# Your First MLP Code

Recitation 1, part 1
Fall 2021

# Part 1 - Overview

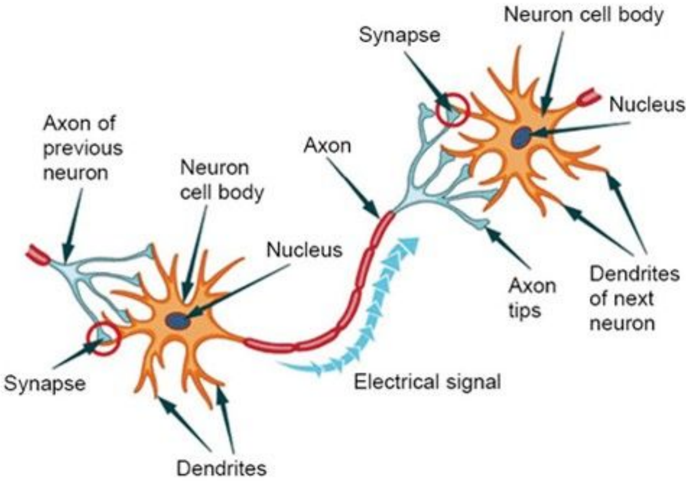
- Neural Networks
- Perceptrons
- Multilayer perceptrons
    - Forward Pass
    - Backpropagation
    - Update Weights

# Neural Networks

- The brain, made up of connected neurons, are the inspirations for artificial neural networks.

# Neural Networks

- A neuron is a node with many inputs and one output.
- A neural network consists of many interconnected neurons -- a “simple” device that receives data at the input and provides a response.
- Information are transmitted from one neuron to another by electrical impulses and chemical signals.



# Perceptrons

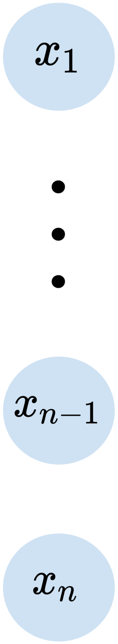
- Perceptron is a single layer neural network.

# Perceptrons

- Perceptron is a single layer neural network.
- The perceptron consists of 4 parts.

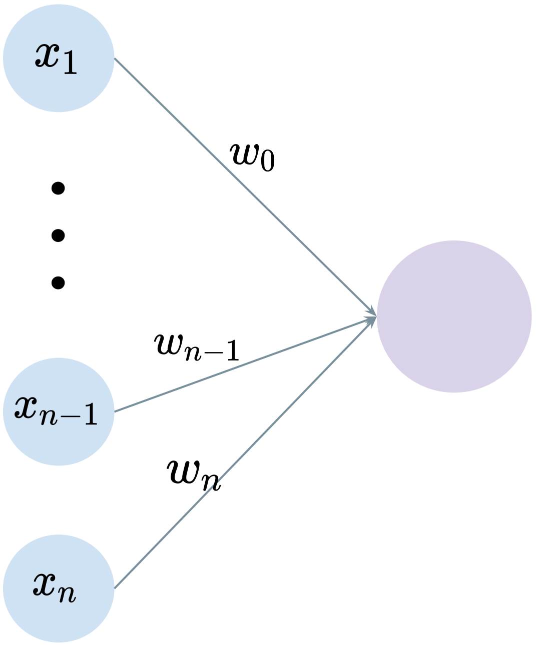
# Perceptrons

- Perceptron is a single layer neural network.
- The perceptron consists of 4 parts.
    - Input values



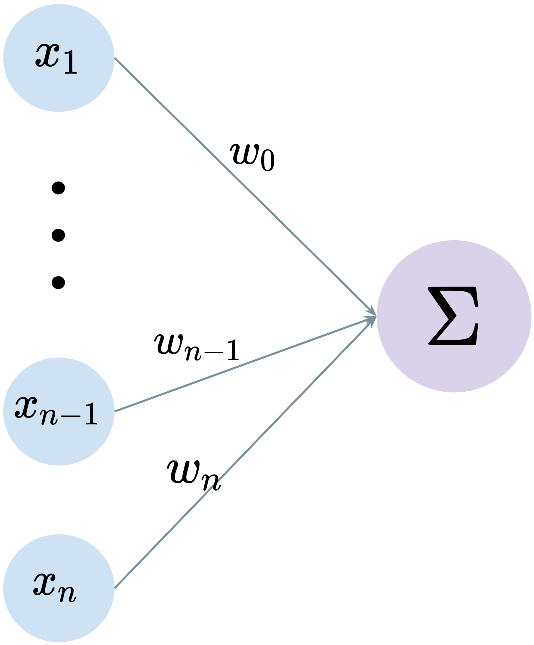
# Perceptrons

- Perceptron is a single layer neural network.
- The perceptron consists of 4 parts.
  - Input values
  - Weights



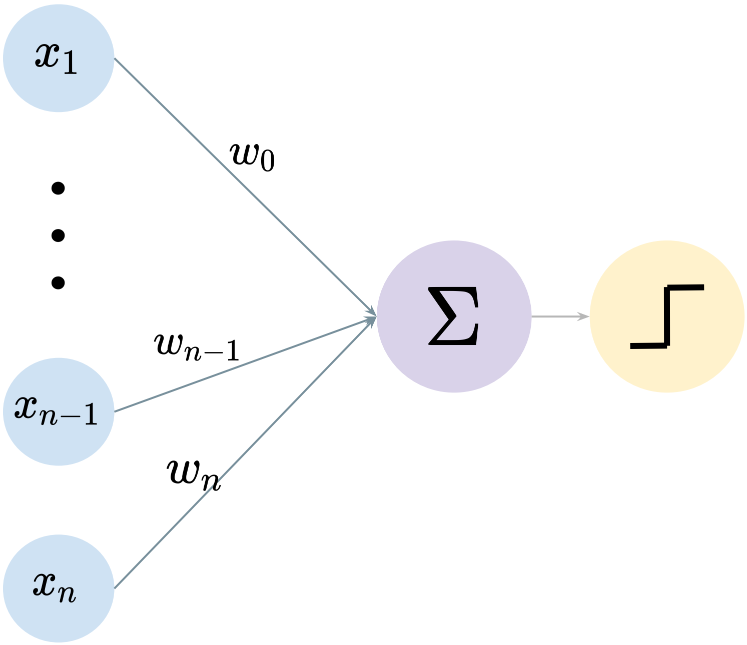
# Perceptrons

- Perceptron is a single layer neural network.
- The perceptron consists of 4 parts.
  - Input values
  - Weights
  - Weighted sums



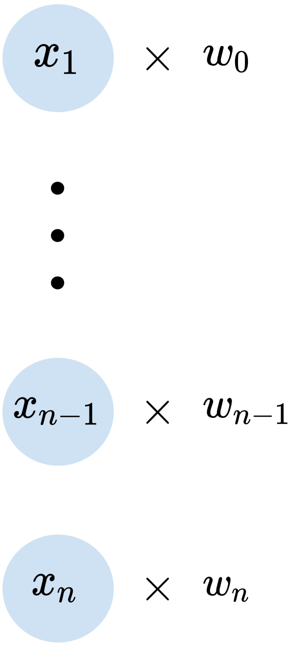
# Perceptrons

- Perceptron is a single layer neural network.
- The perceptron consists of 4 parts.
  - Input values
  - Weights
  - Weighted sums
  - Threshold / Activation functions



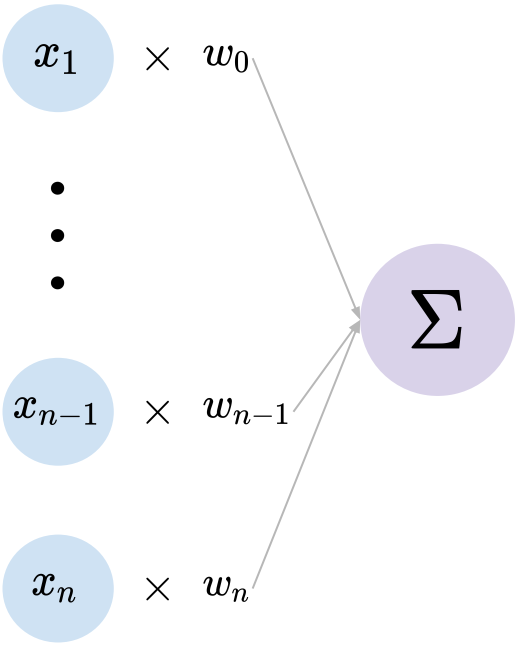
# Perceptrons

- Perceptron is a single layer neural network.
- The perceptron consists of 4 parts.
- The perceptron works on the following steps:
    - Multiply all inputs with their weights



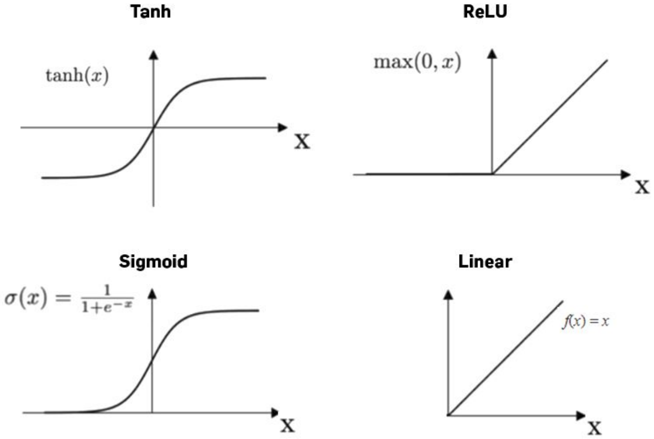
# Perceptrons

- Perceptron is a single layer neural network.
- The perceptron consists of 4 parts.
- The perceptron works on the following steps:
  - Multiply all inputs with their weights
  - Add all multiplied values → weighted sum



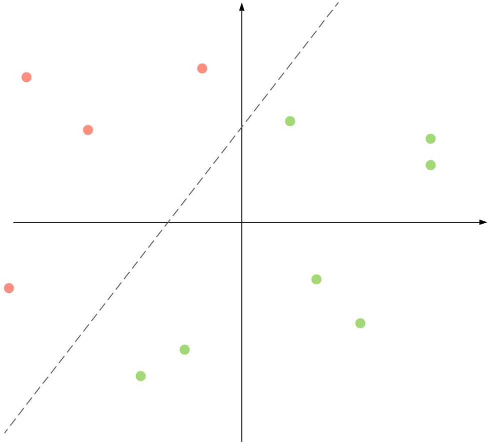
# Perceptrons

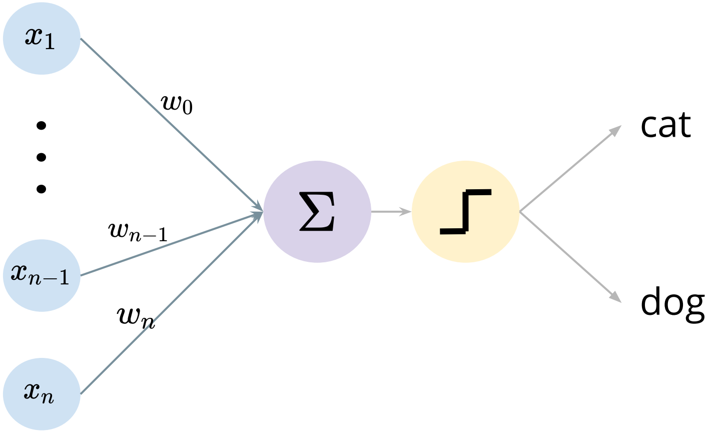
- Perceptron is a single layer neural network.
- The perceptron consists of 4 parts.
- The perceptron works on the following steps:
  - Multiply all inputs with their weights
  - Add all multiplies values → weighted sum
  - Apply the weighted sum to activation function



# Perceptrons

- Perceptron is usually used to classify the data into two parts -- **Linear Binary Classifier**.





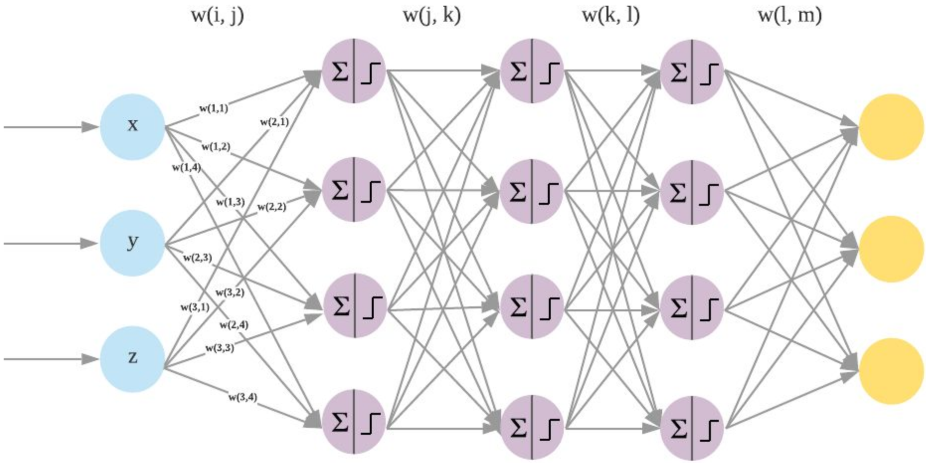
# Multilayer Perceptrons

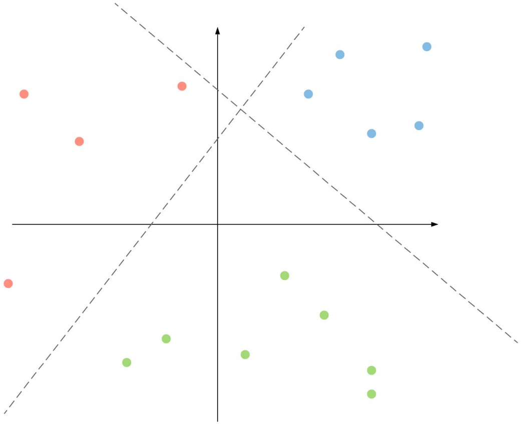
**What if we want to be able to distinguish between more classes?**

# Multilayer Perceptrons

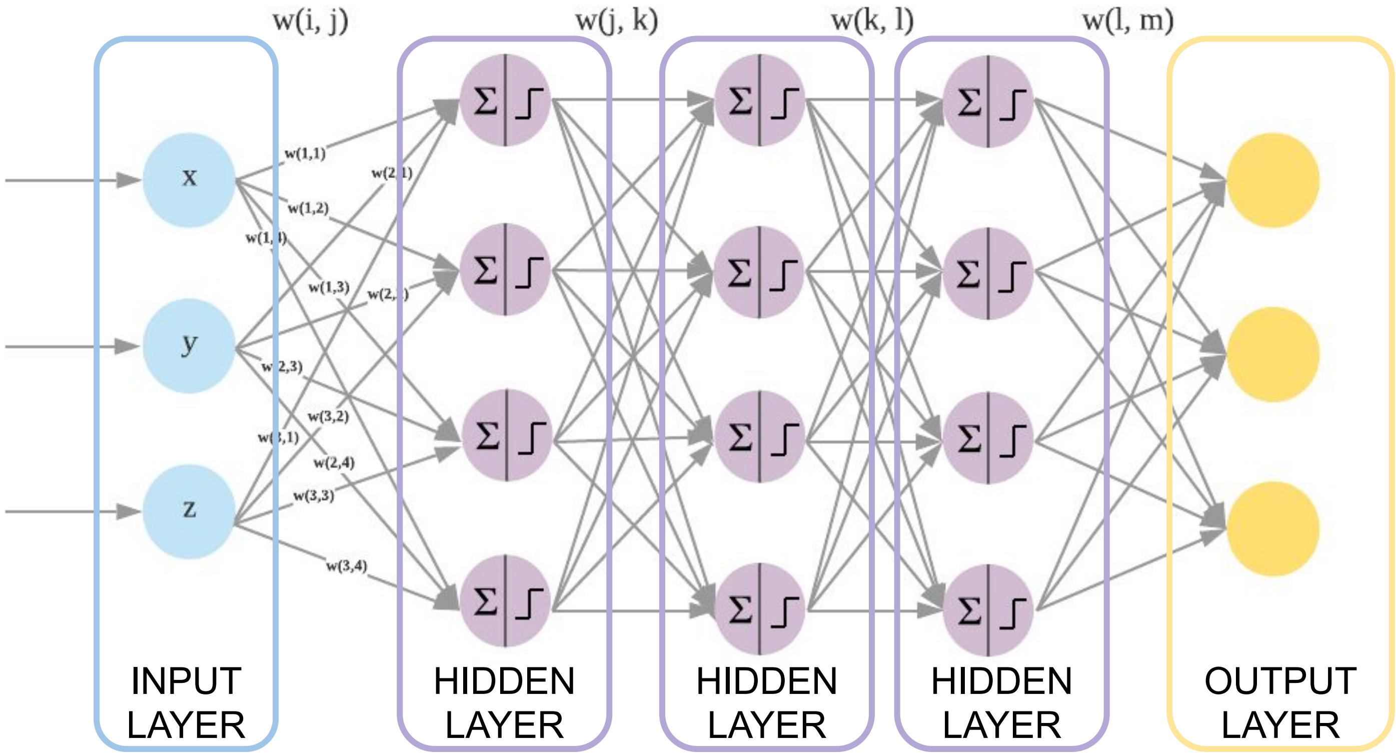
**What if we want to be able to distinguish between more classes?**

- Introduce more perceptrons !



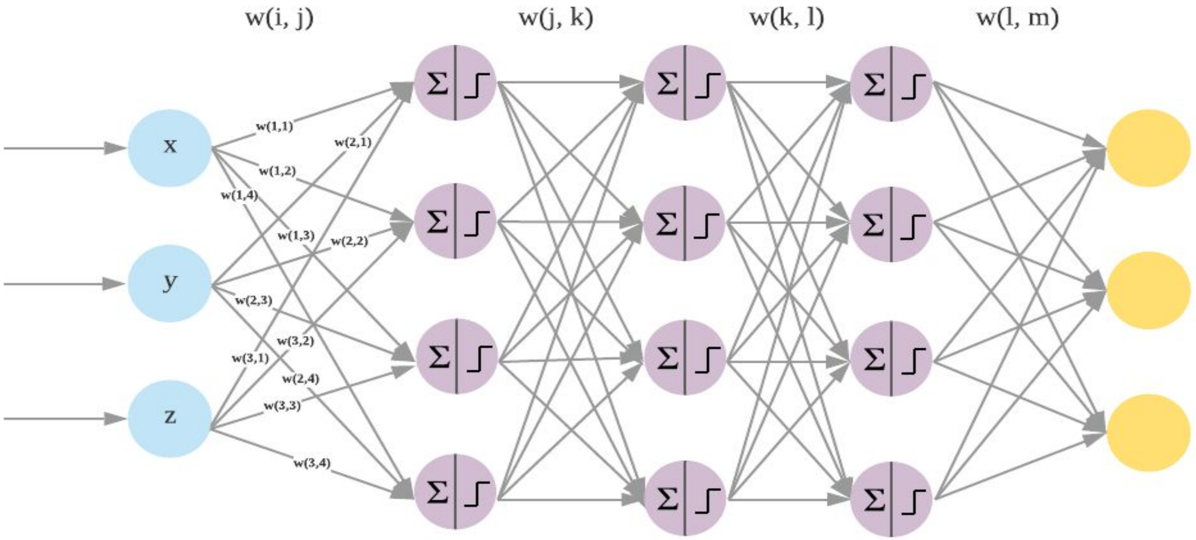


# Multilayer Perceptrons



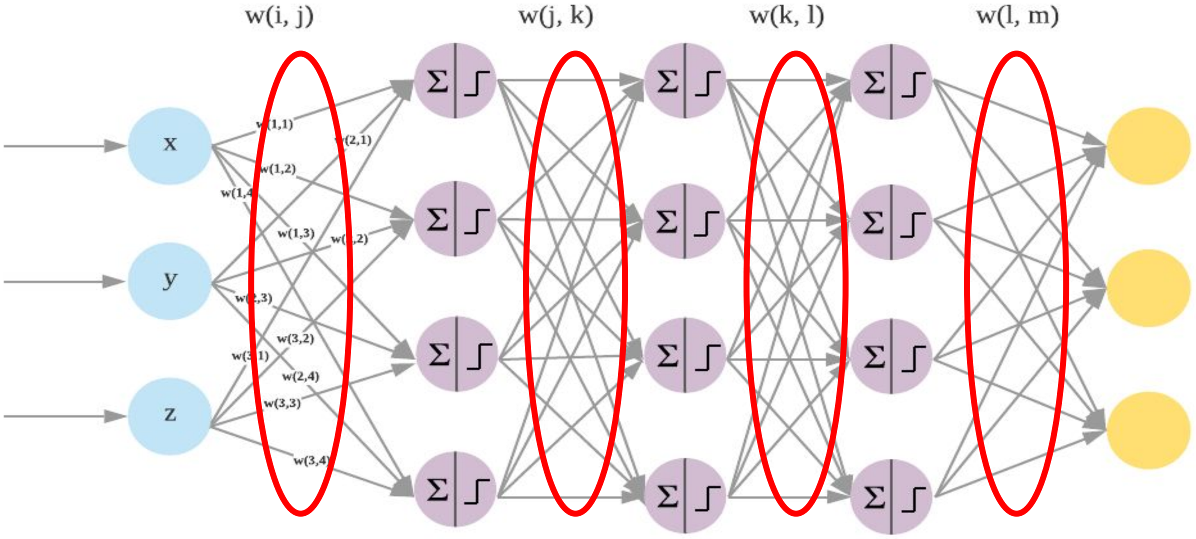
In order to correctly classify things, the network must be **learned**.

# But first, **what** do we need to learn?



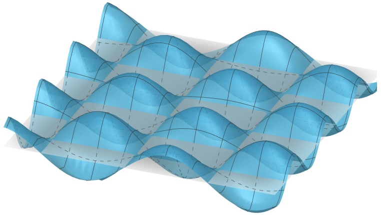
# But first, **what** do we need to learn?

The parameters (or the weights)



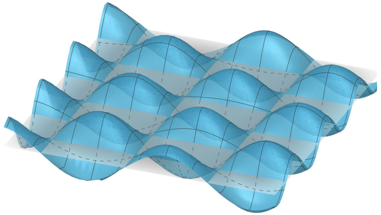
# How do we learn?

- ➔ Actual Function that we are trying to model:
  - ◆ Note: We don't know the actual function.

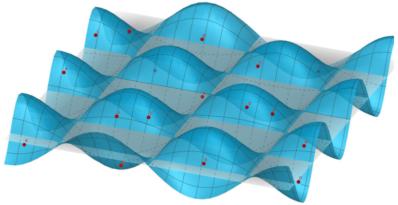


# How do we learn?

- ➔ Actual Function that we are trying to model:
  - ◆ Note: We don’t know the actual function.

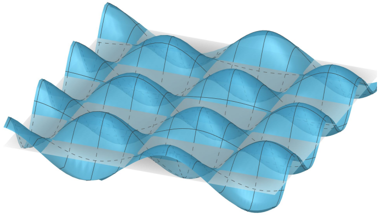


- ➔ We only have several sample data points on this function.

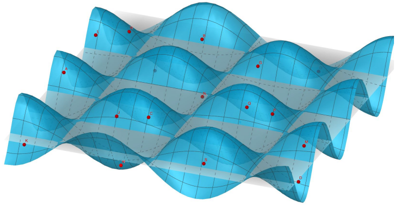


# How do we learn?

- ➔ Actual Function that we are trying to model:
  - ◆ Note: We don't know the actual function.



- ➔ We only have several sample data points on this function.



- ➔ Our goal:
  - ◆ **Estimate the function with the given samples.**

# How do we learn?

- ➔ A measurement of **error**

# How do we learn?

- A measurement of **error**
  - How much off is the **network output** with respect to the **desired output**

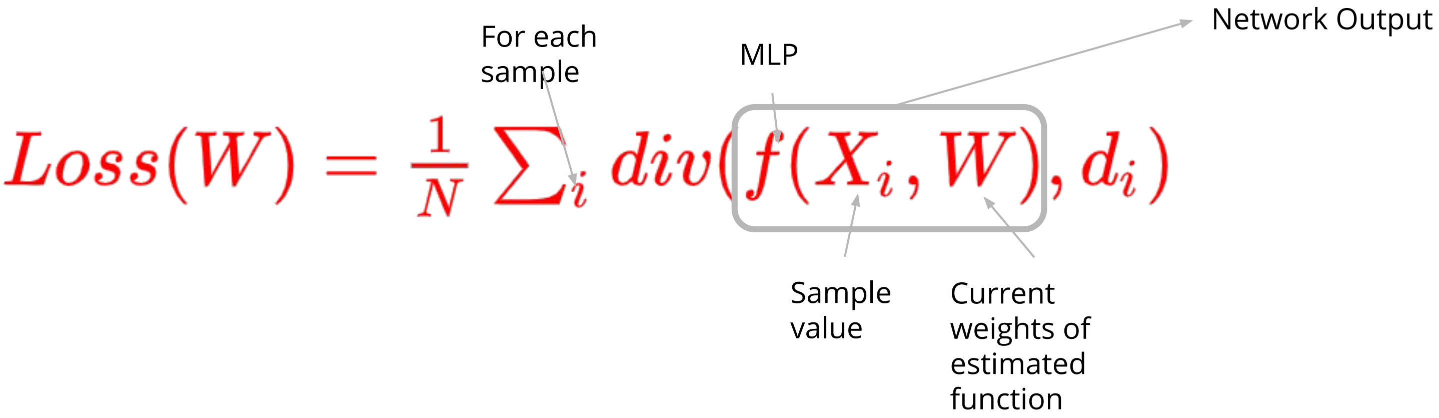
# How do we learn?

- A measurement of **error**
  - How much off is the **network output** with respect to the **desired output**

$$Loss(W) = \frac{1}{N} \sum_i div(f(X_i, W), d_i)$$

# How do we learn?

- ➔ A measurement of **error**
  - ◆ How much off is the **network output** with respect to the **desired output**



$$Loss(W) = \frac{1}{N}\sum_i div(\boxed{f(X_i, W)}, d_i)$$

For each sample

MLP

Network Output

Sample value

Current weights of estimated function

# How do we learn?

- A measurement of **error**
  - How much off is the **network output** with respect to the **desired output**

$$Loss(W) = \frac{1}{N} \sum_i div(\boxed{f(X_i, W)}, d_i)$$

For each sample

MLP

Network Output

Desired Output

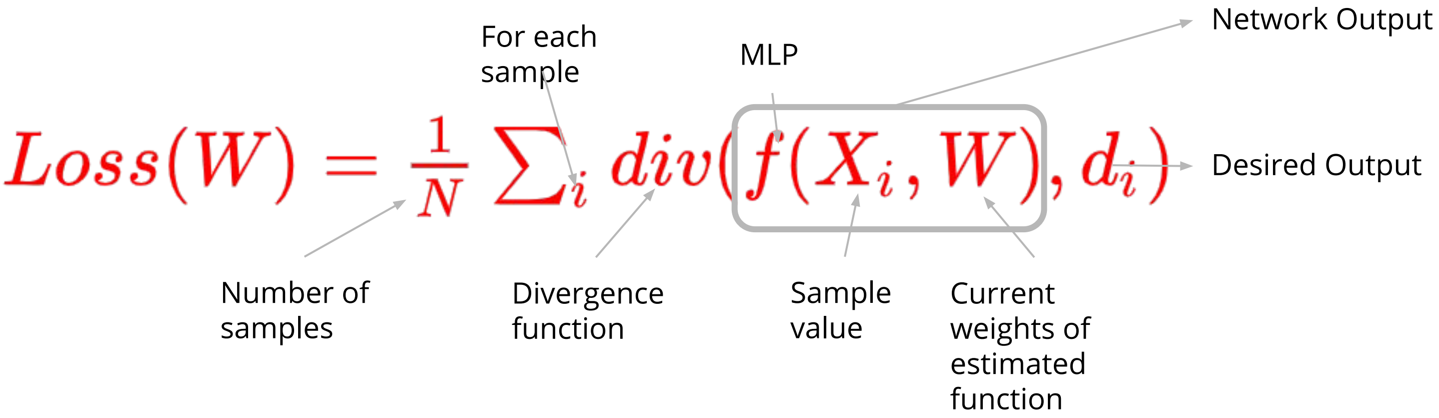
Divergence function

Sample value

Current weights of estimated function

# How do we learn?

- ➔ A measurement of **error**
  - ◆ How much off is the **network output** with respect to the **desired output**



$$Loss(W) = \frac{1}{N}\sum_i div(\boxed{f(X_i, W)}, d_i)$$

For each sample

MLP

Network Output

Desired Output

Number of samples

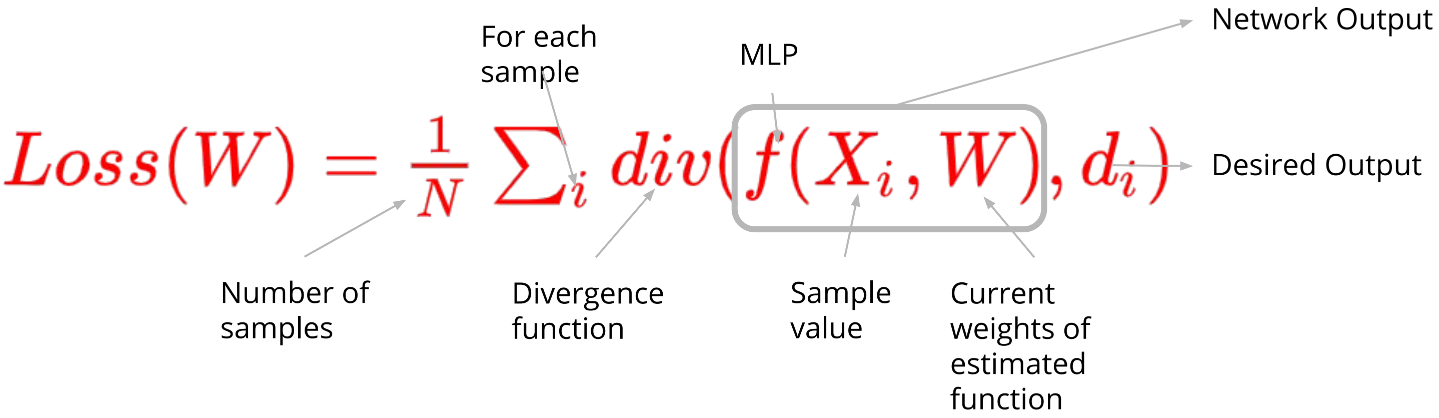
Divergence function

Sample value

Current weights of estimated function

# How do we learn?

- A measurement of **error**
  - How much off is the **network output** with respect to the **desired output**



$$Loss(W) = \frac{1}{N}\sum_i div(\boxed{f(X_i, W)}, d_i)$$

For each sample

MLP

Network Output

Desired Output

Number of samples

Divergence function

Sample value

Current weights of estimated function

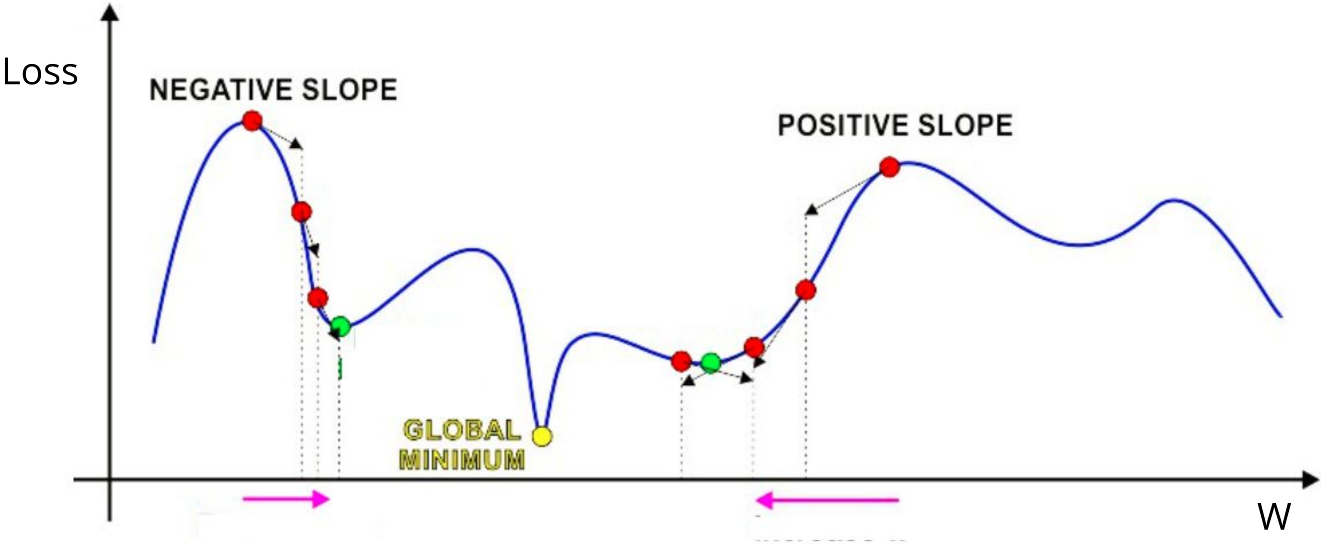
- Our goal (more specifically):
  - Minimize the loss

$$\hat{W} = \arg\min_{W} \; Loss(W)$$

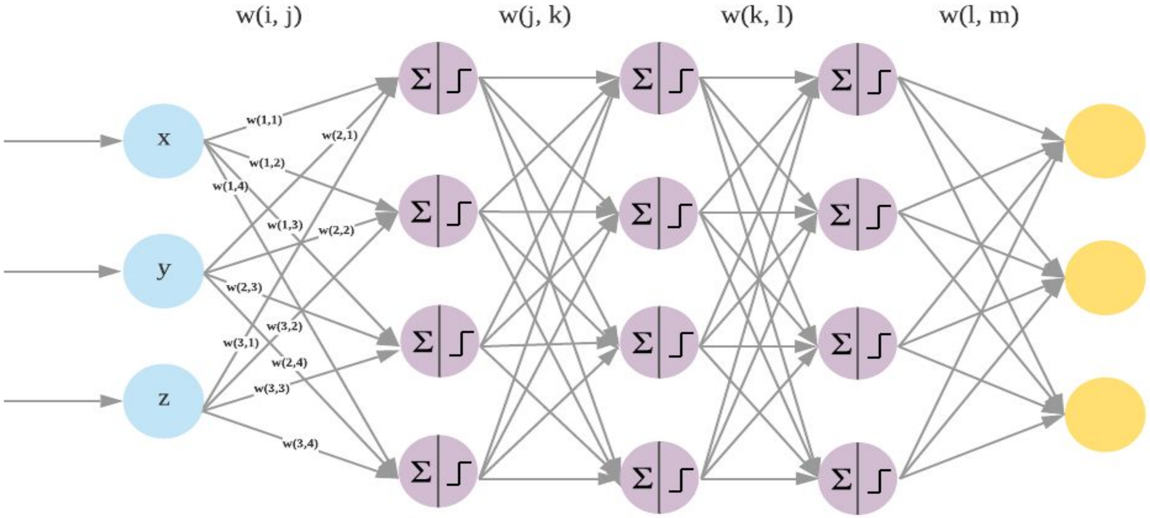
# How do we learn?

# How do we learn?

➔ Gradient Descent

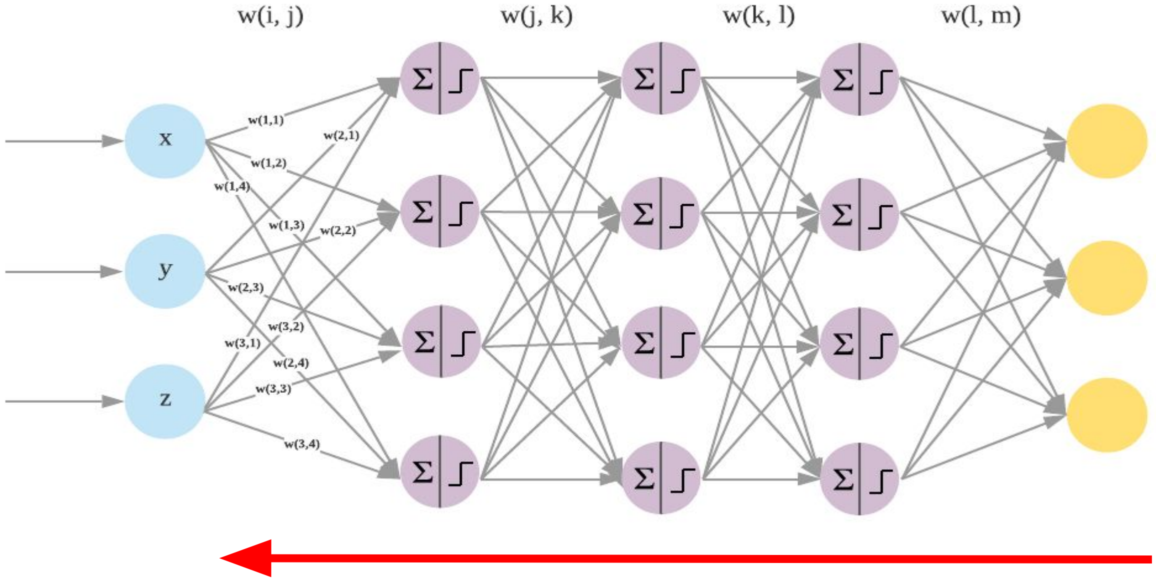


# How do we learn?



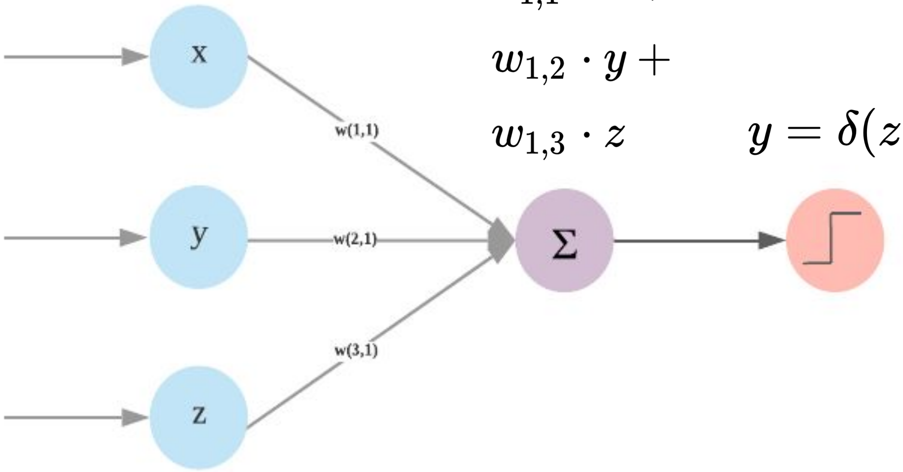
# How do we learn?

➔ Backpropagation



# Forward Pass

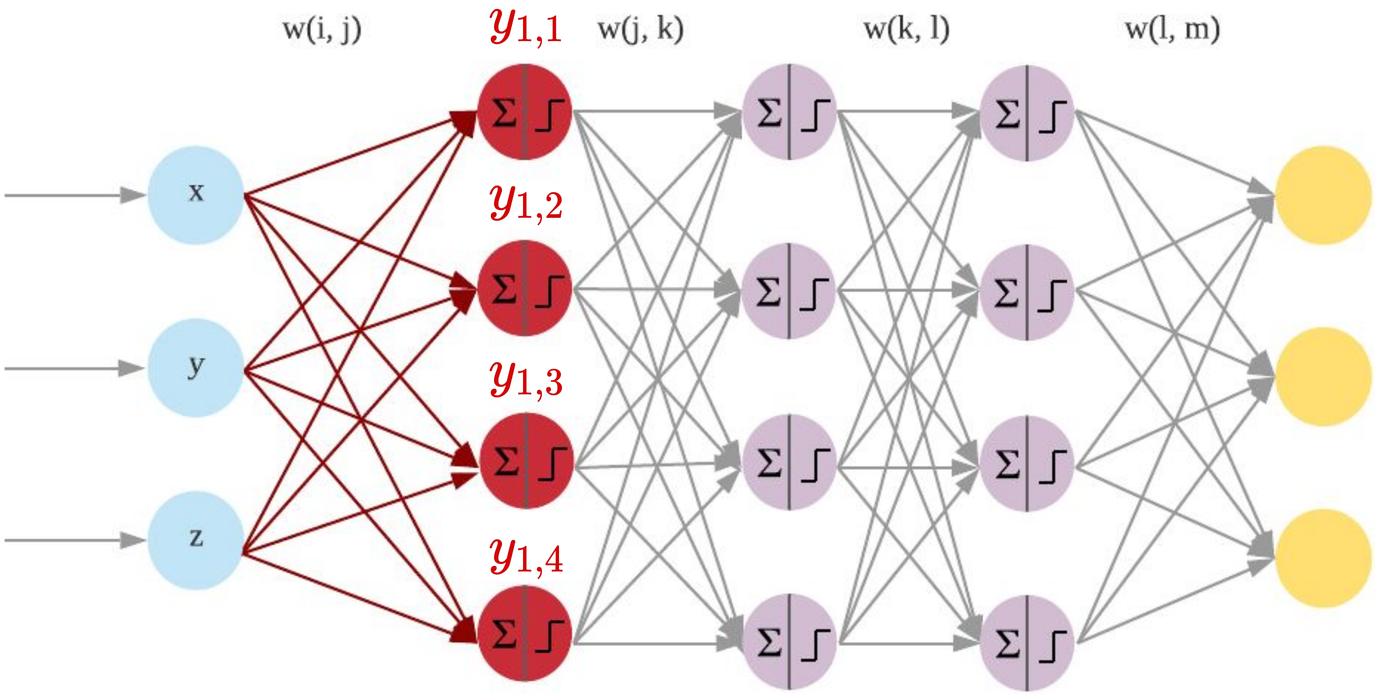
- For each single perceptron



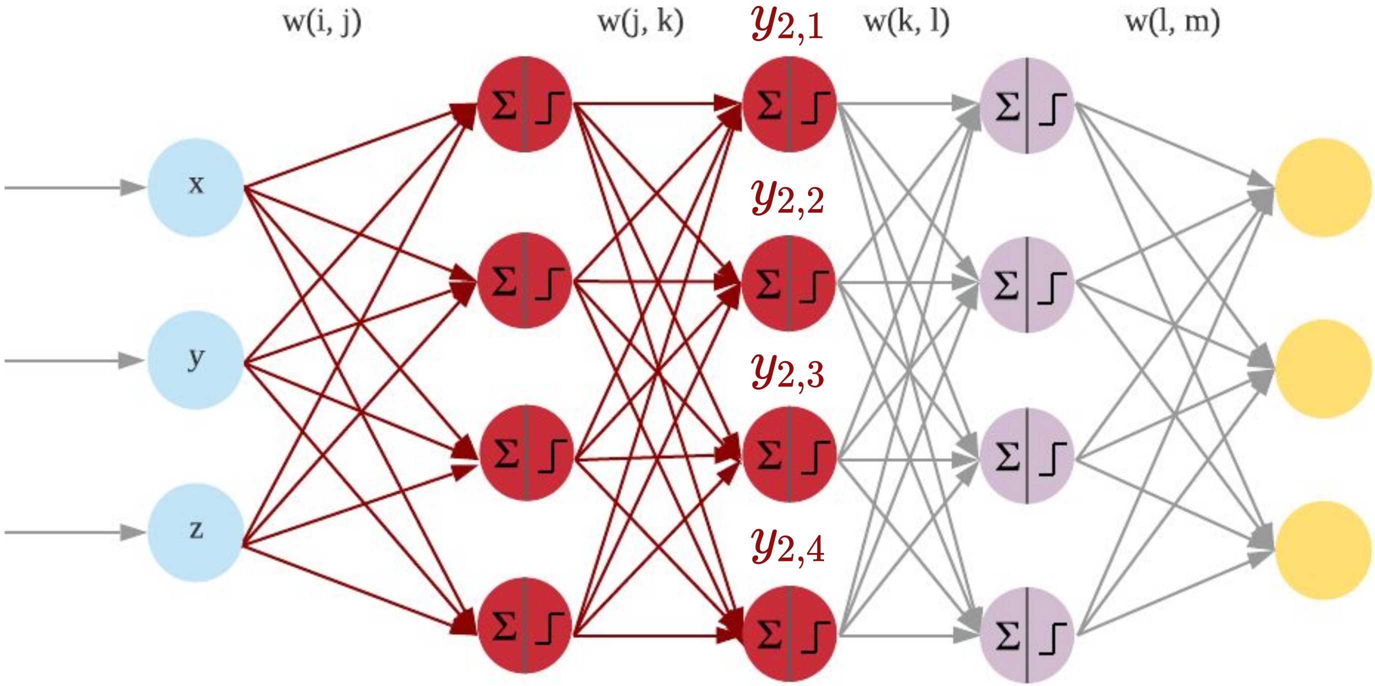
$$\begin{aligned} z = {} & w_{1,1} \cdot x + \\ & w_{1,2} \cdot y + \\ & w_{1,3} \cdot z \end{aligned}$$

$$y = \delta(z)$$

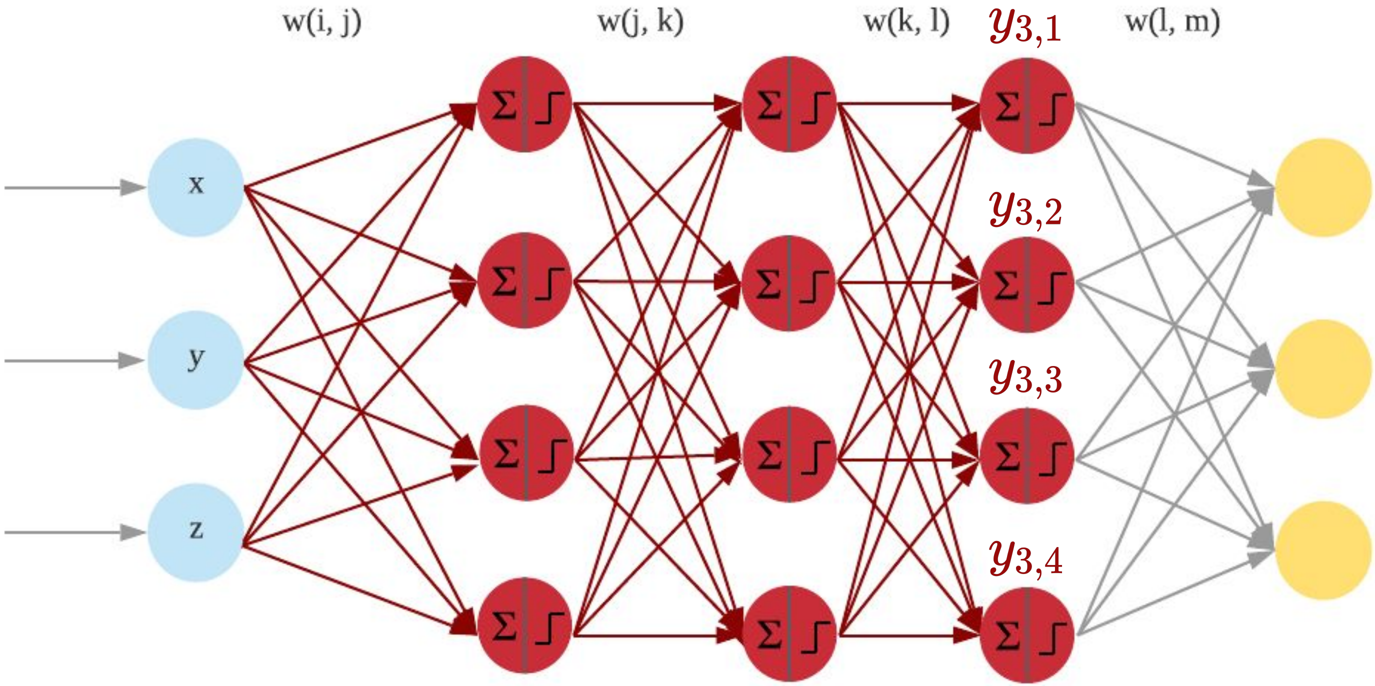
# Forward Pass



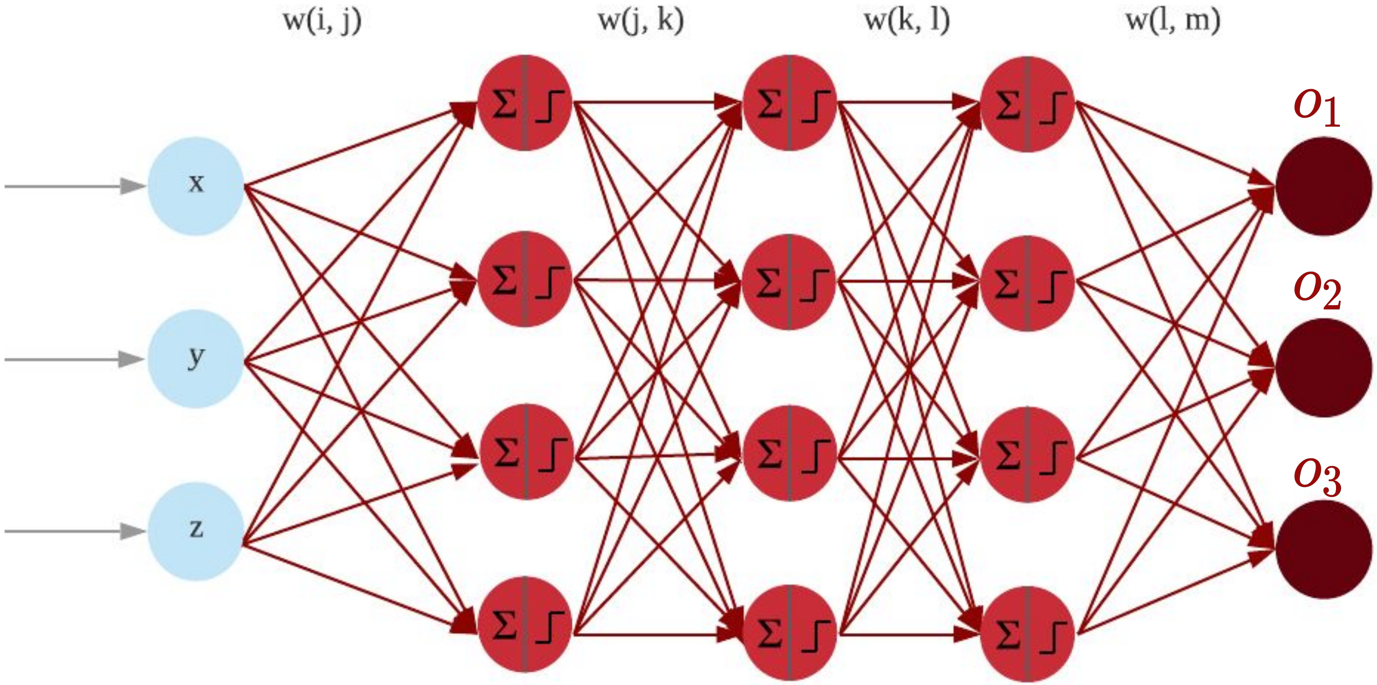
# Forward Pass



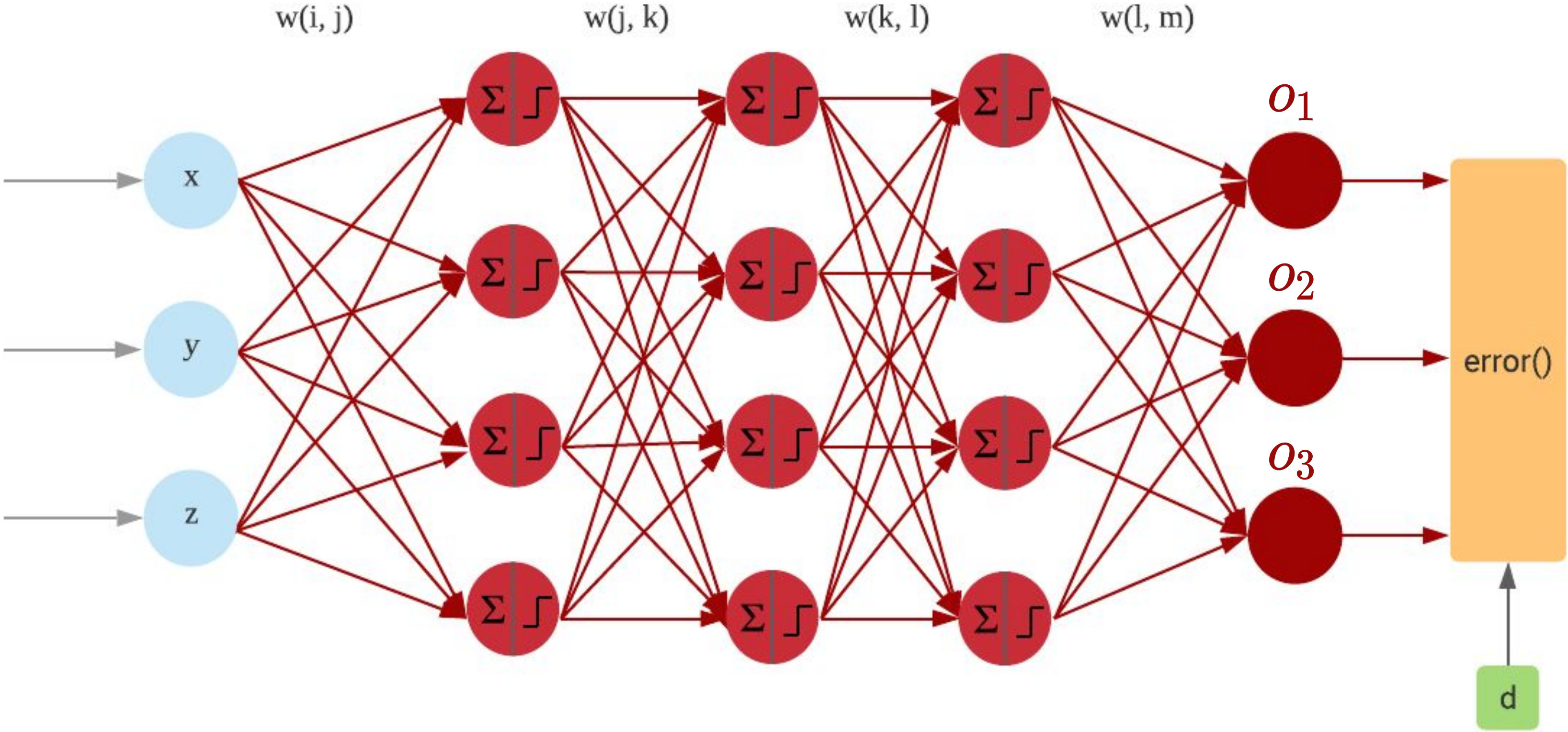
# Forward Pass



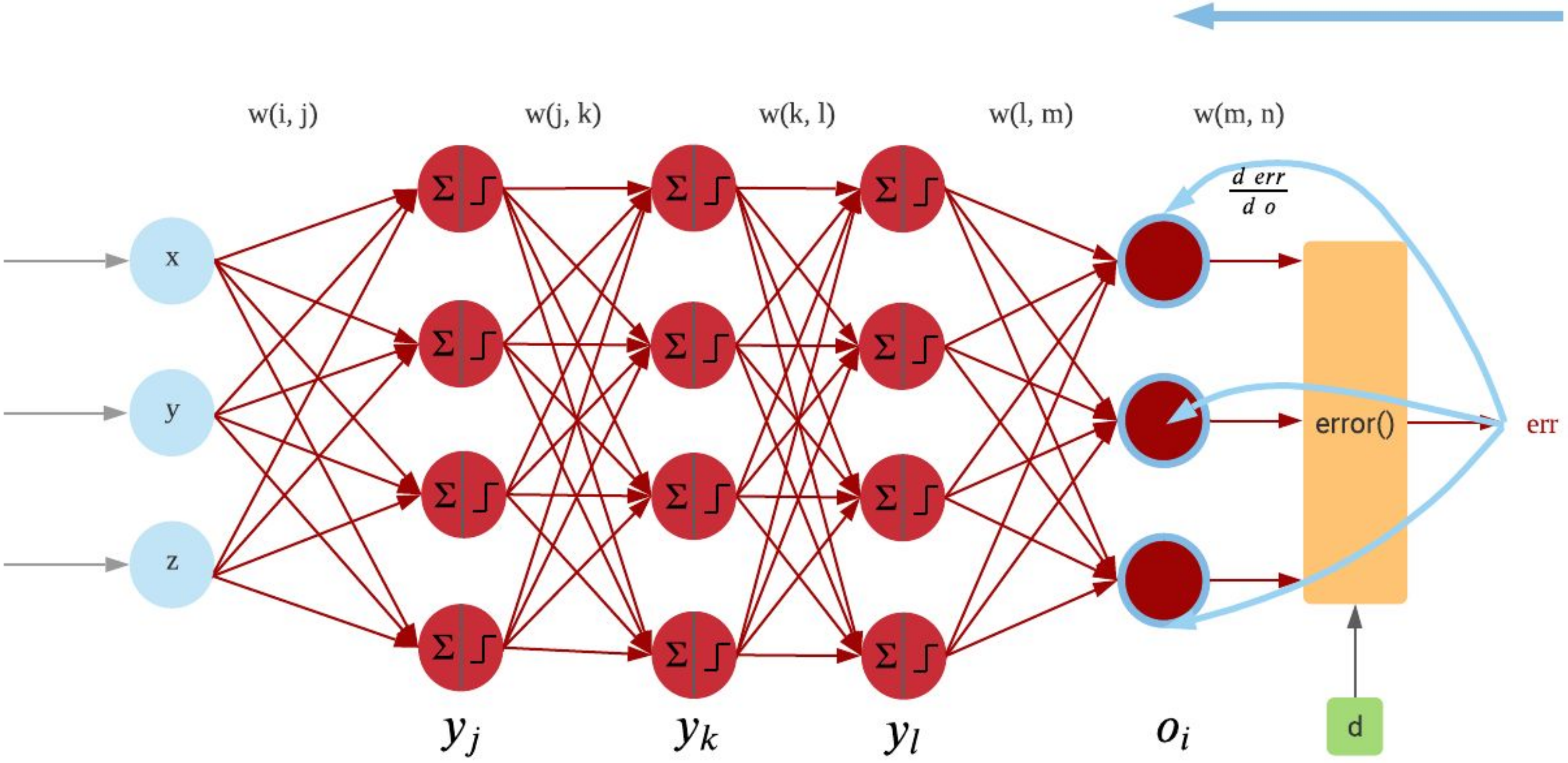
# Forward Pass



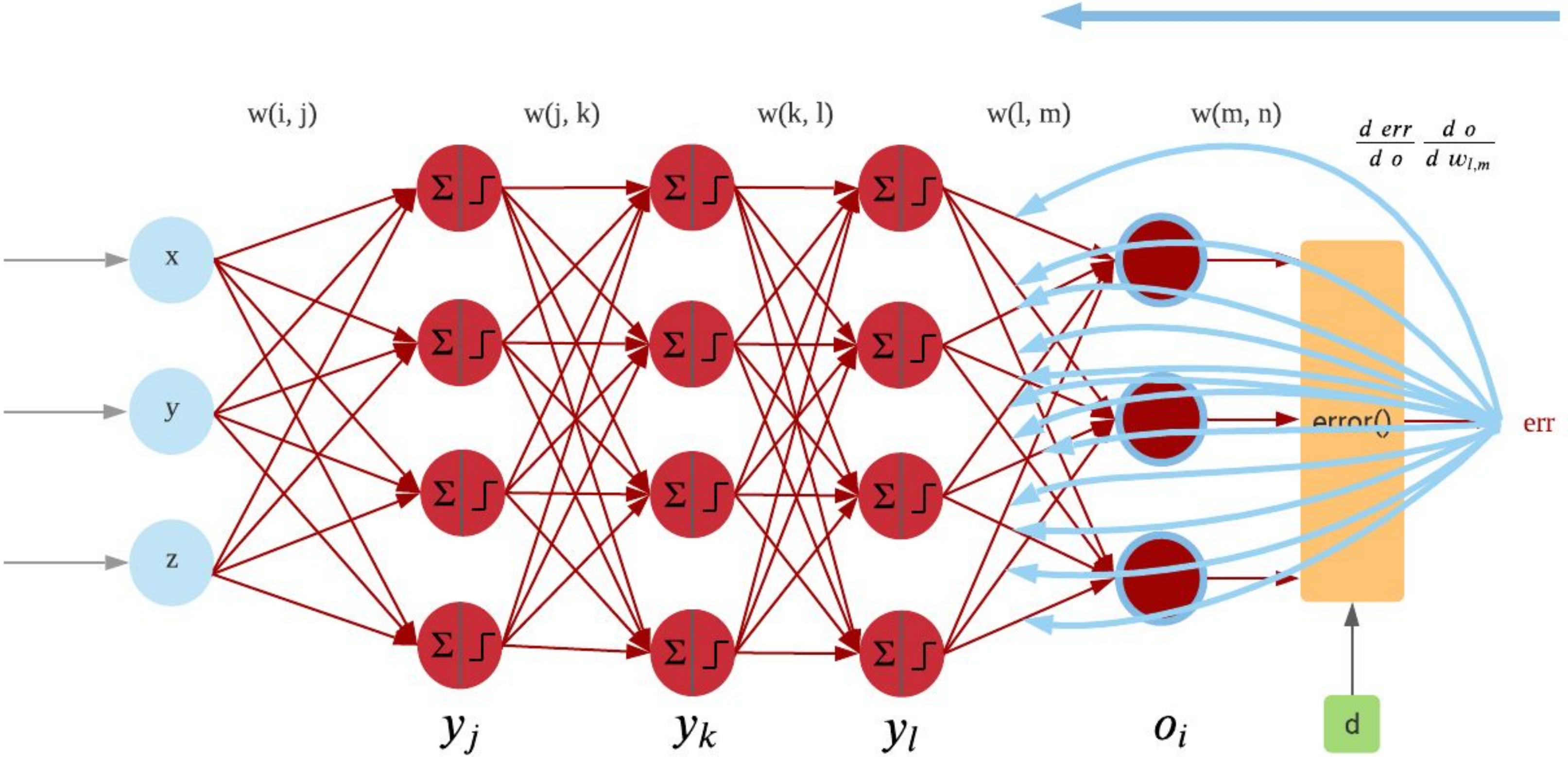
# Error



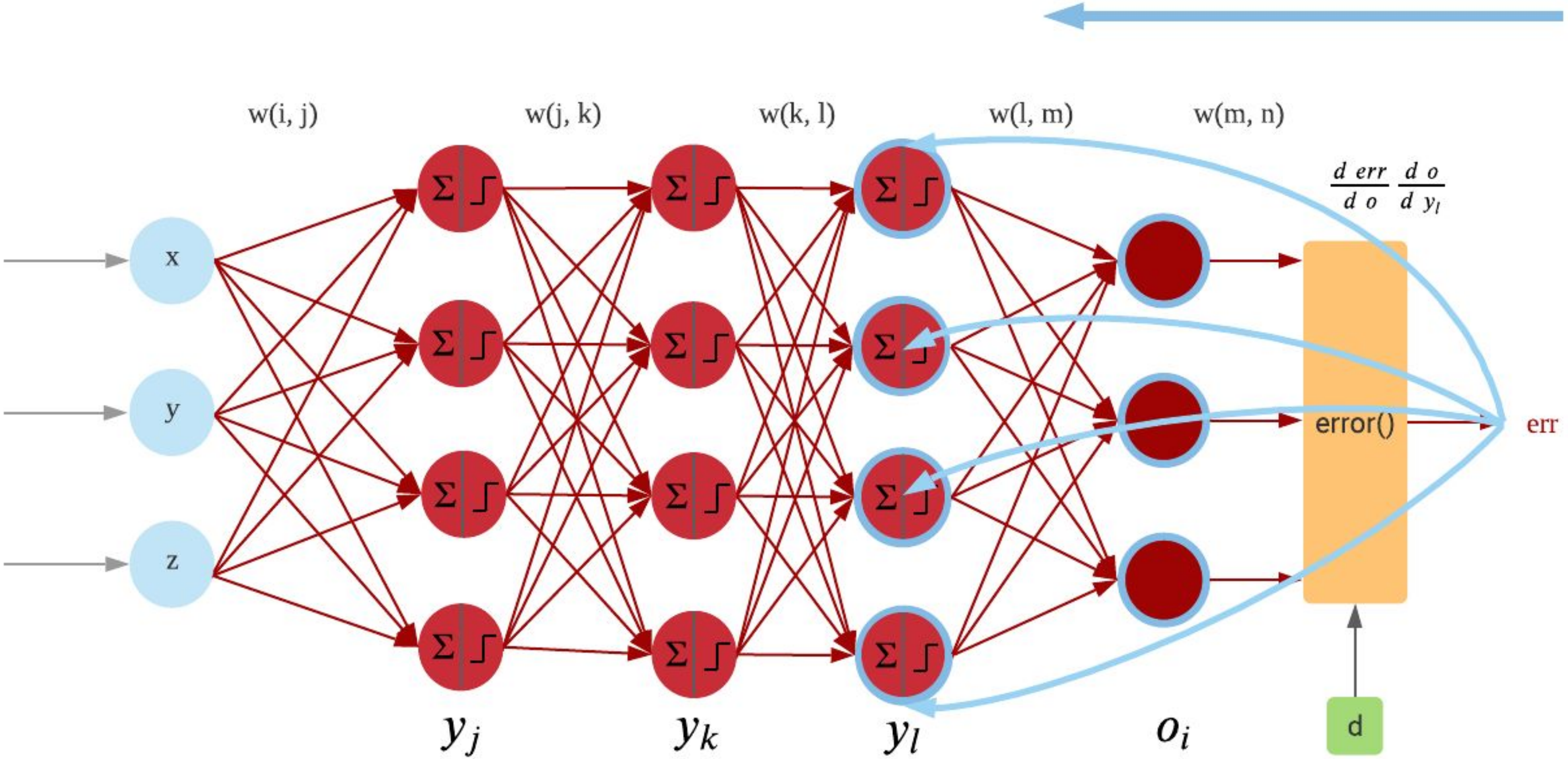
# Backpropagation



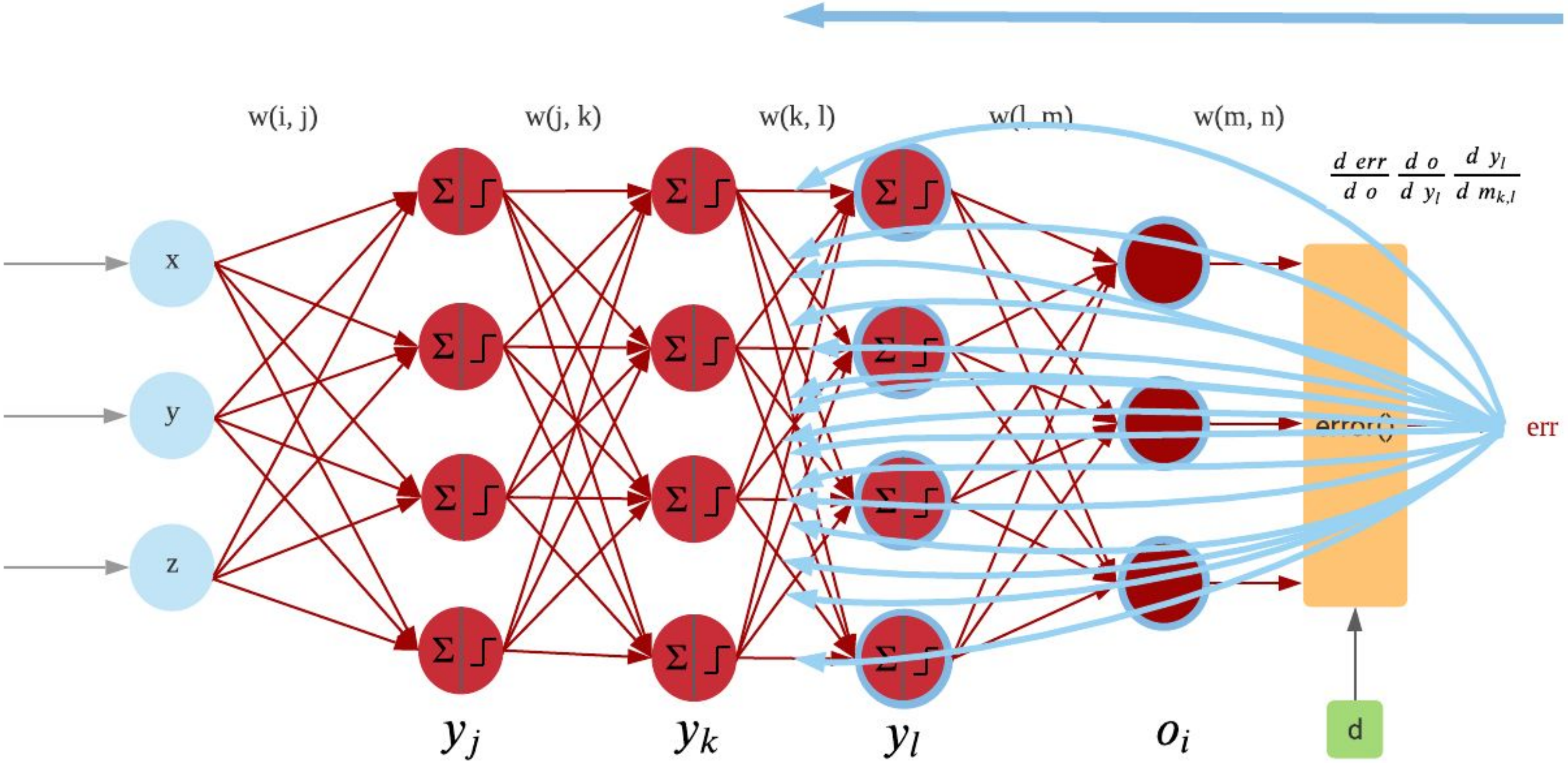
# Backpropagation



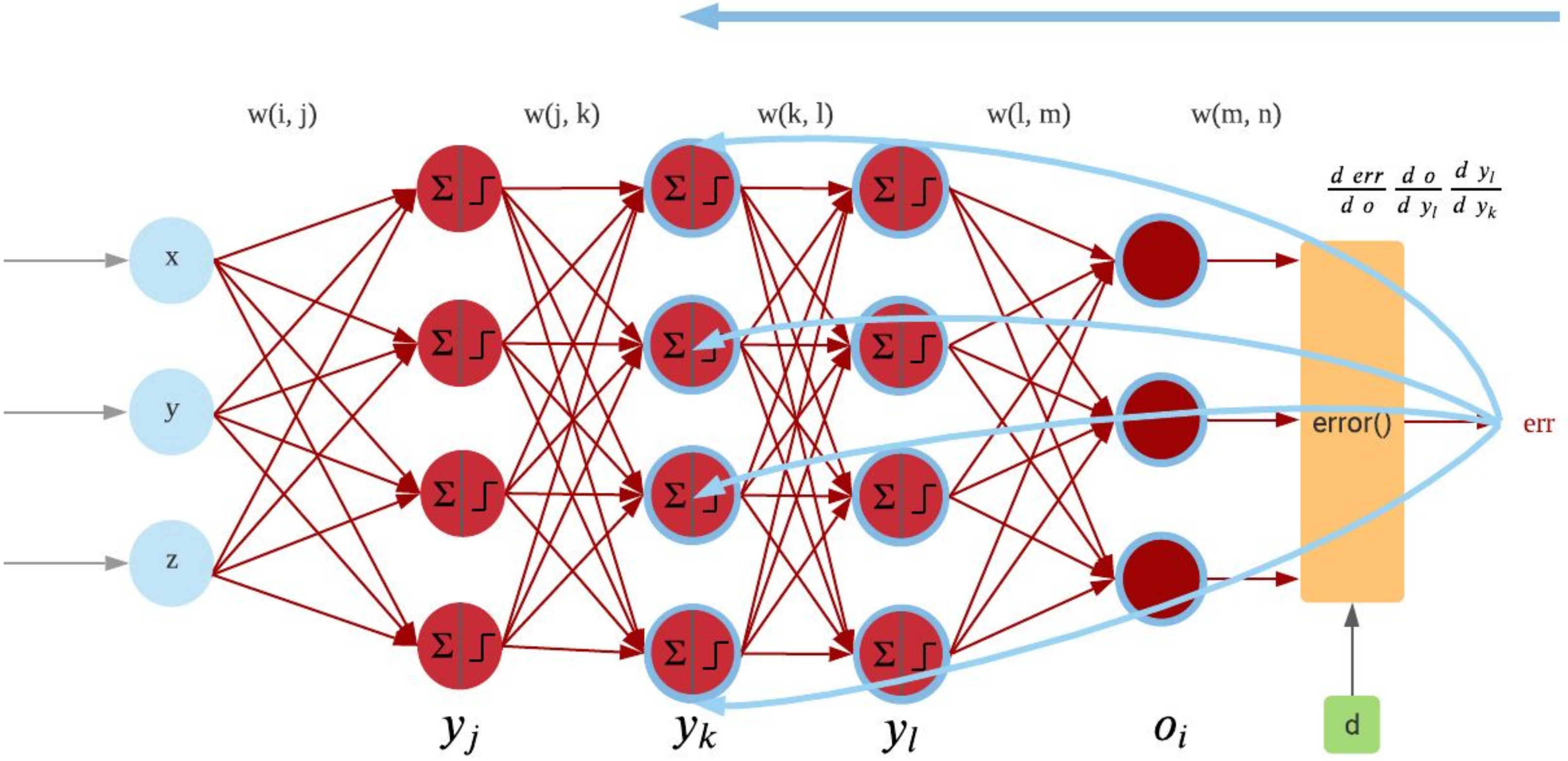
# Backpropagation



# Backpropagation



# Backpropagation



# Backpropagation

……

# Backpropagation

All gradients of weights w.r.t error are calculated!

# Update Weights

$$W \leftarrow W - \eta \cdot \nabla_W Loss(W)$$

learning
rate

gradient

# And finally …

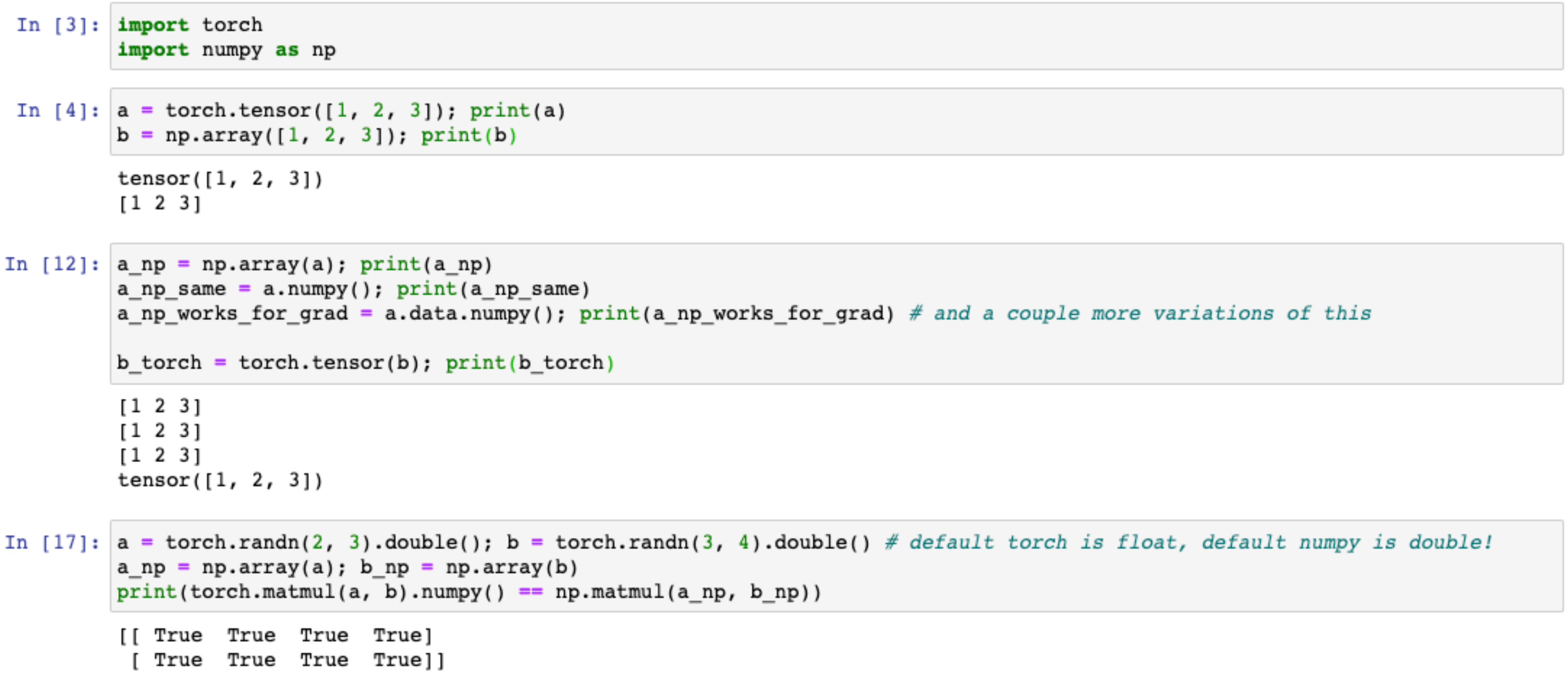
The coding part !

# Summary

1. Tensors
2. CPU & GPU operations
3. Components of Training a Model
    a) Data
    b) Model
    c) Loss Function
    d) Backpropagation
    e) Optimizer

# 1) Tensors

- Pytorch's tensors are very similar to numpy's ndarrays



```
In [3]: import torch
        import numpy as np
```

```
In [4]: a = torch.tensor([1, 2, 3]); print(a)
        b = np.array([1, 2, 3]); print(b)
```

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

```
In [12]: a_np = np.array(a); print(a_np)
         a_np_same = a.numpy(); print(a_np_same)
         a_np_works_for_grad = a.data.numpy(); print(a_np_works_for_grad) # and a couple more variations of this

         b_torch = torch.tensor(b); print(b_torch)
```

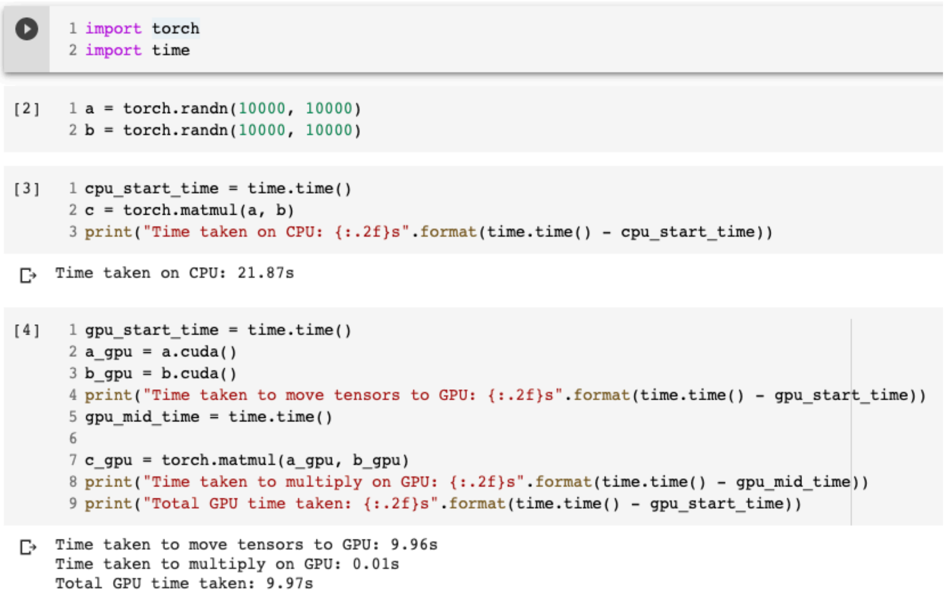
```
[1 2 3]
[1 2 3]
[1 2 3]
tensor([1, 2, 3])
```

```
In [17]: a = torch.randn(2, 3).double(); b = torch.randn(3, 4).double() # default torch is float, default numpy is double!
         a_np = np.array(a); b_np = np.array(b)
         print(torch.matmul(a, b).numpy() == np.matmul(a_np, b_np))
```

```
[[ True  True  True  True]
 [ True  True  True  True]]
```

# 2) CPU & GPU Operations

- The main difference with pytorch’s tensors is that you can perform operations on the GPU as opposed to the CPU.
- Performing tensor (matrix) operations on the GPU is often much faster than working on the CPU



```
import torch
import time
```

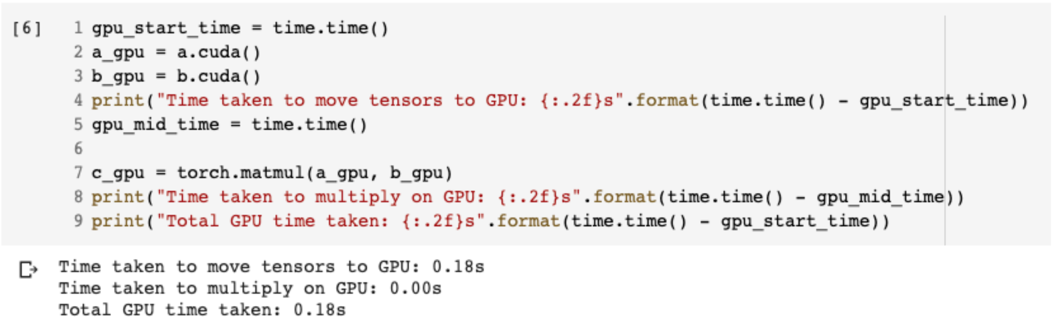
```
[2]  a = torch.randn(10000, 10000)
     b = torch.randn(10000, 10000)
```

```
[3]  cpu_start_time = time.time()
     c = torch.matmul(a, b)
     print("Time taken on CPU: {:.2f}s".format(time.time() - cpu_start_time))
```

```
Time taken on CPU: 21.87s
```

```
[4]  gpu_start_time = time.time()
     a_gpu = a.cuda()
     b_gpu = b.cuda()
     print("Time taken to move tensors to GPU: {:.2f}s".format(time.time() - gpu_start_time))
     gpu_mid_time = time.time()

     c_gpu = torch.matmul(a_gpu, b_gpu)
     print("Time taken to multiply on GPU: {:.2f}s".format(time.time() - gpu_mid_time))
     print("Total GPU time taken: {:.2f}s".format(time.time() - gpu_start_time))
```

```
Time taken to move tensors to GPU: 9.96s
Time taken to multiply on GPU: 0.01s
Total GPU time taken: 9.97s
```

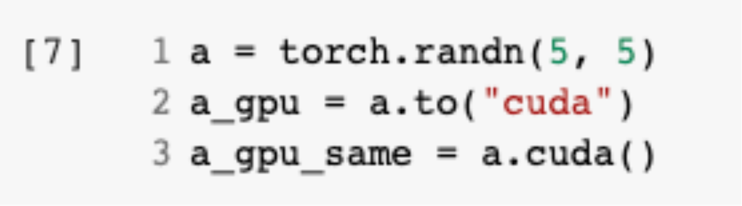


```
[6]  gpu_start_time = time.time()
     a_gpu = a.cuda()
     b_gpu = b.cuda()
     print("Time taken to move tensors to GPU: {:.2f}s".format(time.time() - gpu_start_time))
     gpu_mid_time = time.time()

     c_gpu = torch.matmul(a_gpu, b_gpu)
     print("Time taken to multiply on GPU: {:.2f}s".format(time.time() - gpu_mid_time))
     print("Total GPU time taken: {:.2f}s".format(time.time() - gpu_start_time))
```

```
Time taken to move tensors to GPU: 0.18s
Time taken to multiply on GPU: 0.00s
Total GPU time taken: 0.18s
```

# 2) CPU & GPU Operations

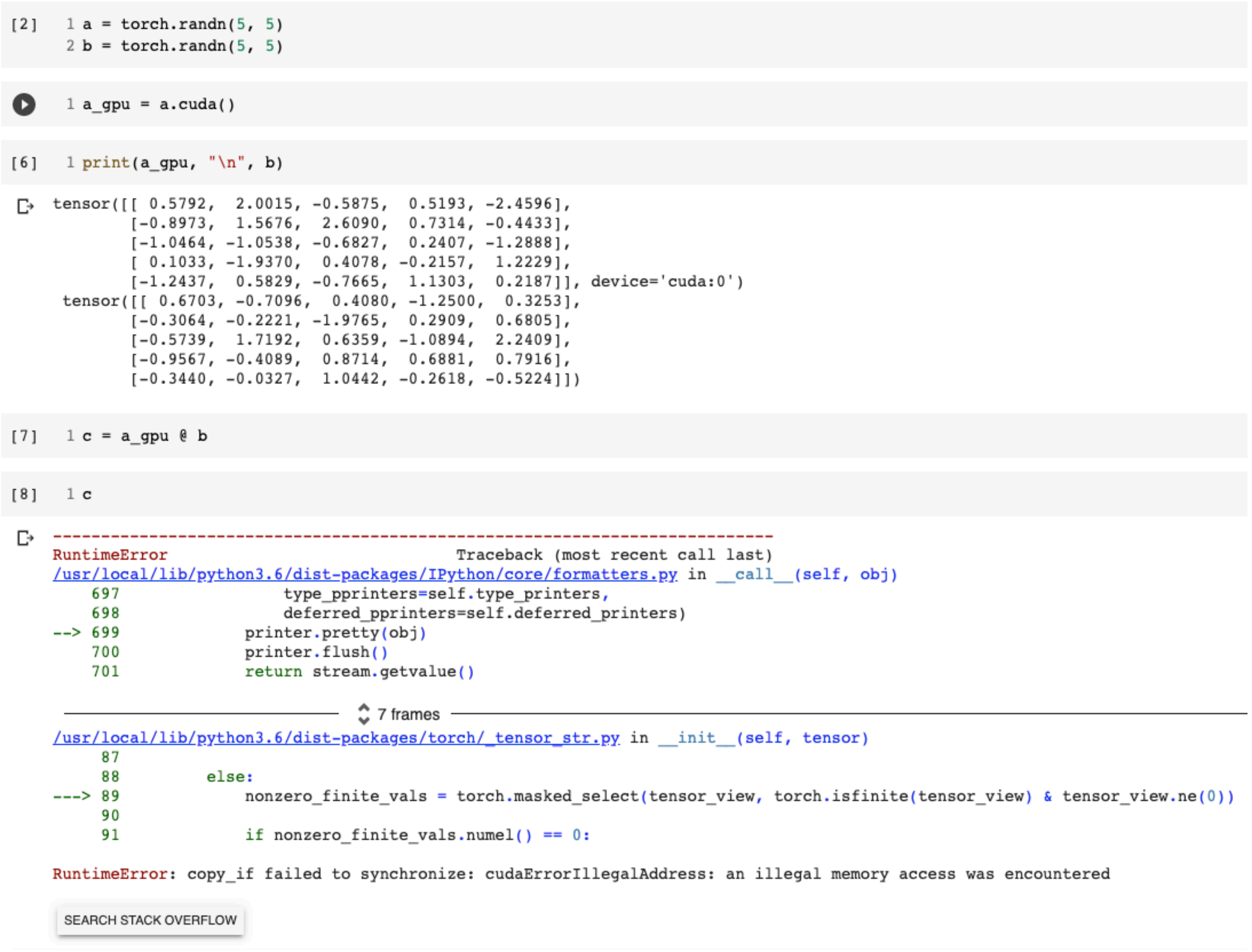
- To do GPU tensor operations, you must first **move** the tensor from the CPU to the GPU



```
a = torch.randn(5, 5)
a_gpu = a.to("cuda")
a_gpu_same = a.cuda()
```

- Operations require all components to be on the same device (CPU or GPU). Operations between CPU and GPU tensors will fail

# 2) CPU & GPU Operations Failure



```
[2]  a = torch.randn(5, 5)
     b = torch.randn(5, 5)
```

```
a_gpu = a.cuda()
```

```
[6]  print(a_gpu, "\n", b)
```

```
tensor([[ 0.5792,  2.0015, -0.5875,  0.5193, -2.4596],
        [-0.8973,  1.5676,  2.6090,  0.7314, -0.4433],
        [-1.0464, -1.0538, -0.6827,  0.2407, -1.2888],
        [ 0.1033, -1.9370,  0.4078, -0.2157,  1.2229],
        [-1.2437,  0.5829, -0.7665,  1.1303,  0.2187]], device='cuda:0')
 tensor([[ 0.6703, -0.7096,  0.4080, -1.2500,  0.3253],
        [-0.3064, -0.2221, -1.9765,  0.2909,  0.6805],
        [-0.5739,  1.7192,  0.6359, -1.0894,  2.2409],
        [-0.9567, -0.4089,  0.8714,  0.6881,  0.7916],
        [-0.3440, -0.0327,  1.0442, -0.2618, -0.5224]])
```

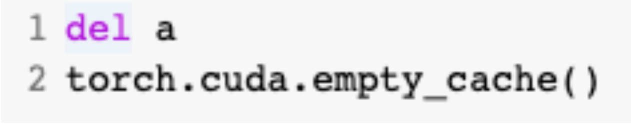
```
[7]  c = a_gpu @ b
```

```
[8]  c
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
/usr/local/lib/python3.6/dist-packages/IPython/core/formatters.py in __call__(self, obj)
    697                 type_pprinters=self.type_printers,
    698                 deferred_pprinters=self.deferred_printers)
--> 699             printer.pretty(obj)
    700             printer.flush()
    701             return stream.getvalue()

                                7 frames
/usr/local/lib/python3.6/dist-packages/torch/_tensor_str.py in __init__(self, tensor)
     87
     88         else:
---> 89             nonzero_finite_vals = torch.masked_select(tensor_view, torch.isfinite(tensor_view) & tensor_view.ne(0))
     90
     91             if nonzero_finite_vals.numel() == 0:

RuntimeError: copy_if failed to synchronize: cudaErrorIllegalAddress: an illegal memory access was encountered
```

# 2) CPU & GPU Operations

- Things to keep in mind:
  - A GPU operation’s runtime comes in two parts: 1) time taken to move a tensor to GPU, 2) time taken for an operation.
    - #2 is very fast on GPU, but sometimes (for small operations), #1 can take much longer. In some cases, it may be faster to perform a certain operation on CPU.
  - GPU memory is quite limited. You will frequently run into the following error:
    - RuntimeError: CUDA out of memory. Tried to allocate 12.50 MiB (GPU 0; 10.92 GiB total capacity; 8.57 MiB already allocated; 9.28 GiB free; 4.68 MiB cached)
    - When this happens, either reduce the batch size or check if there are any dangling unused tensors left on the GPU. You can delete tensors on the GPU and free memory with:



```
del a
torch.cuda.empty_cache()
```

# 2.5) Interruption: Debugging

- I’m going to go through some slides on debugging.
- Although what you are learning from this course is DL, what you will actually be spending most of your time doing is debugging.
- So, it is important to at least talk about it.

# 2.5) Debugging

- This is what you should do when your code breaks:
    1. Look at the error message, expand the list of commands (trace) that led to the error. Then, **go to each of those lines in your code** and see if you can find a problem.
        1. Note: A lot of the trace will not be your code (they will be packages). You should **read the documentation** and see if you are using those correctly.
    2. If you cannot fix the issue, you should **google your problem**. You should read through all of the forums (stackoverflow, pytorch, github issues) and try the suggestions. If you have no clue what the people are saying, **go study the documentation – that's how they learned what they know**.
        1. You might think this is a waste of time, but it's not. This is how you learn to work with new packages/large codebases.
    3. Outside of a class setting, this last step would not exist. You would repeat step 2 or submit your own github issue until the problem is resolved. **However**, since this is a class, you can go on Piazza/go to OH at this step. Only do this if you have actually tried a lot of #2. Generally, there are a lot of **common issues** that TAs will recognize (because we have gone online and looked for this exact error). In these cases, we will tell you to go look up solutions.

# 2.5) CPU & GPU Operations Debugging

- You’ll be running into Cuda errors like:
  - RuntimeError: CUDA error: device-side assert triggered
- This can mean many things. For example:
  - You did an operation between CPU and GPU tensors
  - You did GPU operations between tensors of unexpected shape
    - Likely the most common cause
  - Your types were wrong in some weird way
    - Long when it expects a Float or vice versa is most common.

# 2.5) CPU & GPU Operations Debugging

- However, the line(s) the traceback shows **may NOT be the actual source of error**
  - Because GPU operations are parallelized, and debugging is hard when stuff is run in parallel
- You should try running the entire thing again after setting the following environment variable:
  - CUDA_LAUNCH_BLOCKING=1
- This will force CUDA to do things sequentially, which is more likely to give you a better traceback.
- **Remember to turn this back to** CUDA_LAUNCH_BLOCKING=0 **after**
  - Otherwise your code will be slow.

# 2.5) CPU & GPU Operations Debugging

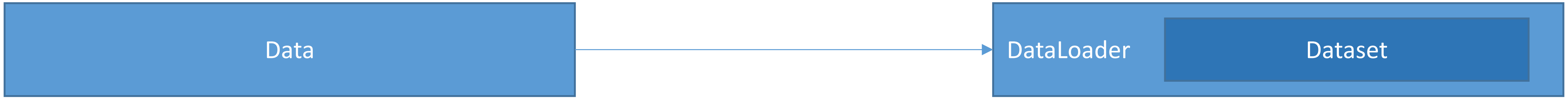
- Some notes on working with Colab
- Colab is great – it’s free.
- However, you will run into some issues that give you a headache. In a prior slide, note that the operation b/n cpu and gpu tensors did not throw an error when you performed it, but afterwards on another command.
- When you get a device-side assert, you probably have to restart the runtime instance.
- If your nvidia-smi shows clogged gpu memory, you have to restart the runtime instance
- …among other issues. Just take note – Colab is free, but you are trading your own time for it.

# 3) Components of Training a Model

- Now we get to training a model in pytorch.
- Colloquially, training a model can be described like this:
  1. We get data – pairs of questions and answers.
  2. For a pair (x, y), we run x through the model to get the model’s answer $\overline{y}$.
  3. Then, a “teacher” gives the model a grade depending on “how wrong” $\overline{y}$ is compared to the true answer y.
  4. Then based on the grade, we figure out who’s fault the error is.
  5. Then, we fix the faults so the model can do better next time.

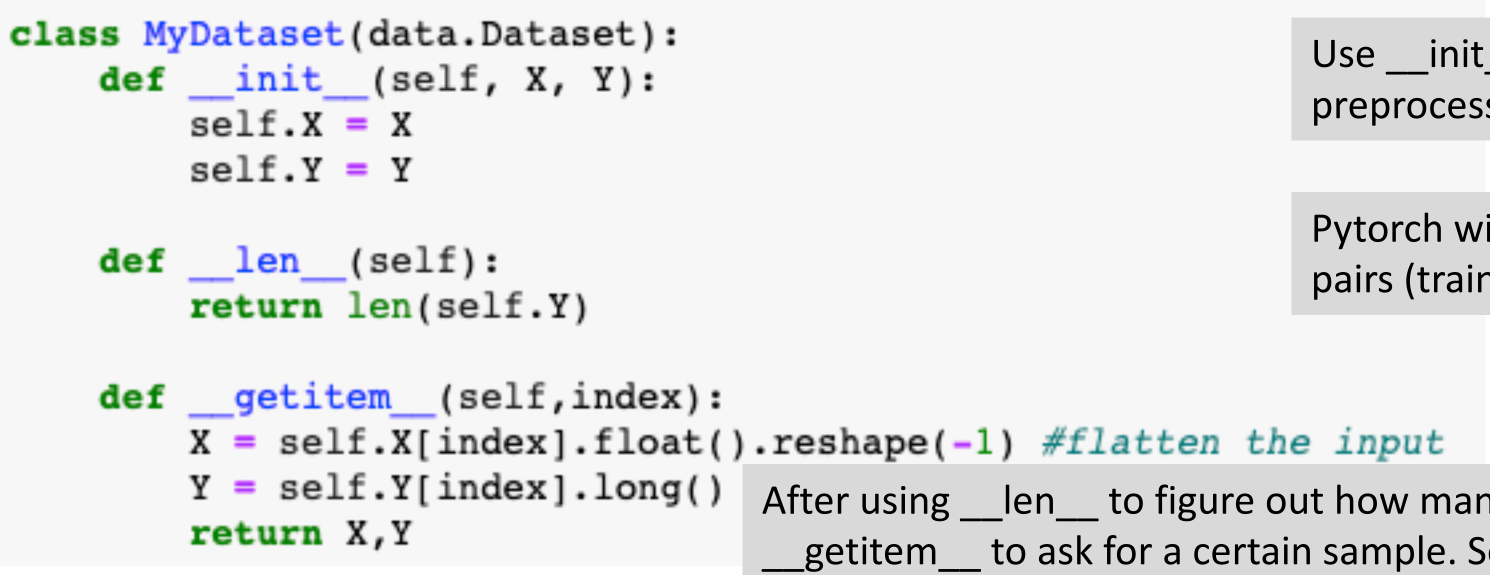
# 3) Training a Model: [Data]

- When training a model, data is generally a long list of (x, y) pairs, where you want the model to see x and predict y.
- Pytorch has two classes you will need to use to deal with data:
  - torch.utils.data.Dataset
  - torch.utils.data.DataLoader
- Dataset class is used to preprocess data and load single pairs (x, y)
- DataLoader class uses your Dataset class to get single pairs and group them into **batches**



# 3) Training a Model: [Dataset]

- When defining a Dataset, there are three class methods that you need to implement: __init__, __len__, __getitem__



```
class MyDataset(data.Dataset):
    def __init__(self, X, Y):
        self.X = X
        self.Y = Y

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

    def __getitem__(self,index):
        X = self.X[index].float().reshape(-1) #flatten the input
        Y = self.Y[index].long()
        return X,Y
```

Use __init__ to load in the data to the class (or preprocess) so it can be accessed later

Pytorch will use __len__ to know how many (x, y) pairs (training samples) are in your dataset

After using __len__ to figure out how many samples there are, pytorch will use __getitem__ to ask for a certain sample. So, __getitem__(i) should return the "i-th" sample, with order chosen by you. You should use __getitem__ to do some final processing on the data before it's sent out.
Caution: __getitem__ will be called maybe millions of times, so make sure you do **as little work** in here as possible for fast code. Try to keep heavy preprocessing in __init__, which is only called once.

# 3) Training a Model: [DataLoader]

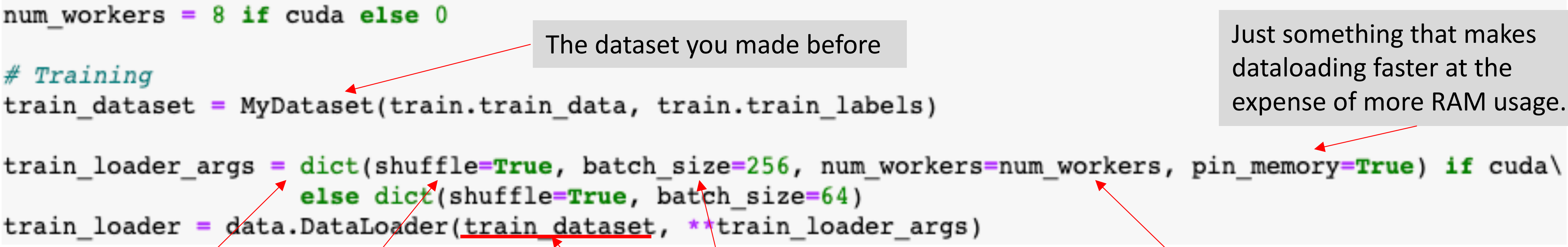
Data

Model

Loss Function

Backpropagation

Optimizer



```
num_workers = 8 if cuda else 0

# Training
train_dataset = MyDataset(train.train_data, train.train_labels)

train_loader_args = dict(shuffle=True, batch_size=256, num_workers=num_workers, pin_memory=True) if cuda\
                    else dict(shuffle=True, batch_size=64)
train_loader = data.DataLoader(train_dataset, **train_loader_args)
```

The dataset you made before

Just something that makes dataloading faster at the expense of more RAM usage.

These are the arguments we're passing to **Training** DataLoader

We'll be going through the entire dataset multiple times. We want to shuffle the Dataset every single time **for the training dataloader.**
For validation/test, you don't want to shuffle.

How many samples per batch? This is a hyperparameter you want to adjust.

Notice that we give our dataset to the DataLoader so it can use it.

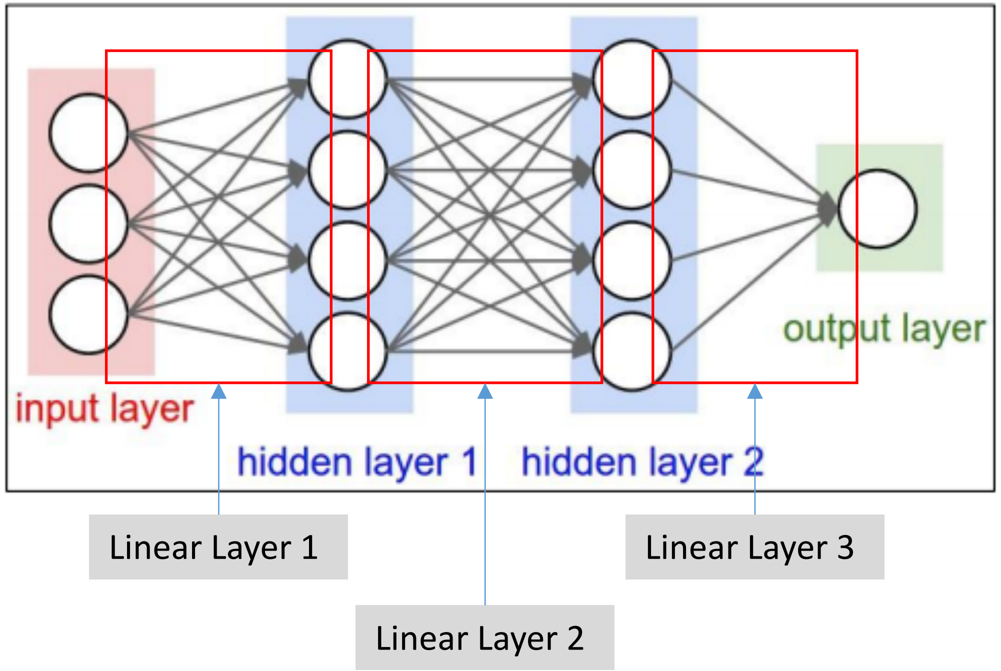
Batches are loaded in parallel – how many workers do you want doing this? Depending on how intensive __getitem__ is, lowering or raising this may speed up dataloading.

# 3) Training a Model: [Model]

- Now, we have our data set up. Next, we need to worry about the model we're going to use.
- This section will be in two parts:
  - How to generate the model you'll use
  - How to run the data sample through the model.

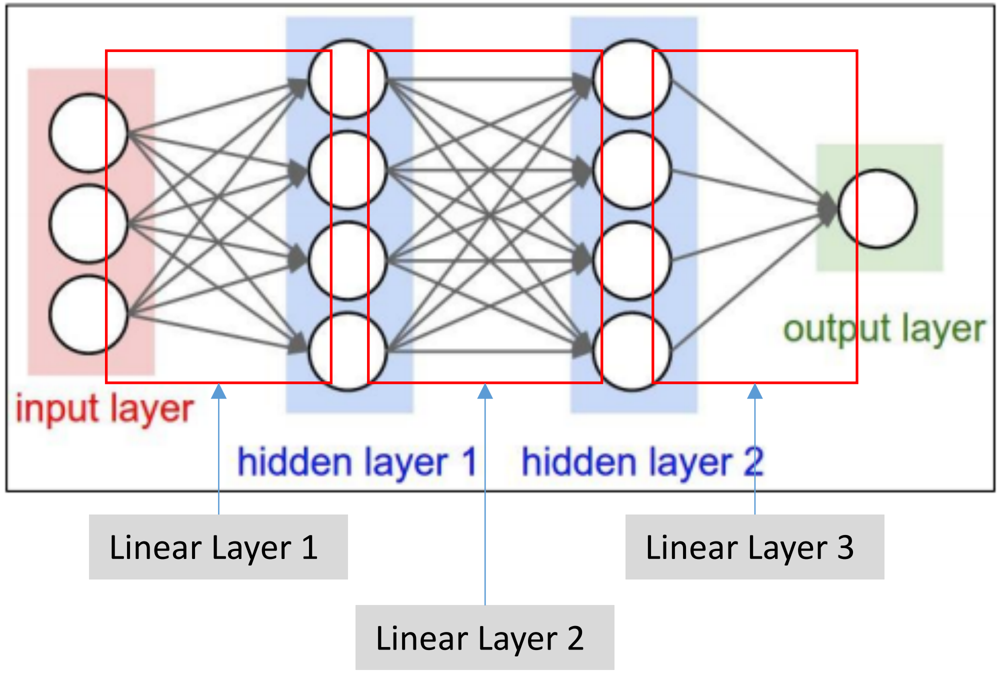
# 3) Training a Model: [Model]

- In class, you went over the Multi-Layer Perceptron (MLP)
- A bunch of these single perceptrons put together are called “Linear layers,” also called “fully-connected (fc)” layers
- Note: a Linear layer is the **connections** between two layers as shown on the right. The layers themselves is an input vector being run through the network.
  - So, the network on the right has **3 layers**



# 3) Training a Model: [Model]

- A key thing in neural networks is **modularity**
- The network on the right can be broken down into 3 essentially same components – Linear layers that differ only in the # of in/out features.
- When coding a network, break down the structure into small parts and take it step by step.
    - This is also a fundamental trend in ML nowadays – using the same structure stacked with varying feature #s



# 3) Training a Model: [Model]

- Now, let’s get into coding a model in pytorch.
- Networks in pytorch are (generally) classes that are based off of the nn.Module class.
- Similar to the Dataset class, pytorch wants you to implement the __init__ and forward methods.
    - __init__: this is where you define the actual model itself (along with other stuff you might need)
    - Forward: given an input x, you run it through the model defined in __init__

Data

Model

Loss Function

Backpropagation

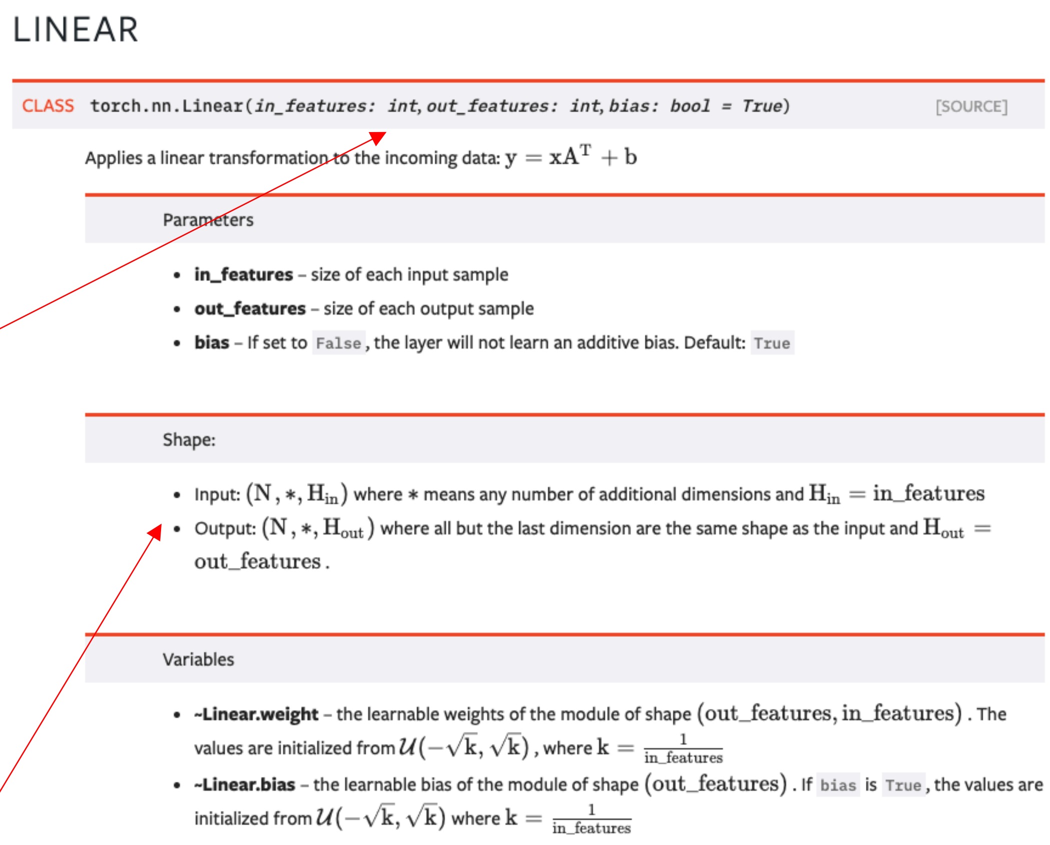
Optimizer

# 3) Training a Model: [Model]

- Pytorch’s nn.Linear class represents a linear layer.
- In fact, nn.Linear is a “model” class itself – it extends nn.Module and has a forward method. We’ll be using this “smaller model” inside our own model. (example of modularity)

nn.Linear takes in in_features and out_features as arguments

nn.Linear takes as input some tensor of shape (N, *, in_features) and outputs
(N, *, out_features).
You can think of this as nn.Linear transforming the last dimension.



LINEAR

CLASS `torch.nn.Linear(in_features: int, out_features: int, bias: bool = True)` [SOURCE]

Applies a linear transformation to the incoming data: $y = xA^T + b$

Parameters

- **in_features** – size of each input sample
- **out_features** – size of each output sample
- **bias** – If set to `False`, the layer will not learn an additive bias. Default: `True`

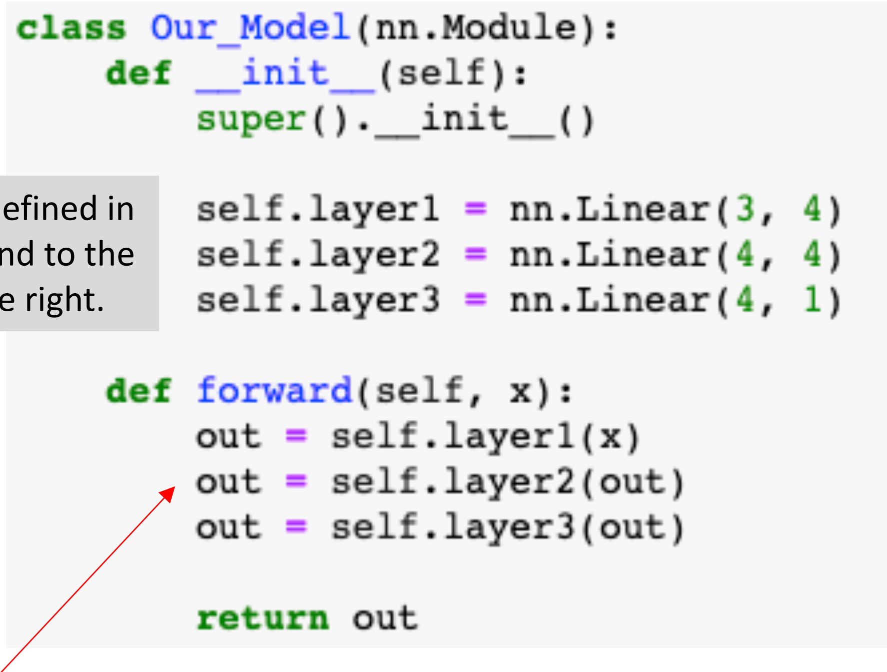
Shape:

- Input: $(N, *, H_{in})$ where $*$ means any number of additional dimensions and $H_{in} = \text{in\_features}$
- Output: $(N, *, H_{out})$ where all but the last dimension are the same shape as the input and $H_{out} = \text{out\_features}$.

Variables

- **~Linear.weight** – the learnable weights of the module of shape $(\text{out\_features}, \text{in\_features})$. The values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$, where $k = \frac{1}{\text{in\_features}}$
- **~Linear.bias** – the learnable bias of the module of shape $(\text{out\_features})$. If `bias` is `True`, the values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{\text{in\_features}}$

# 3) Training a Model: [Model]

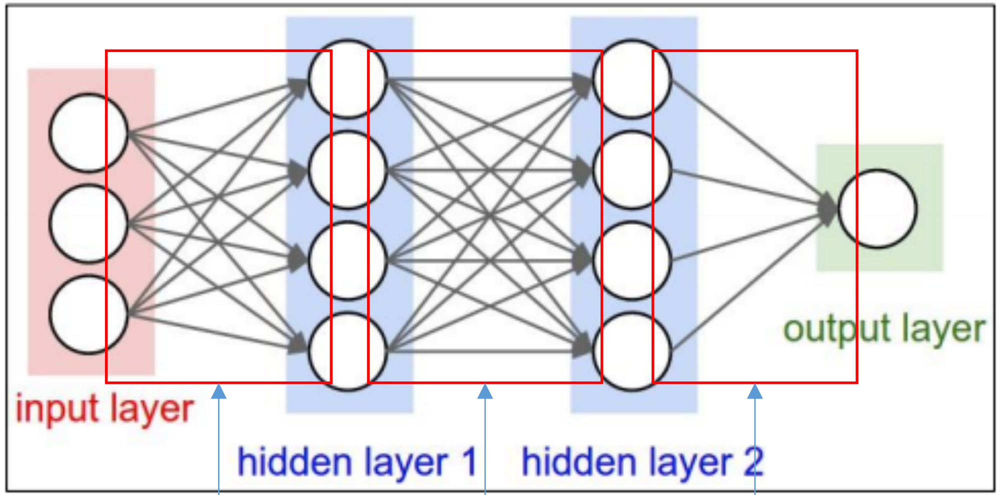


```
class Our_Model(nn.Module):
    def __init__(self):
        super().__init__()

        self.layer1 = nn.Linear(3, 4)
        self.layer2 = nn.Linear(4, 4)
        self.layer3 = nn.Linear(4, 1)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = self.layer3(out)

        return out
```

The three layers defined in __init__ correspond to the three layers on the right.

We run the input x through the layers sequentially, one by one.

Recall: each of the nn.Linear has trainable parameters (weight and bias).
By calling .parameters() on Our_Model, you'll see the weight and bias of each layer thanks to some magic backend.



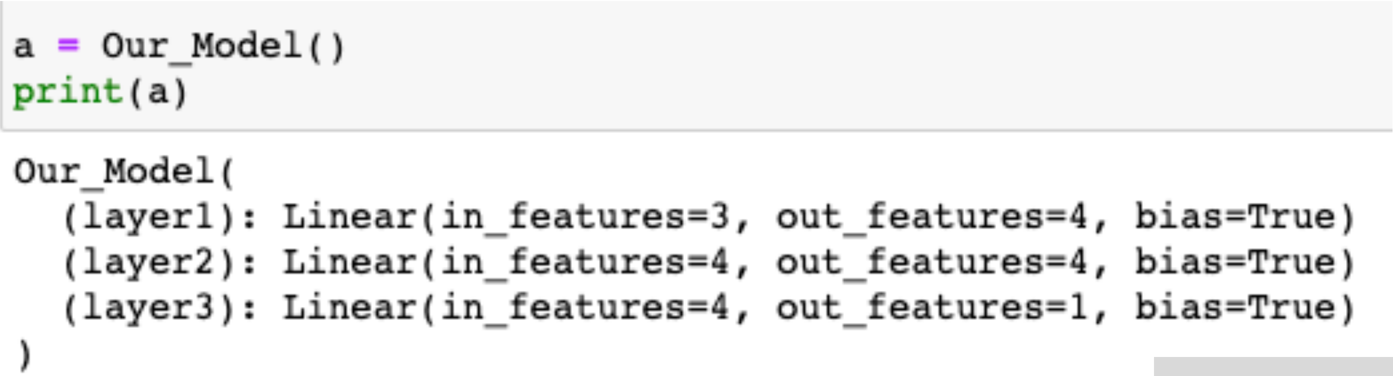
Linear Layer 1

Linear Layer 2

Linear Layer 3

# 3) Training a Model: [Model]

- You can also print the model to see all the components:



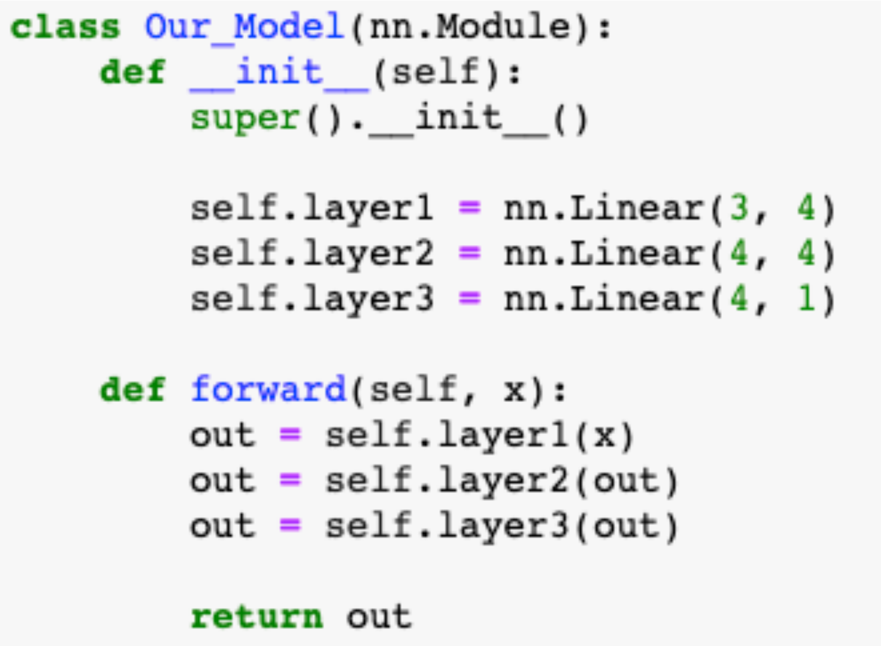
```
a = Our_Model()
print(a)
```

```
Our_Model(
  (layer1): Linear(in_features=3, out_features=4, bias=True)
  (layer2): Linear(in_features=4, out_features=4, bias=True)
  (layer3): Linear(in_features=4, out_features=1, bias=True)
)
```

Caution: These are printed **in order defined in __init__**, and **not** in order **applied in forward()**

# 3) Training a Model: [Model]

- However, it can get annoying to type each of the layers twice – once in __init__ and once in forward.
- Since on the right, we take the output of each layer and directly put it into the next, we can use the **nn.Sequential** module.



```
class Our_Model(nn.Module):
    def __init__(self):
        super().__init__()

        self.layer1 = nn.Linear(3, 4)
        self.layer2 = nn.Linear(4, 4)
        self.layer3 = nn.Linear(4, 1)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = self.layer3(out)

        return out
```

Data

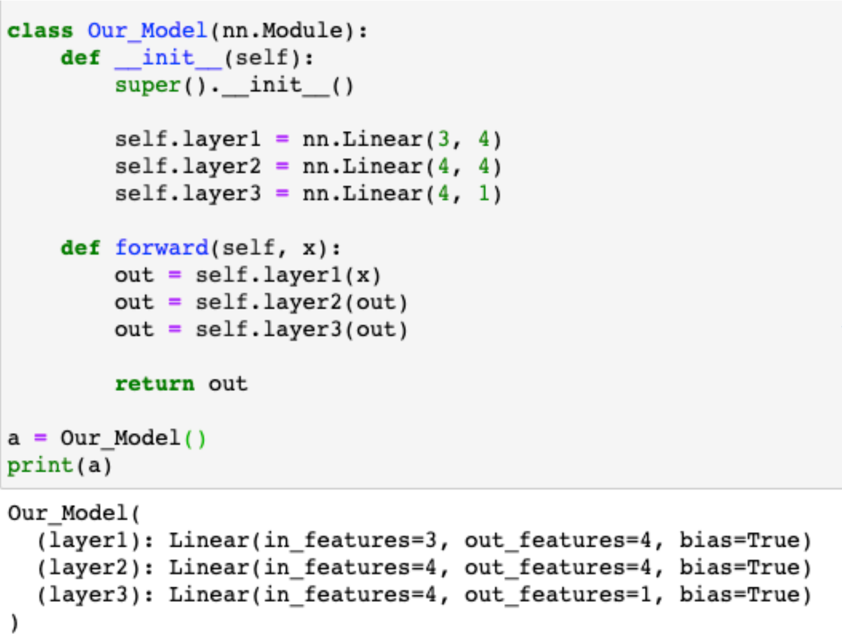
Model

Loss Function

Backpropagation

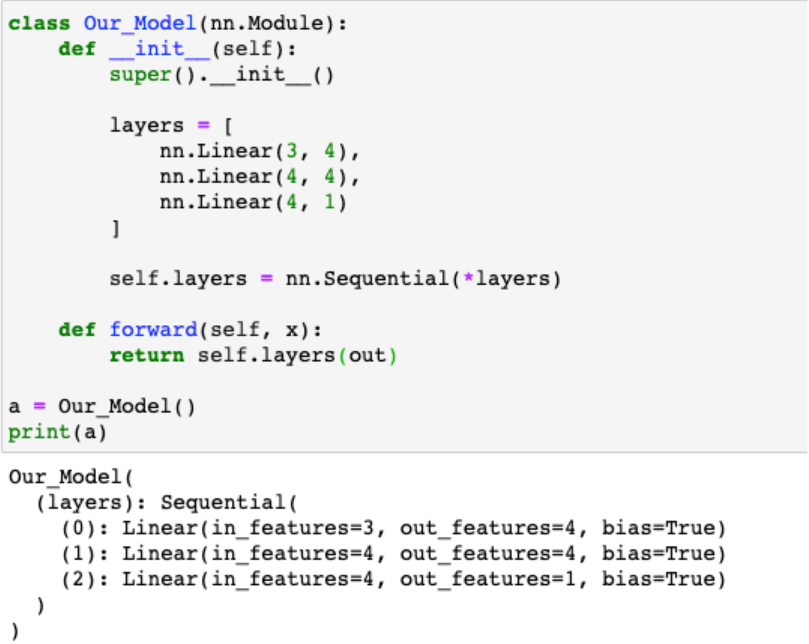
Optimizer

# 3) Training a Model: [Model]



```
class Our_Model(nn.Module):
    def __init__(self):
        super().__init__()

        self.layer1 = nn.Linear(3, 4)
        self.layer2 = nn.Linear(4, 4)
        self.layer3 = nn.Linear(4, 1)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = self.layer3(out)

        return out

a = Our_Model()
print(a)
```

```
Our_Model(
  (layer1): Linear(in_features=3, out_features=4, bias=True)
  (layer2): Linear(in_features=4, out_features=4, bias=True)
  (layer3): Linear(in_features=4, out_features=1, bias=True)
)
```

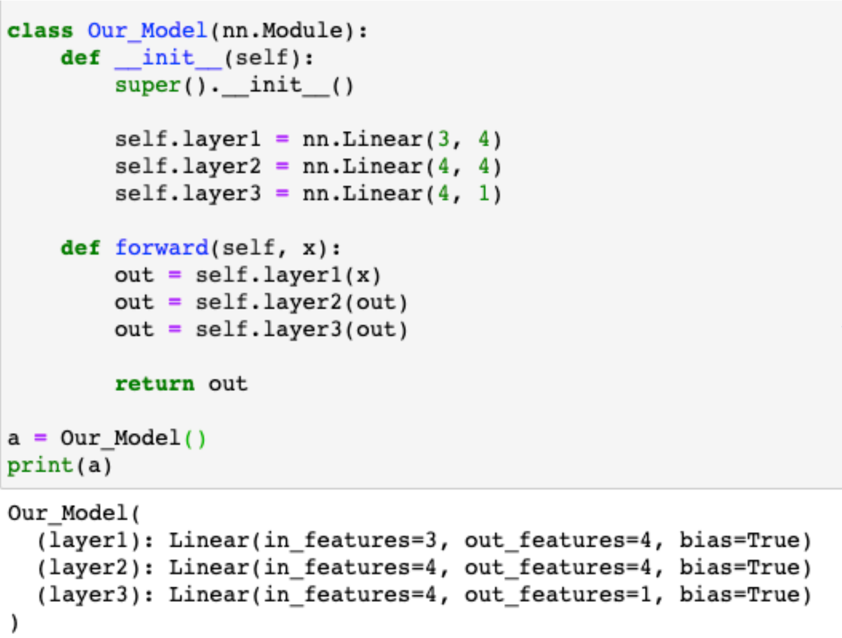


```
class Our_Model(nn.Module):
    def __init__(self):
        super().__init__()

        layers = [
            nn.Linear(3, 4),
            nn.Linear(4, 4),
            nn.Linear(4, 1)
        ]

        self.layers = nn.Sequential(*layers)

    def forward(self, x):
        return self.layers(out)

a = Our_Model()
print(a)
```

```
Our_Model(
  (layers): Sequential(
    (0): Linear(in_features=3, out_features=4, bias=True)
    (1): Linear(in_features=4, out_features=4, bias=True)
    (2): Linear(in_features=4, out_features=1, bias=True)
  )
)
```

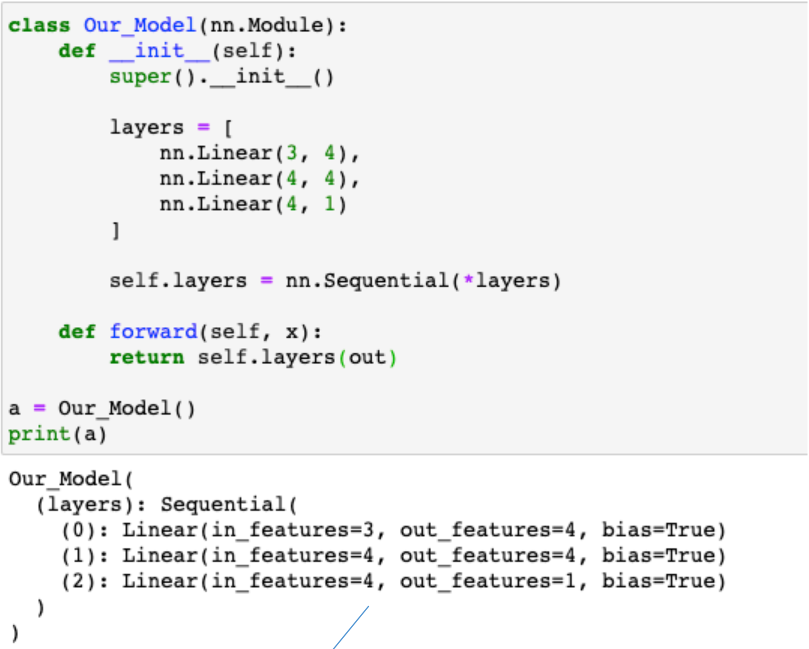
If you're new to python, you might want to look up "args and kwargs" on google to understand what the * operator does. Essentially, it opens up the list and directly puts them in as arguments of nn.Sequential.

# 3) Training a Model: [Model]



```
class Our_Model(nn.Module):
    def __init__(self):
        super().__init__()

        self.layer1 = nn.Linear(3, 4)
        self.layer2 = nn.Linear(4, 4)
        self.layer3 = nn.Linear(4, 1)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = self.layer3(out)

        return out

a = Our_Model()
print(a)
```

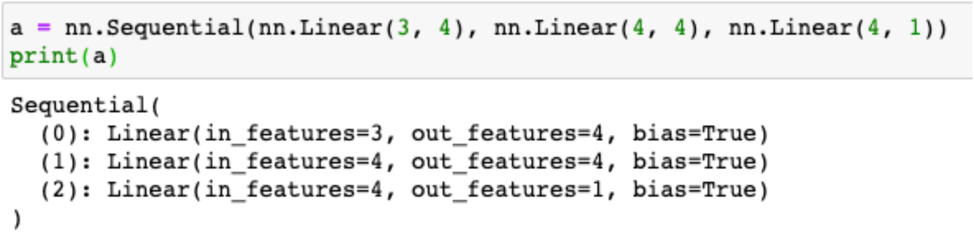
```
Our_Model(
  (layer1): Linear(in_features=3, out_features=4, bias=True)
  (layer2): Linear(in_features=4, out_features=4, bias=True)
  (layer3): Linear(in_features=4, out_features=1, bias=True)
)
```



```
class Our_Model(nn.Module):
    def __init__(self):
        super().__init__()

        layers = [
            nn.Linear(3, 4),
            nn.Linear(4, 4),
            nn.Linear(4, 1)
        ]

        self.layers = nn.Sequential(*layers)

    def forward(self, x):
        return self.layers(out)

a = Our_Model()
print(a)
```

```
Our_Model(
  (layers): Sequential(
    (0): Linear(in_features=3, out_features=4, bias=True)
    (1): Linear(in_features=4, out_features=4, bias=True)
    (2): Linear(in_features=4, out_features=1, bias=True)
  )
)
```

Since nn.Sequential is an nn.Module class anyway, we can make this even simpler:



```
a = nn.Sequential(nn.Linear(3, 4), nn.Linear(4, 4), nn.Linear(4, 1))
print(a)
```

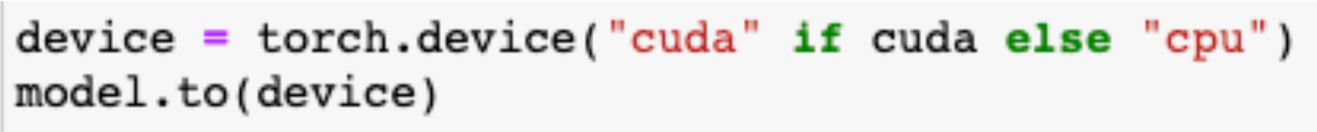
```
Sequential(
  (0): Linear(in_features=3, out_features=4, bias=True)
  (1): Linear(in_features=4, out_features=4, bias=True)
  (2): Linear(in_features=4, out_features=1, bias=True)
)
```

# 3) Training a Model: [Model]

- So far, we only covered the nn.Linear class
- There are many, many more classes in pytorch.
- As a beginner to pytorch, you should definitely have https://pytorch.org/docs/stable/nn.html open. The documentation is very thorough.
- Also, for optimizers: https://pytorch.org/docs/stable/optim.html

# 3) Training a Model: [Model]

- Now that we have our model generated, how do we use it?
- First, we want to put the model on GPU.
- Note that for nn.Module classes, .to(device) is in-place
  - However, for tensors, you must do x = x.to(device)



```
device = torch.device("cuda" if cuda else "cpu")
model.to(device)
```

- Also, models have .train() and .eval() methods.
  - Before training, you should run model.train() to tell the model to save gradients
  - When validating or testing, run model.eval() to tell the model it doesn’t need to save gradients (save memory and time).
  - A common mistake is to forget to toggle back to .train(), then your model doesn’t learn anything.

Data

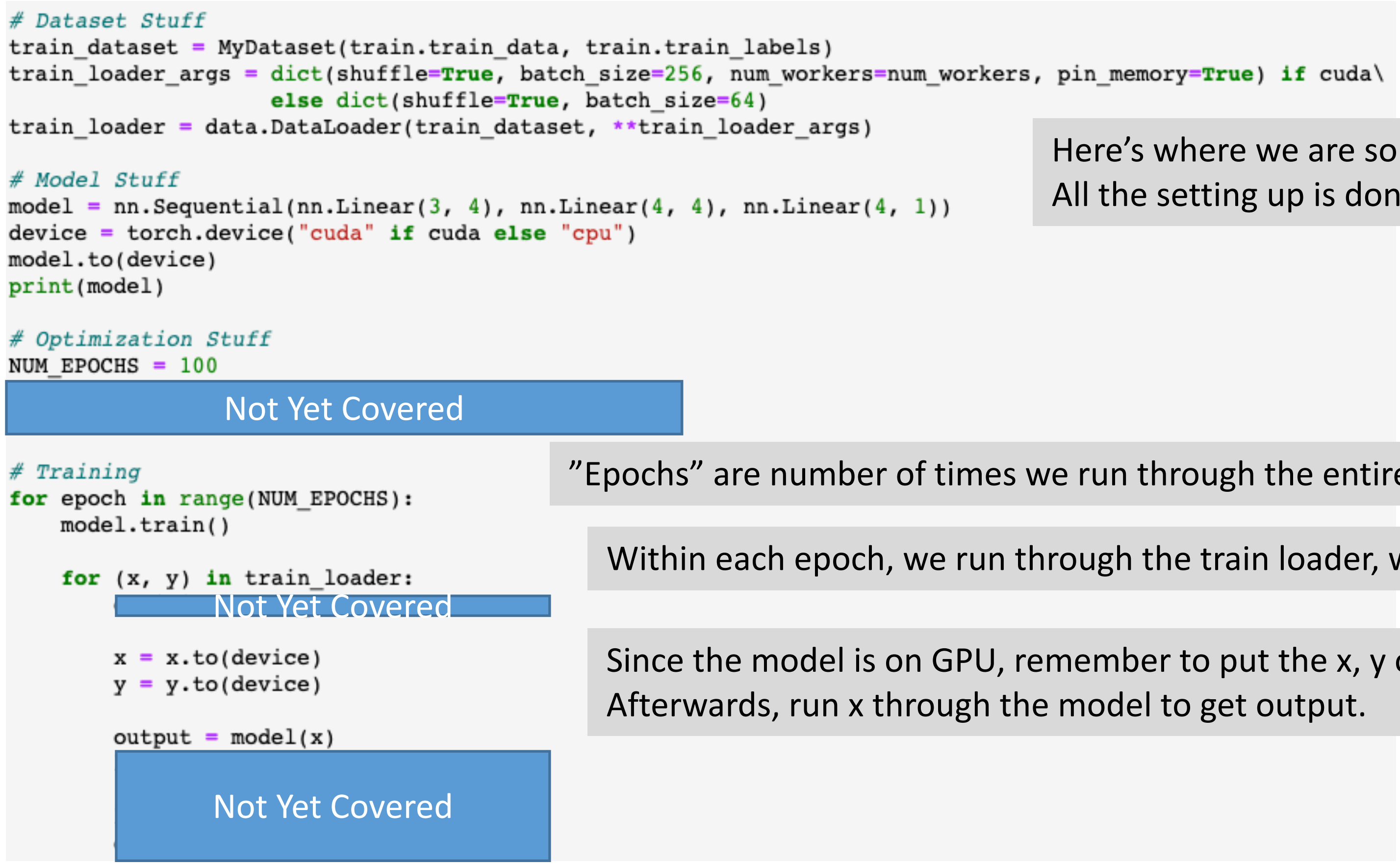
Model

Loss Function

Backpropagation

Optimizer

# 3) Training a Model: [Model]



```
# Dataset Stuff
train_dataset = MyDataset(train.train_data, train.train_labels)
train_loader_args = dict(shuffle=True, batch_size=256, num_workers=num_workers, pin_memory=True) if cuda\
                    else dict(shuffle=True, batch_size=64)
train_loader = data.DataLoader(train_dataset, **train_loader_args)

# Model Stuff
model = nn.Sequential(nn.Linear(3, 4), nn.Linear(4, 4), nn.Linear(4, 1))
device = torch.device("cuda" if cuda else "cpu")
model.to(device)
print(model)

# Optimization Stuff
NUM_EPOCHS = 100
```

Here’s where we are so far.
All the setting up is done up here.

Not Yet Covered

```
# Training
for epoch in range(NUM_EPOCHS):
    model.train()

    for (x, y) in train_loader:
```

Not Yet Covered

```
        x = x.to(device)
        y = y.to(device)

        output = model(x)
```

Not Yet Covered

”Epochs” are number of times we run through the entire dataset.

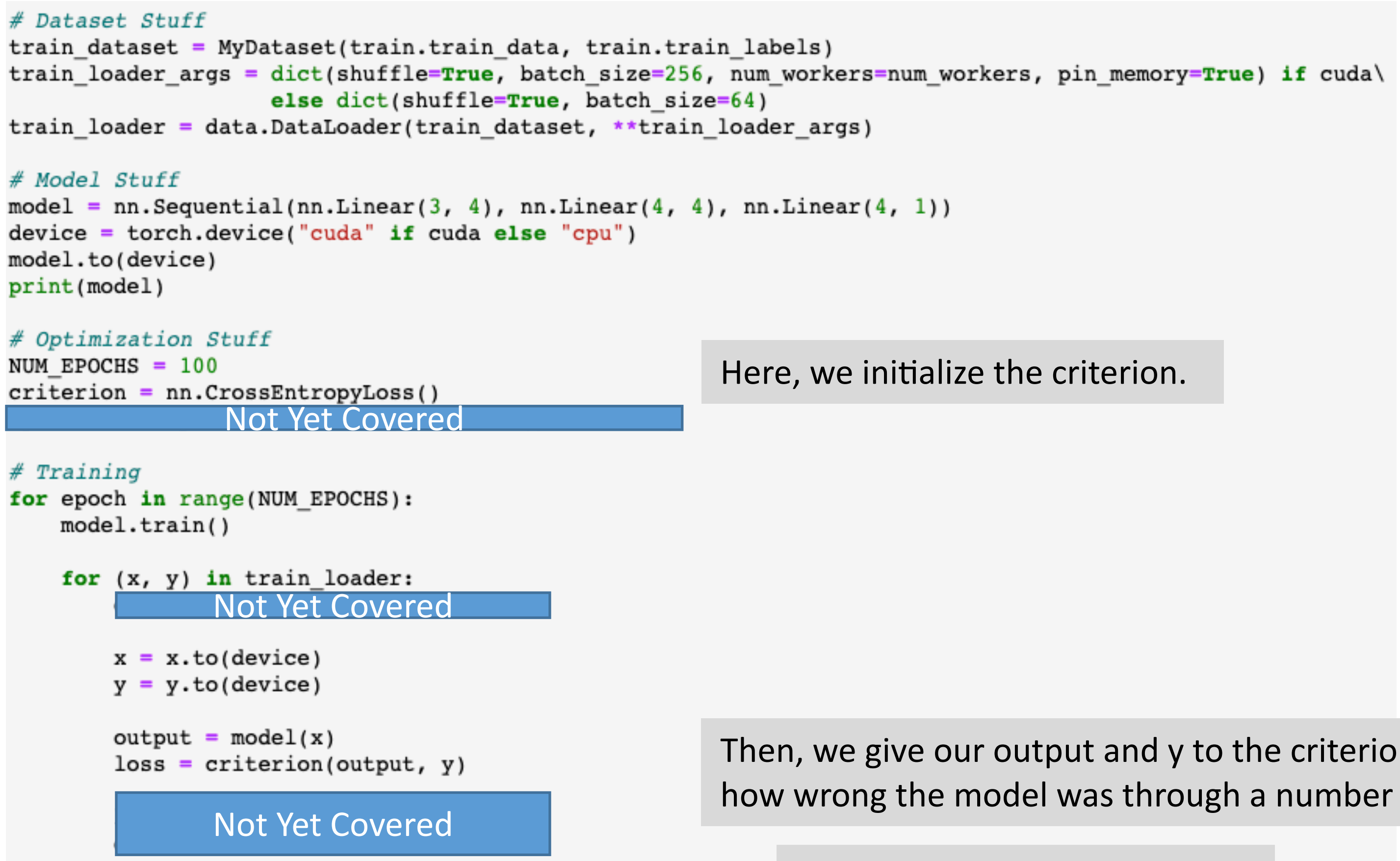
Within each epoch, we run through the train loader, which gives us x, y batched.

Since the model is on GPU, remember to put the x, y on GPU too!
Afterwards, run x through the model to get output.

# 3) Training a Model: [Loss Function]

- To recap, we have run x through our model and gotten “output,” or “$\bar{y}$”
- Recall – we need something to tell us how wrong it is compared to the true answer y.
- We rely on a “loss function,” also called a “criterion” to tell us this.
- The choice of a criterion will depend on the model/application/task, but for classification, a criterion called “CrossEntropyLoss” is commonly used.
- You’ll go over the specifics next lecture.

# 3) Training a Model: [Model]



```
# Dataset Stuff
train_dataset = MyDataset(train.train_data, train.train_labels)
train_loader_args = dict(shuffle=True, batch_size=256, num_workers=num_workers, pin_memory=True) if cuda\
                    else dict(shuffle=True, batch_size=64)
train_loader = data.DataLoader(train_dataset, **train_loader_args)

# Model Stuff
model = nn.Sequential(nn.Linear(3, 4), nn.Linear(4, 4), nn.Linear(4, 1))
device = torch.device("cuda" if cuda else "cpu")
model.to(device)
print(model)

# Optimization Stuff
NUM_EPOCHS = 100
criterion = nn.CrossEntropyLoss()
                Not Yet Covered

# Training
for epoch in range(NUM_EPOCHS):
    model.train()

    for (x, y) in train_loader:
                Not Yet Covered

        x = x.to(device)
        y = y.to(device)

        output = model(x)
        loss = criterion(output, y)

                Not Yet Covered
```

Here, we initialize the criterion.

Then, we give our output and y to the criterion, and it tells us how wrong the model was through a number called “loss”

We want to minimize this loss.

# 3) Training a Model: [Backpropagation]

- Backpropagation is the process of working backwards from the loss and calculating the gradients of every single (trainable) parameter w.r.t the loss.
  - The gradients tell us the direction in which to move to minimize the loss.
- If this is new for you, don’t worry – the next few lectures will make this clear.
- For now, we’ll stick with an intuitive explanation:
  - Backpropagation is a method of “assigning blame”
  - Think about a random parameter “p” in the model. Backprop will give us a number for p: “$\nabla p$”
  - $\nabla p$ tells us “hey, p is not optimal, and it caused the final output to be different from the true answer. This is how much p was wrong, and we should change p this ($\nabla p$) much to make the final output better”

Data

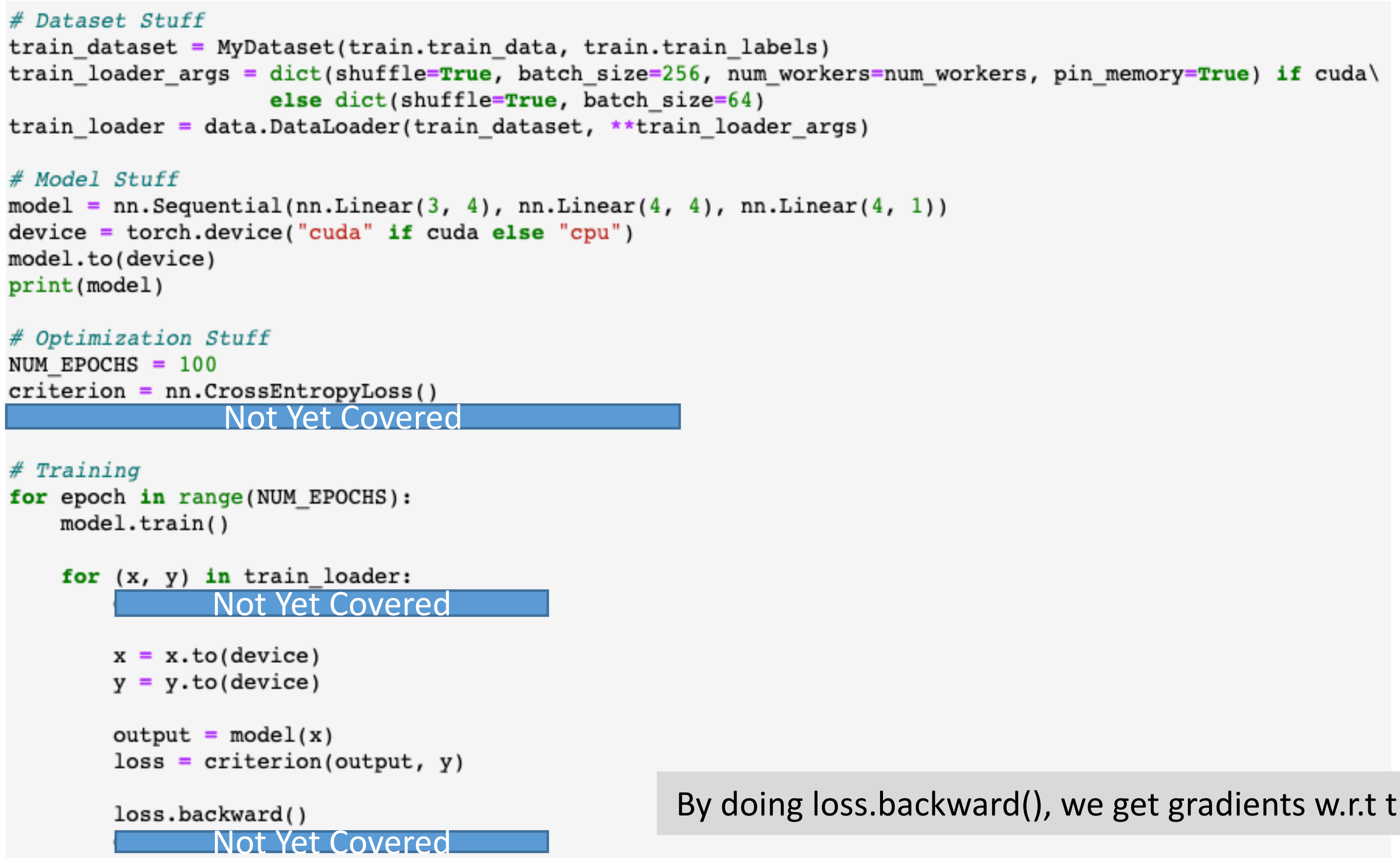
Model

Loss Function

Backpropagation

Optimizer

# 3) Training a Model: [Model]



```
# Dataset Stuff
train_dataset = MyDataset(train.train_data, train.train_labels)
train_loader_args = dict(shuffle=True, batch_size=256, num_workers=num_workers, pin_memory=True) if cuda\
                    else dict(shuffle=True, batch_size=64)
train_loader = data.DataLoader(train_dataset, **train_loader_args)

# Model Stuff
model = nn.Sequential(nn.Linear(3, 4), nn.Linear(4, 4), nn.Linear(4, 1))
device = torch.device("cuda" if cuda else "cpu")
model.to(device)
print(model)

# Optimization Stuff
NUM_EPOCHS = 100
criterion = nn.CrossEntropyLoss()
                Not Yet Covered

# Training
for epoch in range(NUM_EPOCHS):
    model.train()

    for (x, y) in train_loader:
                Not Yet Covered

        x = x.to(device)
        y = y.to(device)

        output = model(x)
        loss = criterion(output, y)

        loss.backward()
                Not Yet Covered
```

By doing loss.backward(), we get gradients w.r.t the loss.

Remember model.train()? That allowed us to compute the gradients. If it had been in the eval state, we wouldn't be able to even compute the gradients, much less train.

# 3) Training a Model: [Optimizer]

- Now, backprop only *computes* the $\nabla$p values – it doesn’t do anything with them.
- Now, we want to *update* the value of p using $\nabla$p. This is the optimizer’s job.
- A crucial component of any optimizer is the “learning rate.” This is a hyperparameter that controls how much we should believe in $\nabla$p.
    - Again, this will be covered in more detail in a future lecture.
    - Ideally, $\nabla$p is a perfect assignment of blame w.r.t the **entire** dataset. However, it’s likely that optimizing to perfectly match the *current* (x, y) sample $\nabla$p was generated from won’t be great for matching the entire dataset.
    - …Among other concerns, the optimizer *weights* the $\nabla$p with the learning rate and use the weighted $\nabla$p to update p.

Data

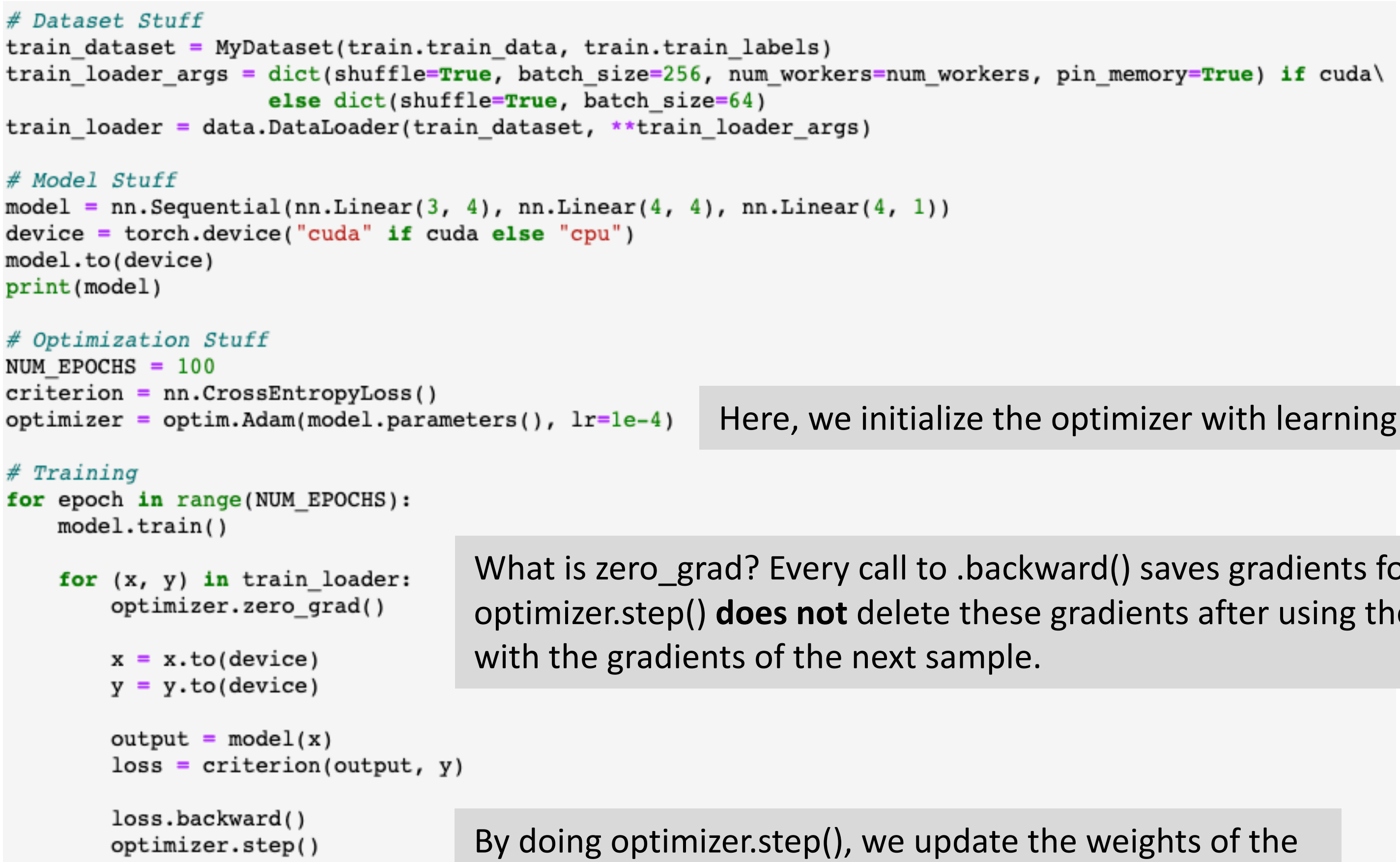
Model

Loss Function

Backpropagation

Optimizer

# 3) Training a Model: [Model]



```
# Dataset Stuff
train_dataset = MyDataset(train.train_data, train.train_labels)
train_loader_args = dict(shuffle=True, batch_size=256, num_workers=num_workers, pin_memory=True) if cuda\
                    else dict(shuffle=True, batch_size=64)
train_loader = data.DataLoader(train_dataset, **train_loader_args)

# Model Stuff
model = nn.Sequential(nn.Linear(3, 4), nn.Linear(4, 4), nn.Linear(4, 1))
device = torch.device("cuda" if cuda else "cpu")
model.to(device)
print(model)

# Optimization Stuff
NUM_EPOCHS = 100
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=1e-4)

# Training
for epoch in range(NUM_EPOCHS):
    model.train()

    for (x, y) in train_loader:
        optimizer.zero_grad()

        x = x.to(device)
        y = y.to(device)

        output = model(x)
        loss = criterion(output, y)

        loss.backward()
        optimizer.step()
```

Here, we initialize the optimizer with learning rate 1e-4

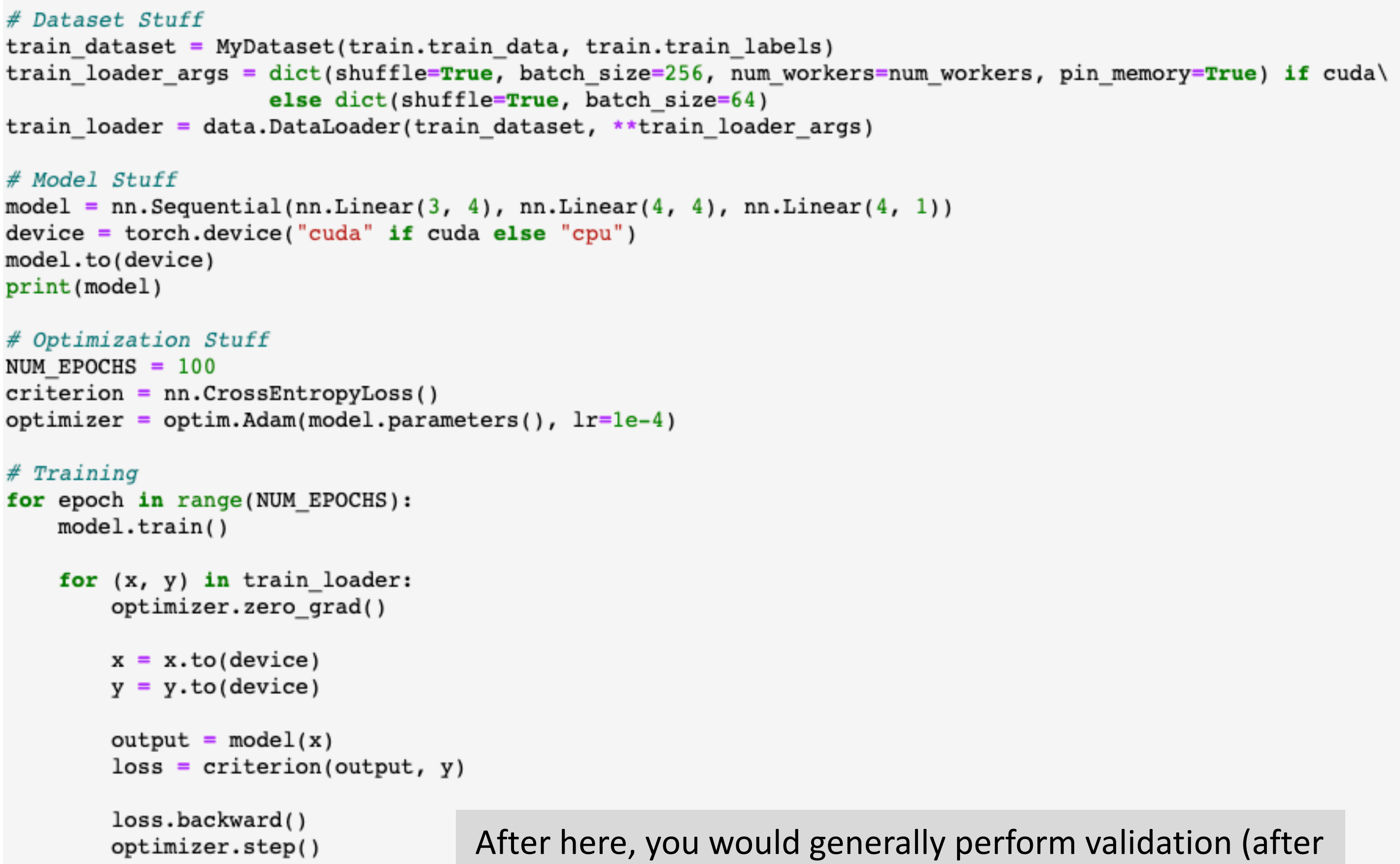
What is zero_grad? Every call to .backward() saves gradients for each parameter in the model. However, calling optimizer.step() **does not** delete these gradients after using them. So, you want to remove them so they don’t interfere with the gradients of the next sample.

By doing optimizer.step(), we update the weights of the model using the computed gradients.

**IMPORTANT**: The general order of these steps are crucial.

1. We run x through the model.
2. We compute the loss.
3. We call loss.backward()
4. Then we step the optimizer.
5. Then, (in this loop or the next), we zero out the gradients.

# 3) Training a Model: Some extras



```
# Dataset Stuff
train_dataset = MyDataset(train.train_data, train.train_labels)
train_loader_args = dict(shuffle=True, batch_size=256, num_workers=num_workers, pin_memory=True) if cuda\
                    else dict(shuffle=True, batch_size=64)
train_loader = data.DataLoader(train_dataset, **train_loader_args)

# Model Stuff
model = nn.Sequential(nn.Linear(3, 4), nn.Linear(4, 4), nn.Linear(4, 1))
device = torch.device("cuda" if cuda else "cpu")
model.to(device)
print(model)

# Optimization Stuff
NUM_EPOCHS = 100
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=1e-4)

# Training
for epoch in range(NUM_EPOCHS):
    model.train()

    for (x, y) in train_loader:
        optimizer.zero_grad()

        x = x.to(device)
        y = y.to(device)

        output = model(x)
        loss = criterion(output, y)

        loss.backward()
        optimizer.step()
```

After here, you would generally perform validation (after every epoch or a couple), to see how your model performs on data it is not trained on.
Validation follows a similar format as training, but without loss.backward() or optimizer.step(). You should check the notebooks for more guidance.

# Link to Example Notebooks

- https://drive.google.com/drive/folders/1dILzOSUDnRSjYIrYnZAOIauRrNbXFqmq?usp=sharing