50.039 Theory and Practice of Deep Learning

W3-S3 Introduction to Deep Learning using the PyTorch framework

Matthieu De Mari

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

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!

```python
class ShallowNeuralNet_PT(torch.nn.Module):
    def __init__(self, n_x, n_h, n_y, device):
        super().__init__()
        self.n_x, self.n_h, self.n_y = n_x, n_h, n_y
        self.W1 = torch.nn.Parameter(torch.zeros(size = (n_x, n_h), requires_grad = True, \
                                                 dtype = torch.float64, device = device))
        torch.nn.init.xavier_uniform_(self.W1.data)
        self.b1 = torch.nn.Parameter(torch.zeros(size = (1, n_h), requires_grad = True, \
                                                 dtype = torch.float64, device = device))
        torch.nn.init.xavier_uniform_(self.b1.data)
        self.W2 = torch.nn.Parameter(torch.zeros(size = (n_h, n_y), requires_grad = True, \
                                                 dtype = torch.float64, device = device))
        torch.nn.init.xavier_uniform_(self.W2.data)
        self.b2 = torch.nn.Parameter(torch.zeros(size = (1, n_y), requires_grad = True, \
                                                 dtype = torch.float64, device = device))
        torch.nn.init.xavier_uniform_(self.b2.data)
        self.loss = torch.nn.BCELoss()
        self.accuracy = BinaryAccuracy()
    def forward(self, inputs):
        return torch.sigmoid(torch.matmul(torch.sigmoid(torch.matmul(inputs, self.W1) + self.b1), self.W2) + self.b2)
    def train(self, inputs, outputs, N_max = 1000, alpha = 1, beta1 = 0.9, beta2 = 0.999, \
              batch_size = 32, lambda_val = 1e-3):
        dataset = torch.utils.data.TensorDataset(inputs, outputs)
        data_loader = torch.utils.data.DataLoader(dataset, batch_size = batch_size, shuffle = True)
        optimizer = torch.optim.Adam(self.parameters(), lr = alpha, betas = (beta1, beta2), eps = 1e-08)
        optimizer.zero_grad()
        self.loss_history = []
        for iteration_number in range(1, N_max + 1):
            for batch in data_loader:
                inputs_batch, outputs_batch = batch
                total_loss = self.loss(self(inputs_batch), outputs_batch.to(torch.float64))\
                    + lambda_val*sum(torch.abs(param).sum() for param in self.parameters()).item()
                self.loss_history.append(total_loss)
                total_loss.backward()
                optimizer.step()
                optimizer.zero_grad()
            if(iteration_number % (N_max//20) == 1):
                pred = self(inputs)
                acc_val = self.accuracy(pred, outputs).item()
                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.

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

```python
# Define folder path as string
# Dataset will be downloaded and stored there.
folder_path = "./data"

# Download (download = True) training data (train = True)
# to folder specified in root parameter.
# The transform parameter specifies that the image samples will
# be converted to Tensors, ready to be used by models.
training_data = datasets.FashionMNIST(root = folder_path, \
                                      train = True, \
                                      download = True, \
                                      transform = ToTensor())
```

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.

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

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

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

```
Image:  torch.Size([1, 28, 28])
Label:  8
```

```
# Display sample
sample_index = 894
img, label = training_data[sample_index]
plt.figure(figsize = (3, 3))
plt.axis("off")
plt.imshow(img.squeeze(), cmap="gray")
plt.title("Image {} - Label = {} ({})".format(sample_index, \
                                               label, \
                                               labels_map[label]))
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

AmesHousing.xlsx

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 AbvGrd	Garage Area	Yr Sold	SalePrice
31770	6	5	1960	1960	1080	1656	0	1656	1	0	3	1	7	528	2010	215
11622	5	6	1961	1961	882	896	0	896	1	0	2	1	5	730	2010	105
14267	6	6	1958	1958	1329	1329	0	1329	1	1	3	1	6	312	2010	172
11160	7	5	1968	1968	2110	2110	0	2110	2	1	3	1	8	522	2010	244
13830	5	5	1997	1998	928	928	701	1629	2	1	3	1	6	482	2010	189
9978	6	6	1998	1998	926	926	678	1604	2	1	3	1	7	470	2010	195
4920	8	5	2001	2001	1338	1338	0	1338	2	0	2	1	6	582	2010	213
5005	8	5	1992	1992	1280	1280	0	1280	2	0	2	1	5	506	2010	191
5389	8	5	1995	1996	1595	1616	0	1616	2	0	2	1	5	608	2010	236
7500	7	5	1999	1999	994	1028	776	1804	2	1	3	1	7	442	2010	189
10000	6	5	1993	1994	763	763	892	1655	2	1	3	1	7	440	2010	175
7980	6	7	1992	2007	1168	1187	0	1187	2	0	3	1	6	420	2010	185
8402	6	5	1998	1998	789	789	676	1465	2	1	3	1	7	393	2010	180
10176	7	5	1990	1990	1300	1341	0	1341	1	1	2	1	5	506	2010	171
6820	8	5	1985	1985	1488	1502	0	1502	1	1	1	1	4	528	2010	212
53504	8	5	2003	2003	1650	1690	1589	3279	3	1	4	1	12	841	2010	538
12134	8	7	1988	2005	559	1080	672	1752	2	0	4	1	8	492	2010	164
11394	9	2	2010	2010	1856	1856	0	1856	1	1	1	1	8	834	2010	394
19138	4	5	1951	1951	864	864	0	864	1	0	2	1	4	400	2010	141
13175	6	6	1978	1988	1542	2073	0	2073	2	0	3	1	7	500	2010	210
11751	6	6	1977	1977	1844	1844	0	1844	2	0	3	1	7	546	2010	190
10625	7	6	1974	1974	1053	1173	0	1173	2	0	3	1	6	528	2010	170
7500	7	5	2000	2000	814	814	860	1674	2	1	3	1	7	663	2010	216
11241	6	7	1970	1970	1004	1004	0	1004	1	0	2	1	5	480	2010	149
12537	5	6	1971	2008	1078	1078	0	1078	1	1	3	1	6	500	2010	149
8450	5	6	1968	1968	1056	1056	0	1056	1	0	3	1	6	304	2010	142
8400	4	5	1970	1970	882	882	0	882	1	0	2	1	4	525	2010	126
10500	4	5	1971	1971	864	864	0	864	1	0	3	1	5	0	2010	115
5858	7	5	1999	1999	1405	1337	0	1337	2	0	2	1	5	511	2010	184
1680	6	5	1971	1971	483	483	504	987	1	1	2	1	5	264	2010	96
1680	5	5	1971	1971	525	525	567	1092	1	1	3	1	6	320	2010	105
1680	6	5	1971	1971	525	525	567	1092	1	1	3	1	6	264	2010	88
4043	6	5	1977	1977	1069	1069	0	1069	2	0	2	1	4	440	2010	127
2280	6	6	1975	1975	855	855	601	1456	2	1	3	1	6	440	2010	149
2280	7	6	1975	1975	836	836	0	836	1	0	2	1	4	308	2010	120
2280	6	5	1978	1978	855	855	601	1456	2	1	3	1	7	440	2010	146
12858	9	5	2009	2010	1590	1627	707	2334	2	1	3	1	10	751	2010	376
11478	8	5	2007	2008	1704	1704	0	1704	2	0	3	1	7	772	2010	306
10159	9	5	2009	2010	1930	1940	0	1940	2	1	3	1	8	606	2010	395
12883	8	5	2009	2010	1544	1544	0	1544	2	0	3	1	7	868	2010	290
12182	7	5	2005	2005	1541	1541	0	1541	2	0	3	1	7	532	2010	220
11520	9	5	2005	2005	1698	1698	0	1698	2	0	3	1	7	730	2010	275
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, and many 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 AI models for predicting house prices, based on various house 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*](https://pandas.pydata.org/docs/user_guide/10min.html)

```python
# Load dataset using pandas, and showing the first five entries
ames_dataset = pd.read_excel("./ames/AmesHousing.xlsx")
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 an **__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**.

```python
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_fetaures_number = 16
        self.input_features = self.data.iloc[:, :16]

        # Extract outputs
        self.output_fetaures_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 an **__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**.

```python
# Instantiate the dataset
ames_dataset = AmesHousingDataset('./ames/AmesHousing.xlsx')

# Fetch sample with index 286
sample_input, sample_output = ames_dataset[286]
# Input is a (16,) numpy array, with the following values
print(type(sample_input), sample_input.shape)
print(sample_input)
# Output is a single value, of type numpy int64
print(type(sample_output), sample_output.shape)
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
```

Restricted

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**.

```python
# Instantiate the dataset
ames_dataset = AmesHousingDataset('./ames/AmesHousing.xlsx')

# Create a DataLoader for the dataset
ames_dataloader = torch.utils.data.DataLoader(ames_dataset, \
                                              batch_size = 32, \
                                              shuffle = True)
```

Writing a custom Dataloader

- We can then 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).

```python
for batch_number, batch in enumerate(ames_dataloader):
    inputs, outputs = batch
    print("---")
    print("Batch number: ", batch_number)
    print(inputs.shape)
    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])
```

Writing a custom Dataloader

- We can then 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.

```
for batch_number, batch in enumerate(ames_dataloader):
    inputs, outputs = batch
    print("---")
    print("Batch number: ", batch_number)
    print(inputs.shape)
    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

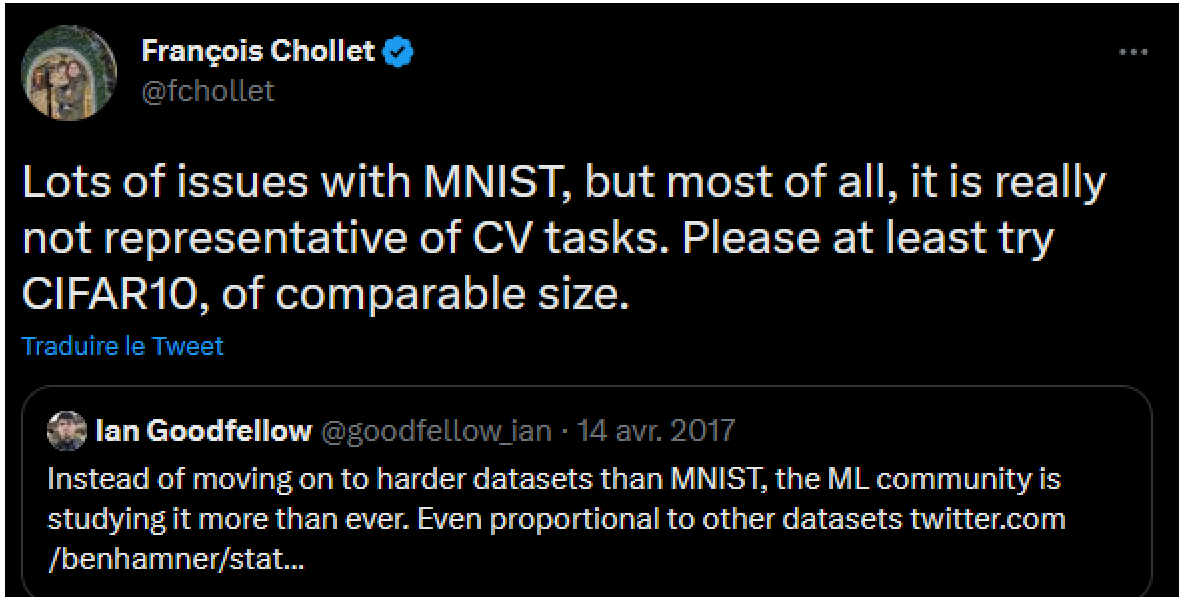
MNIST is a widely-used dataset for the benchmarking of machine learning and computer vision algorithms.

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

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

# Load the data
batch_size = 64
train_dataset = MNIST(root='./mnist/', train = True, download = True, transform = transform_data)
test_dataset = MNIST(root='./mnist/', train = False, download = True, transform = transform_data)
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size = batch_size, shuffle = True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size = batch_size, shuffle = False)
```

```python
# Try the dataloader
for batch_number, batch in enumerate(train_loader):
    inputs, outputs = batch
    print("---")
    print("Batch number: ", batch_number)
    print(inputs.shape)
    print(outputs.shape)
    break
```

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

Introducing the MNIST dataset

Good practice: normalize your data!

In general, do the following:

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

In MNIST, we will then subtract a mean of 0.1307 and divide by a standard deviation of 0.3081.

These values are the original mean and standard deviation of the dataset before normalization!

```
# 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

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

# Load the data
batch_size = 64
train_dataset = MNIST(root='./mnist/', train = True, download = True, transform = transform_data)
test_dataset = MNIST(root='./mnist/', train = False, download = True, transform = transform_data)
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size = batch_size, shuffle = True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size = batch_size, shuffle = False)
```

```python
# Try the dataloader
for batch_number, batch in enumerate(train_loader):
    inputs, outputs = batch
    print("---")
    print("Batch number: ", batch_number)
    print(inputs.shape)
    print(outputs.shape)
    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 $\boldsymbol{p}$** as **output**, with p between 0 and 1.

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

From binary to multi-class classification

Unfortunately, when we have more than 2 classes, **we can no longer rely on a single output value $\boldsymbol{p}$**.

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

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

- Where each $\boldsymbol{p_i}$ corresponds to the **probability of being of class $\boldsymbol{i}$**.
- This could typically be done by asking for **the final layer to produce $\boldsymbol{n_y = 10}$ 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.

$$\sum_{i}^{9} p_i = 1$$

- 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.

To normalize the outputs produced by the final fully connected layer, we will use the **softmax** operation, which is a special activation function.

- It will force the values of the p_i to fall in the range of [0, 1].
- And force their sum to be equal to 1, that is $\sum_{i}^{9} p_i = 1$.

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 $\boldsymbol{P}$ **is guaranteed to be summing up to 1**, i.e.

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

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

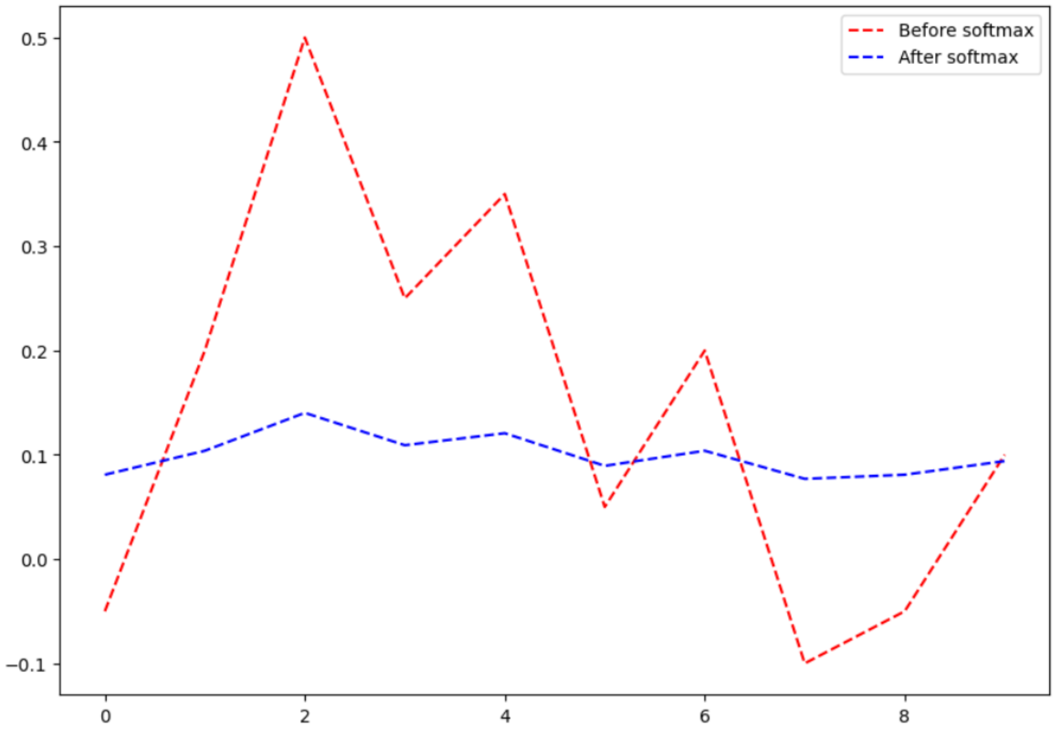
Softmax can be manually implemented as:

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

```
# Ten values that do not sum up to 1
Y = np.array([-1, 4, 10, 5, 7, 1, 4, -2, -1, 2])/20
print(sum(Y))
P = softmax(Y)
print(P)
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 $\boldsymbol{i}$.**

The **predicted class** $\boldsymbol{pred}$ will be the index i of the highest probability value p_i, 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 ($= 28 \times 28$).

```python
class ShallowNeuralNet(torch.nn.Module):
    def __init__(self, n_x, n_h, n_y):
        super().__init__()
        # Define two layers using the nn.Linear()
        self.fc1 = torch.nn.Linear(n_x, n_h)
        self.fc2 = torch.nn.Linear(n_h, n_y)

    def forward(self, x):
        # Flatten images (transform them from 28x28 2D
        # matrices to 784 1D vectors)
        x = x.view(x.size(0), -1)
        # First Wx + b operation
        out1 = self.fc1(x)
        # Using ReLU operation as activation after first layer
        act1 = torch.relu(out1)
        # Second Wx + b operation anbd return
        out2 = self.fc2(act1)
        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.

```python
# Initialize model
model = ShallowNeuralNet(n_x = 784, \
                         n_h = 128, \
                         n_y = 10).to(device)

# Get a single sample
sample = next(iter(train_loader))
data, target = sample
print(data.shape)
print(target.shape)
data1 = data[0].to(device)
target1 = target[0].to(device)
print(data1.shape)
print(target1)

# Forward pass
out2 = model(data1)
act2 =  torch.nn.functional.softmax(out2, dim = 1)
print(out2)
print(act2)
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 (no softmax)** in the variable **out2**.
- Apply the **softmax** operation on **out2**. PyTorch offers a functional implmentation of the softmax:

torch.nn.functional.softmax()

- We can then verify that **softmax will adjust the output of the neural network correctly**.

```python
# Initialize model
model = ShallowNeuralNet(n_x = 784, \
                         n_h = 128, \
                         n_y = 10).to(device)

# Get a single sample
sample = next(iter(train_loader))
data, target = sample
print(data.shape)
print(target.shape)
data1 = data[0].to(device)
target1 = target[0].to(device)
print(data1.shape)
print(target1)

# Forward pass
out2 = model(data1)
act2 =  torch.nn.functional.softmax(out2, dim = 1)
print(out2)
print(act2)
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.

<u>We do not use the softmax operation in the forward.</u>

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

```python
class ShallowNeuralNet(torch.nn.Module):
    def __init__(self, n_x, n_h, n_y):
        super().__init__()
        # Define two layers using the nn.Linear()
        self.fc1 = torch.nn.Linear(n_x, n_h)
        self.fc2 = torch.nn.Linear(n_h, n_y)

    def forward(self, x):
        # Flatten images (transform them from 28x28 2D
        # matrices to 784 1D vectors)
        x = x.view(x.size(0), -1)
        # First Wx + b operation
        out1 = self.fc1(x)
        # Using ReLU operation as activation after first layer
        act1 = torch.relu(out1)
        # Second Wx + b operation anbd return
        out2 = self.fc2(act1)
        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**.

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

- 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:

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

In the formula above, $p_i(x_k)$ denotes the **probability for sample $\boldsymbol{x_k}$ of being of class $\boldsymbol{i}$**. 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**.

...To multi-class cross entropy!

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

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

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 $y_k = 2$, we have:

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

We say that Y_k is the **one-hot vector** for the given scalar value $y_k = 2$.

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.

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

Training our model with Adam GD, as before

```python
# Training model
num_epochs = 5
for epoch in range(num_epochs):
    # Go trough all samples in train dataset
    for i, (images, labels) in enumerate(train_loader):
        # Get from dataloader and send to device
        images = images.to(device)
        labels = labels.to(device)
        # Forward pass
        outputs = model(images)
        # Compute loss
        loss = torch.nn.functional.cross_entropy(outputs, labels)
        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        # Display
        if (i+1) % 300 == 0:
            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 to check its accuracy on test set.

- 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).
- **97% = nicely trained model!**

```python
# Evaluate model accuracy on test after training
# Set model in eval mode!
model.eval()
# Evaluate
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        # Get images and labels from test loader
        images = images.to(device)
        labels = labels.to(device)
        # Forward pass and predict class using max
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        # Check if predicted class matches label
        # and count numbler of correct predictions
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
# Compute final accuracy and display
accuracy = correct/total
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**.

It is now time…

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 40,
- the third layer receive inputs of size 40 and produce outputs of size 20,
- 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.

- 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**.

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

```python
class DenseReLU(torch.nn.Module):
    def __init__(self, n_x, n_y):
        super().__init__()
        # Define Linear layer using the nn.Linear()
        self.fc = torch.nn.Linear(n_x, n_y)

    def forward(self, x):
        # Wx + b operation
        # Using ReLU operation as activation after
        return torch.relu(self.fc(x))
```

```python
class DenseNoReLU(torch.nn.Module):
    def __init__(self, n_x, n_y):
        super().__init__()
        # Define Linear layer using the nn.Linear()
        self.fc = torch.nn.Linear(n_x, n_y)

    def forward(self, x):
        # Wx + b operation
        # No activation function
        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().

```
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

Good practice: This modular approach,

- **defining blocks of layers,**
- and eventually **assembling them in a larger Deep Neural Network network**,

is very common in deep neural networks, especially when the architectures are very heavy and include many layers.

Organize the mess!

```
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

```python
# Training model
num_epochs = 10
for epoch in range(num_epochs):
    # Go trough all samples in train dataset
    for i, (images, labels) in enumerate(train_loader):
        # Get from dataloader and send to device
        images = images.to(device)
        labels = labels.to(device)
        # Forward pass
        outputs = model(images)
        # Compute loss
        loss = torch.nn.functional.cross_entropy(outputs, labels)
        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        # Display
        if (i+1) % 25 == 0:
            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)

```python
# Evaluate model accuracy on test after training
# Set model in eval mode!
model.eval()
# Evaluate
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        # Get images and labels from test loader
        images = images.to(device)
        labels = labels.to(device)
        # Forward pass and predict class using max
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        # Check if predicted class matches label
        # and count numbler of correct predictions
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
# Compute final accuracy and display
accuracy = correct/total
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!)

```python
# Evaluate model accuracy on test after training
# Set model in eval mode!
model.eval()
# Evaluate
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        # Get images and labels from test loader
        images = images.to(device)
        labels = labels.to(device)
        # Forward pass and predict class using max
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        # Check if predicted class matches label
        # and count numbler of correct predictions
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
# Compute final accuracy and display
accuracy = correct/total
print(f'Evaluation after training, test accuracy: {accuracy:.4f}')
```

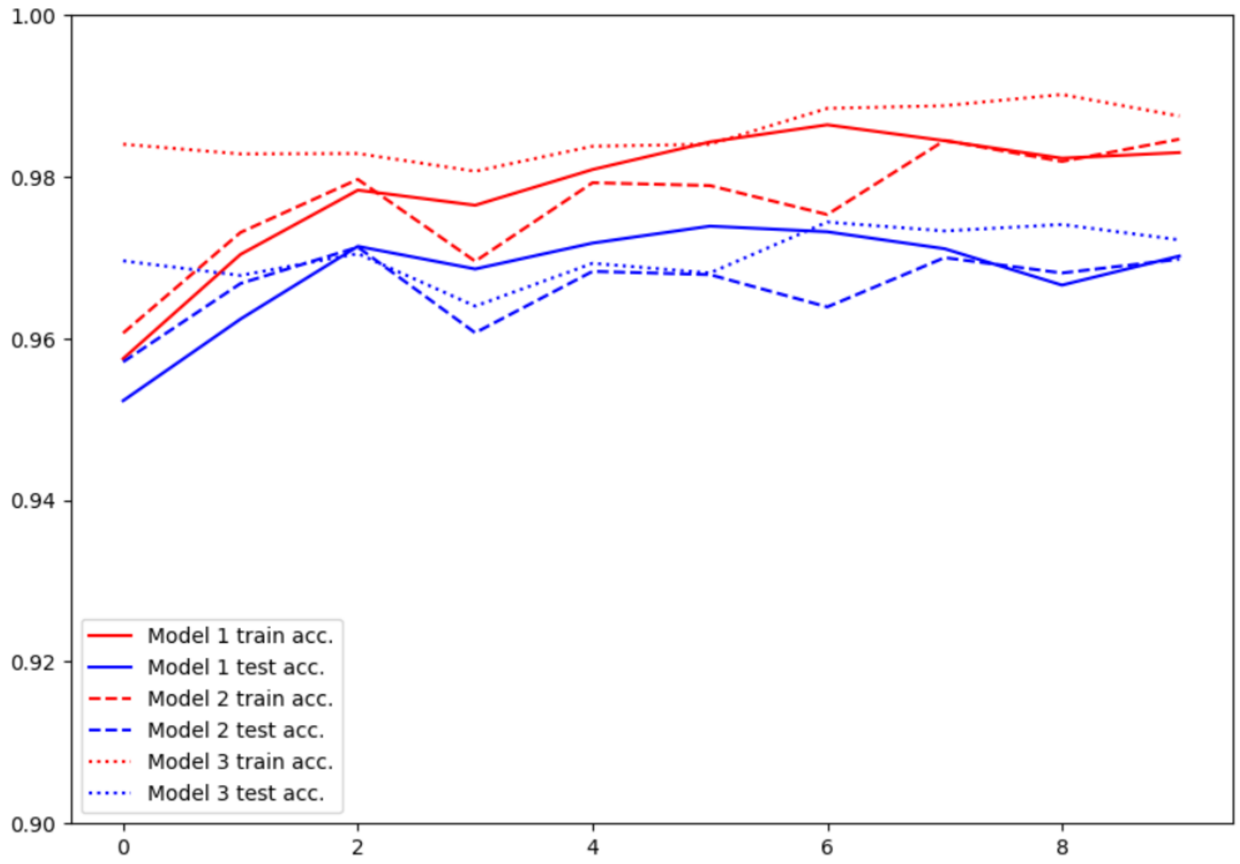
```
Evaluation after training, test accuracy: 0.9367
```

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 compuitational 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

Project announcement!

Let us discuss it now.