform them from 28x28
        # 2D matrices to 784 1D vectors)
        x = x.view(x.size(0), -1)
        # Pass through all four layers
        out = self.combined_layers(x)
        return out
```

```
# Initialize the model and optimizer
model = DeepNeuralNet(n_x = 784, n_h = [80, 40, 20], n_y = 10).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr = 1e-3)
```

Training our model as before

```
1 # Training model
2 num_epochs = 10
3 for epoch in range(num_epochs):
4     # Go through all samples in train dataset
5     for i, (images, labels) in enumerate(train_loader):
6         # Get from dataloader and send to device
7         images = images.to(device)
8         labels = labels.to(device)
9         # Forward pass
10        outputs = model(images)
11        # Compute loss
12        loss = torch.nn.functional.cross_entropy(outputs, labels)
13        # Backward and optimize
14        optimizer.zero_grad()
15        loss.backward()
16        optimizer.step()
17        # Display
18        if (i+1) % 25 == 0:
19            print (f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/{len(train_loader)}], Loss: {loss.item():.4f}')
```

```
Epoch [1/10], Step [25/235], Loss: 2.2469
Epoch [1/10], Step [50/235], Loss: 2.1274
Epoch [1/10], Step [75/235], Loss: 1.9379
Epoch [1/10], Step [100/235], Loss: 1.8415
Epoch [1/10], Step [125/235], Loss: 1.6548
Epoch [1/10], Step [150/235], Loss: 1.4660
Epoch [1/10], Step [175/235], Loss: 1.2895
Epoch [1/10], Step [200/235], Loss: 1.1478
Epoch [1/10], Step [225/235], Loss: 0.9910
Epoch [2/10], Step [25/235], Loss: 0.8251
```

On Deep Neural Networks complexity

We can therefore raise a fairly important question:

What is the appropriate number of layers to use and how many neurons should we use on each layer?

- We established in Week 2 – Notebook 8, that there is no fixed rule for how many layers should be used in a deep neural network.
- Instead, the number of layers, as well as the number of neurons in each layer, should be chosen based on the complexity of the task and the amount of available data.
- In general, deep neural networks with many layers (hundreds or even thousands) can learn very complex patterns in data, but they will require a large amount of data and computational resources to train.

On Deep Neural Networks complexity

- More importantly, if the network is **too deep**, it may also be prone to **overfitting**, which can hinder its generalization performance on unseen data.
- On the other hand, **shallow networks** with fewer layers may be easier to train and require less data, but **they may not be able to learn as complex patterns**.
- **NFL: Finding the optimal number of layers and the optimal architecture of a deep neural network is often a trade-off between model complexity, computational resources, and performance, and requires some experimentation and model selection.**

In fact, our DNN is overfitting at the moment!

**Shallow Neural Net: 96.5% test acc
(not too bad)**

```

1 # Evaluate model accuracy on test after training
2 # Set model in eval mode!
3 model.eval()
4 # Evaluate
5 with torch.no_grad():
6     correct = 0
7     total = 0
8     for images, labels in test_loader:
9         # Get images and labels from test loader
10         images = images.to(device)
11         labels = labels.to(device)
12         # Forward pass and predict class using max
13         outputs = model(images)
14         _, predicted = torch.max(outputs.data, 1)
15         # Check if predicted class matches label
16         # and count number of correct predictions
17         total += labels.size(0)
18         correct += (predicted == labels).sum().item()
19 # Compute final accuracy and display
20 accuracy = correct/total
21 print(f'Evaluation after training, Accuracy: {accuracy:.4f}')
```

Evaluation after training, Accuracy: 0.9649

**Deep Neural Net: 93.7% test acc
(lower, even though we had a lower loss!)**

```

1 # Evaluate model accuracy on test after training
2 # Set model in eval mode!
3 model.eval()
4 # Evaluate
5 with torch.no_grad():
6     correct = 0
7     total = 0
8     for images, labels in test_loader:
9         # Get images and labels from test loader
10         images = images.to(device)
11         labels = labels.to(device)
12         # Forward pass and predict class using max
13         outputs = model(images)
14         _, predicted = torch.max(outputs.data, 1)
15         # Check if predicted class matches label
16         # and count number of correct predictions
17         total += labels.size(0)
18         correct += (predicted == labels).sum().item()
19 # Compute final accuracy and display
20 accuracy = correct/total
21 print(f'Evaluation after training, test accuracy: {accuracy:.4f}')
```

Evaluation after training, test accuracy: 0.9367

Experimenting on layers numbers and sizes

In fact, in Notebook 7, we trained three DNN models:

- **Model 1:** 6 layers (probably too many layers)

$$n_h = [320, 160, 80, 40, 20],$$

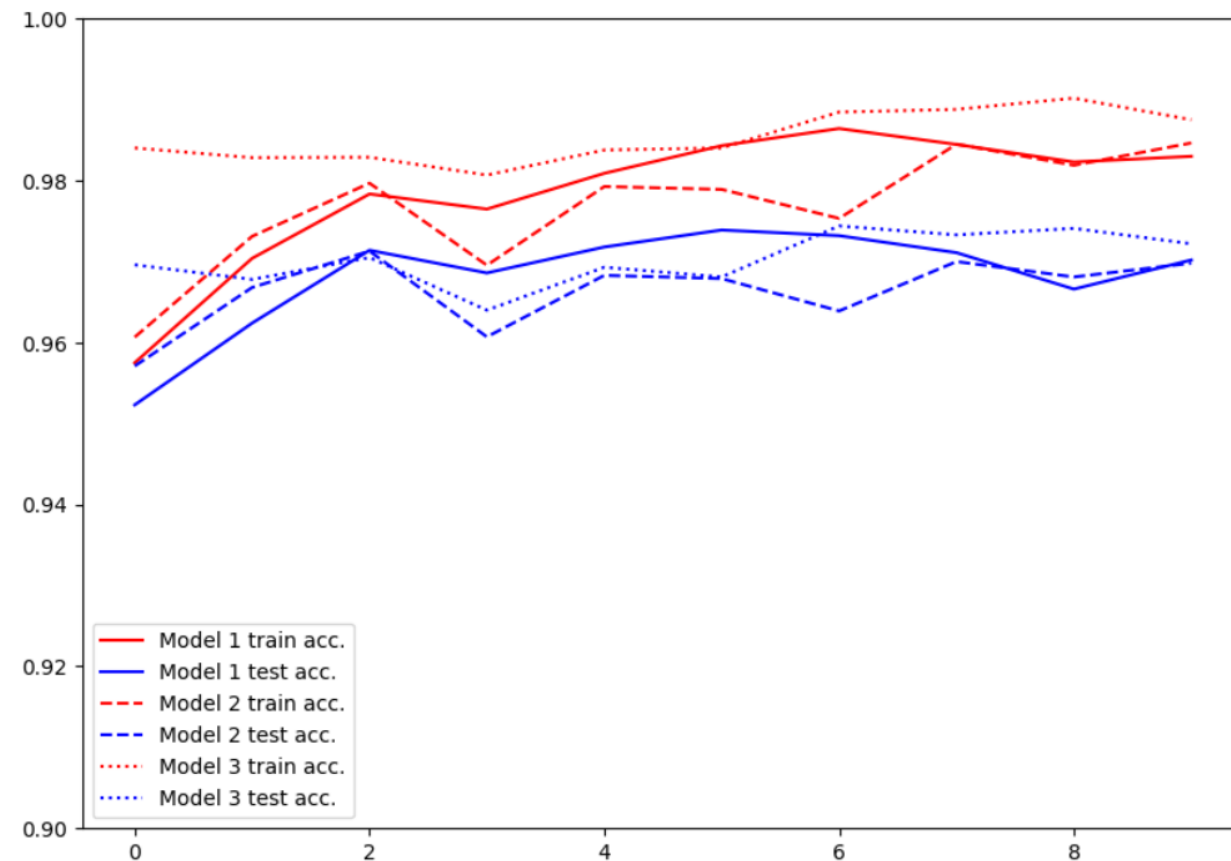
- **Model 2:** 3 layers (layers probably too large)

$$n_h = [400, 200],$$

- **Model 3:** 3 layers (just fine?)

$$n_h = [40, 20].$$

While simpler, model 3 has highest test accuracy!



Introduction (Week 3)

1. What is the **PyTorch library** and its **benefits**?
2. What is a **PyTorch tensor object** and its typical **attributes**?
3. How to implement some typical **tensor operations**?
4. What is **broadcasting** on tensors?
5. What are **tensor locations** in terms of computation?
6. How to **transform our original NumPy shallow Neural Network** class so it uses **PyTorch** now instead?
7. How to implement a **forward, loss** and **accuracy** metric in PyTorch?
8. What are some measurable **performance benefits** of using **PyTorch** over NumPy and **GPUs** over CPUs?

Introduction (Week 3)

9. What is the **autograd/backprop** module in PyTorch, and how does it use a **computational graph** to **compute all derivatives**?
10. How to use the **autograd** to implement **derivatives** and a **vanilla gradient descent**?
11. How to implement **backprop** in PyTorch for our **shallow Neural Network** class?
12. How to use **PyTorch** to implement **advanced optimizers**?
13. How to use **PyTorch** to implement **advanced initializers**?
14. How to use **PyTorch** to implement **regularization**?
15. How to finally revise our **trainer** function to obtain a minimal, yet complete Neural Network in PyTorch?

Introduction (Week 3)

- 16. What are the **Dataset** and **Dataloader** objects in **PyTorch**?
- 17. How to implement a custom **Dataloader** and **Dataset** object in PyTorch?
- 18. How to move from binary classification to **multi-class classification**?
- 19. How to adjust output probabilities using the **softmax** function?
- 20. How to change the **cross-entropy loss** so it works in **multi-class classification**?
- 21. How to implement **building blocks** in PyTorch?
- 22. How to implement and train our first **Deep Neural Network**?
- 23. What are **additional good practices** in PyTorch?

Conclusion (Week 3)

- PyTorch library and its benefits
- Tensor objects, attributes and operations on tensors
- Converting our NumPy shallow neural network into PyTorch
- Parameter objects
- Forward method implementation
- Performance benefits of GPU acceleration
- Using autograd and computational graphs
- Advanced optimizers in PyTorch
- Initializers in PyTorch
- Regularization in PyTorch
- Dataset and Dataloader objects
- Multi-class classification
- Softmax function and multi-class cross entropy loss
- Building blocks in PyTorch
- Our first Deep Neural Network!
- Network size vs. overfitting tradeoff

Homework 1 and project announcement!

Let us discuss it now.