Your First Deep Learning Code

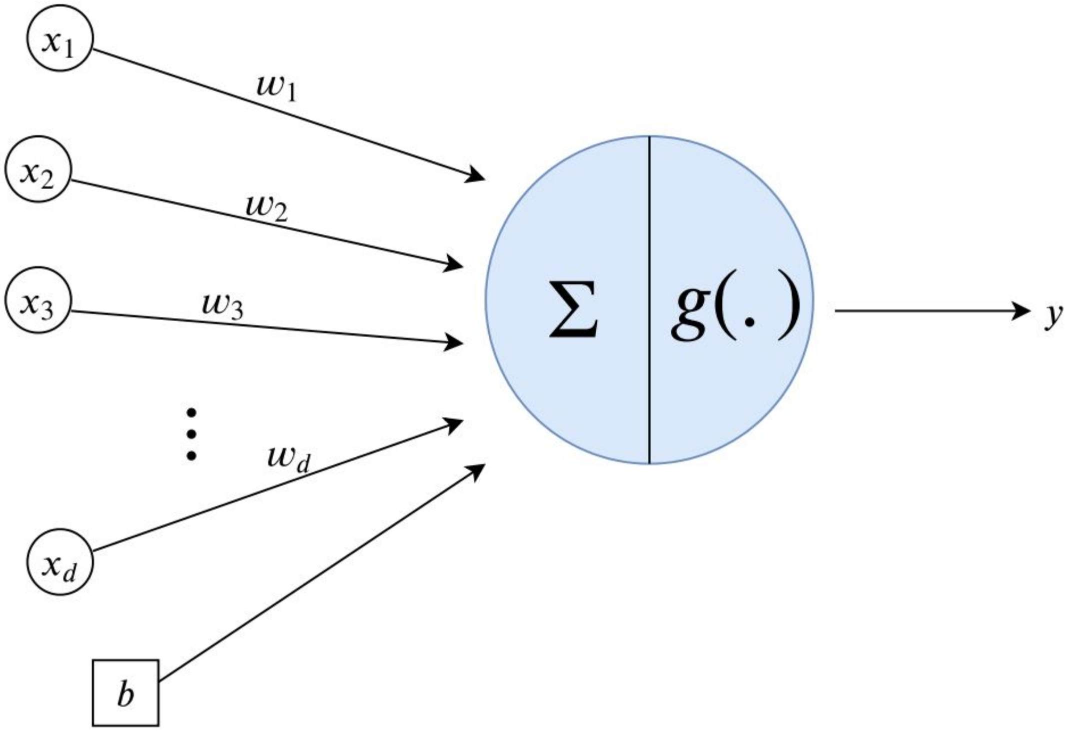
11-785 Spring 2020

Overview

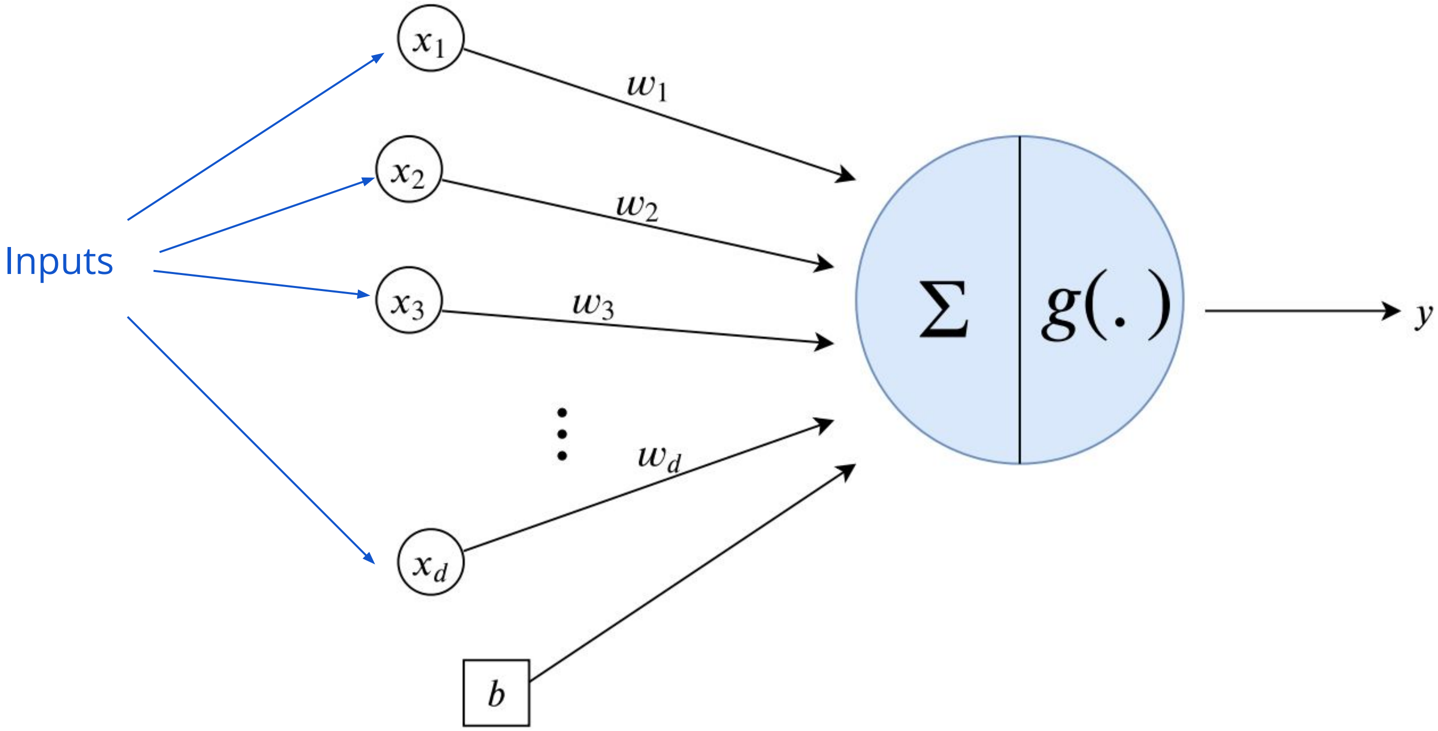
- Neural Networks
- Tensors
- CPU and GPU Operations
- Backpropagation
- Neural Network Modules
- Optimization and Loss
- Saving and Loading
- Common Issues to look out for
- Full NN Example in code

A (Very Brief) Neural Network Primer

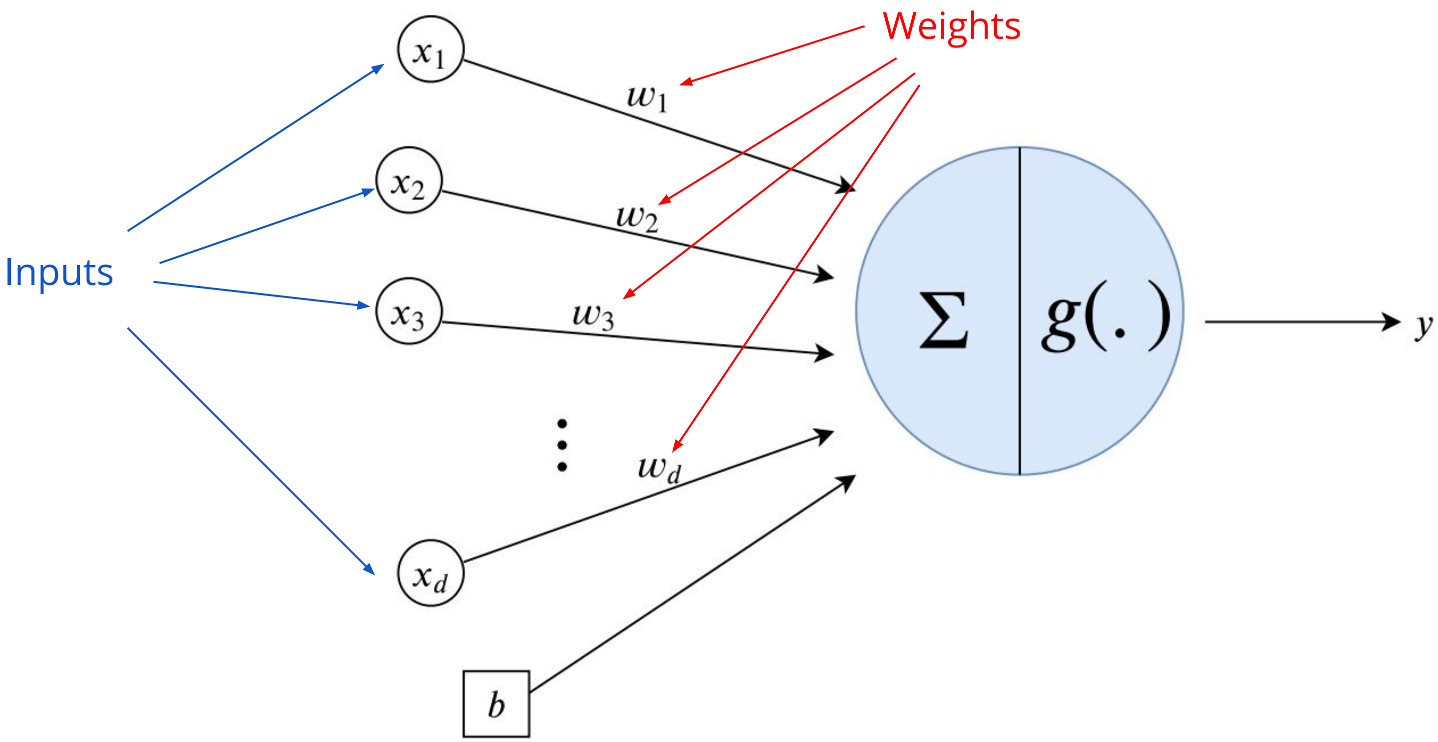
Perceptron (or Artificial Neuron)



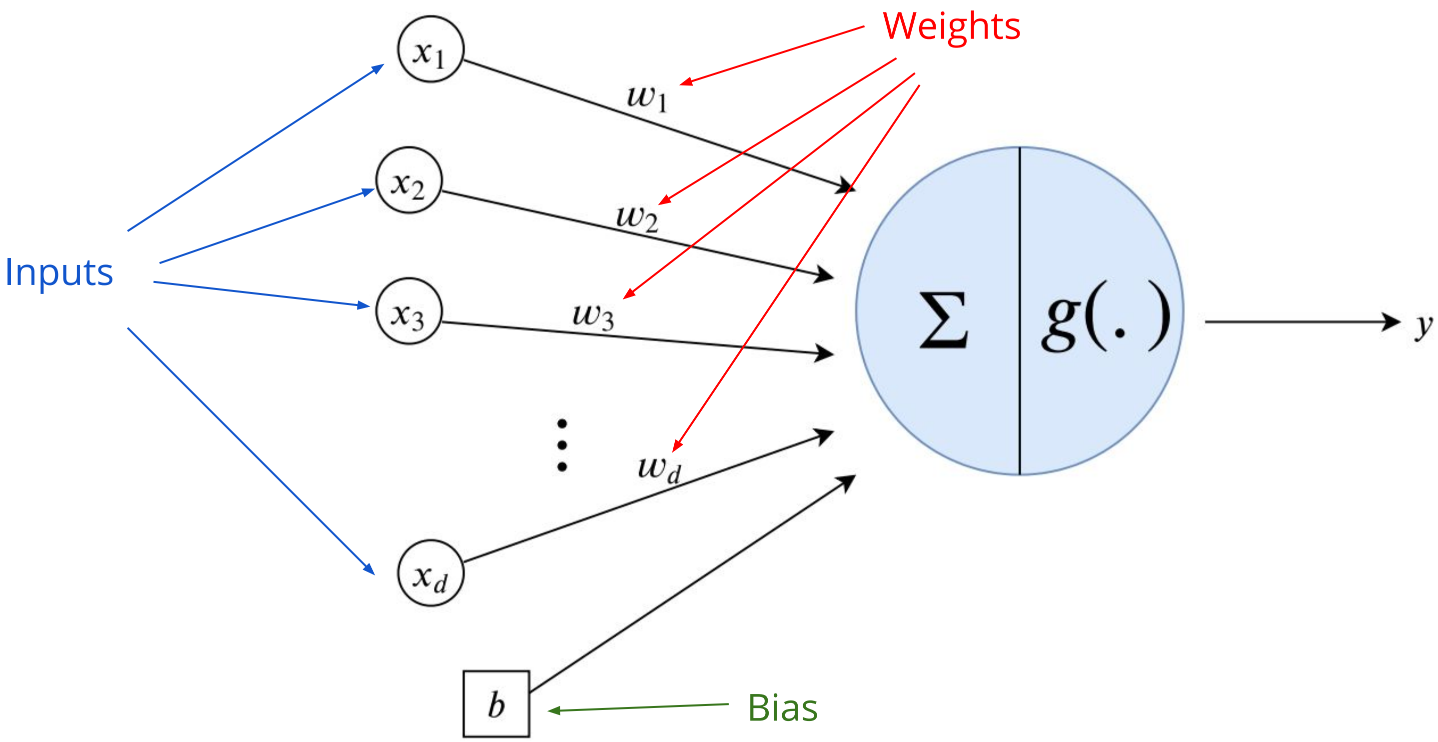
Perceptron (or Artificial Neuron)



Perceptron (or Artificial Neuron)

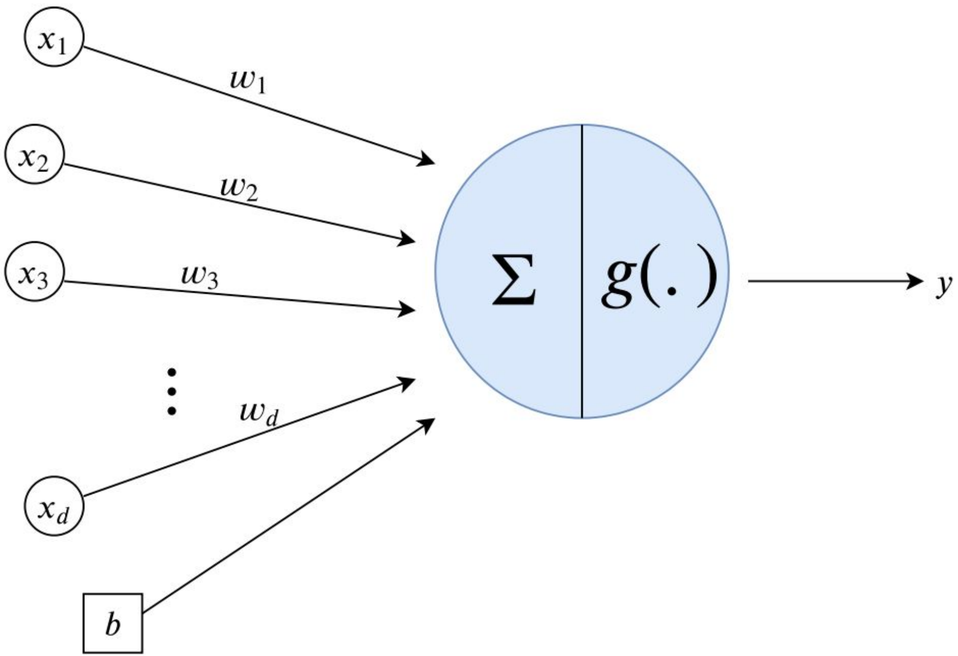


Perceptron (or Artificial Neuron)



Perceptron (or Artificial Neuron)

- Basic computational unit
- Inputs combine linearly

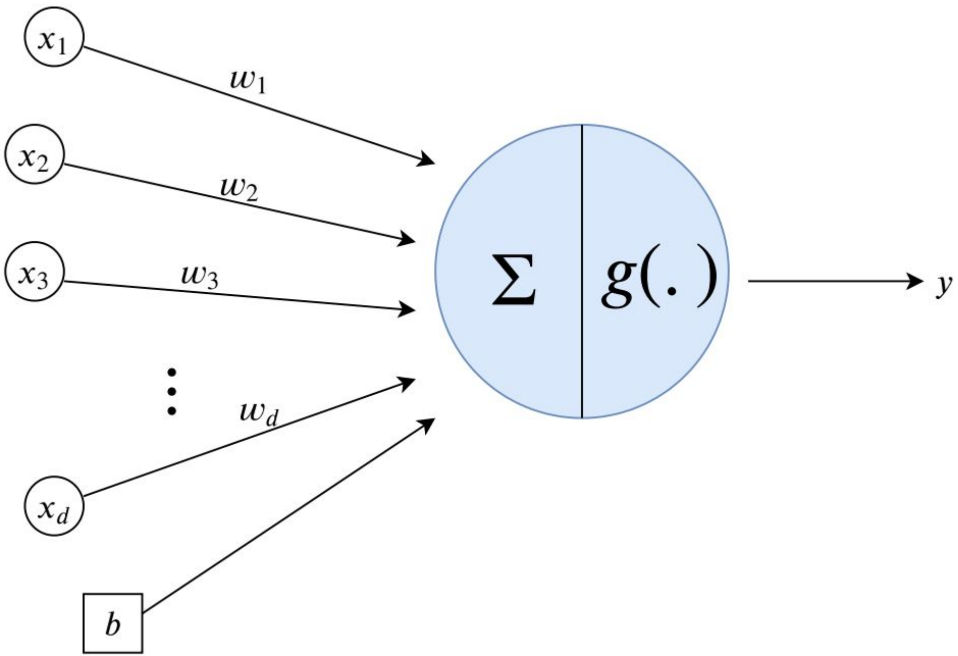


Perceptron (or Artificial Neuron)

- Basic computational unit

- Inputs combine linearly

$$y = g\left(\sum_{i=1}^{d} w_i x_i + b\right)$$

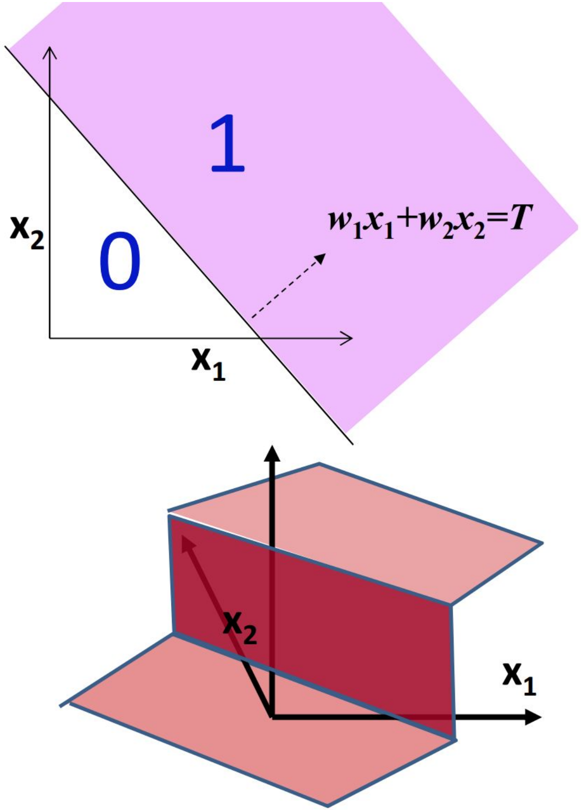


What can a perceptron represent?

- This is a linear classifier

- Here, the activation function is a 0-1 step function.

$$y = \begin{cases} 0 & \mathbf{w}^\top \mathbf{x} + b < 0 \\ 1 & \mathbf{w}^\top \mathbf{x} + b \geq 0 \end{cases}$$

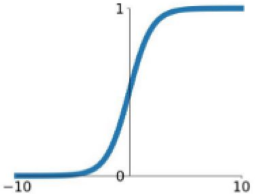


Activation functions

- Instead of a threshold, we can have any arbitrary “activation” function

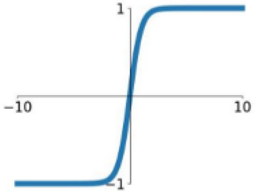
Sigmoid

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



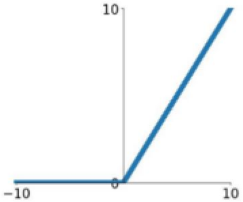
tanh

$$\tanh(x)$$

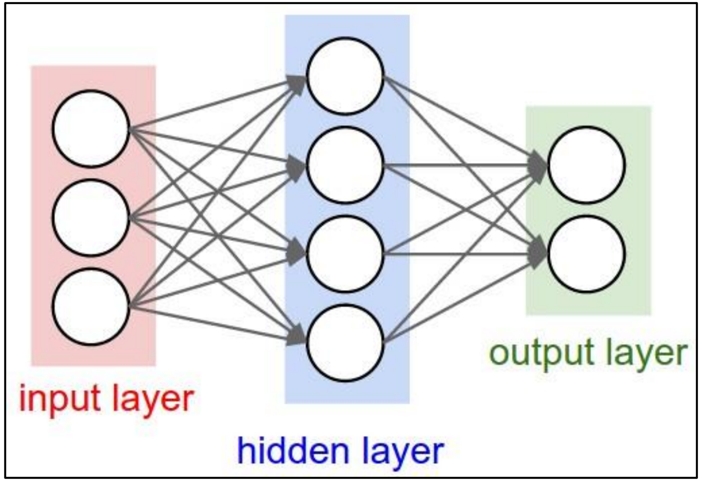


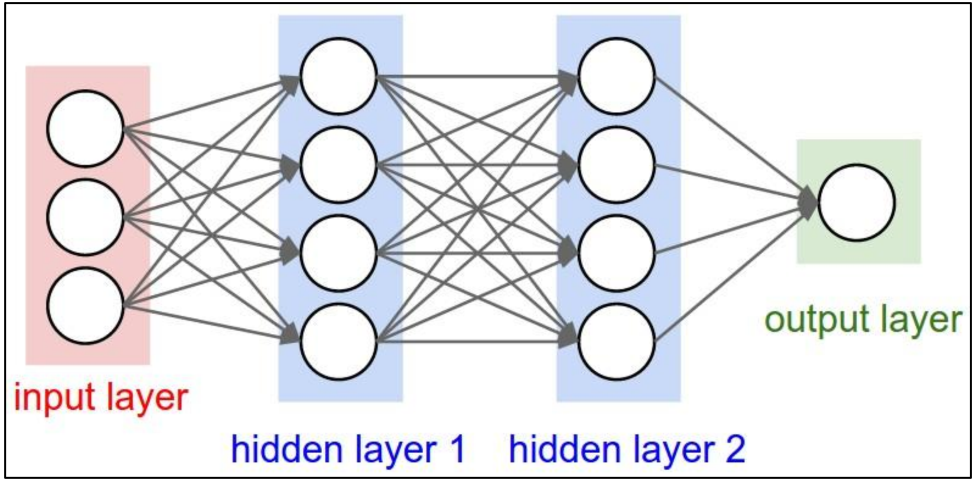
ReLU

$$\max(0, x)$$



Multilayer Perceptron



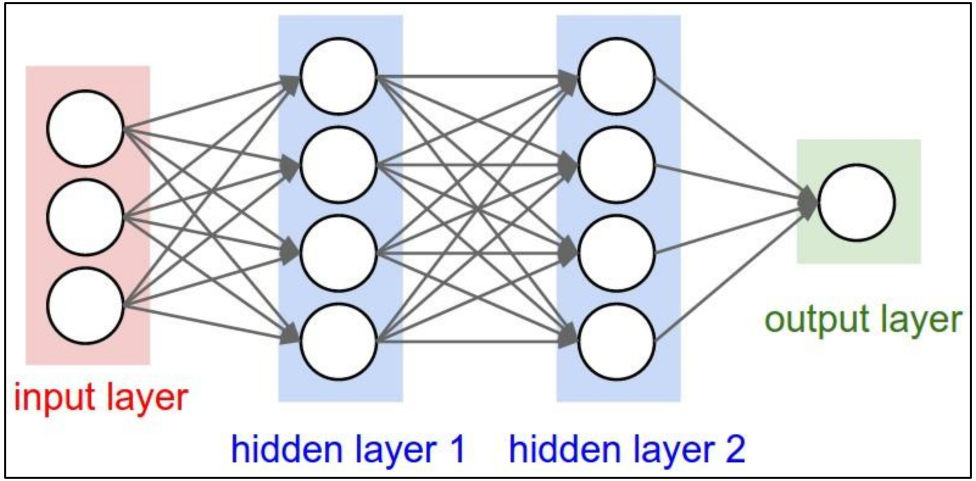


Source: cs231n

Multilayer Perceptron

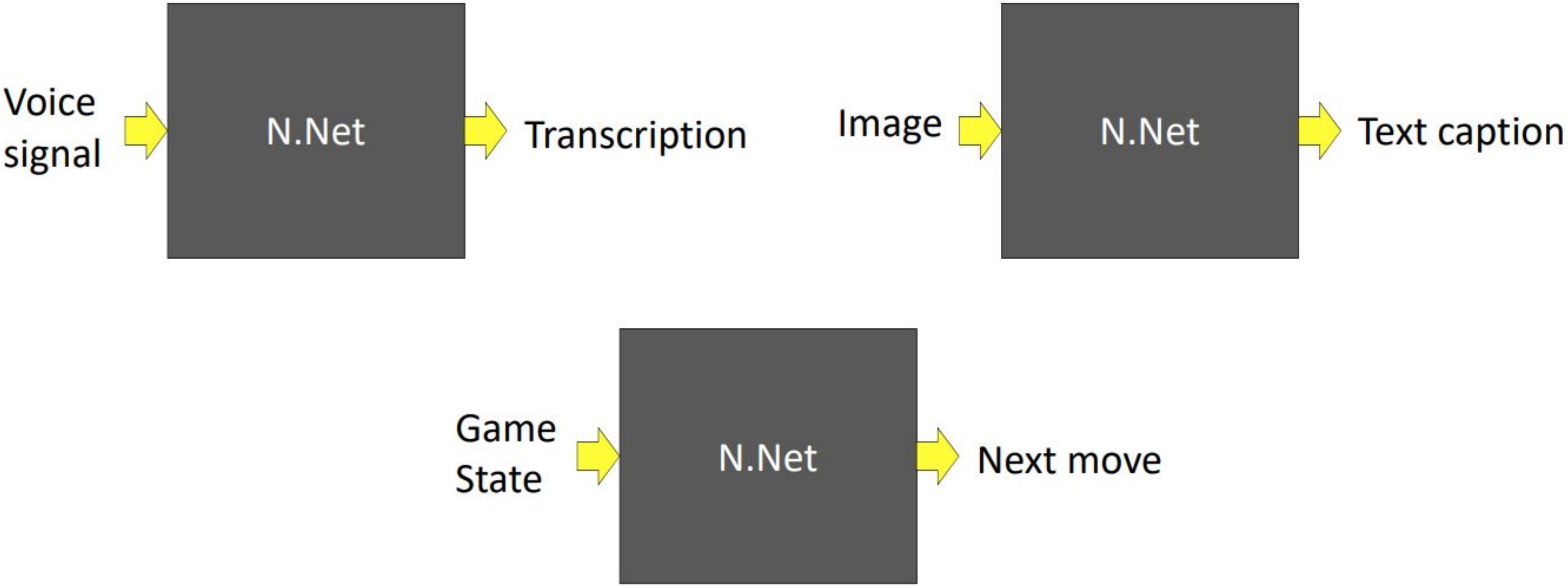
- A “fully-connected” multi-layer network of perceptrons (MLP).

- Much more powerful than a single neuron -- can represent *any* function*.

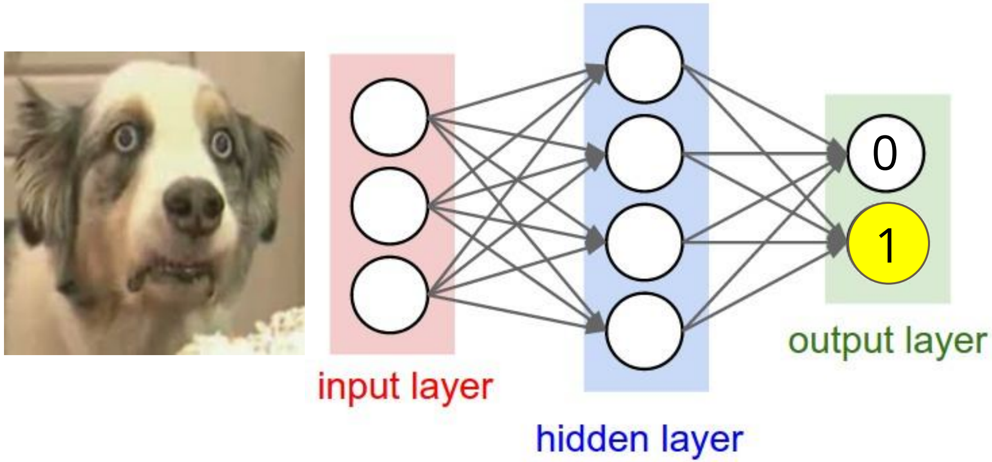


*Universal Approximation Theorem

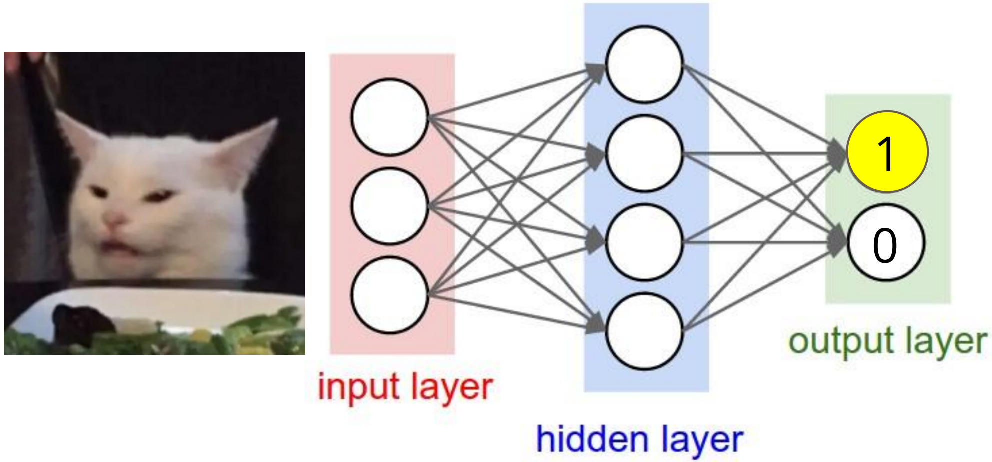
Multilayer Perceptron



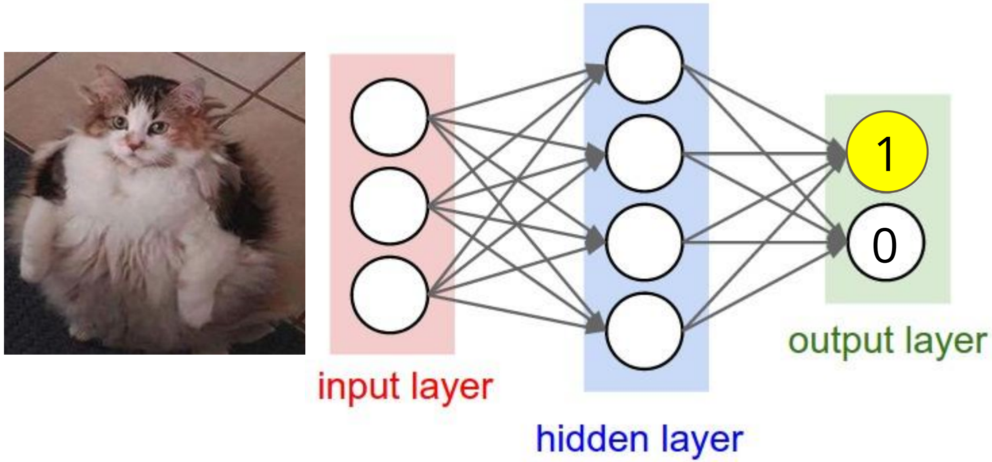
MLPs for Classification



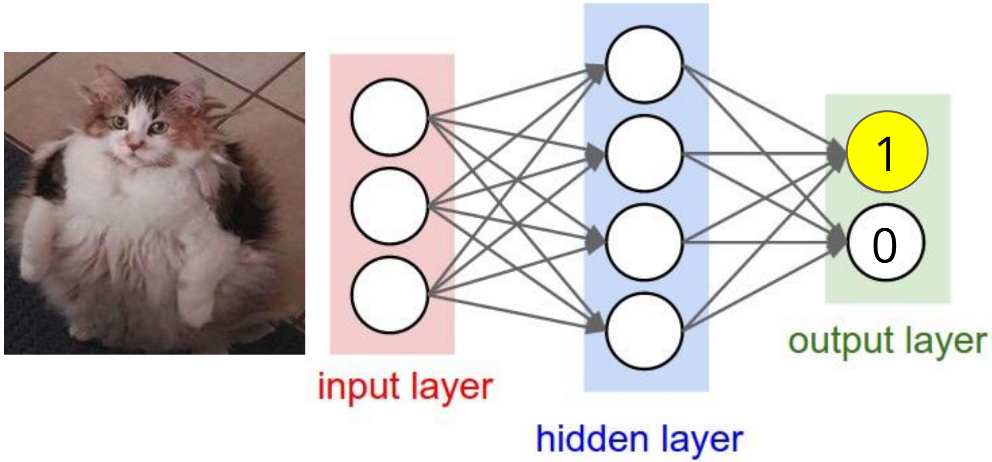
MLPs for Classification



MLPs for Classification

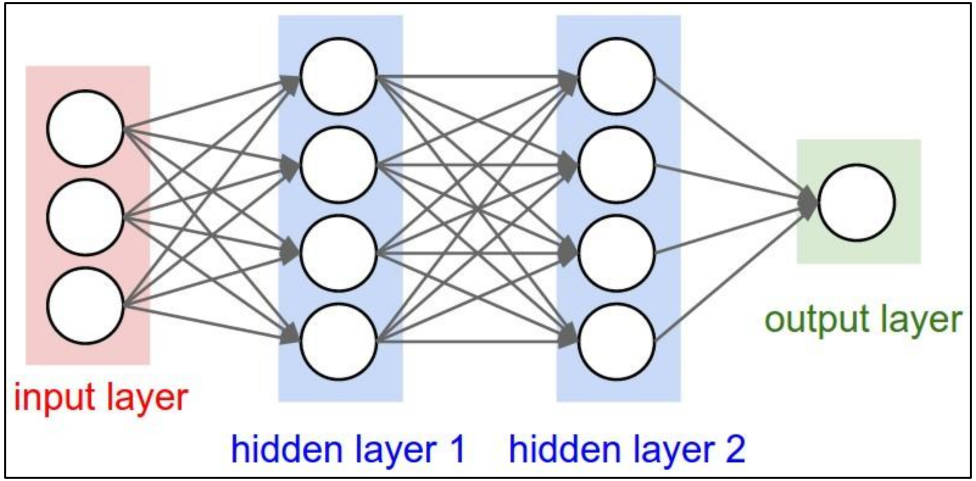


MLPs for Classification



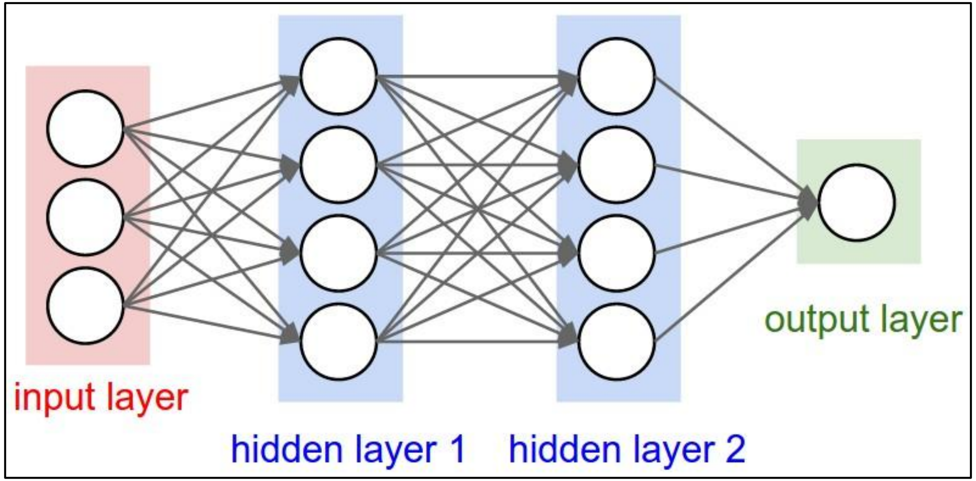
But the network must be learned...

How Do We “Learn”?



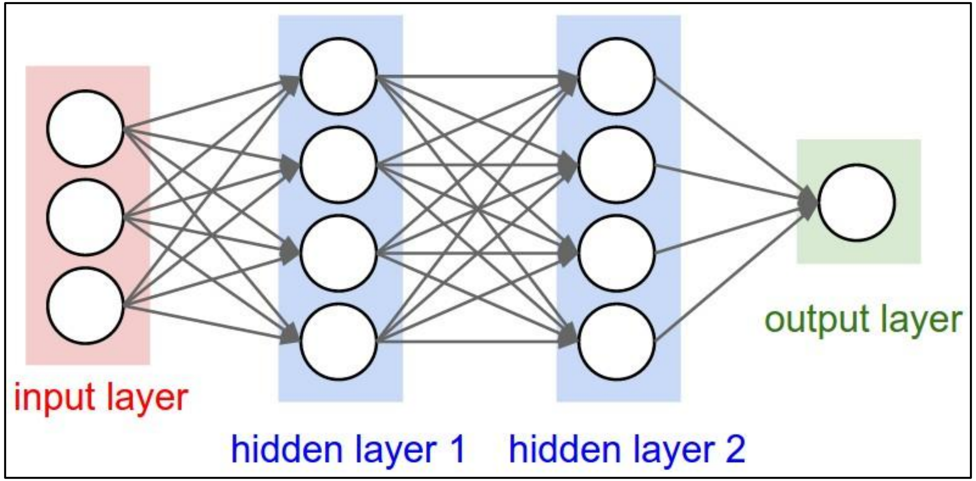
How Do We “Learn”?

- But first: *What* do we learn?



How Do We "Learn"?

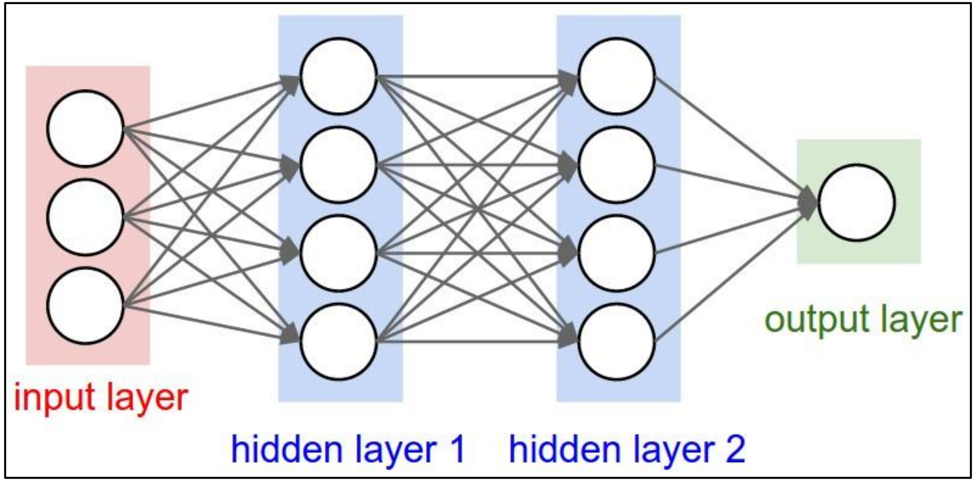
- But first: *What* do we learn?
  The parameters



How Do We "Learn"?

- But first: *What* do we learn?
  The parameters

- What are the parameters?



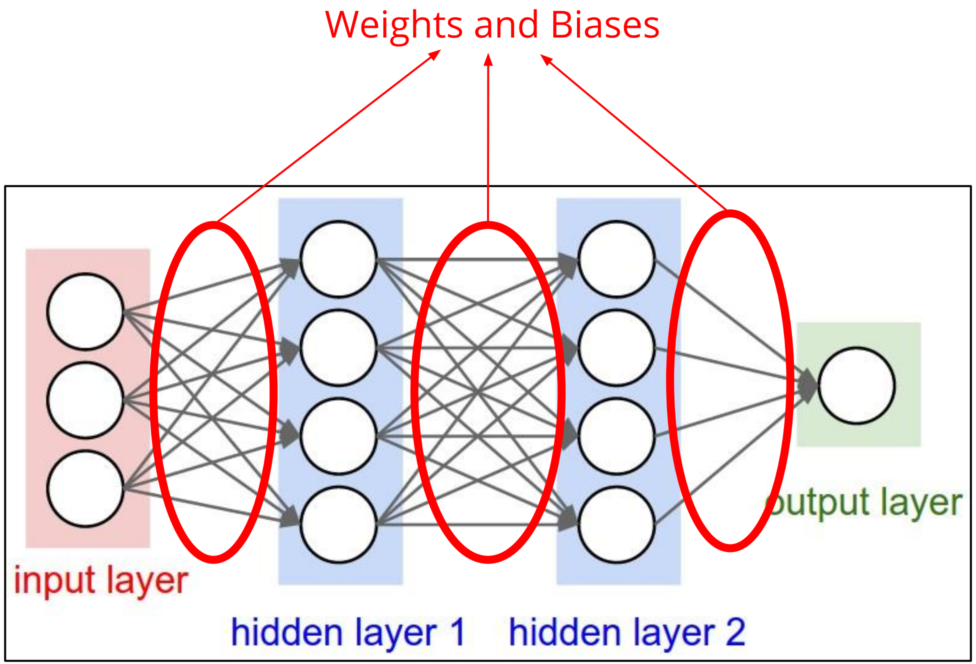
How Do We "Learn"?

- But first: *What* do we learn?
  The parameters

- What are the parameters?
  Weights and biases

$$W_1 = \begin{bmatrix} w_{111} & w_{112} & w_{113} \\ w_{121} & w_{122} & w_{123} \\ w_{131} & w_{132} & w_{133} \\ w_{141} & w_{142} & w_{143} \end{bmatrix} \qquad b_1 = \begin{bmatrix} b_{11} \\ b_{12} \\ b_{13} \\ b_{14} \end{bmatrix}$$

$$W_2 = \begin{bmatrix} w_{211} & \cdots & w_{214} \\ \vdots & \ddots & \\ w_{241} & & w_{244} \end{bmatrix} \qquad b_2 = \begin{bmatrix} b_{21} \\ b_{22} \\ b_{23} \\ b_{24} \end{bmatrix}$$



How Do We "Learn"?

- Suppose we want to classify cats and dogs.

How Do We “Learn”?

- Suppose we want to classify cats and dogs.

- We will provide many input-output example pairs and try to optimize the parameters so that the network output matches **training data** output as closely as possible.

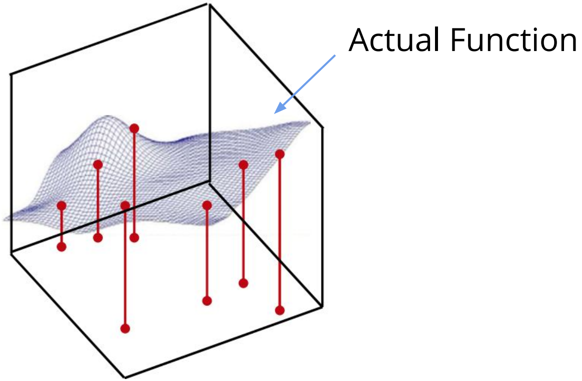
Input	Output
	“Cat”
	“Dog”
	“Dog”
	“Cat”

How Do We “Learn”?

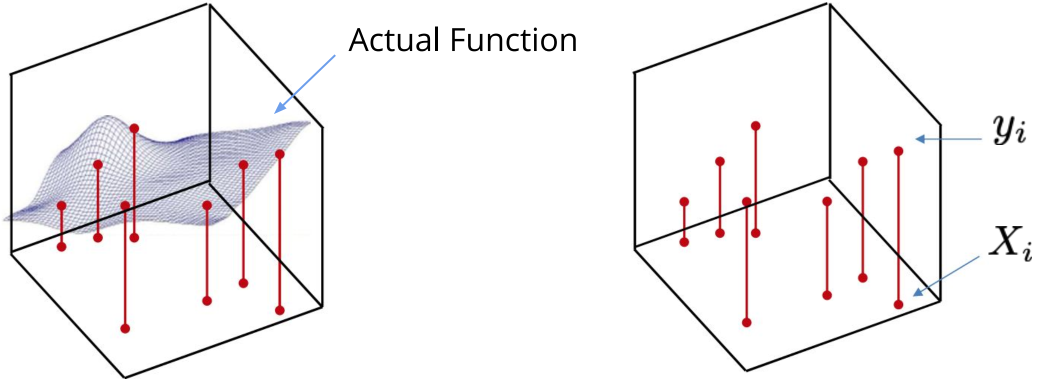
- Suppose we want to classify cats and dogs.
- We will provide many input-output example pairs and try to optimize the parameters so that the network output matches **training data** output as closely as possible.
- Need to quantify the error.

Input	Output
	“Cat”
	“Dog”
	“Dog”
	“Cat”

How Do We “Learn”?

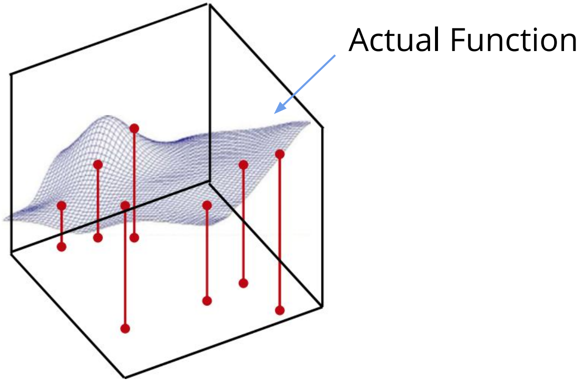


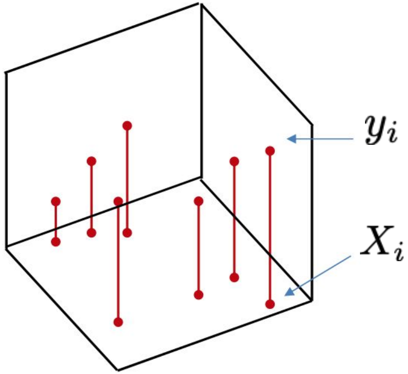
How Do We “Learn”?



- Estimate functions from the samples.

How Do We “Learn”?





- Estimate functions from the samples.

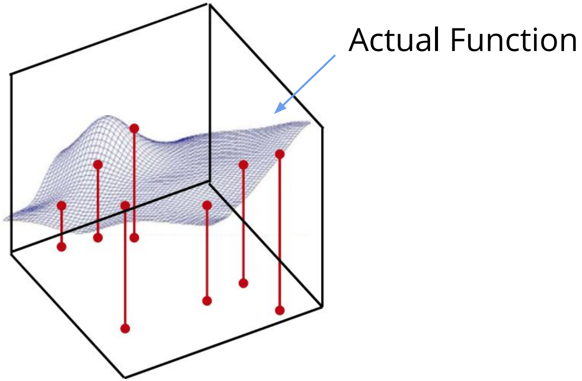
- Need a quantification of the error between the **network output** and the **desired output**

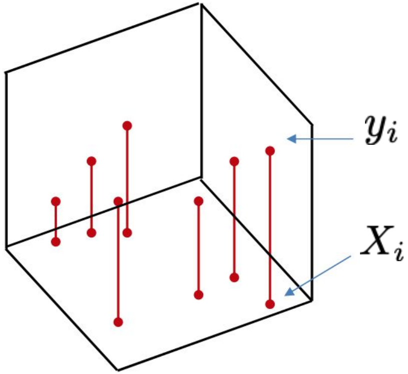
Network output

Desired output

$$\mathcal{L}(W) = \frac{1}{N}\sum_i div(f(X_i; W), y_i)$$

How Do We “Learn”?





- Estimate functions from the samples.
- Need a quantification of the error between the **network output** and the **desired output**
- Optimize parameters to minimize this error.

Network output

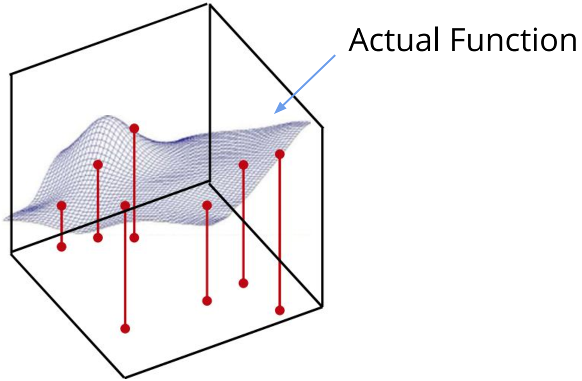
Desired output

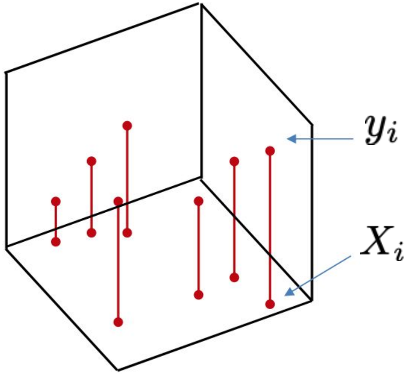
$$\mathcal{L}(W) = \frac{1}{N}\sum_i div(f(X_i; W), y_i)$$

$$\hat{W} = \underset{W}{\operatorname{argmin}}\ \mathcal{L}(W)$$

$$W = \{W_1, b_1, W_2, b_2, \ldots, W_k, b_k\}$$

How Do We “Learn”?





- Estimate functions from the samples.
- Need a quantification of the error between the **network output** and the **desired output**
- Optimize parameters to minimize this error. (How?)

Network output

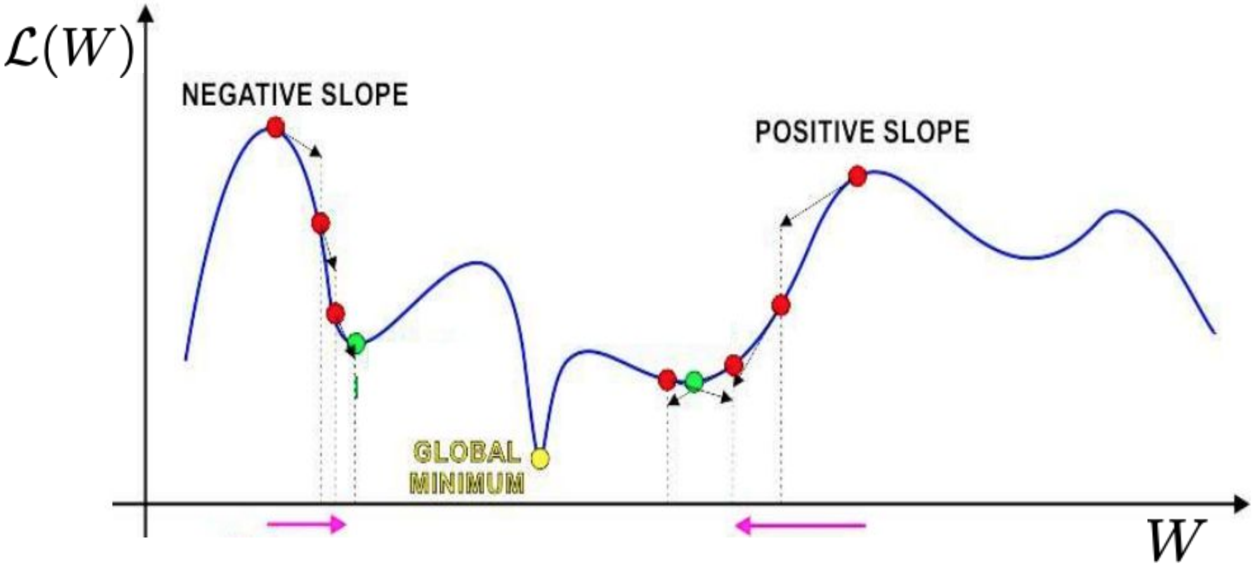
Desired output

$$\mathcal{L}(W) = \frac{1}{N}\sum_i div(f(X_i; W), y_i)$$

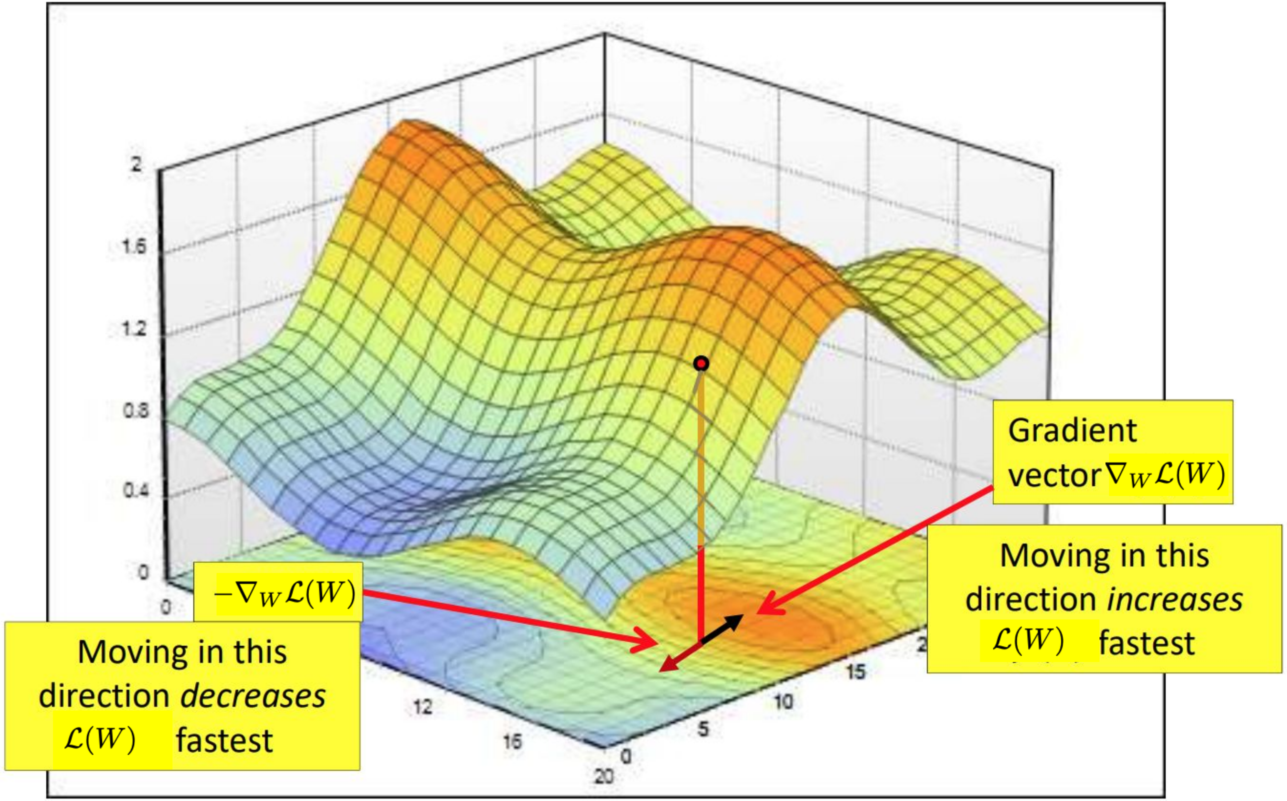
$$\hat{W} = \underset{W}{\operatorname{argmin}}\ \mathcal{L}(W)$$

$$W = \{W_1, b_1, W_2, b_2, \ldots, W_k, b_k\}$$

Gradient Descent



Gradient Descent



Gradient Descent

1. Initialize all the parameters.

Gradient Descent

1. Initialize all the parameters.
2. Repeat until convergence:

Gradient Descent

1. Initialize all the parameters.
2. Repeat until convergence:
   a. Compute loss

$$\mathcal{L}(W) = \frac{1}{N}\sum_i div(f(X_i; W), y_i)$$

Gradient Descent

1. Initialize all the parameters.
2. Repeat until convergence:
   a. Compute loss

$$\mathcal{L}(W) = \frac{1}{N}\sum_i div(f(X_i; W), y_i)$$

   b. Compute gradient of the loss wrt parameters

$$\nabla_W \mathcal{L}(W) \quad \left(\frac{\partial \mathcal{L}}{\partial w_{ijk}} \ \forall i, j, k \text{ and } \frac{\partial \mathcal{L}}{\partial b_{ij}} \ \forall i, j\right)$$

Gradient Descent

1. Initialize all the parameters.
2. Repeat until convergence:
   a. Compute loss

$$\mathcal{L}(W) = \frac{1}{N}\sum_i div(f(X_i; W), y_i)$$

   b. Compute gradient of the loss wrt parameters

$$\nabla_W \mathcal{L}(W) \quad \left(\frac{\partial \mathcal{L}}{\partial w_{ijk}} \;\forall i, j, k \text{ and } \frac{\partial \mathcal{L}}{\partial b_{ij}} \;\forall i, j\right)$$

   c. Update parameters

$$W \leftarrow W - \eta \nabla_W \mathcal{L}(W)$$

Gradient Descent

1. Initialize all the parameters.
2. Repeat until convergence:
    a. Compute loss

    $$\mathcal{L}(W) = \frac{1}{N}\sum_i div(f(X_i; W), y_i)$$

    b. Compute gradient of the loss wrt parameters

    $$\nabla_W \mathcal{L}(W) \quad \left(\frac{\partial \mathcal{L}}{\partial w_{ijk}} \; \forall i,j,k \text{ and } \frac{\partial \mathcal{L}}{\partial b_{ij}} \; \forall i,j\right)$$

    c. Update parameters

    $$W \leftarrow W - \eta \nabla_W \mathcal{L}(W)$$

$$w_{ijk} \leftarrow w_{ijk} - \eta \frac{\partial \mathcal{L}}{\partial w_{ijk}} \qquad b_{ij} \leftarrow b_{ij} - \eta \frac{\partial \mathcal{L}}{\partial b_{ij}}$$

(Scalar form)

Gradient Descent

Your First Deep Learning Code (...finally)

Let’s start with Deep Learning Frameworks

What do they provide?

- Computation (often with some Numpy support)
- GPU support for parallel computation
- Some basic neural layers to combine in your models
- Tools to train your models
- Enforce a general way to code your models
- And most importantly, **automatic backpropagation**

Pytorch

We recommend Pytorch v1.3

You should have access to an environment with it, and hopefully a GPU.

LET'S START!

Overview

- Neural Networks
- Tensors
- CPU and GPU Operations
- Backpropagation
- Neural Network Modules
- Optimization and Loss
- Saving and Loading
- Common Issues to look out for
- Full NN Example in code

Tensors

- Tensors are similar to NumPy's ndarrays, with the addition being that Tensors can also be used on a GPU to accelerate computing.

```
# Create uninitialized tensor
x = torch.FloatTensor(2,3)
# from numpy
np_array = np.random.random((2,3)).astype(float)
x1 = torch.FloatTensor(np_array)
x2 = torch.randn(2,3)
# export to numpy array
x_np = x2.numpy()
# basic operation
x = torch.arange(4,dtype=torch.float).view(2,2)
s = torch.sum(x)
e = torch.exp(x)
# elementwise and matrix multiplication
z = s*e + torch.matmul(x1,x2.t()) # size 2*2
```

Overview

- Neural Networks
- Tensors
- CPU and GPU Operations
- Backpropagation
- Neural Network Modules
- Optimization and Loss
- Saving and Loading
- Common Issues to look out for
- Full NN Example in code

Move Tensors to the GPU

For big computations, GPUs offer significant speedups!

```
# create a tensor
x = torch.rand(3,2)
# copy to GPU
y = x.cuda()
# copy back to CPU
z = y.cpu()
# get CPU tensor as numpy array
# cannot get GPU tensor as numpy array directly
try:
    y.numpy()
except RuntimeError as e:
    print(e)
```

Tensors can be copied between CPU and GPU. It is important that everything involved in a calculation is on the same device.

This portion of the tutorial may not work for you if you do not have a GPU available.

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-ad31a5261faa> in <module>
      9 # cannot get GPU tensor as numpy array directly
     10 try:
---> 11     y.numpy()
     12 except RuntimeError as e:
     13     print(e)

TypeError: can't convert CUDA tensor to numpy. Use Tensor.cpu() to copy the tensor to host memory first.
```

Move Tensors to the GPU

Operations between GPU and CPU tensors will fail. Operations require all arguments to be on the same device.

```
x = torch.rand(3,5)  # CPU tensor
y = torch.rand(5,4).cuda()  # GPU tensor
try:
  torch.mm(x,y)  # Operation between CPU and GPU fails
except TypeError as e:
  print(e)
```

```
torch.mm received an invalid combination of arguments - got (torch.FloatTensor, torc
h.cuda.FloatTensor), but expected one of:
 * (torch.FloatTensor source, torch.FloatTensor mat2)
      didn't match because some of the arguments have invalid types: (torch.FloatTens
or, torch.cuda.FloatTensor)
 * (torch.SparseFloatTensor source, torch.FloatTensor mat2)
      didn't match because some of the arguments have invalid types: (torch.FloatTens
or, torch.cuda.FloatTensor)
```

Move Tensors to the GPU

Typical code should be compatible with both CPU & GPU (device agnostic). Include if statements or utilize helper functions so it can operate with or without the GPU.

```python
# Put tensor on CUDA if available
x = torch.rand(3,2)
if torch.cuda.is_available():
    x = x.cuda()
    print(x, x.dtype)

# Do some calculations
y = x ** 2
print(y)

# Copy to CPU if on GPU
if y.is_cuda:
    y = y.cpu()
    print(y, y.dtype)
```

```
tensor([[0.1084, 0.5432],
        [0.2185, 0.3834],
        [0.3720, 0.5374]], device='cuda:0') torch.float32
tensor([[0.0117, 0.2951],
        [0.0477, 0.1470],
        [0.1383, 0.2888]], device='cuda:0')
tensor([[0.0117, 0.2951],
        [0.0477, 0.1470],
        [0.1383, 0.2888]]) torch.float32
```

Overview

- Neural Networks
- Tensors
- CPU and GPU Operations
- Backpropagation
- Neural Network Modules
- Optimization and Loss
- Saving and Loading
- Common Issues to look out for
- Full NN Example in code

Backpropagation

1. Initialize parameters

2. Repeat until convergence:

    a. Compute Loss

    b. *** Compute gradients of the Loss function wrt parameters**

    c. Update parameters

In a nutshell: Backpropagation is an algorithm to compute the gradients of the loss function wrt the parameters *efficiently* using the chain-rule of calculus.

Backpropagation in Pytorch

Pytorch can retro-compute gradients for any succession of operations. Use the **.backward()** method.

```python
# Create a differentiable tensor
# NOTE: This is wrong because only float tensors can require gradients
x = torch.arange(0,4, requires_grad=True)
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-14-e4e17071c696> in <module>()
      1 # Create a differentiable tensor
      2 # NOTE: This is wrong because only float tensors can require gradients
----> 3 x = torch.arange(0,4, requires_grad=True)

RuntimeError: Only Tensors of floating point dtype can require gradients
```

Backpropagation in Pytorch

Solution

```
x = torch.arange(0,4, dtype=torch.float, requires_grad=True)
print(x.dtype)
# Calculate y = sum(x**2)
y = torch.sum(x**2)
# Calculate gradient (dy/dx = 2x)
y.backward()
# Print values
print(x)
print(y)
print(x.grad)
```

```
torch.float32
tensor([0., 1., 2., 3.], requires_grad=True)
tensor(14., grad_fn=<SumBackward0>)
tensor([0., 2., 4., 6.])
```

Overview

- Neural Networks
- Tensors
- CPU and GPU Operations
- Backpropagation
- Neural Network Modules
- Optimization and Loss
- Saving and Loading
- Common Issues to look out for
- Full NN Example in code

Neural Networks in Pytorch

As you know a neural network:

- Is a function connecting an input to an output
- Depends on (lots of) parameters

In Pytorch, a neural network is a class that implements the base class torch.nn.Module. You are provided with some pre-implemented networks such as torch.nn.Linear which is a single layer perceptron.

CLASS `torch.nn.Linear(in_features, out_features, bias=True)`

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

```
net = torch.nn.Linear(4,2)
```

Neural Networks in Pytorch

- The **.forward()** function applies the function

```
x = torch.arange(0,4).float()
y = net.forward(x)
y = net(x) # Alternatively
print(y)
```

```
tensor([-0.4807, -0.7048])
```

- **The .parameters()** method gives access to all the network parameters

```
for param in net.parameters():
    print(param)
```

```
Parameter containing:
tensor([[-0.1506,  0.3700, -0.4565,  0.4557],
        [-0.4525, -0.0645, -0.3689,  0.4634]])
Parameter containing:
tensor([ 0.1931,  0.3287])
```

Let’s write an MLP

```
class MyNet0(nn.Module):
    def __init__(self,input_size, hidden_size, output_size):
        super(MyNetworkWithParams,self).__init__()
        self.layer1_weights = nn.Parameter(torch.randn(input_size,hidden_size))
        self.layer1_bias = nn.Parameter(torch.randn(hidden_size))
        self.layer2_weights = nn.Parameter(torch.randn(hidden_size,output_size))
        self.layer2_bias = nn.Parameter(torch.randn(output_size))

    def forward(self,x):
        h1 = torch.matmul(x,self.layer1_weights) + self.layer1_bias
        h1_act = torch.max(h1, torch.zeros(h1.size())) # ReLU
        output = torch.matmul(h1_act,self.layer2_weights) + self.layer2_bias
        return output
net=MyNet0(4,16,2)
```

All attributes of Parameter type become network parameters

Let’s write an MLP

A better way:

```
class MyNet1(torch.nn.Module):
    def __init__(self,input_size, hidden_size, output_size):
        super().__init__()
        self.layer1 = torch.nn.Linear(input_size,hidden_size)
        self.layer2 = torch.nn.Sigmoid()
        self.layer3 = torch.nn.Linear(hidden_size,output_size)

    def forward(self, input_val):
        h = input_val
        h = self.layer1(h)
        h = self.layer2(h)
        h = self.layer3(h)
        return h
net = MyNet1(4,16,2)
```

You can use small networks inside big networks. Parameters of subnetworks will be “absorbed”

Let's write an MLP

Even better:

```
def generate_net(input_size,hidden_size, output_size):
    return nn.Sequential(nn.Linear(input_size,hidden_size),
                         nn.ReLU(),
                         nn.Linear(hidden_size,output_size))

net = generate_net(4,16,2)
```

This is a shortcut for simple feedforward networks.

So all you need in HW1 P2, but probably not in later homeworks

Let's write an MLP

Your own classes might be useful in bigger networks:

```python
def relu_mlp(size_list):
    layers = []
    for i in range(len(size_list)-2):
        layers.append(nn.Linear(size_list[i],size_list[i+1]))
        layers.append(nn.ReLU())
    layers.append(nn.Linear(size_list[-2],size_list[-1]))
    return nn.Sequential(*layers)

my_big_MLP = nn.Sequential(
    relu_mlp([1000,512,512,256]),
    nn.Sigmoid(),
    relu_mlp([256,128,64,32,10]))
```

Allows a sort of "tree structure"

Overview

- Neural Networks
- Tensors
- CPU and GPU Operations
- Backpropagation
- Neural Network Modules
- Optimization and Loss
- Saving and Loading
- Common Issues to look out for
- Full NN Example in code

Final Layers and Losses

torch.nn.CrossEntropyLoss includes both the softmax and the loss criterion and is stable (uses the log softmax)

```
x = torch.tensor([np.arange(4), np.zeros(4),np.ones(4)]).float()
y = torch.tensor([0,1,0])
criterion = nn.CrossEntropyLoss()

output = net(x)
loss = criterion(output,y)
print(loss)
```

```
tensor(2.4107)
```

Here the input x is 2-dimensional: it is a **batch** of input vectors (which is usually the case)

Use the Optimizer

You must use an optimizer subclass of **torch.nn.Optimizer.** The optimizer is initialized with the parameters that you want to update.

```
optimizer = torch.optim.SGD(net.parameters(), lr=0.01)
```

The **.step()** method will apply gradient descent on all these parameters, using the gradients they contain.

```
optimizer.step()
```

Use the Optimizer

Remember that gradients **accumulate** in Pytorch.

If you want to apply several iterations of gradient descent, gradients must be set to zero before each optimization step.

```
n_iter = 100
for i in range(n_iter):
    optimizer.zero_grad() # equivalent to net.zero_grad()
    output = net(x)
    loss = criterion(output,y)
    loss.backward()
    optimizer.step()
```

Overview

- Neural Networks
- Tensors
- CPU and GPU Operations
- Backpropagation
- Neural Network Modules
- Optimization and Loss
- Saving and Loading
- Common Issues to look out for
- Full NN Example in code

Saving and Loading

```
# get dictionary of keys to weights using `state_dict`
net = torch.nn.Sequential(
    torch.nn.Linear(28*28,256),
    torch.nn.Sigmoid(),
    torch.nn.Linear(256,10))
print(net.state_dict().keys())
```

```
odict_keys(['0.weight', '0.bias', '2.weight', '2.bias'])
```

```
# save a dictionary
torch.save(net.state_dict(),'test.t7')
# load a dictionary
net.load_state_dict(torch.load('test.t7'))
```

```
<All keys matched successfully>
```

Overview

- Neural Networks
- Tensors
- CPU and GPU Operations
- Backpropagation
- Neural Network Modules
- Optimization and Loss
- Saving and Loading
- Common Issues to look out for
- Full NN Example in code

Common Issues to Look Out For

Tensor Operations

- GPU + CPU
- Size mismatch in vector multiplications
- (*) is NOT matrix multiplication

```
x = 2* torch.ones(2,2)
y = 3* torch.ones(2,2)
print(x * y)
print(x.matmul(y))
```

```
tensor([[ 6.,  6.],
        [ 6.,  6.]])
tensor([[ 12.,  12.],
        [ 12.,  12.]])
```

Common Issues to Look Out For

Tensor Operations

- **.view()** is not transposition

```
x = torch.tensor([[1,2,3],[4,5,6]])
print(x)
print(x.t())
print(x.view(3,2))
```

```
tensor([[ 1,  2,  3],
        [ 4,  5,  6]])
tensor([[ 1,  4],
        [ 2,  5],
        [ 3,  6]])
tensor([[ 1,  2],
        [ 3,  4],
        [ 5,  6]])
```

GPU Memory Error

```
net = nn.Sequential(nn.Linear(2048,2048),nn.ReLU(),
                    nn.Linear(2048,2048),nn.ReLU(),
                    nn.Linear(2048,2048),nn.ReLU(),
                    nn.Linear(2048,2048),nn.ReLU(),
                    nn.Linear(2048,2048),nn.ReLU(),
                    nn.Linear(2048,2048),nn.ReLU(),
                    nn.Linear(2048,120))
x = torch.ones(256,2048)
y = torch.zeros(256).long()
net.cuda()
x.cuda()
crit=nn.CrossEntropyLoss()
out = net(x)
loss = crit(out,y)
loss.backward()
```

Common Issues to Look Out For

```
net = nn.Linear(4,2)
x = torch.tensor([1,2,3,4])
y = net(x)
print(y)
```

Is there a problem?

What is it?...

Common Issues to Look Out For

Type error

```
net = nn.Linear(4,2)
x = torch.tensor([1,2,3,4])
y = net(x)
print(y)
```

```
RuntimeError: Expected object of type torch.LongTensor but found type torch.FloatTensor
```

```
x = x.float()
x = torch.tensor([1.,2.,3.,4.])
```

Common Issues to Look Out For

```python
class MyNet(nn.Module):
    def __init__(self,n_hidden_layers):
        super(MyNet,self).__init__()
        self.n_hidden_layers=n_hidden_layers
        self.final_layer = nn.Linear(128,10)
        self.act = nn.ReLU()
        self.hidden = []
        for i in range(n_hidden_layers):
            self.hidden.append(nn.Linear(128,128))


    def forward(self,x):
        h = x
        for i in range(self.n_hidden_layers):
            h = self.hidden[i](h)
            h = self.act(h)
        out = self.final_layer(h)
        return out
```

What’s the problem?

Common Issues to Look Out For

Parameter Issue

```
class MyNet(nn.Module):
    def __init__(self,n_hidden_layers):
        super(MyNet,self).__init__()
        self.n_hidden_layers=n_hidden_layers
        self.final_layer = nn.Linear(128,10)
        self.act = nn.ReLU()
        self.hidden = []
        for i in range(n_hidden_layers):
            self.hidden.append(nn.Linear(128,128))


    def forward(self,x):
        h = x
        for i in range(self.n_hidden_layers):
            h = self.hidden[i](h)
            h = self.act(h)
        out = self.final_layer(h)
        return out
```

Hidden Layers are not module parameters

They will not be optimized

Common Issues to Look Out For

Solution

```python
class MyNet(nn.Module):
    def __init__(self,n_hidden_layers):
        super(MyNet,self).__init__()
        self.n_hidden_layers=n_hidden_layers
        self.final_layer = nn.Linear(128,10)
        self.act = nn.ReLU()
        self.hidden = []
        for i in range(n_hidden_layers):
            self.hidden.append(nn.Linear(128,128))
        self.hidden = nn.ModuleList(self.hidden)

    def forward(self,x):
        h = x
        for i in range(self.n_hidden_layers):
            h = self.hidden[i](h)
            h = self.act(h)
        out = self.final_layer(h)
        return out
```

Pytorch Debugging

If you have an error/bug in your code, or question about Pytorch:

- **Always try to figure it out by yourself first,** that's how you learn the most, for any strange behavior in your code, try printing the outputs, inputs, parameters and errors
- **Use the debugger:** `import pdb; pdb.set_trace()`
- **Tons of online resources,** great pytorch documentation, and basically every error is somewhere on stackoverflow.
- **Use Piazza -** First check if someone else has encountered the same bug before making a new post.
- **Come to office hours.**

Overview

- Neural Networks
- Tensors
- CPU and GPU Operations
- Backpropagation
- Neural Network Modules
- Optimization and Loss
- Saving and Loading
- Common Issues to look out for
- Full NN Example in code

Pytorch Example

Open the notebook MNIST_example.ipynb