CS3485

Deep Learning for Computer Vision

Lec 5: Pytorch I – MLPs

Announcements

- Quiz 1:
  - Just graded! Let me know if you guys have questions!
  - Answers on Canvas
- Lab 1:
  - It was graded! as I said, I when very easy on the grade. Don’t expect that for the next labs!
  - **Make sure to read the comments**!
- Come to office hours if you need more detail on the next labs!
- Finally, a little recap from last time!

(Tentative) Lecture Roadmap

Basics of Deep Learning

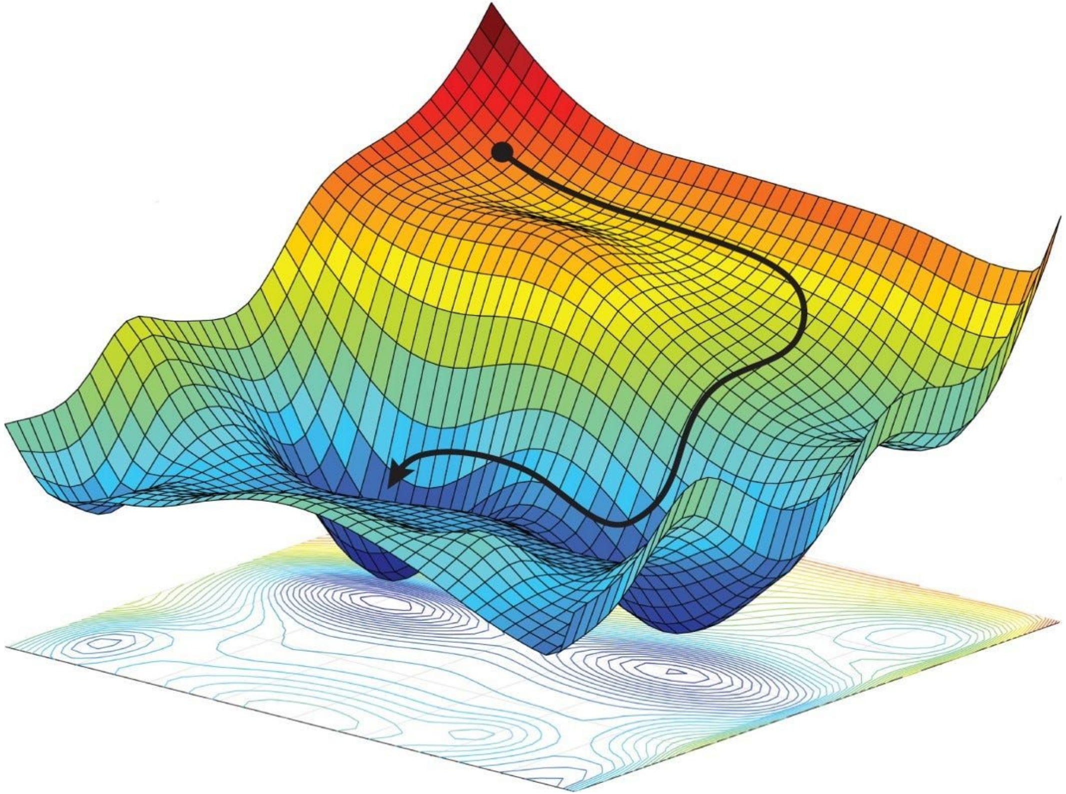
- Intro to Computer Vision
- Linear Classifiers and Perceptron
- Multilayer Perceptron
- Optimization in Deep Learning
- Pytorch I – MLPs
- Pytorch II – Images and Regularization
- Convolutional Neural Networks
- Data Augmentation and Deep CNNs
- Transfer Learning and Residual Nets
- Inception Net and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

Deep Learning and Computer Vision in Practice

- Intro to MLOps
- Intro to Object Detection
- Fast Object Detection
- Intro to Image Segmentation
- Applications of Detection and Segmentation
- Autoencoders
- Image Generation with GANs
- Advanced GANs
- The Attention Mechanism
- Transformers and ChatGPT
- Image Generation by Prompt
- Misc. Topics

Optimization and Neural Networks

- Last time we saw the major ingredient behind the power of deep learning: **Gradient Descent (GD)**
- Then we saw a few tricks that improve performance, such as stochasticity and momentum.
- Today we'll see how is applied to Neural Networks and we'll see how to implement this process in a computer using **Pytorch**.



Recap: Gradient Descent

- To minimize a **differentiable** function* $f(x)$ one can use **Gradient Descent (GD)**, which starting from some x_0, it finds x_1 such that $f(x_1)$ is lower than $f(x_2)$, and then repeats.
- It uses the derivative of f, defined as df/dx, to check its slope at each point to know where to go next.
- GD works just like a climber who wants to quickly go down a mountain:
  - He first steps around where he is, in order to “feel” the **slope** of his location,
  - Then decides to take the direction where the slope is the **steepest**,
  - After that he walks a **step** on that direction.
  - He then **repeats** the process until he is at the bottom of the mountain.

* We’ll work on the general case for now and get back to Neural Networks/Deep Learning later.

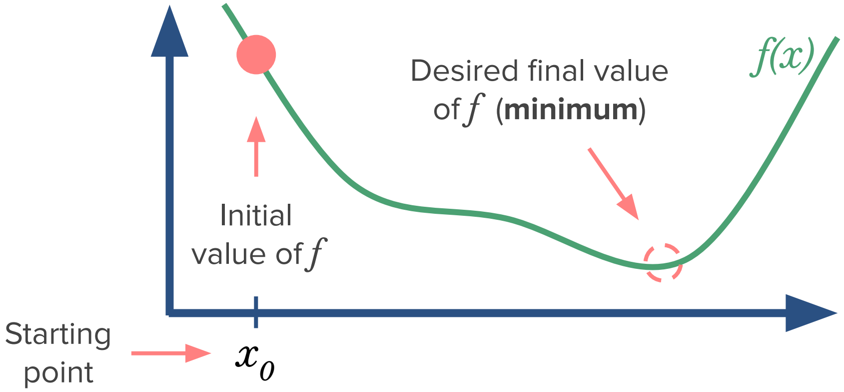
Recap: Gradient Descent in 1D

- We use this intuition to mathematically formulate our **minimizer** for functions in 1D:

$$x_{t+1} = x_t - \eta \frac{df}{dx}(x_t)$$

  where η (called step size or **learning rate**) is a constant*. This equation simply says:
  - If you are at x_t, the next point you should go to is on the opposite direction of the slope of f at x_t.
  - Then walk a step of size proportional to how steep that slope is in the direction.

- With this definition, the **gradient descent algorithm** in 1D is very simple:
  1. Pick a random starting point x_0,
  2. Repeat for t = 0, 1, 2, ... until $|grad| < \epsilon$**
     a. Compute $grad = df(x_t)/dx$
     b. Update x as in $x_{t+1} = x_t - \eta \times grad$



* η reads like "eta".
** ϵ ("epsilon") is just a small number set by the user.

Recap: Gradient Descent in 1D

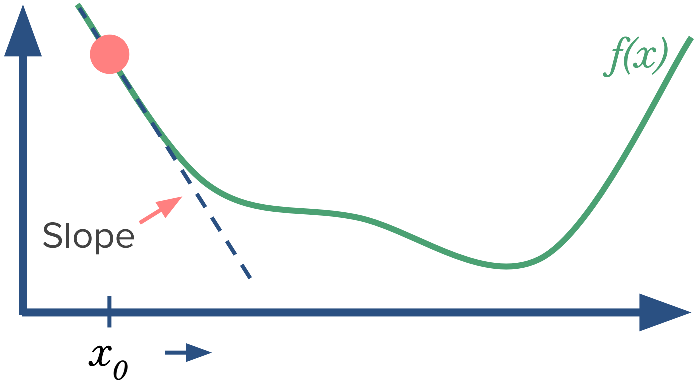
- We use this intuition to mathematically formulate our **minimizer** for functions in 1D:

$$x_{t+1} = x_t - \eta \frac{df}{dx}(x_t)$$

where η (called step size or **learning rate**) is a constant*. This equation simply says:

  - If you are at x_t, the next point you should go to is on the opposite direction of the slope of f at x_t.
  - Then walk a step of size proportional to how steep that slope is in the direction.

- With this definition, the **gradient descent algorithm** in 1D is very simple:
  1. Pick a random starting point x_0,
  2. Repeat for $t = 0, 1, 2, \ldots$ until $|grad| < \epsilon$**
     a. Compute $grad = df(x_t)/dx$
     b. Update x as in $x_{t+1} = x_t - \eta \times grad$



* η reads like "eta".

** ϵ ("epsilon") is just a small number set by the user.

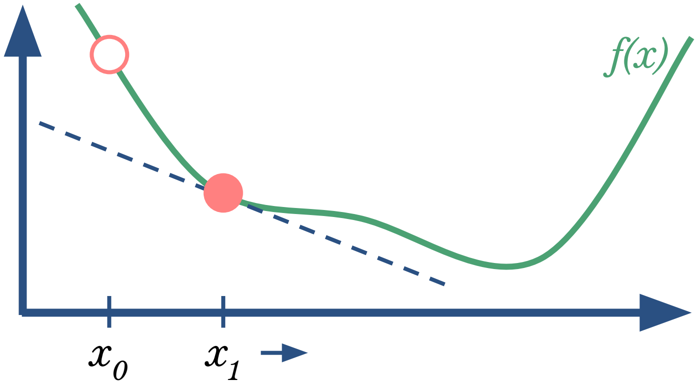
Recap: Gradient Descent in 1D

- We use this intuition to mathematically formulate our **minimizer** for functions in 1D:

$$x_{t+1} = x_t - \eta \frac{df}{dx}(x_t)$$

where η (called step size or **learning rate**) is a constant*. This equation simply says:
  - If you are at x_t, the next point you should go to is on the opposite direction of the slope of f at x_t.
  - Then walk a step of size proportional to how steep that slope is in the direction.

- With this definition, the **gradient descent algorithm** in 1D is very simple:
  1. Pick a random starting point x_0,
  2. Repeat for $t = 0, 1, 2, \ldots$ until $|grad| < \epsilon$**
     a. Compute $grad = df(x_t)/dx$
     b. Update x as in $x_{t+1} = x_t - \eta \times grad$



* η reads like "eta".
** ϵ ("epsilon") is just a small number set by the user.

Recap: Gradient Descent in 1D

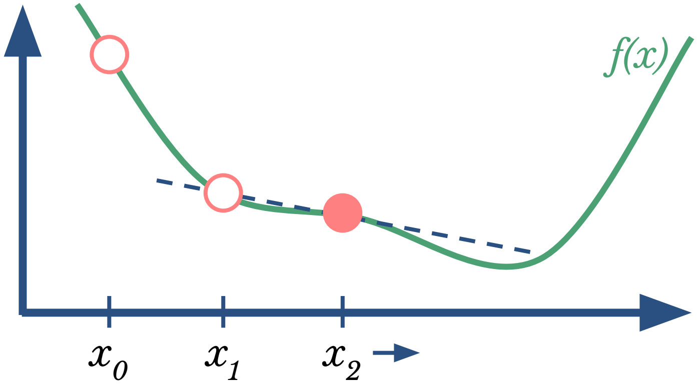
- We use this intuition to mathematically formulate our **minimizer** for functions in 1D:

$$x_{t+1} = x_t - \eta \frac{df}{dx}(x_t)$$

where η (called step size or **learning rate**) is a constant*. This equation simply says:

- If you are at x_t, the next point you should go to is on the opposite direction of the slope of f at x_t.
- Then walk a step of size proportional to how steep that slope is in the direction.

- With this definition, the **gradient descent algorithm** in 1D is very simple:
  1. Pick a random starting point x_0,
  2. Repeat for $t = 0, 1, 2, \ldots$ until $|grad| < \epsilon$**
     a. Compute $grad = df(x_t)/dx$
     b. Update x as in $x_{t+1} = x_t - \eta \times grad$



* η reads like "eta".
** ϵ ("epsilon") is just a small number set by the user.

Recap: Gradient Descent in 1D

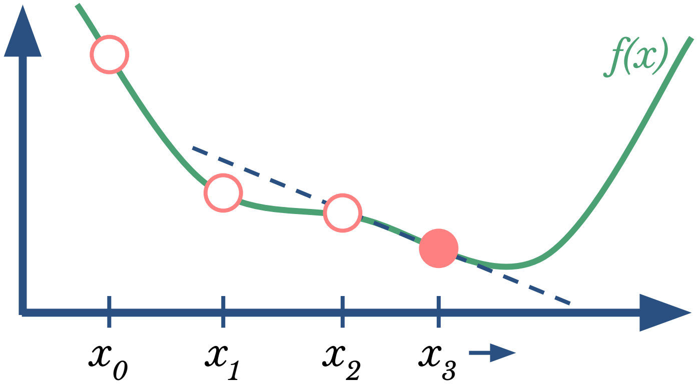
- We use this intuition to mathematically formulate our **minimizer** for functions in 1D:

$$x_{t+1} = x_t - \eta \frac{df}{dx}(x_t)$$

where η (called step size or **learning rate**) is a constant*. This equation simply says:

- If you are at x_t, the next point you should go to is on the opposite direction of the slope of f at x_t.
- Then walk a step of size proportional to how steep that slope is in the direction.

- With this definition, the **gradient descent algorithm** in 1D is very simple:
  1. Pick a random starting point x_0,
  2. Repeat for $t = 0, 1, 2, \ldots$ until $|grad| < \epsilon$**
     a. Compute $grad = df(x_t)/dx$
     b. Update x as in $x_{t+1} = x_t - \eta \times grad$



* η reads like "eta".
** ϵ ("epsilon") is just a small number set by the user.

Recap: Gradient Descent in 1D

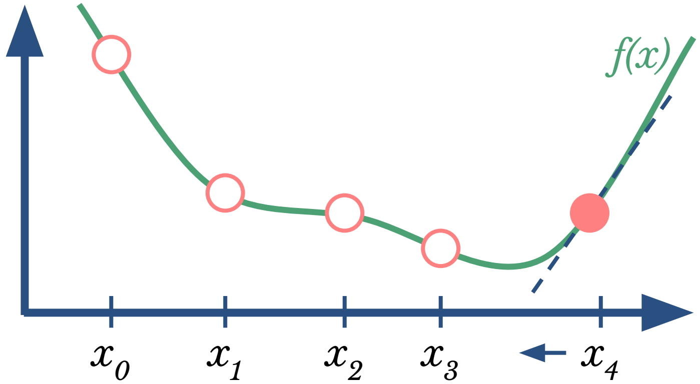
- We use this intuition to mathematically formulate our **minimizer** for functions in 1D:

$$x_{t+1} = x_t - \eta \frac{df}{dx}(x_t)$$

where η (called step size or **learning rate**) is a constant*. This equation simply says:

  - If you are at x_t, the next point you should go to is on the opposite direction of the slope of f at x_t.
  - Then walk a step of size proportional to how steep that slope is in the direction.

- With this definition, the **gradient descent algorithm** in 1D is very simple:
  1. Pick a random starting point x_0,
  2. Repeat for $t = 0, 1, 2, \ldots$ until $|grad| < \epsilon$**
     a. Compute $grad = df(x_t)/dx$
     b. Update x as in $x_{t+1} = x_t - \eta \times grad$



* η reads like "eta".

** ϵ ("epsilon") is just a small number set by the user.

Recap: Gradient Descent in 1D

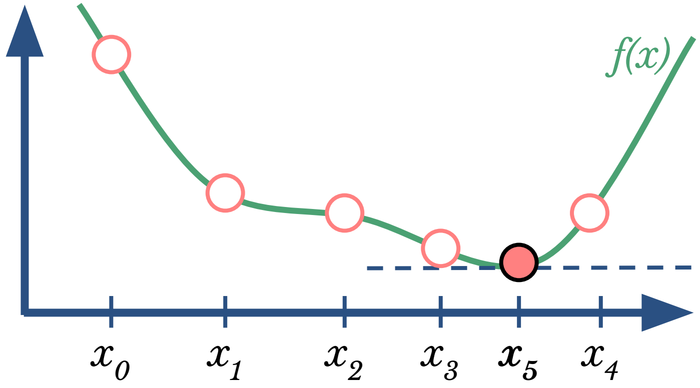
- We use this intuition to mathematically formulate our **minimizer** for functions in 1D:

$$x_{t+1} = x_t - \eta \frac{df}{dx}(x_t)$$

where η (called step size or **learning rate**) is a constant*. This equation simply says:

  - If you are at x_t, the next point you should go to is on the opposite direction of the slope of f at x_t.
  - Then walk a step of size proportional to how steep that slope is in the direction.

- With this definition, the **gradient descent algorithm** in 1D is very simple:
  1. Pick a random starting point x_0,
  2. Repeat for $t = 0, 1, 2, \ldots$ until $|grad| < \epsilon$**
     a. Compute $grad = df(x_t)/dx$
     b. Update x as in $x_{t+1} = x_t - \eta \times grad$



* η reads like "eta".

** ϵ ("epsilon") is just a small number set by the user.

Chain rule and Backpropagation

- After seeing all this theory of optimization, we only miss one thing: **how can we apply it to the neural networks we saw before???**
- Well, the first step is to write out the function we need to minimize.
- If we are using cross-entropy loss, this is the average loss function for our network:

$$L(\theta) = \frac{1}{n}\sum_{i=1}^{n} l(NN_\theta(x^{(i)}), y^{(i)}) = -\frac{1}{n}\sum_{i=1}^{n}[y^{(i)}]^\top \log(\mathrm{softmax}(W_L a(W_{L-1}\cdots a(W_0 x^{(i)})...)))$$

- Now we "just" need to compute the gradient of $L(\theta)$ with respect to θ! Although not straightforward, one just has to use the **Chain Rule** from calculus.
- Say you have two **differentiable** functions f and g. Let $y = f(g(x))$ and $u = g(x)$ for a value x. Then we have that:

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}$$

Chain rule and Backpropagation

- *Example*: if $f(x) = x^2$ and $g(x) = 3x^3 + 2$, then the derivative of $y = f(g(x))$ (call $u = g(x)$):

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx} = (2u)(9x^2) = (2(3x^3+2))(9x^2) = 54x^5+36x^2$$

- Using a similar approach one can consider $y = f_1(f_2(f_3 ... f_n(x)...))$. Let $u_1 = f_2(f_3 ... f_n(x)...)$, $u_2 = (f_3 ... f_n(x)...)$ and so on. Then we have that:

$$\frac{dy}{dx} = \frac{dy}{du_1}\frac{du_1}{du_2}\frac{du_2}{du_3}\cdots\frac{du_n}{dx}$$

- For (simple) neural networks, one only has to apply the chain rule to get the weight updates*.
- In that case the first step it to “see” our loss definition as a series of composed function such as $y = f_1(f_2(f_3 ... f_n(x)...))$.

* Mathematically speaking, the ReLU activation is not differentiable, which should complicate things. In practice, however, the deep learning community simply simply **disregards** this issue. This paper goes in detail about this issue.

Chain rule and Backpropagation

- To make things simple, let's consider a network of just one hidden layer, the loss on only one datapoint (called x with true label y) and that we just want to optimize W_0. Then:

$$u_0(W_0) = l(NN_\theta(x), y) = -y^\top \log(\text{softmax}(W_1 a(W_0 x)))$$

- Let $u_1(z) = -y^\top z$, $u_2(z) = log(z)$, $u_3(z) = \text{softmax}(z)$, $u_4(z) = W_1 z$, $u_5(z) = a(z)$, $u_6(z) = z^\top x$, where z is a **vector** or a **matrix**. Then have that $u_0 = u_1(u_2(u_3(u_4(u_5(u_6(W_0))))))$ and that

$$\frac{du_0}{dW_0} = \frac{du_0}{du_1}\frac{du_1}{du_2}\frac{du_2}{du_3}\frac{du_3}{du_4}\frac{du_4}{du_5}\frac{du_5}{du_6}\frac{du_6}{dW_0}$$

- Now things are much easier: for example, using matrix calculus*, we have $du_1/dz = -y$, $du_4(z)/dz = W_1$ and so on (*Remember that one has to compute Jacobians sometimes*).
- Note that you'd need to do something similar to W_1 to optimize it too.

* Here, you can find more refreshing information on derivatives with respect to vectors and matrices

Exercises (*In pairs*)

- Recall the expression of a general multilayer perceptron with L layers:

$$L(\theta) = -\frac{1}{n}\sum_{i=1}^{n}[y^{(i)}]^\top \log(\text{softmax}(W_L a(W_{L-1}\cdots a(W_0 x^{(i)})...)))$$

  How many "operations" are there in between $W_0 x^{(i)}$ and the final sum in terms of L? Based on that number, how many multiplications would take place to compute the final $dL(\theta)/dW_0$? For example, there are 6 operations in the following expression:

$$-y^\top \log(\text{softmax}(W_1 a(W_0 x)))$$

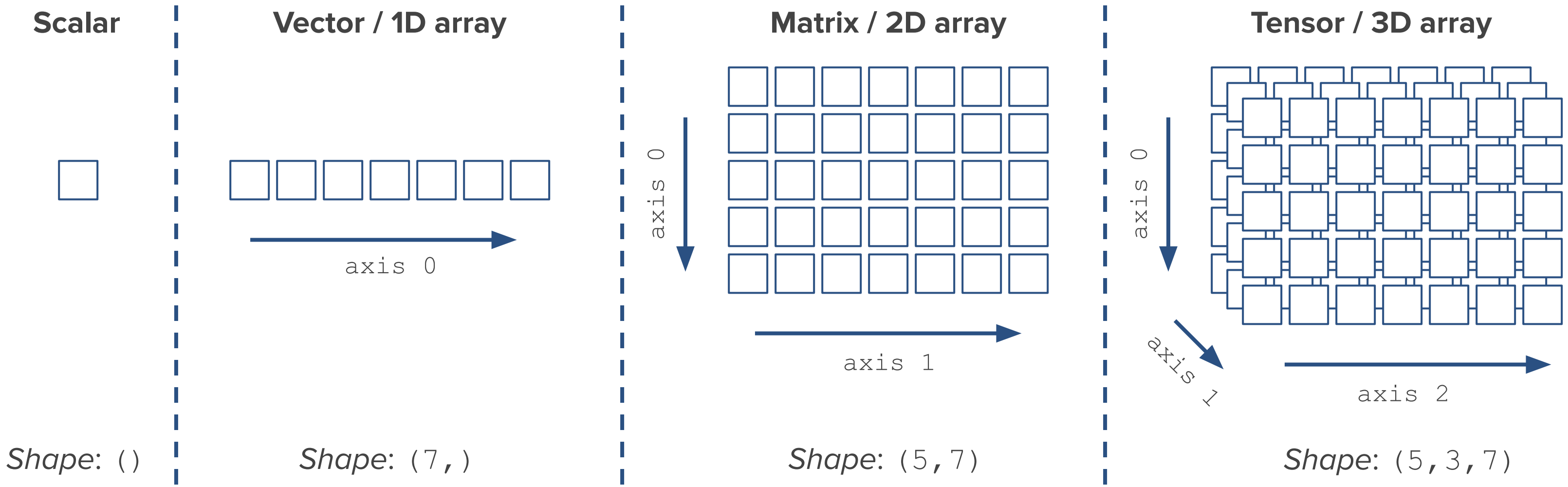
  and 6 multiplications are needed to compute its derivative over W_0.
- What happens to the final derivative $dL(\theta)/dW_0$ if many of the derivatives inside its chain rule are smaller than 1?

PyTorch

- After we learned all this 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.

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 $3{\times}4$ 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 + x)
print(x * 10)
```

```
tensor([[ 2,  4,  6,  8],
        [10, 12, 14, 16]])
tensor([[10, 20, 30, 40],
        [50, 60, 70, 80]])
```

  - 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 function `backward()`, which computes gradients according to `f` and store them in the tensors it finds in its computation pipeline (like `x`).

```
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)
```

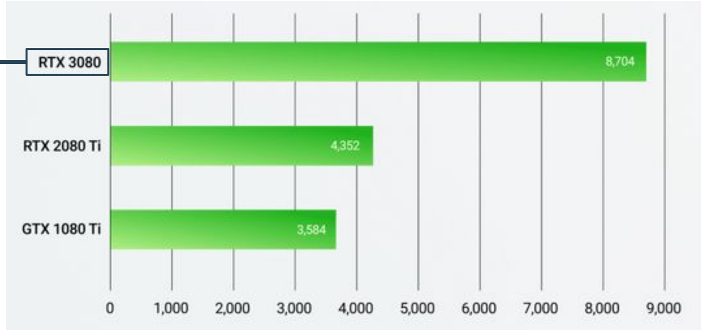
```
tensor([ 4., -2.])
```

- 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 much more quickly than Numpy.
- This is due to the fact that PyTorch tensors are 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('cuda')`, and in the CPU (with `.cpu()`) and compare their performances with regular Numpy:

```
x, y = x_t.to('cuda'), y_t.to('cuda')
%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
```

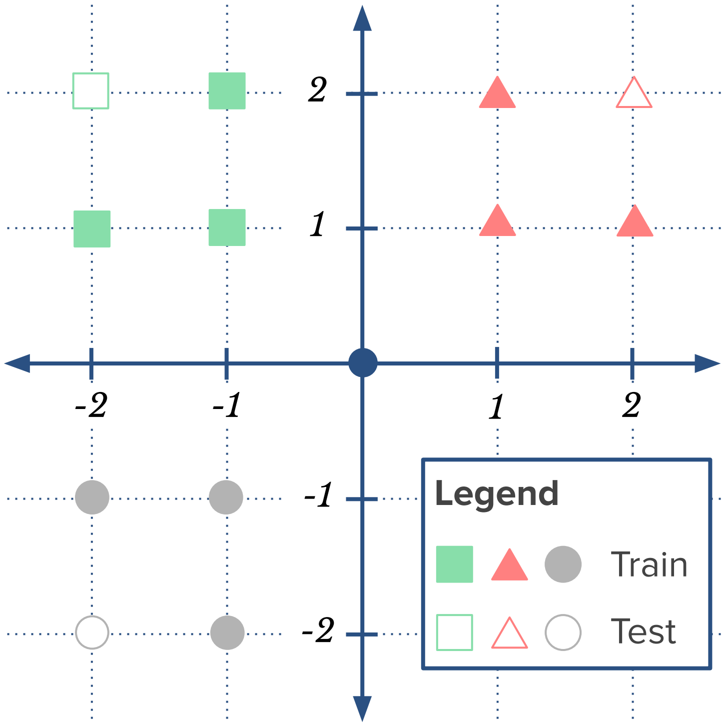
```
%timeit z = np.matmul(x_n,y_n)
```

```
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:

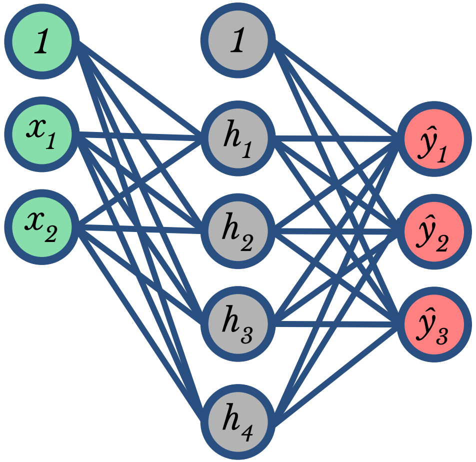
```
x_train = [[-2,-1], [-1,-1], [-1,-2],
           [2,-1],  [1,-1],  [1,-2],
           [2,1],   [1,1],   [1,2]]
y_train = [[1, 0, 0], [1, 0, 0], [1, 0, 0],
           [0, 1, 0], [0, 1, 0], [0, 1, 0],
           [0, 0, 1], [0, 0, 1], [0, 0, 1]]

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
```



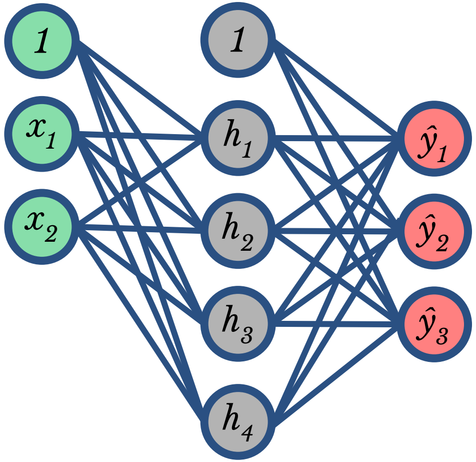
Our First Neural Network: Architecture

- 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`.



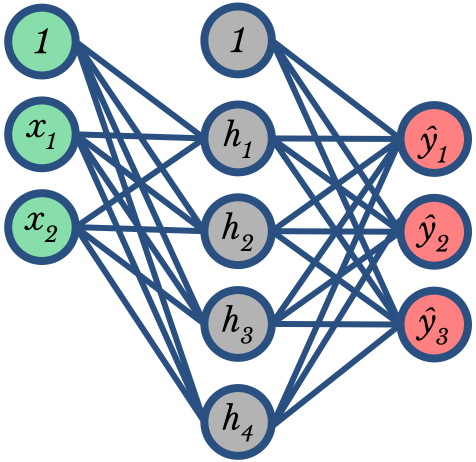
Our First Neural Network: Architecture

- 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.

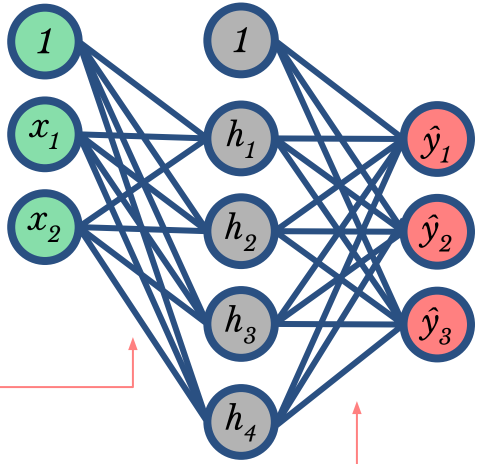
Notice: no softmax!



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 simplicity, we will not consider mini-batches (which makes SGD “not stochastic” for now), so we’ll use all the data to compute one step in gradient descent.
- When training a network in PyTorch, we have to repeat *4* main steps in within `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
```

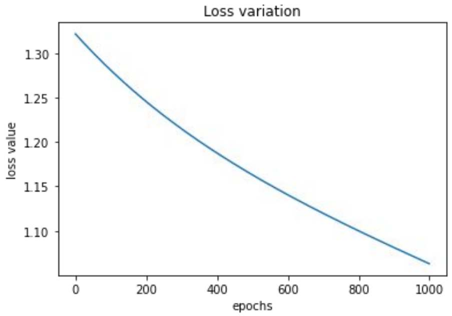
Our First Neural Network: Training!

- 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())
```

... 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: Training!

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

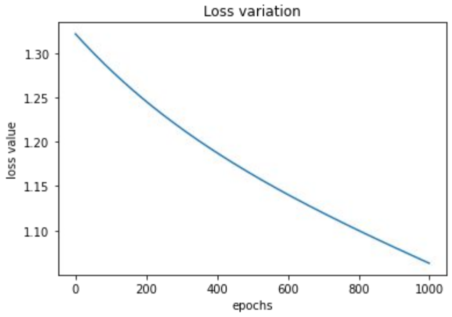
```
n_epochs = 1000
loss_history = []
for    in range(n_epochs):
   opt.zero_grad()
   loss_value = loss_func(mynet(X_train),Y_train)
   loss_value.backward()
   opt.step()

   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`.

Homework (*In pairs*)

- 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.
- Graph the loss variation of epochs on those experiments.
- Create an MLP that learns to classify the data in this dataset (from a few lectures ago):

Click here to open code in Colab

```
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
x, y = make_blobs(n_samples=400, centers=4, cluster_std=2, random_state=10)
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.4, random_state=2)
```

Train the network on you training data and test it on your test data. Add an `accuracy()` function that computes final classification accuracy. You'll need to use the function `torch.nn.functional.one_hot()` from Pytorch (*More on it here*).