Deep Learning in Life Sciences

Lecture 1: Introduction to Deep Learning

Bartek Wilczyński
University of Warsaw
bartek@mimuw.edu.pl
http://regulomics.mimuw.edu.pl

CHARLES UNIVERSITY
SORBONNE UNIVERSITÉ
UNIVERSITÄT HEIDELBERG ZUKUNFT SEIT 1386
UNIVERSITY OF WARSAW
UNIVERSITÀ DEGLI STUDI DI MILANO
4eu+ EUROPEAN UNIVERSITY ALLIANCE

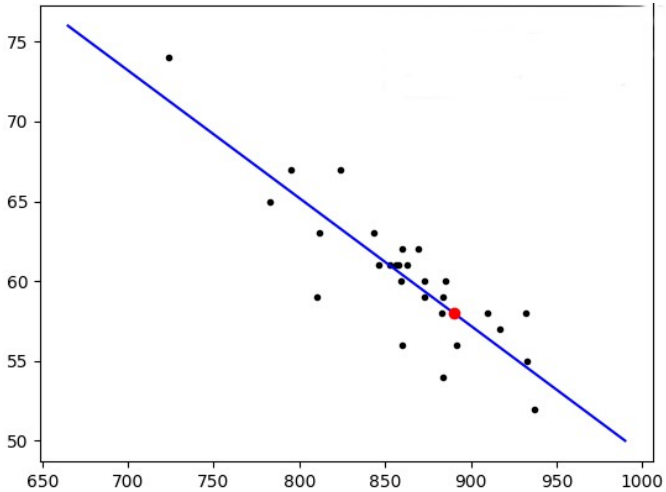
Overview of this lecture

- Basic concepts of ML in 6 slides
  (reminding ourselves of the course prerequisites)
- Background on Artificial Neural Networks
  (why, what, when and how of ANNs)
- Deep learning system basics
  (how are modern ANN models built and trained)
- Short intro into PyTorch architecture
  (what software are we going to use for the course)

Section I – ML basics

ML: Models and their parameters

In machine learning (ML), we start from a **set of data points** sampled from a **larger population** and our goal is to choose the values of **model parameters**, so that the model is **optimal with respect to the data** and **usable for new data** in the larger population



In a task of basic linear regression our model parameters are slope and intercept of a line and our optimization criteria is the mean square error

ML: Supervised vs unsupervised learning

Machine learning can be used in a **supervised** setting, where our model is typically a function

$$f(x,\Theta) \rightarrow Y$$

Our data are a collection of $<x_i, y_i>$ paired observations, where x_i is a sample input and y_i is a corresponding output value. We are working with a certain function f that depends on parameters Θ that are chosen to minimize the difference between $f(x_i,\Theta)$ and y_i.

Regression and **classification** are typical examples of supervised learning.

In the **unsupervised** setting, we are usually only presented with observations $<x_i>$, without knowing the desired outputs $<y_i>$. We are still usually trying to find some function

$$f(x,\Theta) \rightarrow Y$$

that is transforms our observations in some desirable way. Typical cases of unsupervised learning are **clustering** or **dimensionality reduction.** The model optimization criteria needs to be based on some external measure.

ML: Loss function and model optimization

In order to find the best model, we need an optimization criteria, usually called **loss function**, that takes all (or some) of the data and a model with parameters and gives us a score

$$L(X,[Y],f,\Theta) \rightarrow R$$

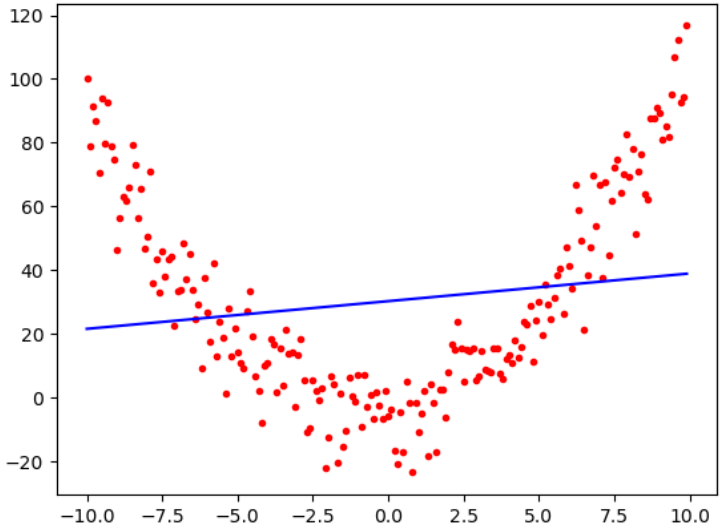
As the smaller the score, the better the model, we will need to use some sort of optimization method to find the model parameters, that minimize the loss function

$$\text{argmin}_{\Theta} L(X,[Y],f,\Theta) \ .$$

This process is usually parametrized to some level, but the parameters of the optimization process are called hyper-parameters and are not part of Θ.

ML: Model capacity vs underfitting

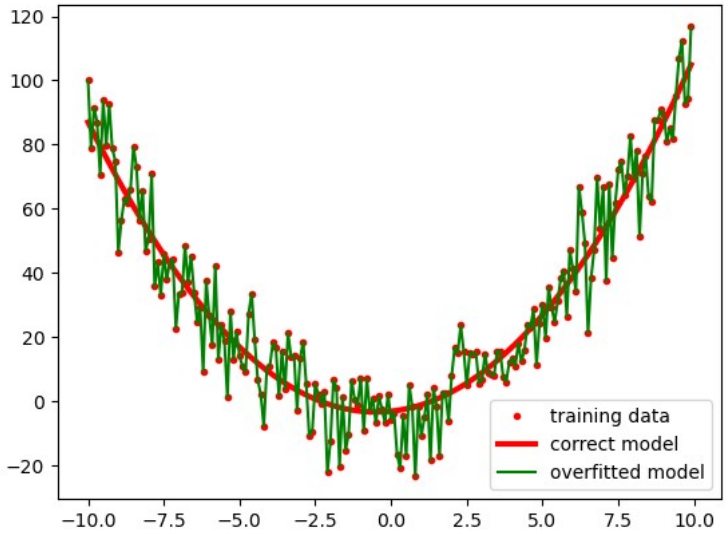
- Different ML models can differ in their complexity (usually correlated to the number of parameters) or as we sometimes say **model capacity**
- If we choose a model with too few parameters, it cannot describe the data well, and this leads to **under-fitting,** the situation where even the optimal model has large values of the loss function



In a task of basic linear regression, trying to fit a linear model to a data generated from a quadratic function, we can see a case of under-fitting, the model is not flexible enough – it lacks capacity to describe the data well

ML: Generalization vs overfitting

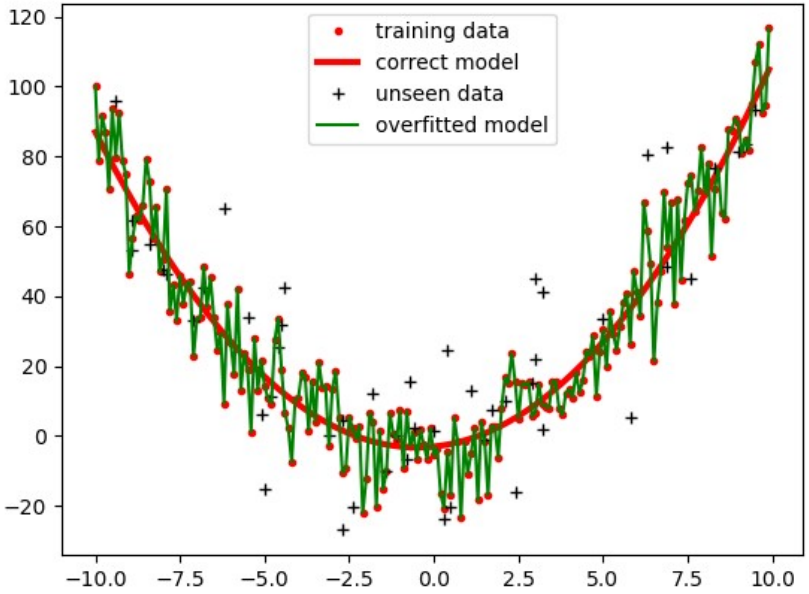
- Increasing model complexity, we can achieve a better fit (with lower loss value), but matching the data perfectly may lead to "**overfitting**"
- Typically we expect some noise in the data and we would like our model to **generalize,** i.e. perform well on new data that we did not use for training
- However, in order to be able to judge if we are indeed over-fitting or not, we would ideally need to know the actual function we are trying to model...



If we allow for more complex models than the straight line, we can see that we can get the actual model of the quadratic equation, but if we increase the number of parameters further, we can fit every bit of noise in the data

ML: Training, Validation and Testing datasets

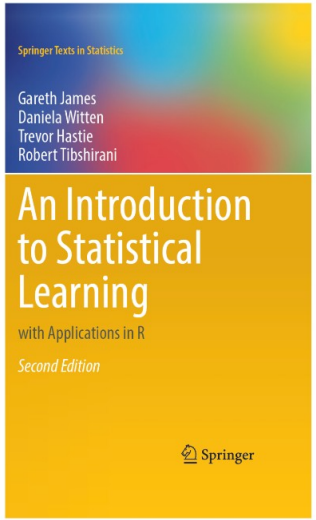
- We can artificially divide our dataset into **training** and **testing** subsets
- We will use only the training data for model optimization and we can see that the model is overfitted, when the loss score is much better on the training set than on the test sets
- If we are also tuning model hyperparameters, we can use an additional **validation** subset of observations. We then train on the training data, tune hyperparameters based on the validation data, and report the metrics on the testing dataset

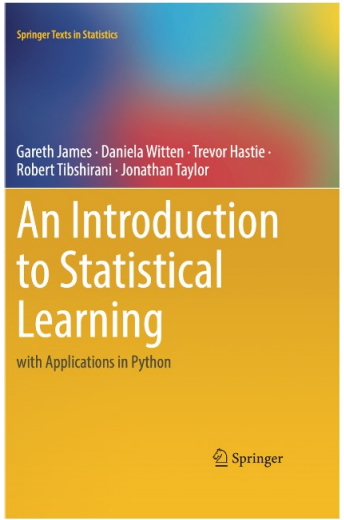


When we add some data that was not used in training, we can see that indeed they are similarly distributed around the “correct” model, but not nearly as close to the overfitted model as the training data

More info on modern ML techniques

- If any of the topics we discussed so far are not clear to you, you should familiarize yourself with these ideas ASAP.
- One way is the book on the right, available for free at: http://statlearning.com
- One of the versions is in python and in addition to a free PDF, there are many code samples on the website accompanying the book

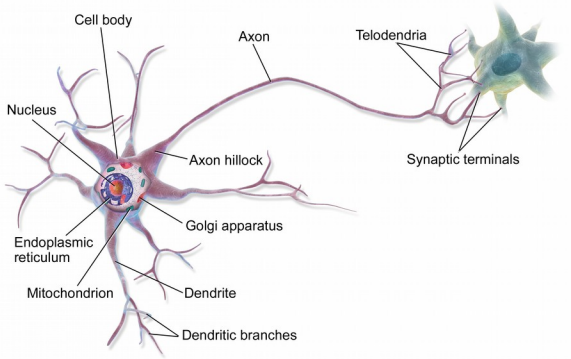


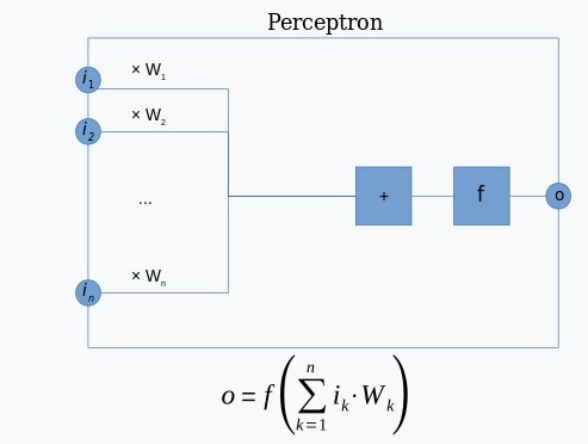


Section II – ANN backstory

ANN backstory: McCulloch and Pitts Perceptron (1943)

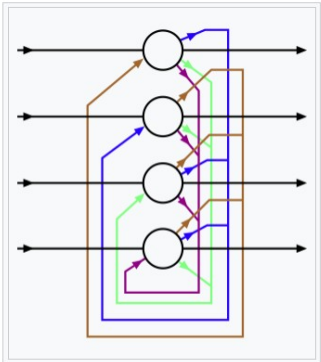
- At the very beginning of research on computing machines, people had ideas of following the architecture of the biological neuron
- A computing machine would be built from a lot of artificial neurons (called perceptrons by McCulloch and Pitts)
- Each perceptron would aggregate many signals from other neurons, by computing a weighted sum of the inputs (weights w_i are parts of the Θ, as well as its bias b)
- Then apply a monotonic activation function (originally a step function) f to this sum, and calculate its ouput that is then connected as input to other perceptrons
- Perceptron machines were built in 1950's by Rosenblatt and were able to perform some computational tasks, however, their limitations, i.e. their inability to handle non-monotonic functions, such as XOR, made them at the time inferior to computers with Von Neuman architecture



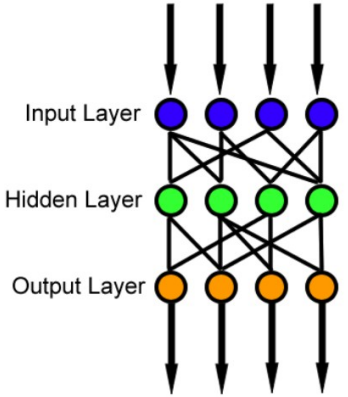


ANN backstory – 1980's: Hopfield nets and Back-propagation in Feed-forward networks

- After a decade of lowered interest in neural networks (1970's – the first AI "winter") two ideas brought the neural networks back into the forefront of the computing research
- **Hopfield networks** were shown to converge to non-linear functions by John Hopfield
- Multi-layered feed forward networks were shown not only to be able to represent non-linear functions, but they were relatively easily trainable using the **back-propagation algorithm**
- The **back-propagation algorithm** is simply an application of a chain-rule to compute the gradient of the loss function with respect to all the weights in the network. Then, we can use **gradient descent** methods to identify how to change the weight values to lower the Loss function.
- Even though these were essentially all the necessary theoretical tools for modern ANNs, the computing infrastructure was deeply insufficient for effective ANN learning, so the field went back into the "second AI winter" of the 1990's



A Hopfield net with four nodes



ANN backstory: image recognition and modern deep learning breakthrough

- In the late 1990s, the availability of the large training datasets and vast improvements to the computing infrastructure allowed for a *renaissance* of Neural Network research
- In the 1990s pioneering work of Yan Le Cun on image processing using convolutional neural networks lead to the development of working ANNs for handwritten letter recognition in 1998 (LeNet5)
- Around 2006, availability of GPU computing platform such as CUDA provided a leap in ANN learning performance
- In 2012, improved CNN model AlexNet developed by Knizhevsky et al outperformed all other approaches by 10% in the ImageNet image classification challlenge. AlexNet, among other improvements pioneered usage of the simplified **RelU activation unit**
- In 2014 recurrent neural network seq2seq was succesfully used for machine translation
- Since 2017, transformer networks were used to the language processing leading to the current explosive growth of Large Language Models such as ChatGPT

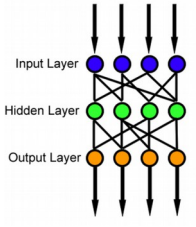
Section III – Modern deep learning

Overview of section III

- Multi-layered feed-forward networks and back-propagation
- Types of activation units: Sigmoid, step function and RelU
- Different optimization methods: SGD, momentum and ADAM
- Dropout as an example of regularization in ANNs

Multi-layered (deep) feed-forward neural networks

- Technically, any network consisting of at least three layers (including input and output), i.e. **containing at least 2 stacked layers of perceptrons** can be considered a “deep” neural network
- In such networks, the middle, **hidden layers**, are taking the outputs of the previous layers as inputs.
- Today, typical model (such as ImageNet → fig.) can have dozens or even hundreds of such layers
- If we allow connections between all neurons in consecutive layers, we call them **fully connected layers**
- The output of these network can be computed very quickly by feeding forward the values from inputs through all layers to the output.
- Current CPU/GPU hardware can do it very fast, especially if the activation function is as simple as ReLU



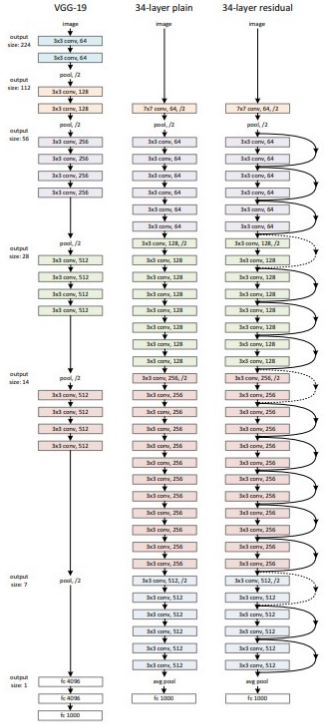


Figure 3. Example network architectures for ImageNet. **Left**: the VGG-19 model [41] (19.6 billion FLOPs) as a reference. **Middle**: a plain network with 34 parameter layers (3.6 billion FLOPs). **Right**: a residual network with 34 parameter layers (3.6 billion FLOPs). The dotted shortcuts increase dimensions. **Table 1** shows more details and other variants.

Backpropagation algorithm

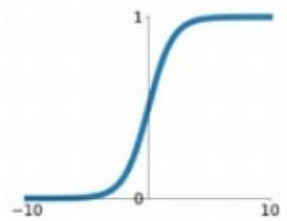
- Assuming that we have a set of training examples, where both inputs $X=\{x_i\}$ and outputs $Y=\{y_i\}$ are given, for any network computing a function $f(x_i,\Theta)$ we can compute the Loss function $L(f(x_i,\Theta),y_i)$
- The Loss function is basically giving us some well behaved difference between the response our network gives us with the weights Θ and the desired output
- Since our function f is just a nested application of the perceptron activation function, and the Loss function is just applied to f and y, the gradient (a matrix of partial derivatives with respect to all weights and biases) of the loss function can be computed efficiently for any pair $<x_i, y_i>$
- The gradient value gives us for each parameter (weight or bias) the direction it should be changed to make the loss function lower for a given example $<x_i, y_i>$
- In practice, this is done layer by layer, starting from output to the first perceptron layer to make the computation more efficient

Types of activation functions

- Originally proposed activation function for perceptrons was the step function
- At the inception of back-propagation algorithm the most widely used were the sigmoid and tanh functions due to their differentiability
- Currently, the ReLU (rectified linear unit) function is the most popular due to its simplicity and ease of computation – its values (for propagation) are just x or 0 and its derivatives (for backpropagation) are just 0 or 1

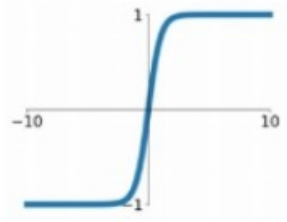
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



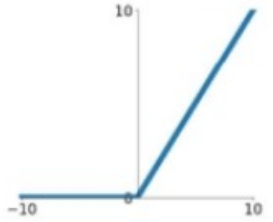
tanh

$$\tanh(x)$$



ReLU

$$\max(0, x)$$



Stochastic gradient descent algorithm

- Now that we have the gradient of the loss function for any observation, we need to find an efficient way to create a learning loop
- The simplest method is to take the loop over all observations (an epoch) and adapt the weights according to the gradient function with some learning rate ($\eta << 1$, e.g. 10^{-5})
- If we do this in random order in each epoch, this is a variant of a method called Stochastic Gradient Descent (SGD)
- Usually we need multiple learning epochs to reach convergence

```
Choose Initial weights and biases Θ
until change in Loss is low enough:
   #new epoch
   For each training pair <x_i, y_i>:
      Calculate the gradient ∇L(f(x_i, Θ), y_i)
      for each parameter w in Θ:
         w = w - η ∇L(f(x_i, Θ), y_i) (w)
```

- For technical reasons (memory size in GPU) usually epochs will be divided further into batches, so that the gradient is averaged over multiple samples that are loaded into GPU all at once

Adaptive optimizers momentum and ADAM

- SGD is applicable practically to any data, but it cannot adapt to the fact that some weights have similar gradients for many observations, while others are oscillating
- The **momentum** algorithm simply adds a rescaled gradient from the previous step to the current gradient – this leads to faster learning of weights with consistent gradients and slower learning for weights with oscillating gradients
- ADAM (ADAptive Moment estimation) takes this idea further, by looking at first and second moments of the gradient and adding two "forgetting" factors β_1 and β_2 to the estimation process.

$$m_w^{(t+1)} \leftarrow \beta_1 m_w^{(t)} + (1-\beta_1)\nabla_w L^{(t)}$$
$$v_w^{(t+1)} \leftarrow \beta_2 v_w^{(t)} + (1-\beta_2)\left(\nabla_w L^{(t)}\right)^2$$

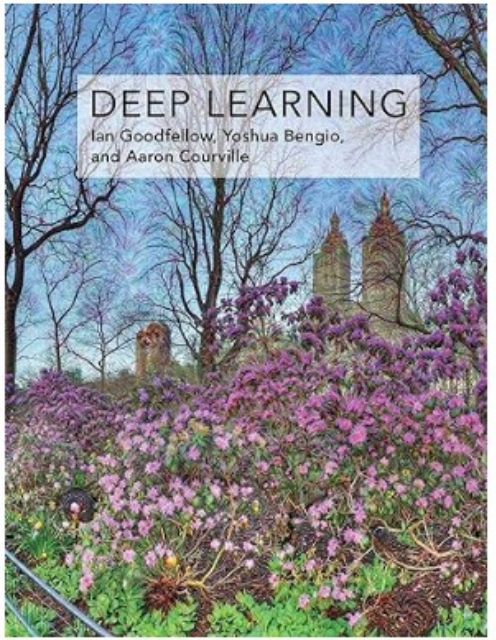
- This method is very effective at differentiating effective learning rates for weights with consistent and inconsistent gradients, while remaining relatively effective

Regularization in deep learning: Dilution&dropout to prevent overfitting

- Typically, deep learning models have very large capacity, and as such are prone to over-fitting,
- In general, as the network weights are parameters in deep learning, we can use many regularization techniques that are used in standard ML (e.g. L_1, L_2)
- However, arguably the most popular regularization techniques for deep learning are based on randomized removal of subsets of weights during training
  - In case of dilution – during each training epoch (or batch) we select a random subset (the size of this subset is a learning parameter) of weights that will not get updated
  - In case of dropout – we select all weights associated with a certain subset of neurons that will not be used at all
- Dropout is a very robust technique that prevents overfitting by requiring the flow of information to be distributed through multiple nodes.
- It slows down the process of training, but usually it is worth to wait longer to have a model with much better generalization

More on modern deep learning theory

- The “Deep learning” book by Goodfellow et al. is very good for getting more details on the theory of deep learning
- You can buy the printed book from MIT Press or bookstores
- There is a free online version at: https://www.deeplearningbook.org/



Section IV – short PyTorch primer

Overview of Section III

- Running PyTorch on your own computer and Google colab
- Representation of data in PyTorch – tensors
- Using dataloaders to load your data
- Defining multilayer ANNs in PyTorch
- Standard train-test loop
- Saving/Loading your models in PyTorch

Installation of PyTorch

- We will be using mostly PyTorch library for training neural networks
  - one of the most popular deep learning libraries,
  - very effective,
  - programmed in python
  - has support in google collab
- If you need to install it on your own computer, there is documentation here: https://pytorch.org/get-started/locally/
- Running it in google colab is documented here: https://pytorch.org/tutorials/beginner/colab.html
- If you install on your local machine, make sure that you can use your GPU for training the networks. If you don't have a compatible GPU, the learning will be much slower

Data representation in PyTorch tensors

- Torch tensors are basically an extension of numpy arrays with additional interfaces for copying them between GPU and CPU
- Tensors are used both to store model parameters (weights and biases) and your training and testing data
- If you have any experience with numpy arrays, you will be able to use most of the functions and methods on tensors
- Torch tensors facilitate fast CPU/GPU operations, so any data used in torch need to be stored in tensors and we control we a tensor is stored by the `device` argument
- Another feature of torch tensors is autograd – an optional argument that needs to be set to True for model parameters, as it allows tensors to track their gradients during training

Datasets & DataLoaders

- Since datasets for deep learning tend to be large, PyTorch has an infrastructure called DataLoaders that can be used for many types of datasets:
  - Image datasets
  - Text datasets
  - Audio datasets
- Many such datasets are available as examples with DataLoaders ready, others can be used after adapting an existing Dataset/dataloader or writing your own classes for that

Sample DataSet

- Bu

```python
import os
import pandas as pd
from torchvision.io import read_image

class CustomImageDataset(Dataset):
    def __init__(self, annotations_file, img_dir, transform=None, target_transform=None):
        self.img_labels = pd.read_csv(annotations_file)
        self.img_dir = img_dir
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return len(self.img_labels)

    def __getitem__(self, idx):
        img_path = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])
        image = read_image(img_path)
        label = self.img_labels.iloc[idx, 1]
        if self.transform:
            image = self.transform(image)
        if self.target_transform:
            label = self.target_transform(label)
        return image, label
```

DataLoader to get your training/testing datasets

- Bu

```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)
```

CLASS torch.utils.data.DataLoader(*dataset, batch_size=1, shuffle=None, sampler=None, batch_sampler=None, num_workers=0, collate_fn=None, pin_memory=False, drop_last=False, timeout=0, worker_init_fn=None, multiprocessing_context=None, generator=None, *, prefetch_factor=None, persistent_workers=False, pin_memory_device=''*) [SOURCE]

Data loader combines a dataset and a sampler, and provides an iterable over the given dataset.

The `DataLoader` supports both map-style and iterable-style datasets with single- or multi-process loading, customizing loading order and optional automatic batching (collation) and memory pinning.

See `torch.utils.data` documentation page for more details.

PyTorch multilayer network definition

- We define ANN models by defining layers and the way they are connected in PyTorch
- We can use classes like nn.sequential or nn.ReLU to build networks with typical architectures quickly

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

We create an instance of `NeuralNetwork`, and move it to the `device`, and print its structure.

```
model = NeuralNetwork().to(device)
print(model)
```

Out:

```
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

Setting the optimization parameters

- In order to start the training process, we need to select:
  - The loss function (there are many available, like nn.MSELoss, nn.NLLLoss, etc.)
  - The optimization algorithm, called the Optimizer in PyTorch (again, many are available, like SGD, ADAM, etc.)
  - Parameters for both, if applicable, as well as some more global parameters like batch size, etc.
- Once we have selected all that, we can set out to write the main loops for training/testing our model

The main training loop

```python
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    # Set the model to training mode - important for batch normalization and dropout layers
    # Unnecessary in this situation but added for best practices
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * batch_size + len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

The tesating loop

```
def test_loop(dataloader, model, loss_fn):
    # Set the model to evaluation mode - important for batch normalization and dropout layers
    # Unnecessary in this situation but added for best practices
    model.eval()
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    # Evaluating the model with torch.no_grad() ensures that no gradients are computed during
test mode
    # also serves to reduce unnecessary gradient computations and memory usage for tensors with
requires_grad=True
    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n")
```

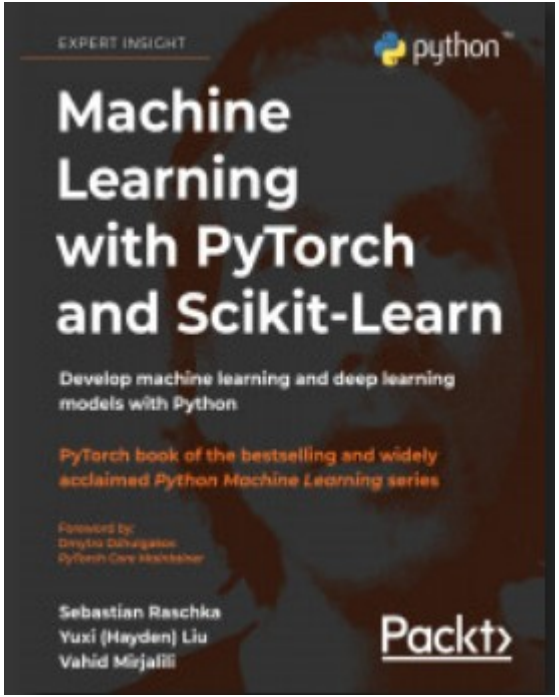
Saving/Loading your model params

- Naturally, once you have trained your model, you may want to keep it for future use
- PyTorch provides convenience functions for you there:
  - Torch.save – saves your model to a file
  - Torch.load - loads the weights from a file.
- Typically these are used in connection with model.state_dict() and model.load_state_dict() as in the examples below:

```
model = models.vgg16(weights='IMAGENET1K_V1')
torch.save(model.state_dict(), 'model_weights.pth')
-----------------------------------------------------
model = models.vgg16() # we do not specify ``weights``, i.e. create untrained model
model.load_state_dict(torch.load('model_weights.pth'))
```

More worked examples of using PyTorch and Scikit-Learn

- If you want to read more about the python libraries we will be using in this lecture, the most natural way is to start from the project websites, both of which have large and well written documentation and many tutorials:
  - https://scikit-learn.org/
  - https://pytorch.org/
- Another way is to read a book by Raschka, Liu and Mirjalili →



Saving/Loading your model params

- Naturally, once you have trained your model, you may want to keep it for future use
- PyTorch provides convenience functions for you there:
  - Torch.save – saves your model to a file
  - Torch.load - loads the weights from a file.
- Typically these are used in connection with model.state_dict() and model.load_state_dict() as in the examples below:

```
model = models.vgg16(weights='IMAGENET1K_V1')
torch.save(model.state_dict(), 'model_weights.pth')
-----------------------------------------------------
model = models.vgg16() # we do not specify ``weights``, i.e. create untrained model
model.load_state_dict(torch.load('model_weights.pth'))
```