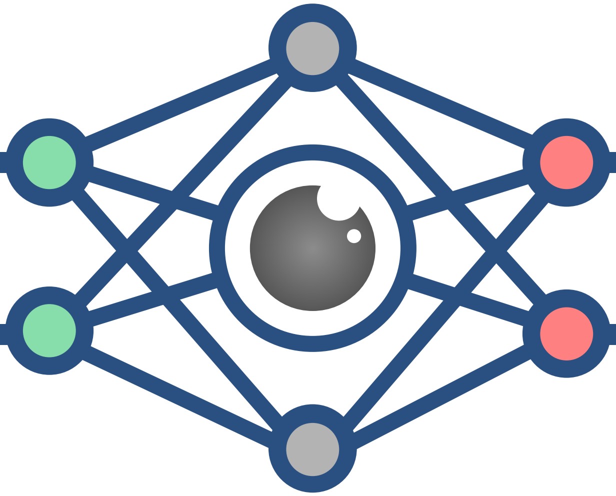
CS3485

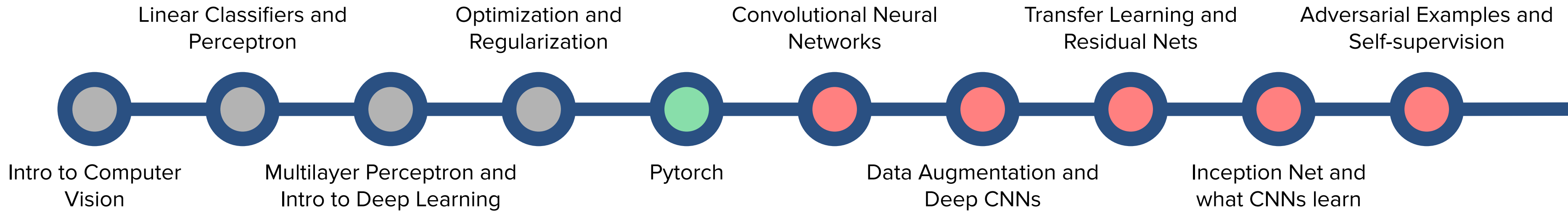
Deep Learning for Computer Vision



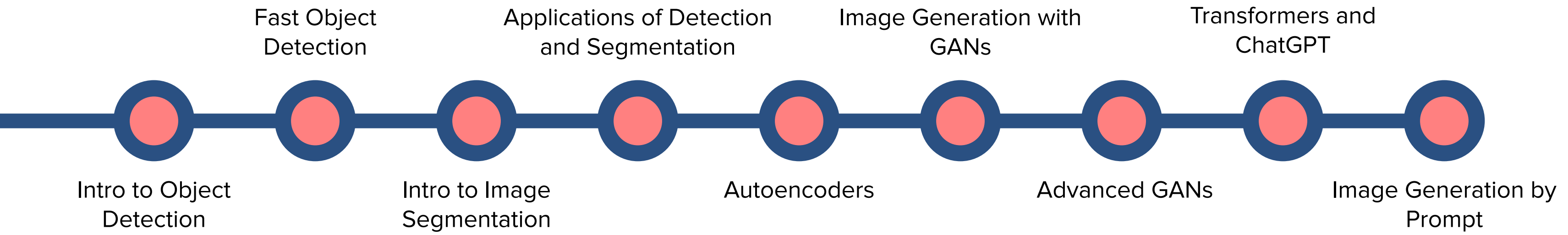
Lec 5: PyTorch

(Tentative) Lecture Roadmap

Basics of Deep Learning


Intro to Computer Vision
Linear Classifiers and Perceptron
Multilayer Perceptron and Intro to Deep Learning
Optimization and Regularization
Pytorch
Convolutional Neural Networks
Data Augmentation and Deep CNNs
Transfer Learning and Residual Nets
Inception Net and what CNNs learn
Adversarial Examples and Self-supervision

Computer Vision Tasks


Intro to Object Detection
Fast Object Detection
Intro to Image Segmentation
Applications of Detection and Segmentation
Autoencoders
Image Generation with GANs
Advanced GANs
Transformers and ChatGPT
Image Generation by Prompt

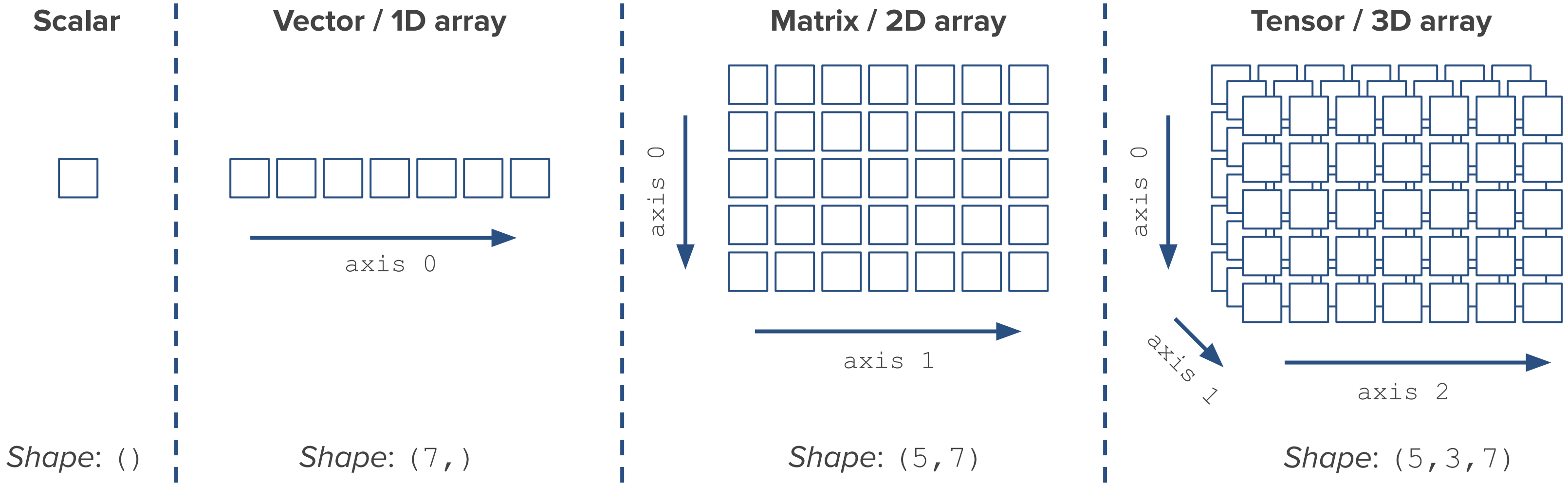
PyTorch

- After we learned all this previous theory on Deep Learning, it is finally time to implement it and solve real problem.
- To that goal, we'll use a Python library called **PyTorch**, which provides many more features, and it is much more optimized for Deep Learning development than Scikit-learn, which we used previously.
- Created in 2016 by Facebook, PyTorch has become the de facto library for DL in many industries and most of the Artificial Intelligence research is done with it nowadays.

PyTorch

Tensors

- The main data structure used in PyTorch is a **tensor**, which is a generalization of vectors and matrices:



- We can create tensors / arrays of more dimensions (4, 5, ...) following the same principle.

Initializing a tensor

- We initialize a tensor by calling `torch.tensor()` on a list of numerical elements:

```
import torch
x = torch.tensor([[1,2]])
y = torch.tensor([[1],[2]])
```

- Just like in Numpy, we can access the tensors’ shapes and data types:

```
print(x.shape, y.shape)
print(x.dtype)
```

```
torch.Size([1,2]) torch.Size([2,1])
torch.int64
```

- The data type of all elements within a tensor **is the same**! If a tensor contains data of different data types, entire tensor is coerced to the most generic data type: `float`.

```
x = torch.tensor([False, 1, 2.0])
print(x)
```

```
tensor([0., 1., 2.])
```

Initializing a tensor

- Just like Numpy and usually with the same command names, we can initialize tensors with built-in functions. For example, in the following example different tensors of size 3×4 are created using these functions:

```
t1 = torch.zeros((3, 4))      # tensor of zeros
t2 = torch.ones((3, 4))       # tensor of ones
t3 = torch.randint(low=0, high=10, size=(3,4)) # tensor of random integers between 0 and 10
t4 = torch.rand(3, 4)         # tensor of random floats 0 and 1
t5 = torch.randn((3,4))       # tensor of random floats normally distributed
```

- Finally, one can convert a Numpy array into a Pytorch tensor and vice-versa:

```
x = np.array([[10,20,30],[2,3,4]])
y = torch.tensor(x)
z = y.numpy()
print(type(x), type(y), type(z))
```

```
<class 'numpy.ndarray'> <class 'torch.Tensor'> <class 'numpy.ndarray'>
```

Operations in tensors

- There are many useful operations we can do with tensors, most of them similar to how Numpy works:
  - Addition and multiplication by a scalar:

```
x = torch.tensor([[1,2,3,4],
                  [5,6,7,8]])
print(x * 10)
print(x.add(10)) # here, x is changed "in-place"
```

```
tensor([[10, 20, 30, 40],
        [50, 60, 70, 80]])
tensor([[11, 12, 13, 14],
        [15, 16, 17, 18]])
```

  - Matrix transposition and multiplication (example below uses `x` from above):

```
print(torch.matmul(x, x.T)) # or x @ x.T
```

```
tensor([[ 30,  70],
        [ 70, 174]])
```

  - Indexing and concatenation (example below uses `x` from above):

```
y = torch.tensor([9, 10, 11, 12])
print(torch.cat([x[1, :], y], axis = 0))
```

```
tensor([ 5,  6,  7,  8,  9, 10, 11, 12])
```

Operations in tensors

- Tensor reshaping:

```
y = torch.tensor([[2, 3], [1, 0]])
z = y.view(4,1)  # 4 rows and 1 column
w = y.view(-1,4) # The other dimension is inferred if using "-1"
print(z)
print(w)
```

```
tensor([[2],
        [3],
        [1],
        [0]])
tensor([[2, 3, 1, 0]])
```

- Maximum value and index:

```
x = torch.arange(16).view(4,4)
print(x)
print(x.max()) # Maximum over the whole tensor

vals, indx = x.max(dim=1) # Maximum over each row
print(vals)
print(indx) # We could use "argmax()" to get just the indices
```

```
tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11],
        [12, 13, 14, 15]])
tensor(15)
tensor([ 3,  7, 11, 15])
tensor([3, 3, 3, 3])
```

- Standard mathematical operations: `abs`, `floor`, `sin`, `cos`, `exp`, `mean`, `round`...

Gradients with Autograd

- One of the main operations in PyTorch is to **compute the gradients** of a tensor object.
- It uses a technique called **Automatic Differentiation (Autograd)**, which enables us to do it by evaluating the derivative of a function **specified by a computer program**.
- In PyTorch, the way we to use it starts by specifying that a tensor requires a gradient to be calculated via the parameter `requires_grad`:

```
x = torch.tensor([2., -1.], requires_grad=True)
```

- Say you have the following function of $x = [x_1, x_2]$:

$$f(x_1, x_2) = x_1^2 + x_2^2$$

which can be computed in PyTorch as:

```
f = x.pow(2).sum()
```

Gradients with Autograd

- Now, we know that the gradient of f is $[2x_1, 2x_2]$.
- We get this in PyTorch by first using the (very important) function `backward()`:

```
f.backward()
```

(As the name of it hints at, `backward()` is where the backpropagation in NN happens).

- Now we compute the gradient of f at the point x from the previous slide with `x.grad`:

```
ans = x.grad
print(ans)
```

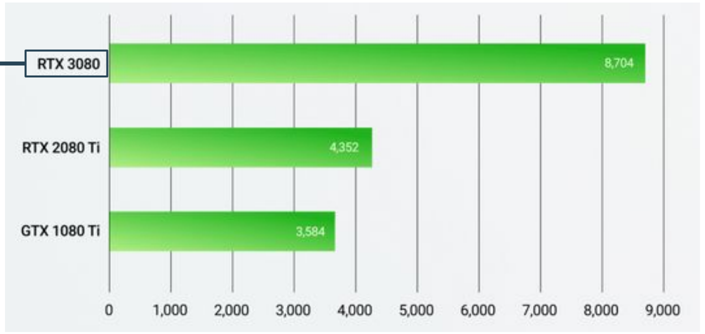
```
tensor([ 4., -2.])
```

- There's one catch with PyTorch autograd: the function you want to compute the gradient of **should return a scalar**. Loss functions fit in that category.

PyTorch's tensors vs NumPy's arrays

- Despite the similarities, PyTorch performs certain mathematical operations more quickly than Numpy.
- This is mainly due to the fact that PyTorch tensor is optimized to work with a **Graphics Processing Unit (GPU)**, instead of a Central Processing Unit (CPU), although they also work in CPUs.
- GPUs make **parallelizable operations** (such as matrix multiplication) much quicker, because of the sheer amount of computational cores it has available (between 700 and 9000).
- A usual CPU (which, in general, have less than 64 cores) would be much slower than a GPU.

Number of cores per GPU model



PyTorch's tensors vs NumPy's arrays

- Let's check that with an experiment. Create random matrices with PyTorch and Numpy:

```
x_t, y_t = torch.rand(1, 6400), torch.rand(6400, 5000)
x_n, y_n = np.random.random((1, 6400)),  np.random.random((6400, 5000))
```

- Then check if **CUDA (a parallel computing platform)** is available to be used.

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'     # If CUDA isn't available, we use the CPU.
```

- We can store our PyTorch tensors in the GPU (if it is available) with `.to(device)`, and in the CPU (with `.cpu()`) and compare their performances with regular Numpy:

```
x, y = x_t.to(device), y_t.to(device)
%timeit z = x@y
```

```
100 loops: 515 µs per loop
```

```
x, y = x_t.cpu(), y_t.cpu()
%timeit z = x@y
```

```
100 loops: 9.04 ms per loop
```

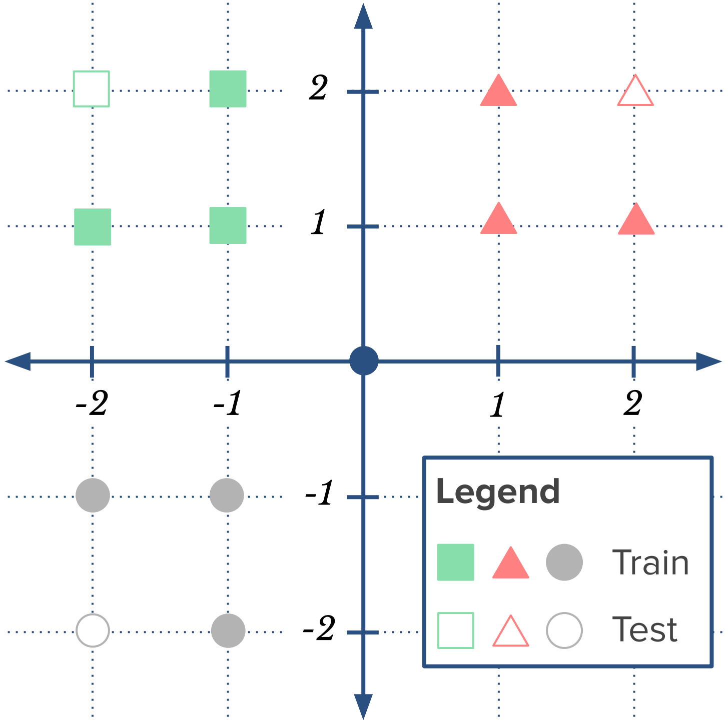
```
%timeit z = np.matmul(x_n,y_n)
```

```
100 loops: 18.8 ms per loop
```

Our First Neural Network: Dataset

- Now we are ready to build and train our first neural network in PyTorch!
- We'll first instantiate the training data on the right with what we saw so far:

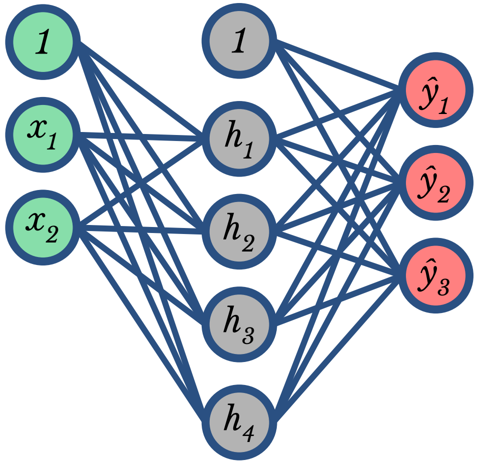
```
x train = [[-2,-1], [-1,-1], [-1,-2],
           [2,-1],  [1,-1],  [1,-2],
           [2,1],   [1,1],   [1,2]]
y train = [[1, 0, 0], [1, 0, 0], [1, 0, 0],
           [0, 1, 0], [0, 1, 0], [0, 1, 0],
           [0, 0, 1], [0, 0, 1], [0, 0, 1]]

X_train = torch.tensor(x_train).float()
Y_train = torch.tensor(y_train).float()

device = 'cuda' if torch.cuda.is_available() else 'cpu'
X train = X train.to(device)
Y_train = Y_train.to(device)
```



Our First Neural Network: Architecture

- Let's define our network. For simplicity, we'd like a network with
  - **One hidden layer** with **four units** (*shown below*),
  - **ReLU activation functions** between the hidden and the output layer.
- In Torch, we have to create a class for our network that inherits `torch`'s nn.Module.
- That class should implement the constructor and `forward()` methods:

```
import torch.nn as nn
class MyNeuralNet(nn.Module):
   def __init__(self):
       super().__init__()
       self.input_to_hidden_layer = nn.Linear(2,4)
       self.hidden_layer_activation = nn.ReLU()
       self.hidden_to_output_layer = nn.Linear(4,3)
   def forward(self, x):
       x = self.input_to_hidden_layer(x)
       x = self.hidden_layer_activation(x)
       x = self.hidden_to_output_layer(x)
       return x
```



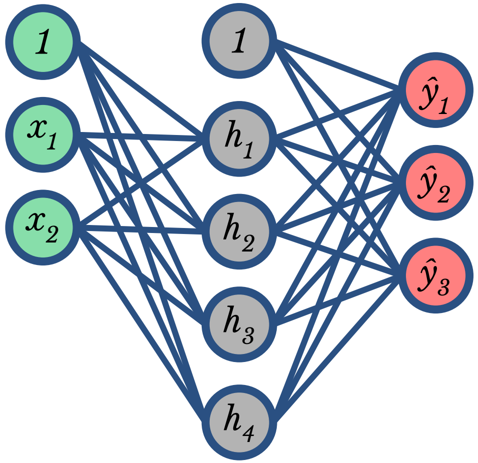
Our First Neural Network: Architecture

- Let’s define our network. For simplicity, we’d like a network with
  - **One hidden layer** with **four units** (*shown below*),
  - **ReLU activation functions** between the hidden and the output layer.
- In Torch, we have to create a class for our network that inherits `torch`’s nn.Module.
- That class should implement the constructor and `forward()` methods:

```
import torch.nn as nn
class MyNeuralNet(nn.Module):
   def __init__(self):
       super().__init__()
       self.input_to_hidden_layer = nn.Linear(2,4)
       self.hidden_layer_activation = nn.ReLU()
       self.hidden_to_output_layer = nn.Linear(4,3)
   def forward(self, x):
       x = self.input_to_hidden_layer(x)
       x = self.hidden_layer_activation(x)
       x = self.hidden_to_output_layer(x)
       return x
```

In the constructor, you should declare the layers and functions you need.

In `forward()`, you explain how the layer would be composed such that to transform the network input `x` into the output in `return`.



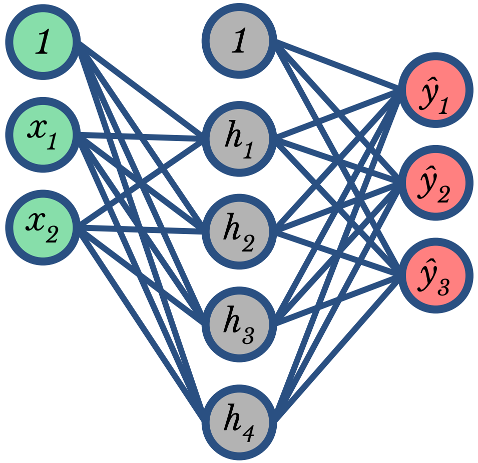
Our First Neural Network: Architecture

- Let's define our network. For simplicity, we'd like a network with
  - **One hidden layer** with **four units** (*shown below*),
  - **ReLU activation functions** between the hidden and the output layer.
- In Torch, we have to create a class for our network that inherits `torch`'s nn.Module.
- That class should implement the constructor and `forward()` methods:

```
import torch.nn as nn
class MyNeuralNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.input_to_hidden_layer = nn.Linear(2,4)
        self.hidden_layer_activation = nn.ReLU()
        self.hidden_to_output_layer = nn.Linear(4,3)
    def forward(self, x):
        x = self.input_to_hidden_layer(x)
        x = self.hidden_layer_activation(x)
        x = self.hidden_to_output_layer(x)
        return x
```

A `Linear` layer is the type of layer that connects all layer inputs to all layer outputs. Notice that you have to specify how many inputs and outputs.

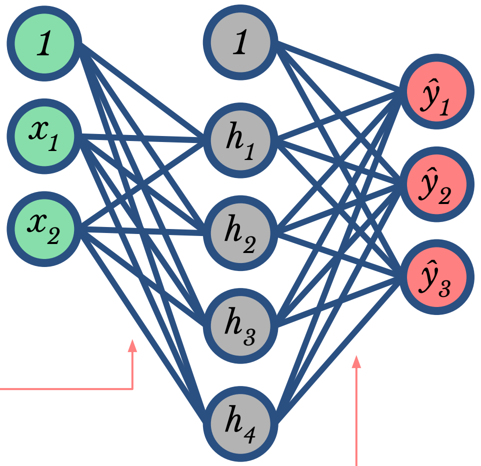
Notice: no softmax!



Our First Neural Network: Architecture

- Let’s define our network. For simplicity, we’d like a network with
  - **One hidden layer** with **four units** (*shown below*),
  - **ReLU activation functions** between the hidden and the output layer.
- In Torch, we have to create a class for our network that inherits `torch`’s nn.Module.
- That class should implement the constructor and `forward()` methods:

```
import torch.nn as nn
class MyNeuralNet(nn.Module):
   def __init__(self):
       super().__init__()
       self.input_to_hidden_layer = nn.Linear(2,4)
       self.hidden_layer_activation = nn.ReLU()
       self.hidden_to_output_layer = nn.Linear(4,3)
   def forward(self, x):
       x = self.input_to_hidden_layer(x)
       x = self.hidden_layer_activation(x)
       x = self.hidden_to_output_layer(x)
       return x
```



Our First Neural Network: Optimizer and Loss

- The next step is to **instantiate a network** of the class `MyNeuralNet`:

```
mynet = MyNeuralNet().to(device)
```

Here we also register the network **weights (which are tensors)** to the device.

- Then we need to **define the loss function** that we optimize for. Since we have three classes, we'll use Cross Entropy, which can be used in PyTorch as:

```
loss_func = nn.CrossEntropyLoss()
```

(again, this loss also computes the softmax operation to its inputs).

- Finally, we define our optimizer. For now, let's use our simplest option: Stochastic Gradient Descent (SGD).

```
from torch.optim import SGD
opt = SGD(mynet.parameters(), lr = 0.001) # "lr" is the learning rate.
```

Our First Neural Network: Training!

- Good! Now, we are ready to train our network on our dataset!
- For now, we will not consider mini-batches, so we'll use all the data to compute one step in gradient descent.
- When training a network in PyTorch, we have to go over 4 main steps in a `for` loop:
  1. **Zero the gradients** saved in the optimizer: PyTorch accumulates them by default.
  2. **Compute the loss** for current set of data: the current data is the whole dataset for now.
  3. **Compute the new gradients**: this operation is done via the AutoGrad's `backward()`.
  4. **Make a gradient descent step**: this operation is done via `opt.step()`.
- Then we repeat it for a given amount of epochs. Here's how it looks like:

```
n_epochs = 1000
for _ in range(n_epochs):
   opt.zero_grad()             # flush the previous epoch's gradients
   loss_value = loss_func(mynet(X_train),Y_train) # compute loss
   loss_value.backward()      # perform back-propagation
   opt.step()                  # update the weights according to the gradients computed
```

Our First Neural Network: Training!

- How well we are doing during training? We can track the loss value over the epochs ...

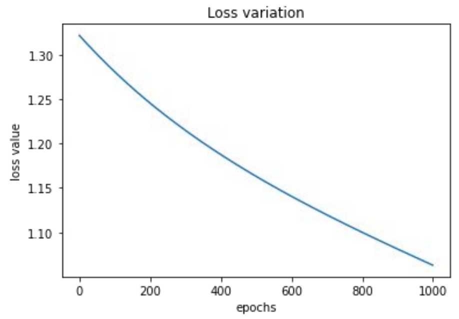
```
n_epochs = 1000
loss_history = []
for   in range(n_epochs):
   opt.zero_grad()
   loss_value = loss_func(mynet(X_train),Y_train)
   loss_value.backward()
   opt.step()

   loss_history.append(loss_value.detach().cpu().numpy())
```

Why is it so complicated? We just want a number! Well, `loss_value` is a tensor on the GPU that can be used to compute gradients. We need to remove all that to get the loss value (a number). So we do:

- `detach()` removes `requires_grad`.
- `cpu()` moves the tensor to the cpu.
- `numpy()` converts the tensor to an array.

... and plot it using Matplotlib:

```
import matplotlib.pyplot as plt
plt.plot(loss_history)
plt.title('Loss variation')
plt.xlabel('epochs')
plt.ylabel('loss value')
```



Our First Neural Network: Checking Parameters

- We can use `mynet.parameters()` to check what weights we've learned after training:

```
for par in mynet.parameters():
    print(par)
```

```
Parameter containing:
tensor([[ 0.0207,  0.6736],
        [-0.6257, -0.1910],
        [ 0.1345,  0.4238],
        [-0.0057, -0.0278]], device='cuda:0', requires_grad=True)
Parameter containing:
tensor([ 0.3481, -0.5513, -0.5184, -0.0614], device='cuda:0',
       requires_grad=True)
Parameter containing:
tensor([[-0.3208, -0.1217,  0.3756, -0.0855],
        [-0.0237, -0.1747, -0.2482, -0.2043],
        [ 0.0442, -0.1720, -0.3428,  0.2704]], device='cuda:0',
       requires_grad=True)
Parameter containing:
tensor([-0.3330, -0.0685, -0.2763], device='cuda:0', requires_grad=True)
```

Our First Neural Network: Checking Parameters

- We can use `mynet.parameters()` to check what weights we've learned after training:

```
for par in mynet.parameters():
   print(par)
```

```
Parameter containing:
tensor([[ 0.0207,  0.6736],
      [-0.6257, -0.1910],
      [ 0.1345,  0.4238],
      [-0.0057, -0.0278]], device='cuda:0', requires_grad=True)
Parameter containing:
tensor([ 0.3481, -0.5513, -0.5184, -0.0614], device='cuda:0',
      requires_grad=True)
Parameter containing:
tensor([[-0.3208, -0.1217,  0.3756, -0.0855],
      [-0.0237, -0.1747, -0.2482, -0.2043],
      [ 0.0442, -0.1720, -0.3428,  0.2704]], device='cuda:0',
      requires_grad=True)
Parameter containing:
tensor([-0.3330, -0.0685, -0.2763], device='cuda:0', requires_grad=True)
```

Weights from the input layer to the hidden layer.

Biases on the input layer.

Weights from the hidden layer to the output layer.

Biases on the hidden layer.

Our First Neural Network: Testing!

- Let’s test how our network performs on the test data. First, let’s get the test data:

```
x_test = [[-2,-2], [2,-2], [2,2]]
y_test = [[1, 0, 0], [0, 1, 0], [0, 0, 1]] # This means that the test labels are [0, 1, 2]

X_test, Y_test = torch.tensor(x_test).float().to(device), torch.tensor(y_test).float().to(device)
```

- Now, we simply need to **feed the test data to the network** and get the predictions:

```
Y_pred = mynet(X_test)
print(Y_pred.cpu().detach().numpy())
print(torch.argmax(Y_pred, dim=1).cpu().numpy())
```

```
[[ 0.69065624  0.26163384  0.29026318]
 [-0.01491237  0.0594516   0.10389088]
 [ 0.13534957  0.03114225  0.16994081]]
[0 2 2]
```

Note that we don’t get the softmax’s probabilities as we never added that layer in. This is okay, since our final predictions are the indices where the max prediction occur*.

- **In this run**, we didn’t get all points correctly classified. How can we improve?

* If you want the softmaxes anyway, you first define the softmax function as `softmax = nn.Softmax()` and the apply it to `Y_pred.`

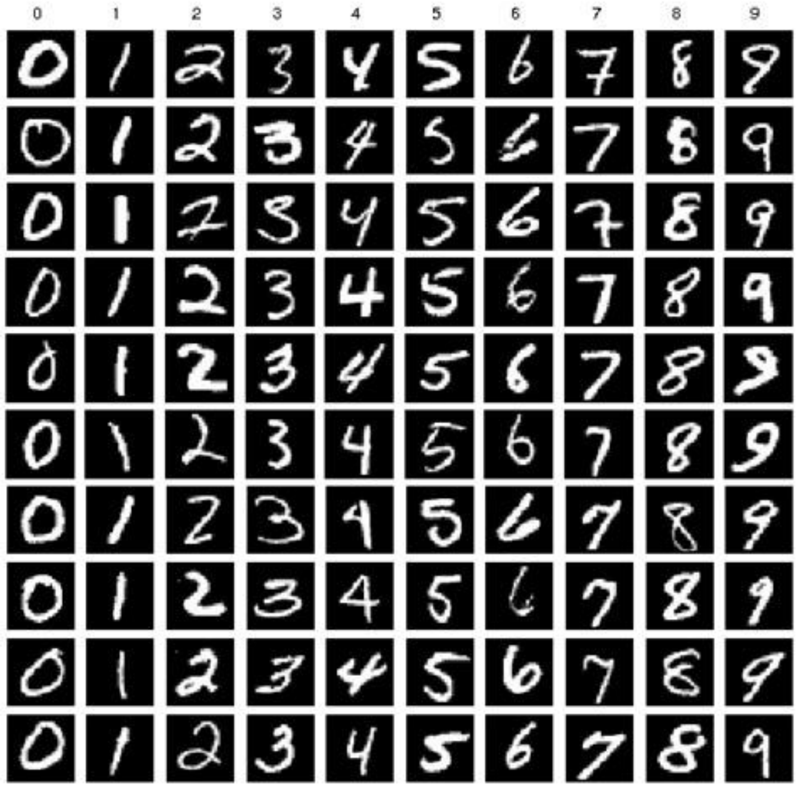
Exercise (*In pairs*)

Click here to open code in Colab

- Change the previous experiment by the following ways:
  - Keeping the same network as before, increase the number of epochs.
  - Keeping the one hidden layers and number of epochs, add more units to it.
  - Keeping the same number of units per layer and number of epochs, increase the number of hidden layers.
- For all these new experiments, make sure to graph the loss variation of epochs. Also, use the library `time.time()` to time the average epoch elapsed time.

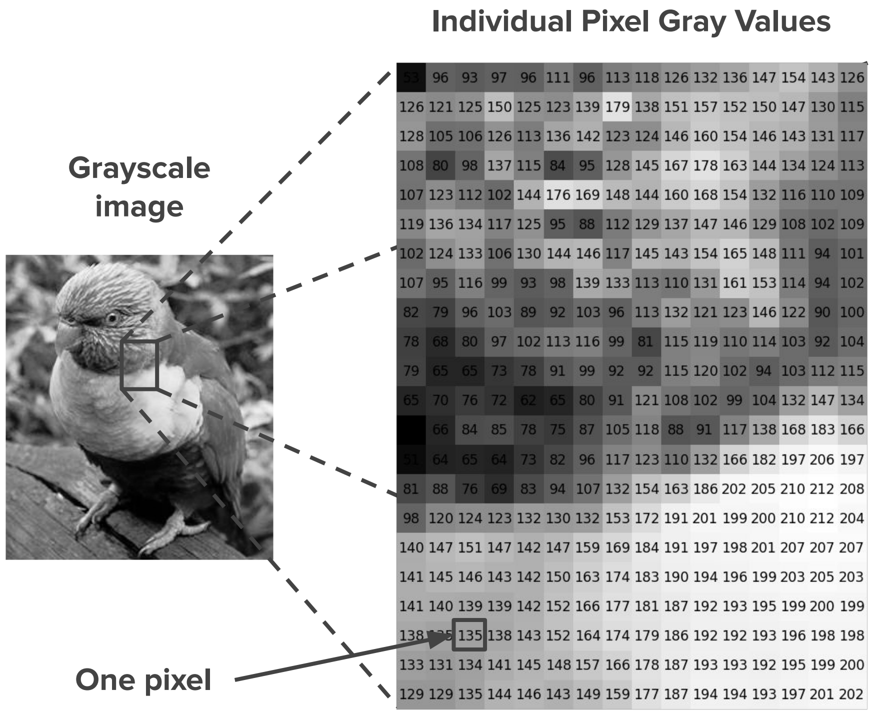
Making more complex networks

- The toy dataset we saw before isn't a good example of realistic data used in practice.
- A more realistic (and more convenient to Computer Vision) dataset in the **MNIST handwritten digits.**
- The database contains $28{\times}28$ grayscale images representing the digits 0 through 9 (some depicted on the right).
- The data is split into two subsets, with 60000 images for training and 10000 images for testing.
- Before we go there, we should ask ourselves: **what is an image**?



What is an image?

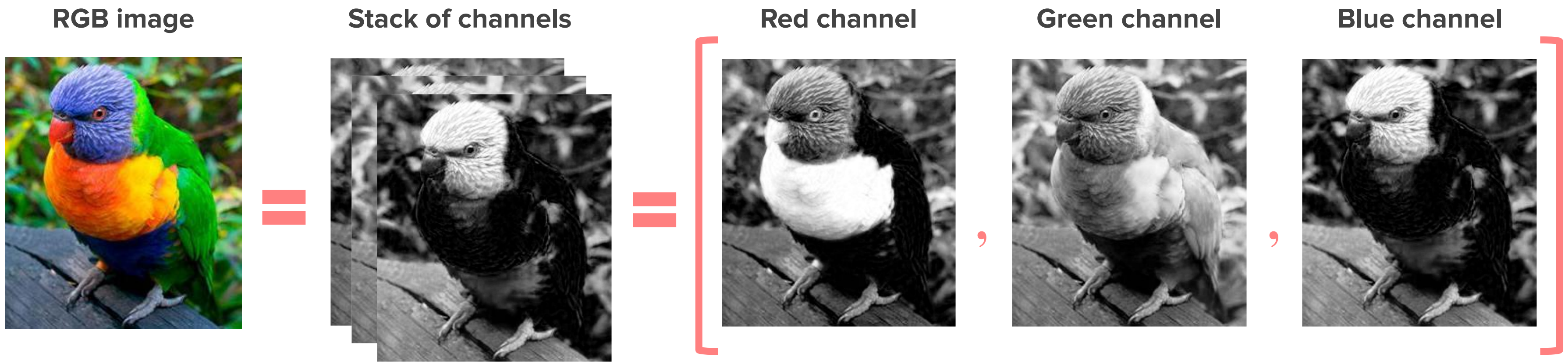
- An **image** as a data structure is simply a matrix (or matrices) of integer numbers ranging from 0 to 255 (one byte of data).
- **Pixels** are the individual subdivisions of an image, each with one sole color.
- A **grayscale image** is an image that only needs one of these matrices to represent its data.
- In a GS image, each of its pixels is colored with a **shade of gray,** whose color ranges from **black** (value 0) to **white** (value 255).



53	96	93	97	96	111	96	113	118	126	132	136	147	154	143	126
126	121	125	150	125	123	139	179	138	151	157	152	150	147	130	115
128	105	106	126	113	136	142	123	124	146	160	154	146	143	131	117
108	80	98	137	115	84	95	128	145	167	178	163	144	134	124	113
107	123	112	102	144	176	169	148	144	160	168	154	132	116	110	109
119	136	134	117	125	95	88	112	129	137	147	146	129	108	102	109
102	124	133	106	130	144	146	117	145	143	154	165	148	111	94	101
107	95	116	99	93	98	139	133	113	110	131	161	153	114	94	102
82	79	96	103	89	92	103	96	113	132	121	123	146	122	90	100
78	68	80	97	102	113	116	99	81	115	119	110	114	103	92	104
79	65	65	73	78	91	99	92	92	115	120	102	94	103	112	115
65	70	76	72	62	65	80	91	121	108	102	99	104	132	147	134
	66	84	85	78	75	87	105	118	88	91	117	138	168	183	166
51	64	65	64	73	82	96	117	123	110	132	166	182	197	206	197
81	88	76	69	83	94	107	132	154	163	186	202	205	210	212	208
98	120	124	123	132	130	132	153	172	191	201	199	200	210	212	204
140	147	151	147	142	147	159	169	184	191	197	198	201	207	207	207
141	145	146	143	142	150	163	174	183	190	194	196	199	203	205	203
141	140	139	139	142	152	166	177	181	187	192	193	195	199	200	199
138	135	$\boxed{135}$	138	143	152	164	174	179	186	192	192	193	196	198	198
133	131	134	141	145	148	157	166	178	187	193	193	192	195	199	200
129	129	135	144	146	143	149	159	177	187	194	194	193	197	201	202

What is an image?

- For colored images, we make each individual pixel contain 3 integers, each in $[0, 255]$.
- These values represent the the intensities of Red, Blue and Green (RGB)* in that pixel, in this order, giving the name for thai kind of image an **RGB image**.
- Each **color channel** (R, G or B) is then represented by a gray scale image and the full image is the stacking of its channels.



* There are other ways to define a colored image that is not RGB (ex.: CYMK = Cyan, Yellow, Magenta and Black).

Loading the data

- With that out of the way, we can now (down)load the MNIST dataset, which PyTorch provides (along with other datasets) in the `torchvision` module.

```python
from torchvision import datasets

data_folder = '~/data/MNIST'
mnist_train = datasets.MNIST(data_folder, download=True, train=True)
X_train, y_train = mnist_train.data, mnist_train.targets
```

- We also create a class for our dataset that inherits from Torch's `Dataset`.
- Note that two pre-processing actions take place on the data:
  - **Rescaling**: it makes all data values be in the range [0, 1].
  - **Reshaping**: turn images into vectors.

```python
from torch.utils.data import Dataset
class MNISTDataset(Dataset):
    def __init__(self, x, y):
        x = x.float()/255     # Data rescaling
        x = x.view(-1,28*28)  # Data reshaping
        self.x, self.y = x, y
    def __len__(self):
        return len(self.x)
    def __getitem__(self, ix):
        x, y = self.x[ix], self.y[ix]
        return x.to(device), y.to(device)
```

Visualizing the data

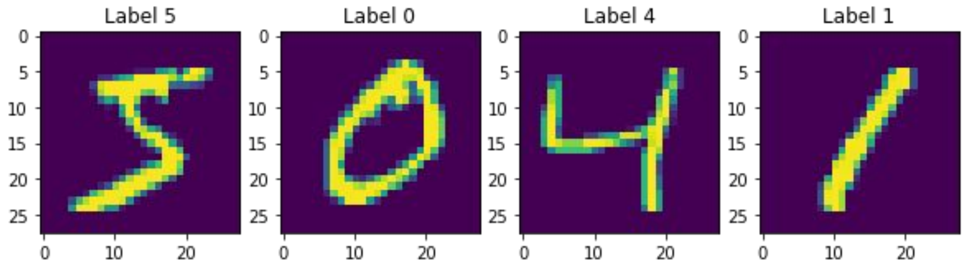
- It is always a good practice to check how the data looks like before working on it.

```
print(x_train.shape)
print(y_train.shape)
```

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

- This is an example of how PyTorch organizes the data dimensions: **`(N,C,H,W)`**
  - First the number of datapoints, (**`N`**); then the number of channels in each point, (**`C`**); then the height (**`H`**) and width of each point (**`W`**).
  - If either dimension (`N`, `C`, `H` or `W`) is just 1, it won't show up.
- We can check the data itself. Here are the first *4* data points and their true labels:

```
import matplotlib.pyplot as plt
plt.figure(figsize=(10,3))
for i in range(4):
    plt.subplot(1,4,i+1)
    plt.imshow(x_train[i])
    plt.title(f"Label {y_train[i]}")
plt.show()
```



Dataloader

- We then instantiate an `MNISTDataset` using the data we just downloaded:

```
train_dataset = MNISTDataset(x_train, y_train)
```

- Now we can use that dataset object in conjunction to what Torch calls a Dataloader:

```
from torch.utils.data import DataLoader
train_dl = DataLoader(train_dataset, batch_size=32, shuffle=True)
```

- In our context, a `DataLoader` object acts like a Python generator that yields a set of datapoints and their corresponding labels (a mini-batch) at a time.
- The number of data points it yields is controlled by `batch_size`.
- The `shuffle` parameter makes the loader shuffle the data once all its batches were yielded, in order to produce new batches for the next time it has to yield data.
- This object will coordinate the usage of mini-batches in our network learning process.

Network, Optimizers and Loss

- As before, we define and instantiate a similar NN, now with 1000 hidden units:

```
class MNISTNeuralNet(nn.Module):
   def __init__(self):
       super().__init__()
       self.input_to_hidden_layer = nn.Linear(28 * 28, 1000) # The input size is the number Of pixels
       self.hidden_layer_activation = nn.ReLU()              # in an individual image.
       self.hidden_to_output_layer = nn.Linear(1000, 10)
   def forward(self, x):
       x = self.input_to_hidden_layer(x)
       x = self.hidden_layer_activation(x)
       x = self.hidden_to_output_layer(x)
       return x
model = MNISTNeuralNet().to(device)
```

- We'll use Cross Entropy again and, now, the ADAM optimizer (with learning rate of 0.01):

```
from torch.optim import Adam
opt = Adam(model.parameters(), lr=1e-2)
loss_func = nn.CrossEntropyLoss()
```

Training the network

- This time, we’ll create a function that performs those four typical in PyTorch training:

```
def train_batch(x, y, model, opt, loss_fn):
   model.train() # We’ll see the meaning of this line later today

   opt.zero_grad()   # Flush memory
   batch_loss = loss_func(model(x), y)   # Compute loss
   batch_loss.backward()   # Compute gradients
   opt.step()   # Make a GD step
   return batch_loss.detach().cpu().numpy() # Removes grad, sends data to cpu, converts tensor to array
```

- We’ll also keep track of the accuracy of our model using the function:

```
@torch.no_grad() # This decorator is used to tell PyTorch that nothing here is used for training
def accuracy(x, y, model):
   model.eval() # We’ll see the meaning of this line later today

   prediction = model(x) # Check model prediction
   argmaxes = prediction.argmax(dim=1) # Compute the predicted labels for the batch
   s = torch.sum((argmaxes == y).float())/len(y) # Compute accuracy
   return s.cpu().numpy()
```

Training the network

- Now let’s train our network!

```python
losses, accuracies, n_epochs = [], [], 5
for epoch in range(n_epochs):
   print(f"Running epoch {epoch + 1} of {n_epochs}")

   epoch_losses, epoch_accuracies = [], []
   for batch in train_dl:
       x, y = batch
       batch_loss = train_batch(x, y, model, opt, loss_func)
       epoch_losses.append(batch_loss)
   epoch_loss = np.mean(epoch_losses)

   for batch in train_dl:
       x, y = batch
       batch_acc = accuracy(x, y, model)
       epoch_accuracies.append(batch_acc)
   epoch_accuracy = np.mean(epoch_accuracies)

   losses.append(epoch_loss)
   accuracies.append(epoch_accuracy)
```

Training the network

- Now let's train our network!

```
losses, accuracies, n_epochs = [], [], 5
for epoch in range(n_epochs):
   print(f"Running epoch {epoch + 1} of {n_epochs}")

   epoch_losses, epoch_accuracies = [], []
   for batch in train_dl:
       x, y = batch
       batch_loss = train_batch(x, y, model, opt, loss_func)
       epoch_losses.append(batch_loss)
   epoch_loss = np.mean(epoch_losses)

   for batch in train_dl:
       x, y = batch
       batch_acc = accuracy(x, y, model)
       epoch_accuracies.append(batch_acc)
   epoch_accuracy = np.mean(epoch_accuracies)

   losses.append(epoch_loss)
   accuracies.append(epoch_accuracy)
```

Runs gradient descent on each batch (of size 32) at a time.

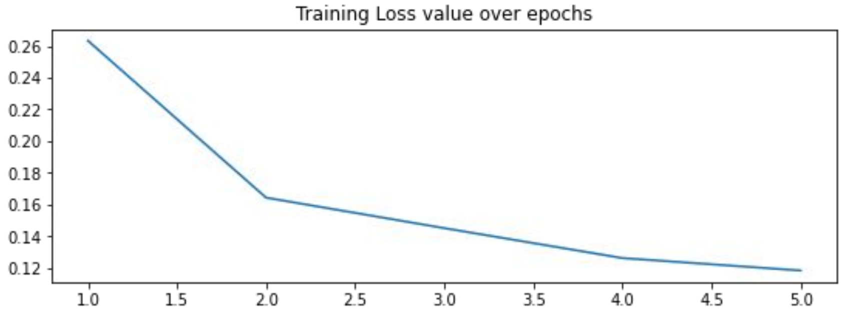
Computes the classification accuracy on the training data also using the same Dataloader

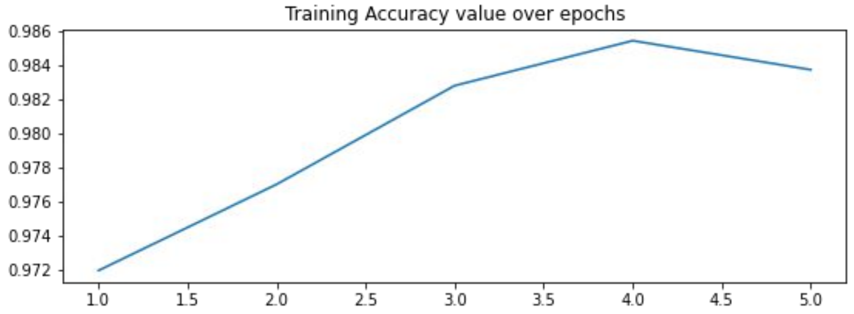
Keeps track of losses and training accuracy.

Plotting the training statistics

- We plot our train and test performance over epochs (*your results can be slightly different*):

```
import matplotlib.pyplot as plt

epochs = np.arange(n_epochs) + 1
plt.figure(figsize=(20,3))
plt.subplot(121)
plt.title('Training Loss value over epochs')
plt.plot(epochs, losses)
plt.subplot(122)
plt.title('Training Accuracy value over epochs')
plt.plot(epochs, accuracies)
```





Testing the model

- Finally we test the learned model on the test data. First we load the data and create a dataset object and a dataloader with it:

```
mnist_test = datasets.MNIST(data_folder, download=True, train=False)
x_test, y_test = mnist_test.data, mnist_test.targets

test_dataset = MNISTDataset(x_test, y_test)
test_dl = DataLoader(test_dataset, batch_size=32, shuffle=True)
```

and then we compute the accuracy of the trained model on that dataset:

```
epoch_accuracies = []
for ix, batch in enumerate(iter(test_dl)):
   x, y = batch
   epoch_accuracies.append(accuracy(x, y, model))

print(f"Test accuracy: {np.mean(epoch_accuracies)}")
```

```
Test accuracy: 0.9637579917907715
```

Improving the Result: Batch-Normalization

- Not bad: 96% of accuracy! But we can try to improve it. One technique used to improve a deep learning model is called **Batch Normalization (BN)**.
- This is similar to the rescaling phase during preprocessing: we don't want the outputs of each layer to have a very large range. Additionally, we also change their mean.
- Basically, if $x_1, x_2, \ldots, x_m$ are the **outputs of layer** for a given data batch, we first compute their mean and standard deviation:

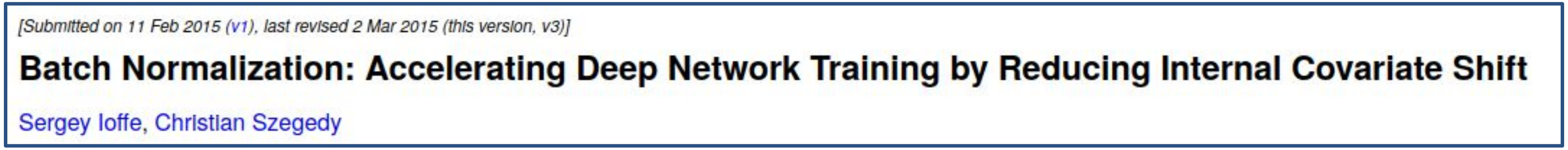
$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i, \ \ \sigma_B = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2}$$

- Then we rescale and shift each output using the following formula using the parameters γ and β, which will become the new standard deviation and mean of the xs, respectively:

$$\tilde{x}_i = \gamma\frac{(x_i - \mu_B)}{\sqrt{\sigma_B^2 + \epsilon}} + \beta$$

Improving the Result: Batch-Normalization

- This simple change was show to accelerate training and improve learning performance:

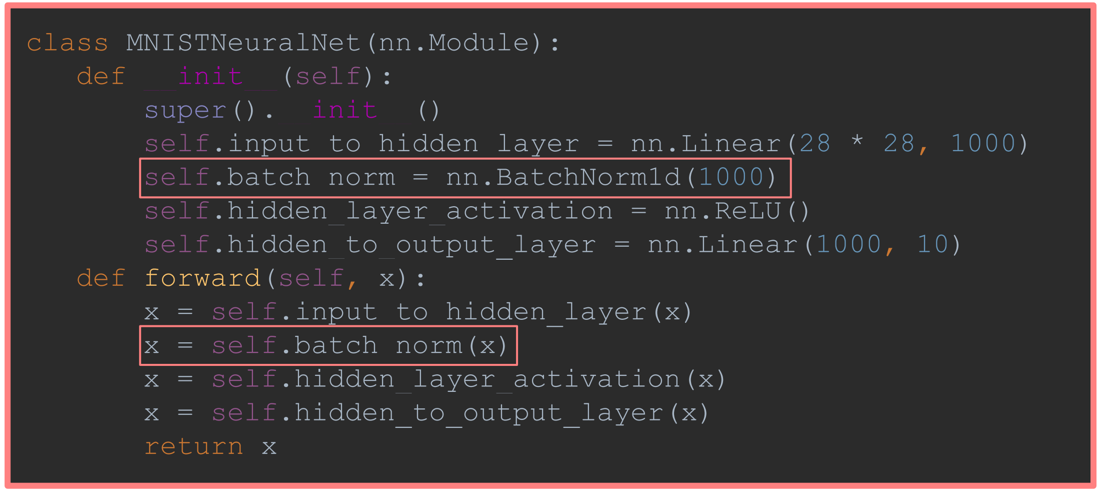


[Submitted on 11 Feb 2015 (v1), last revised 2 Mar 2015 (this version, v3)]

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Sergey Ioffe, Christian Szegedy

- In PyTorch, you can add it step via `nn.BatchNorm1d()` and it acts like a module in the network definition.
- On the right, we change our network to have a BN step right before the activation function of the hidden layer.



```
class MNISTNeuralNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.input_to_hidden_layer = nn.Linear(28 * 28, 1000)
        self.batch_norm = nn.BatchNorm1d(1000)
        self.hidden_layer_activation = nn.ReLU()
        self.hidden_to_output_layer = nn.Linear(1000, 10)
    def forward(self, x):
        x = self.input_to_hidden_layer(x)
        x = self.batch_norm(x)
        x = self.hidden_layer_activation(x)
        x = self.hidden_to_output_layer(x)
        return x
```

Improving the Result: Dropout

- Finally, we add a regularizer via dropout to our network, via the `nn.Dropout()` module:

```
class MNISTNeuralNet(nn.Module):
   def __init__(self):
       super().__init__()
       self.input_to_hidden_layer = nn.Linear(28 * 28, 1000)
       self.batch_norm = nn.BatchNorm1d(1000)
       self.hidden_layer_activation = nn.ReLU()
       self.dropout = nn.Dropout(0.25)
       self.hidden_to_output_layer = nn.Linear(1000, 10)
   def forward(self, x):
       x = self.input_to_hidden_layer(x)
       x = self.batch_norm(x)
       x = self.hidden_layer_activation(x)
       x = self.dropout(x)
       x = self.hidden_to_output_layer(x)
       return x
```

In this case, we say the the weights in the following layer (`hidden_to_output_layer`) will have 25% of its units turned off randomly.

Model evaluation and training

- Having seen BN and Dropout modules, we are ready to understand what each of why we have `model.train()` and `model.eval()` in the following functions:

```
def train_batch(x, y, model, opt, loss_fn):
   model.train()
   (...)
```

```
def accuracy(x, y, model):
   model.eval()
   (...)
```

- They tell PyTorch to switch from “learning mode” to “evaluation mode”.
- This helps inform modules such as Dropout and BN, which are designed to behave differently during training and evaluation.
- For instance, in training mode, BN updates the mean of each new batch, whereas, for evaluation/testing mode, these updates do not happen.
- You can call either `model.eval()` or `model.train(mode=False)` to tell PyTorch that you are testing.

Exercise (*In pairs*)

Click here to open code in Colab

- Run the previous results on the MNIST dataset using the dropout and BN modules as shown previously, keeping all the previous parameters (such as number of epochs) the same. Keep track of learning time, final training accuracy and test accuracy.
- Now, remove the BN module and focus on dropout. Change p to 0.1 and then to 0.9 and note the change in train/test accuracy and learning time.

Extra Material:

- There is this very good tutorial called learnpytorch.io that helps you learn PyTorch through examples. Pretty good stuff in there.
- Tensorflow (PyTorch's competitor) has this nice website where you can train different networks in different datasets, under different parameters. It is really worth play with it