Lecture 4: Introduction to Pytorch & Neural Networks

Contents

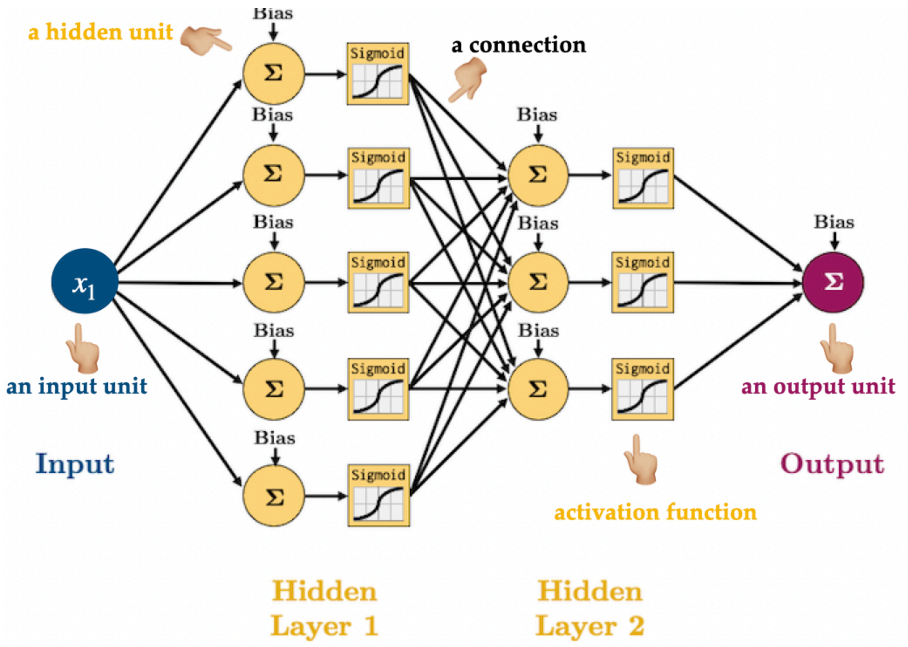
Lecture Learning Objectives

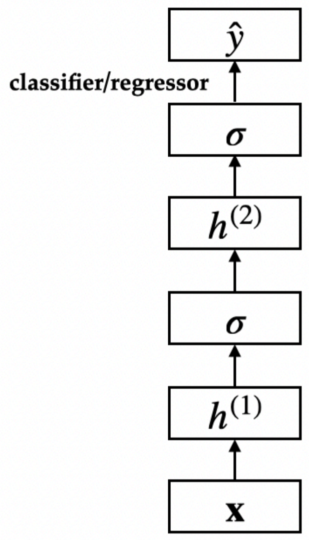
- Describe the difference between `Numpy` and `torch` arrays (`np.array` vs. `torch.Tensor`)
- Explain forward pass in neural networks using vector and matrix notation.
- Explain fundamental concepts of neural networks such as layers, nodes, activation functions, etc.
- Create a simple neural network in PyTorch for regression or classification

Imports

```python
import sys
import numpy as np
import pandas as pd
import sys, os
sys.path.append(os.path.join(os.path.abspath(".."), "code"))
from plotting import *

import torch
from torchsummary import summary
from torch import nn, optim
from torch.utils.data import DataLoader, TensorDataset
from sklearn.datasets import make_regression, make_circles, make_blobs
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression


DATA_DIR = DATA_DIR = os.path.join(os.path.abspath(".."), "data/")
```

Recap





Here are the computations in the forward pass

$$h_i^{(1)} = \phi^{(1)} \sum_j w_{ij}^{(1)} x_j + b_i^{(1)}$$

$$h_i^{(2)} = \phi^{(2)} \sum_j w_{ij}^{(2)} h_j^{(1)} + b_i^{(2)}$$

$$y_i = \sum_j w_{ij}^{(3)} h_j^{(2)} + b_i^{(3)}$$

Assume the following dimensionality

- $\mathbf{X}_{n\times d}$, i.e., each batch has n examples
- $\mathbf{W}^{(l)}_{\text{hidden\_size}\times\text{input\_size}}$, where hidden_size and input_size are of the corresponding layer
- The bias vectors $\mathbf{b}^{(l)}$ are row vectors, as they are typicallt stored as row vectors in PyTorch.

This can be written in vectorized form as:

$$\mathbf{h}^{(1)} = \phi^{(1)}(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)\mathrm{T}})$$

$$\mathbf{h}^{(2)} = \phi^{(2)}(\mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)\mathrm{T}})$$

$$\mathbf{y} = (\mathbf{W}^{(3)}\mathbf{h}^{(2)} + \mathbf{b}^{(3)\mathrm{T}})$$

And in matrix form as:

$$\mathbf{H}^{(1)} = \phi^{(1)}(\mathbf{X}\mathbf{W}^{(1)\mathrm{T}} + \mathbf{b}^{(1)})$$

$$\mathbf{H}^{(2)} = \phi^{(2)}(\mathbf{H}^{(1)}\mathbf{W}^{(2)\mathrm{T}} + \mathbf{b}^{(2)})$$

$$\mathbf{Y} = (\mathbf{H}^{(2)}\mathbf{W}^{(3)\mathrm{T}} + \mathbf{b}^{(3)})$$

- Assume broadcasting is used to add the bias vector to each row of the matrix.

? ? Questions for you

Exercise 4.1

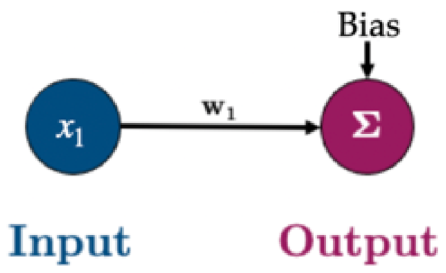
Select all of the following statements which are TRUE.

- (A) In a neural network, the number of nodes in the hidden layer must always be greater than the number of input nodes.
- (B) Activation functions in a neural network are used to introduce non-linearity into the network model.
- (C) In regression problems, it is advisable to apply a sigmoid function to the output, similar to other nodes.
- (D) The same activation function must be used for all neurons in a feedforward neural network.
- (E) During the forward pass, the bias in each neuron is added to the weighted sum of inputs before applying the activation function.

V's Solutions!

Motivation

So far, our focus has been on optimizing simple linear models using gradient descent.

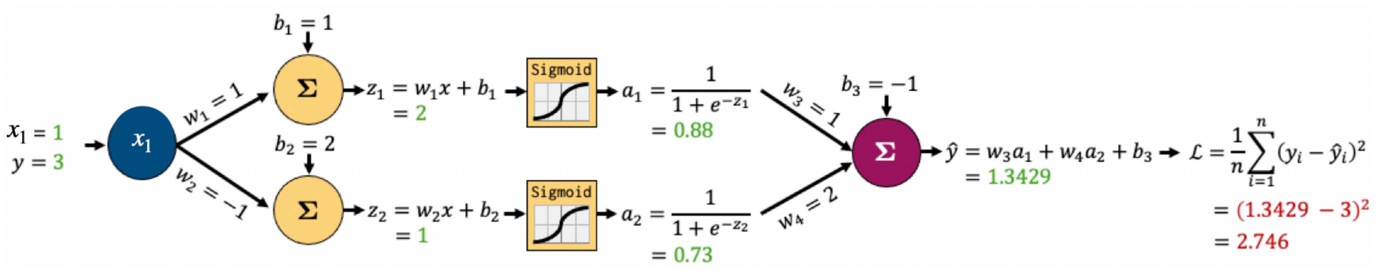


Here's the overview of this process:

- **Initialization:** Start with an initial set of parameters, often randomly chosen.
- **Forward pass:** Generate predictions using the current values of the parameters. (E.g., $\hat{y}_i = x_1 w_1 + Bias$ in the toy example above)
- **Loss calculation:** Evaluate the loss, which quantifies the discrepancy between the model's predictions and the actual target values.
- **Gradient calculation:** Compute the gradient of the loss function with respect to each parameter either on a batch or the full dataset. This gradient indicates the direction in which the loss is increasing and its magnitude.

- **Parameter Update**: Adjust the parameters in the opposite direction of the calculated gradient, scaled by the learning rate. This step aims to reduce the loss by moving the parameters toward values that minimize it.

How can we calculate the gradient of the loss with respect to the parameters in a more complicated network?


$x_1 = 1$
$y = 3$
x_1
$w_1 = 1$
$w_2 = -1$
$b_1 = 1$
$b_2 = 2$
Σ
$z_1 = w_1 x + b_1 = 2$
$z_2 = w_2 x + b_2 = 1$
Sigmoid
$a_1 = \frac{1}{1+e^{-z_1}} = 0.88$
$a_2 = \frac{1}{1+e^{-z_2}} = 0.73$
$w_3 = 1$
$w_4 = 2$
$b_3 = -1$
$\hat{y} = w_3 a_1 + w_4 a_2 + b_3 = 1.3429$
$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = (1.3429 - 3)^2 = 2.746$

- The parameters of the model are: w_1, w_2, w_3, w_4, b_1, b_2, b_3.
- To optimize these parameters, we need to calculate the gradients of the loss function $\mathcal{L}$ with respect to each parameter. So we need to calculate $\frac{\partial \mathcal{L}}{\partial w_1}$, $\frac{\partial \mathcal{L}}{\partial w_2}$, $\frac{\partial \mathcal{L}}{\partial w_3}$, $\frac{\partial \mathcal{L}}{\partial w_4}$, $\frac{\partial \mathcal{L}}{\partial b_1}$, $\frac{\partial \mathcal{L}}{\partial b_2}$, and $\frac{\partial \mathcal{L}}{\partial b_3}$
- To calculate $\frac{\partial \mathcal{L}}{\partial w_3}$, it's essential to determine $\frac{\partial \mathcal{L}}{\partial \hat{y}}$ and $\frac{\partial \hat{y}}{\partial w_3}$
- To calculate $\frac{\partial \mathcal{L}}{\partial w_1}$, we need the product of multiple derivatives $\frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_1} \frac{\partial a_1}{\partial z_1} \frac{\partial z_1}{\partial w_1}$
- As the network becomes more complex, this process of gradient calculation grows increasingly intricate.
- This method of efficiently computing these gradients is known as **backpropagation** or **backprop** for short.
- In our next lecture, we'll delve into a toy example of backpropagation. For the moment, let's focus on how we can leverage PyTorch to automate the gradient computation using backpropagation.

What do we want from a deep learning framework?

- **Automatic Differentiation:** Essential for efficiently computing gradients
- **GPU Support:** To leverage the power of GPUs for faster computation
- **Optimization and Inspection of Computation Graph:** Tools to optimize performance and the ability to inspect or visualize the computation graph for debugging and understanding model behaviour.

There are several popular deep-learning frameworks

- PyTorch by Facebook AI
- TensorFlow and Keras by Google
- Apache MXNet
- Microsoft Cognitive Toolkit (CNTK)

Here and here you will find some resources discussing PyTorch vs. TensorFlow.

Introduction to PyTorch

Why PyTorch

PyTorch is a Python-based tool for scientific computing. Some pros of PyTorch are

- Python-friendly
- Good documentation and community support
- Open source
- Plenty of projects out there using PyTorch
- Dynamic graphs

In general, PyTorch does two main things:

- Provides an n-dimensional array object similar to that of `Numpy`, with the difference that it can be manipulated using GPUs
- Computes gradients (through automatic differentiation)

PyTorch’s Tensor

- In PyTorch a tensor is just like NumPy’s `ndarray` that we have become so familiar with.
- A key difference between PyTorch’s `torch.Tensor` and Numpy’s `np.array` is that `torch.Tensor` was constructed to integrate with GPUs and PyTorch’s computational graphs

`ndarray` vs `tensor`

- Creating and working with tensors is much the same as with Numpy `ndarrays`
- You can create a tensor with `torch.tensor()` in various ways:

```
a = torch.tensor([1, 2, 3.])
```

```
b = torch.tensor([1, 2, 3])
```

Let's see the datatype of each tensor:

```
for _ in [a, b]:
    print(f"{_}, dtype: {_.dtype}")
```

```
tensor([1., 2., 3.]), dtype: torch.float32
tensor([1, 2, 3]), dtype: torch.int64
```

- PyTorch comes with most of the `Numpy` functions we're already familiar with:

```
torch.zeros(2, 2)  # zeroes
```

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

```
torch.ones(2, 2)  # ones
```

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

```
8.4119e-01 + 1.0
```

```
1.84119
```

```
torch.randn(3, 2)  # random normal
```

```
tensor([[ 3.3095e-01,  8.4195e-02],
        [-5.8819e-02, -2.2231e-02],
        [ 1.7321e+00,  1.3052e-03]])
```

```
torch.rand(2, 3, 2)  # rand uniform
```

```
tensor([[[0.6557, 0.0705],
         [0.2940, 0.8813],
         [0.3465, 0.9291]],

        [[0.5641, 0.8527],
         [0.2531, 0.7962],
         [0.4087, 0.9570]]])
```

- Just like in NumPy we can look at the shape of a tensor with the `.shape` attribute:

```
x = torch.rand(2, 3, 2, 2)
x.shape
```

```
torch.Size([2, 3, 2, 2])
```

```
x.ndim
```

```
4
```

Tensors and Data Types

- Different dtypes have different memory and computational implications (see the end of Lecture 1)
- In Pytorch we'll be building networks that require thousands, millions, or even billions of floating point calculations
- In such cases, using a smaller dtype like `float32` can significantly speed up computations and reduce memory requirements
- The **default float dtype in Pytorch is `float32`**, as opposed to Numpy's `float64`
- In fact some operations in Pytorch will even throw an error if you pass a high-memory `dtype`

```
torch.tensor([1., 2]).dtype
```

```
torch.float32
```

```
print(np.array([3.14159]).dtype)
print(torch.tensor([3.14159]).dtype)
```

```
float64
torch.float32
```

- But just like in Numpy, you can always specify the particular dtype you want using the `dtype` argument:

```
print(torch.tensor([3.14159], dtype=torch.float64).dtype)
```

```
torch.float64
```

Operations on Tensors

- Tensors operate just like `ndarrays` and have a variety of familiar methods that can be called off them:

```
a = torch.rand(1, 3)
b = torch.rand(3, 1)
```

```
a
```

```
tensor([[0.5859, 0.1689, 0.0097]])
```

```
b
```

```
tensor([[0.5054],
        [0.7828],
        [0.2662]])
```

```
a + b  # broadcasting betweean a 1 x 3 and 3 x 1 tensor
```

```
tensor([[1.0914, 0.6743, 0.5152],
        [1.3687, 0.9517, 0.7925],
        [0.8521, 0.4351, 0.2759]])
```

```
h1 = torch.rand(10, 5)
b1 = torch.rand(1, 5)
```

```
h1, b1
```

```
(tensor([[0.9325, 0.9678, 0.5175, 0.8935, 0.3224],
         [0.6078, 0.6781, 0.1328, 0.8796, 0.0106],
         [0.6879, 0.1605, 0.8332, 0.6184, 0.6062],
         [0.5368, 0.7816, 0.2784, 0.4438, 0.2548],
         [0.7811, 0.7999, 0.1664, 0.2497, 0.6277],
         [0.9849, 0.8704, 0.2895, 0.6063, 0.2859],
         [0.1511, 0.0261, 0.1775, 0.5240, 0.3120],
         [0.0372, 0.9534, 0.9307, 0.2683, 0.3008],
         [0.8368, 0.6715, 0.2800, 0.3972, 0.0025],
         [0.6760, 0.5109, 0.6439, 0.7346, 0.5061]]),
 tensor([[0.8902, 0.8157, 0.6049, 0.6193, 0.4495]]))
```

```
h1 + b1 # broadcasting between a 10 x 5 and 1 x 5 tensor
```

```
tensor([[1.8226, 1.7836, 1.1223, 1.5128, 0.7719],
        [1.4979, 1.4939, 0.7377, 1.4989, 0.4601],
        [1.5781, 0.9763, 1.4380, 1.2377, 1.0558],
        [1.4270, 1.5973, 0.8833, 1.0632, 0.7043],
        [1.6712, 1.6156, 0.7712, 0.8691, 1.0773],
        [1.8751, 1.6862, 0.8943, 1.2256, 0.7354],
        [1.0413, 0.8418, 0.7824, 1.1434, 0.7615],
        [0.9274, 1.7691, 1.5356, 0.8876, 0.7504],
        [1.7270, 1.4873, 0.8849, 1.0165, 0.4521],
        [1.5662, 1.3266, 1.2487, 1.3540, 0.9556]])
```

```
a,b
```

```
(tensor([[0.5859, 0.1689, 0.0097]]),
 tensor([[0.5054],
         [0.7828],
         [0.2662]]))
```

```
a * b  # element-wise multiplication
```

```
tensor([[0.2961, 0.0854, 0.0049],
        [0.4587, 0.1322, 0.0076],
        [0.1560, 0.0450, 0.0026]])
```

```
a @ b # matrix multiplication
```

```
tensor([[0.4309]])
```

```
a.mean()
```

```
tensor(0.2548)
```

```
a.sum()
```

```
tensor(0.7645)
```

Indexing

- Once again, same as Numpy

```
X = torch.rand(5, 2)
print(X)
```

```
tensor([[0.5990, 0.4399],
        [0.5381, 0.7130],
        [0.0608, 0.4086],
        [0.7359, 0.2971],
        [0.4526, 0.6855]])
```

```
print(X[0, :])
print(X[0])
print(X[:, 0])
```

```
tensor([0.5990, 0.4399])
tensor([0.5990, 0.4399])
tensor([0.5990, 0.5381, 0.0608, 0.7359, 0.4526])
```

Numpy Bridge

- Sometimes we might want to convert a tensor back to a NumPy array
- We can do that using the `.Numpy()` method

```
X = torch.rand(3,3)
print(type(X))
X_numpy = X.numpy()
print(type(X_numpy))
```

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

Using GPU with PyTorch

- GPU is a graphical processing unit (as opposed to a CPU: central processing unit)
- GPUs were originally developed for gaming. They are very fast at performing operations on large amounts of data by performing them in parallel (think about updating the value of all pixels on a screen very quickly as a player moves around in a game)
- More recently, GPUs have been adapted for more general purpose programming
- Neural networks can typically be broken into smaller computations that can be performed in parallel on a GPU
- PyTorch is tightly integrated with CUDA (Compute Unified Device Architecture), a software layer developed by Nvidia that facilitates interactions with an Nvidia GPU (if you have one)
- You can check if you have a CUDA GPU:

```
torch.cuda.is_available()
```

```
False
```

- In May 2022, PyTorch also announced GPU-accelerated PyTorch training on Mac (see here) using Apple's Metal Performance Shaders (MPS).
- If you're using a Mac equipped with **Apple silicon** (M1 or M2), you can benefit from its GPU cores to train your PyTorch models:

```
torch.backends.mps.is_available()
```

```
True
```

- When training on a machine that has a GPU, you need to tell PyTorch you want to use it
- You'll see the following at the top of most PyTorch code:

```
# device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
device = torch.device('mps' if torch.backends.mps.is_available() else 'cpu')
print(device)
```

```
mps
```

- You can then use the `device` argument when creating tensors to specify whether you wish to use a CPU or GPU
- Or if you want to move a tensor between the CPU and GPU, you can use the `.to()` method:

```
X = torch.rand(2, 2, 2, device=device)
X.device
```

```
device(type='mps', index=0)
```

```
X.to('cpu')
```

```
tensor([[[0.4596, 0.1801],
         [0.6962, 0.1487]],

        [[0.6878, 0.0603],
         [0.6538, 0.8214]]])
```

```
X.device
```

```
device(type='mps', index=0)
```

```
# X.to('cuda')  # will give me an error as I don't have a CUDA GPU
```

Gradient computation

We'll learn in the next lecture that the process of using the chain rule in a backward manner (from the last computation back to its root at the variable) is called **back-propagation**. For now, let's just learn how we make PyTorch compute the gradient for us using back-propagation.

PyTorch provides a `.backward()` method on every tensor that takes part in gradient computation. This enables us to **do gradient descent without worrying about the structure of our model**, since we no longer need to compute the derivative ourselves.

At a high-level, the idea is that the loss function is a series of elementary computations performed on the weight parameters, along with some constant data points and biases. So if we track computations that involve the weights all the way to the loss function, we can compute the gradient of the loss function using back-propagation w.r.t to the weights. More on this in the next lecture.

If we now define `w` using PyTorch tensors, we can use `.backward()` on our loss function to compute the derivative w.r.t to `w`:

```
X = torch.tensor([1.0, 2.0, 3.0], requires_grad=False)
w = torch.tensor([1.0], requires_grad=True)  # Random initial weight
y = torch.tensor([2.0, 4.0, 6.0], requires_grad=False)  # Target values
mse = ((X * w - y)**2).mean()
mse.backward()
w.grad
```

```
tensor([-9.3333])
```

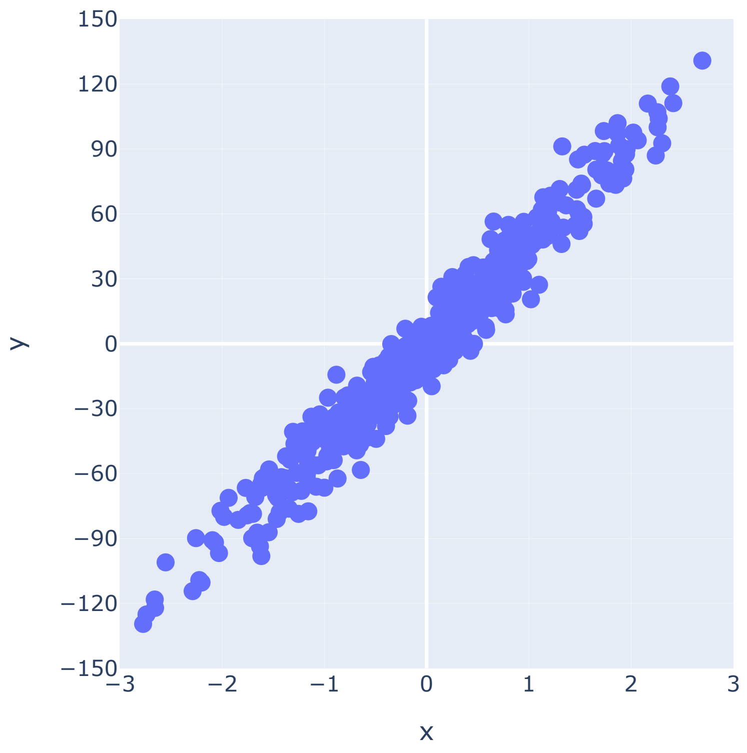
```
X@(X*w - y) * (2/3)
```

```
tensor(-9.3333, grad_fn=<MulBackward0>)
```

Simple Linear Regression with PyTorch

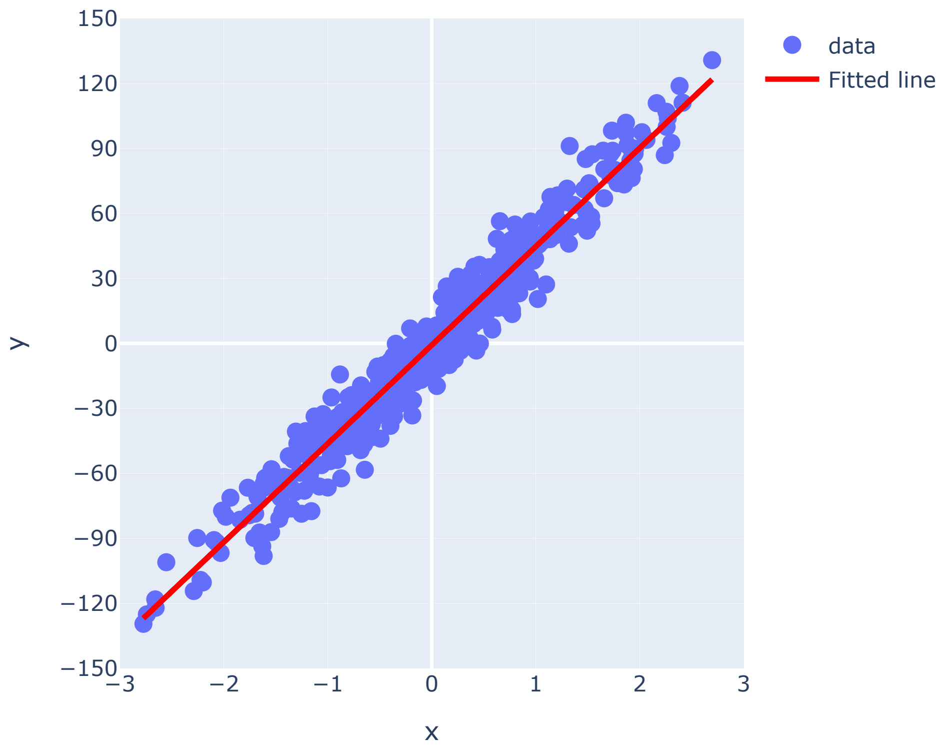
- Let’s create a simple regression dataset with 500 observations:

```
X, y = make_regression(n_samples=500, n_features=1, random_state=0, noise=10.0)
plot_regression(X, y)
```



- We know how to fit a simple linear regression to this data using sklearn:

```python
sk_model = LinearRegression().fit(X, y)
plot_regression(X, y, sk_model.predict(X))
```



- Here are the parameters of that fitted line:

```python
print(f"w_0: {sk_model.intercept_:.2f} (bias/intercept)")
print(f"w_1: {sk_model.coef_[0]:.2f}")
```

```
w_0: -0.77 (bias/intercept)
w_1: 45.50
```

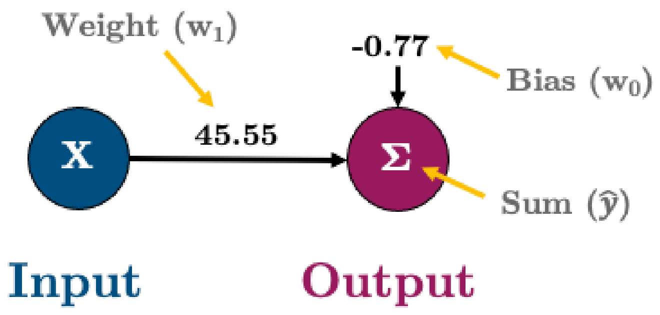
- As an equation, that looks like this:

$$\hat{y} = -0.77 + 45.50x$$

- Or in matrix form:

$$\begin{bmatrix}\hat{y_1}\\ \hat{y_2}\\ \vdots\\ \hat{y_n}\end{bmatrix} = \begin{bmatrix}1 & x_{11}\\ 1 & x_{21}\\ \vdots & \vdots\\ 1 & x_{n1}\end{bmatrix}\begin{bmatrix}-0.77\\ 45.55\end{bmatrix}$$

- Or in graph form I'll represent it like this:



Linear Regression with a Neural Network in PyTorch

- So let's implement the above in PyTorch to start gaining an intuition about neural networks
- Every neural network model you build in PyTorch has to inherit from `torch.nn.Module`
- Remember class inheritance from DSCI 511? Inheritance allows us to inherit commonly needed functionality without having to write code ourselves

- Think about sklearn models: they all inherit common methods like `.fit()`, `.predict()`, `.score()`, etc. When creating a neural network, we define our own architecture but still want common functionality which we inherit from `torch.nn.Module`.
- Let's create a model called `linearRegression` and then I'll talk you through the syntax:

```
class linearRegression(nn.Module):  # our class inherits from nn.Module and we can call

    def __init__(self, input_size, output_size):
        super().__init__()  # super().__init__() makes our class inherit everything from

        self.linear = nn.Linear(input_size, output_size,)  # this is a simple linear lay

    def forward(self, x):
        out = self.linear(x)
        return out
```

Let's step through the above:

```
class linearRegression(nn.Module):

    def __init__(self, input_size, output_size):
        super().__init__()
```

Here we're creating a class called `linearRegression` and inheriting the methods and attributes of `nn.Module`

(hint: try typing `help(linearRegression)` to see all the things we inherited from `nn.Module`).

```
        self.linear = nn.Linear(input_size, output_size)
```

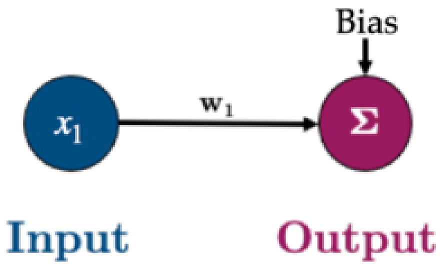
Here we're defining a "Linear" layer, which just means `wX + b`, i.e., the weights of the network, multiplied by the inputs plus the bias.

```
    def forward(self, x):
        out = self.linear(x)
        return out
```

PyTorch networks created with `nn.Module` must have a `forward()` method. It accepts the input data `x` and passes it through the defined operations. In this case, we are passing `x` into our linear layer and getting an output `out`.

- After defining the model class, we can create an instance of that class:

```
model = linearRegression(input_size=1, output_size=1)
```



- We can check out our model using `print()`:

```python
print(model)
```

```
linearRegression(
  (linear): Linear(in_features=1, out_features=1, bias=True)
)
```

- Or the more useful `summary()` (which we imported at the top of this notebook with `from torchsummary import summary`):

```python
summary(model);
```

```
=================================================================
Layer (type:depth-idx)                   Param #
=================================================================
├─Linear: 1-1                            2
=================================================================
Total params: 2
Trainable params: 2
Non-trainable params: 0
=================================================================
```

- Notice how we have two parameters? We have one for the weight (`w1`) and one for the bias (`w0`)
- These were initialized **randomly** by PyTorch when we created our model. They can be accessed with `model.state_dict()`:

```python
model.state_dict()
```

```
OrderedDict([('linear.weight', tensor([[-0.8378]])),
             ('linear.bias', tensor([-0.1262]))])
```

- Our `X` and `y` data are currently Numpy arrays but they need to be PyTorch tensors

- Let's convert them:

```
X_t = torch.tensor(X, dtype=torch.float32)
y_t = torch.tensor(y, dtype=torch.float32)
```

```
X_t.shape
```

```
torch.Size([500, 1])
```

- We have a working model right now and could tell it to give us some output with this syntax:

```
model(X_t[0])
```

```
tensor([-0.6459], grad_fn=<ViewBackward0>)
```

That's just a raw prediction, and far from the actual value of:

```
y_t[0]
```

```
tensor(31.0760)
```

It's because our model is not trained yet.

What does training mean? In the context of what we've learned so far, it means that we haven't yet done an SGD run to find optimal weights.

- As we learned in the past few lectures, to fit our model we need:
  1. **a loss function** (called "`criterion`" in PyTorch) to tell us how good/bad our predictions are. We'll use mean squared error, `torch.nn.MSELoss()`. See the list of different loss functions in PyTorch here.
  2. **an optimization algorithm** to help optimize model parameters. We'll use SGD, `torch.optim.SGD()`. See the list of different optimization algorithms in PyTorch here.

```
LEARNING_RATE = 0.02
criterion = nn.MSELoss()  # loss function
optimizer = torch.optim.SGD(model.parameters(), lr=LEARNING_RATE)  # optimization algori
```

- Before we train I'm going to create a **data loader** to help batch my data
- We'll talk more about these in the next lecture and in lab but they are just generators that yield data to us on request (remember generators from 511?)
- We'll use a `BATCH_SIZE=50` (which should give us 10 batches because we have 500 data points)

```
BATCH_SIZE = 50
dataset = TensorDataset(X_t, y_t)
dataloader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True)
dataloader
```

```
<torch.utils.data.dataloader.DataLoader at 0x165ed9670>
```

- We should have 10 batches:

```
len(dataloader)
```

```
10
```

- We can look at a batch using this syntax:

```
XX, yy = next(iter(dataloader))
```

```
print(f"Shape of feature data (X) in batch: {XX.shape}")
print(f"Shape of response data (y) in batch: {yy.shape}")
```

```
Shape of feature data (X) in batch: torch.Size([50, 1])
Shape of response data (y) in batch: torch.Size([50])
```

Let's write code for doing a typical SGD with 10 epochs, but using automatic differentiation of PyTorch:

```python
def trainer(model, criterion, optimizer, dataloader, epochs=5, verbose=True):
    """Simple training wrapper for PyTorch network."""

    for epoch in range(epochs):
        losses = 0

        for X, y in dataloader:

            optimizer.zero_grad()       # Clear gradients w.r.t. parameters
            y_hat = model(X).flatten()  # Forward pass to get output
            loss = criterion(y_hat, y)  # Calculate loss
            loss.backward()             # Getting gradients w.r.t. parameters
            optimizer.step()            # Update parameters
            losses += loss.item()       # Add loss for this batch to running total

        if verbose: print(f"epoch: {epoch + 1}, loss: {losses / len(dataloader):.4f}")
```

OK, before starting the training, here are the model parameters before training for reference:

```python
model.state_dict()
```

```
OrderedDict([('linear.weight', tensor([[-0.8378]])),
             ('linear.bias', tensor([-0.1262]))])
```

```python
trainer(model, criterion, optimizer, dataloader, epochs=30, verbose=True)
```

```
epoch: 1, loss: 1621.4079
epoch: 2, loss: 768.1087
epoch: 3, loss: 392.0068
epoch: 4, loss: 225.3830
epoch: 5, loss: 151.7468
epoch: 6, loss: 119.4527
epoch: 7, loss: 104.7970
epoch: 8, loss: 98.4027
epoch: 9, loss: 95.5253
epoch: 10, loss: 94.3038
epoch: 11, loss: 93.6724
epoch: 12, loss: 93.3628
epoch: 13, loss: 93.3368
epoch: 14, loss: 93.2055
epoch: 15, loss: 93.1900
epoch: 16, loss: 93.2099
epoch: 17, loss: 93.1983
epoch: 18, loss: 93.2386
epoch: 19, loss: 93.2982
epoch: 20, loss: 93.2073
epoch: 21, loss: 93.2578
epoch: 22, loss: 93.2604
epoch: 23, loss: 93.1815
epoch: 24, loss: 93.2830
epoch: 25, loss: 93.2164
epoch: 26, loss: 93.1991
epoch: 27, loss: 93.1660
epoch: 28, loss: 93.2206
epoch: 29, loss: 93.2389
epoch: 30, loss: 93.2719
```

- Now our model has been trained, our parameters should be different than before:

```
model.state_dict()
```

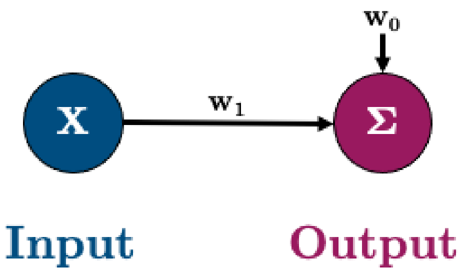
```
OrderedDict([('linear.weight', tensor([[45.4673]])),
             ('linear.bias', tensor([-0.7401]))])
```

- Comparing to our sklearn model, we get the same answer:

```
pd.DataFrame({"w0": [sk_model.intercept_, model.state_dict()['linear.bias'].item()],
              "w1": [sk_model.coef_[0], model.state_dict()['linear.weight'].item()]},
             index=['sklearn', 'pytorch']).round(2)
```

	w0	w1
sklearn	-0.77	45.50
pytorch	-0.74	45.47

- We got pretty close
- We could do better by changing the number of epochs or the learning rate
- So here is our simple network once again:



- By the way, check out what happens if we run `trainer()` again:

```
trainer(model, criterion, optimizer, dataloader, epochs=20, verbose=True)
```

```
epoch: 1, loss: 93.2702
epoch: 2, loss: 93.2755
epoch: 3, loss: 93.2373
epoch: 4, loss: 93.2641
epoch: 5, loss: 93.2035
epoch: 6, loss: 93.2739
epoch: 7, loss: 93.2235
epoch: 8, loss: 93.2000
epoch: 9, loss: 93.2454
epoch: 10, loss: 93.2384
epoch: 11, loss: 93.2720
epoch: 12, loss: 93.2199
epoch: 13, loss: 93.1864
epoch: 14, loss: 93.1870
epoch: 15, loss: 93.2335
epoch: 16, loss: 93.1753
epoch: 17, loss: 93.1911
epoch: 18, loss: 93.3436
epoch: 19, loss: 93.1874
epoch: 20, loss: 93.2109
```

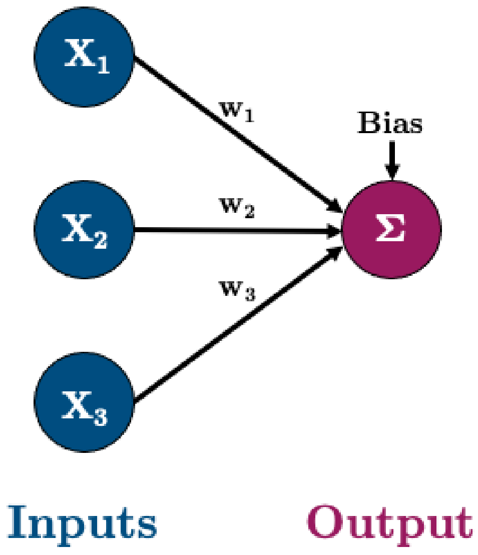
- **Our model continues where we left off**
- This may or may not be what you want. We can start from scratch by re-making the `model` and `optimizer`.

```
pd.DataFrame({"w0": [sk_model.intercept_, model.state_dict()['linear.bias'].item()],
              "w1": [sk_model.coef_[0], model.state_dict()['linear.weight'].item()]},
             index=['sklearn', 'pytorch']).round(2)
```

	w0	w1
sklearn	-0.77	45.5
pytorch	-0.77	45.5

Multiple Linear Regression with a Neural Network

- Okay, let's do a multiple linear regression now with 3 features
- So our network will look like this:



- Let's go ahead and create some data:

```
# Create dataset
X, y = make_regression(n_samples=500, n_features=3, random_state=0, noise=10.0)
X_t = torch.tensor(X, dtype=torch.float32)
y_t = torch.tensor(y, dtype=torch.float32)

# Create dataloader
dataset = TensorDataset(X_t, y_t)
dataloader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True)
```

- And let’s create the above model:

```
X_t.shape
```

```
torch.Size([500, 3])
```

```
model = linearRegression(input_size=3, output_size=1)
```

- We should now have 4 parameters (3 weights and 1 bias)

```
summary(model, (3,));
```

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Linear: 1-1                            [-1, 1]                   4
==========================================================================================
Total params: 4
Trainable params: 4
Non-trainable params: 0
Total mult-adds (M): 0.00
==========================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.00
Estimated Total Size (MB): 0.00
==========================================================================================
```

- Looks good to me! Let’s train the model and then compare it to sklearn’s `LinearRegression()`:

```
model.state_dict()
```

```
OrderedDict([('linear.weight', tensor([[-0.2864, -0.4463, -0.4538]])),
             ('linear.bias', tensor([0.0270]))])
```

```
criterion = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.05)
trainer(model, criterion, optimizer, dataloader, epochs=100, verbose=False)
```

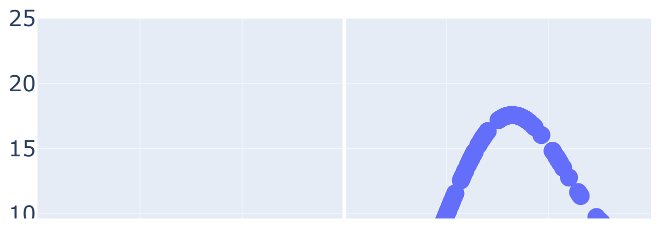
```
sk_model = LinearRegression().fit(X, y)
pd.DataFrame({"w0": [sk_model.intercept_, model.state_dict()['linear.bias'].item()],
              "w1": [sk_model.coef_[0], model.state_dict()['linear.weight'][0, 0].item()
              "w2": [sk_model.coef_[1], model.state_dict()['linear.weight'][0, 1].item()
              "w3": [sk_model.coef_[2], model.state_dict()['linear.weight'][0, 2].item()
             index=['sklearn', 'pytorch']).round(2)
```

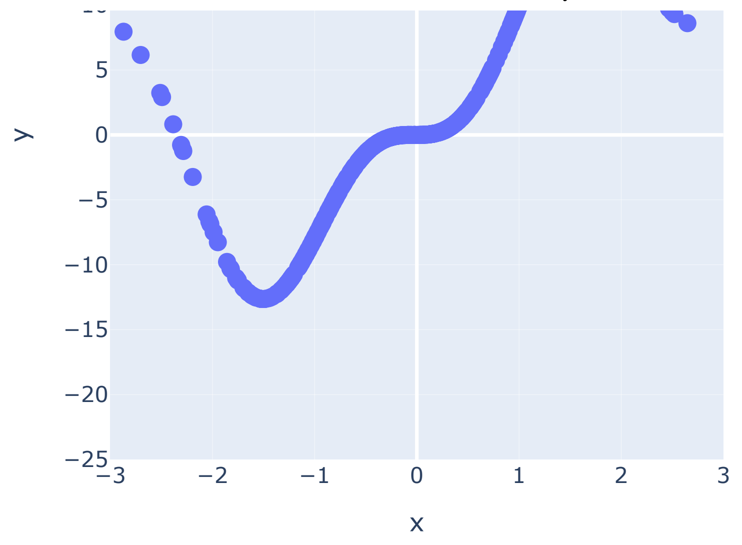
	w0	w1	w2	w3
sklearn	0.43	0.62	55.99	11.14
pytorch	0.25	0.85	55.89	11.07

Non-linear Regression with a Neural Network

- Okay so we can make a simple network to imitate simple and multiple *linear* regression
- For example, what happens when we have more complicated datasets like this?

```
# Create dataset
np.random.seed(2020)

X = np.sort(np.random.randn(500))
y = X ** 2 + 15 * np.sin(X) **3

X_t = torch.tensor(X[:, None], dtype=torch.float32)
y_t = torch.tensor(y, dtype=torch.float32)

# Create dataloader
dataset = TensorDataset(X_t, y_t)
dataloader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True)

plot_regression(X, y, y_range=[-25, 25], dy=5)
```





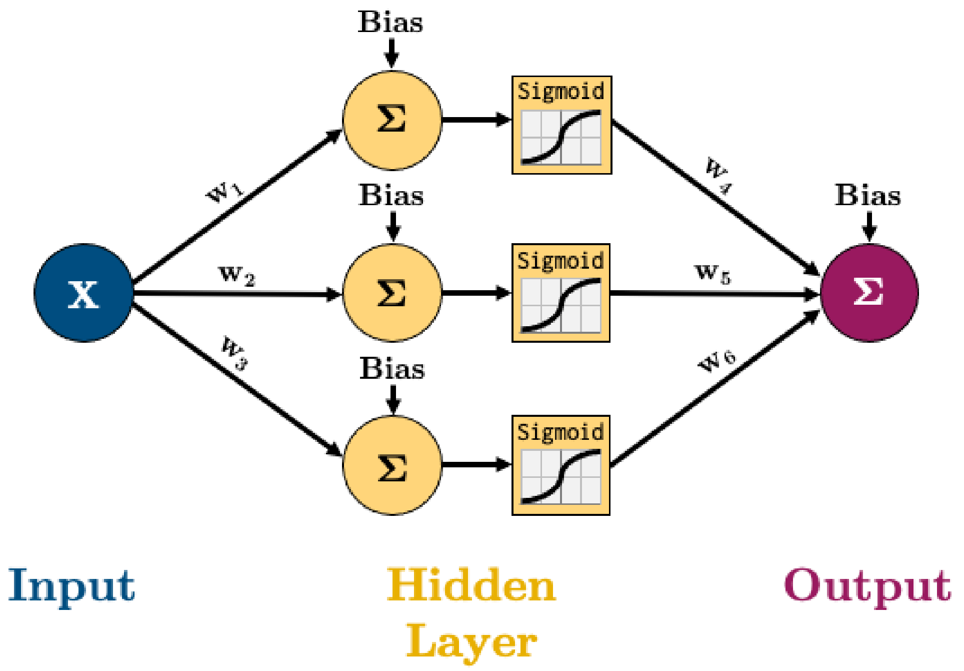
- This is obviously non-linear, and we need to introduce some **non-linearities** into our network
- These non-linearities are what make neural networks so powerful and they are called **"activation functions"**
- We are going to create a new model class that includes a non-linearity, that is, a sigmoid function:

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

- We'll talk more about activation functions later, but note how the sigmoid function non-linearly maps `x` to a value between 0 and 1
- Okay, so let's create the following network:

```
X_t.shape
```

```
torch.Size([500, 1])
```



- All this means is that the value of each node in the hidden layer will be transformed by the “activation function”, thus introducing non-linear elements to our model
- There’s **two main ways** of creating the above model in PyTorch, I’ll show you both:

```python
class nonlinRegression(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()

        self.hidden = nn.Linear(input_size, hidden_size)
        self.output = nn.Linear(hidden_size, output_size)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.hidden(x)       # input -> hidden layer
        x = self.sigmoid(x)      # sigmoid activation function in hidden layer
        x = self.output(x)       # hidden -> output layer
        return x
```

- Note how our `forward()` method now passes `x` through the `nn.Sigmoid()` function after the hidden layer
- The above method is very clear and flexible, but I prefer using `nn.Sequential()` to combine my layers together in the constructor:

```
class nonlinRegression(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()

        self.main = torch.nn.Sequential(
            nn.Linear(input_size, hidden_size),  # input -> hidden layer
            nn.Sigmoid(),                        # sigmoid activation function in hidde
            nn.Linear(hidden_size, output_size)  # hidden -> output layer
        )

    def forward(self, x):
        x = self.main(x)
        return x
```

- Let's make an instance of our new class and confirm it has 10 parameters (6 weights + 4 biases):

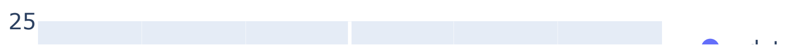
```
model = nonlinRegression(1, 3, 1)
summary(model, (1,));
```

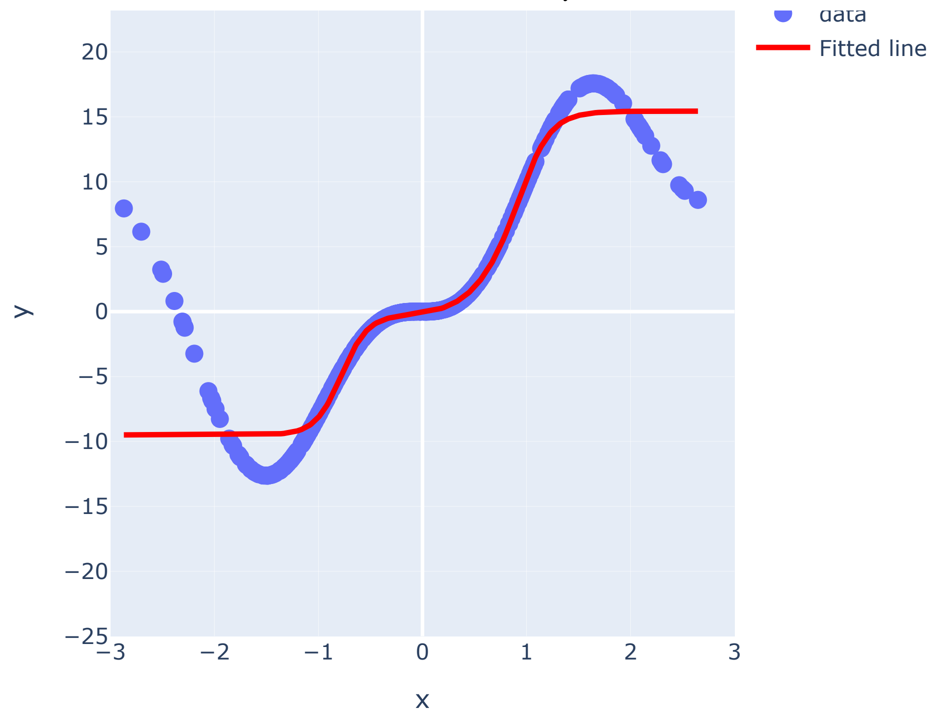
```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Sequential: 1-1                        [-1, 1]                   --
|    └─Linear: 2-1                       [-1, 3]                   6
|    └─Sigmoid: 2-2                      [-1, 3]                   --
|    └─Linear: 2-3                       [-1, 1]                   4
==========================================================================================
Total params: 10
Trainable params: 10
Non-trainable params: 0
Total mult-adds (M): 0.00
==========================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.00
Estimated Total Size (MB): 0.00
==========================================================================================
```

- Okay, let's train:

```
criterion = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

trainer(model, criterion, optimizer, dataloader, epochs=1000, verbose=False)
```

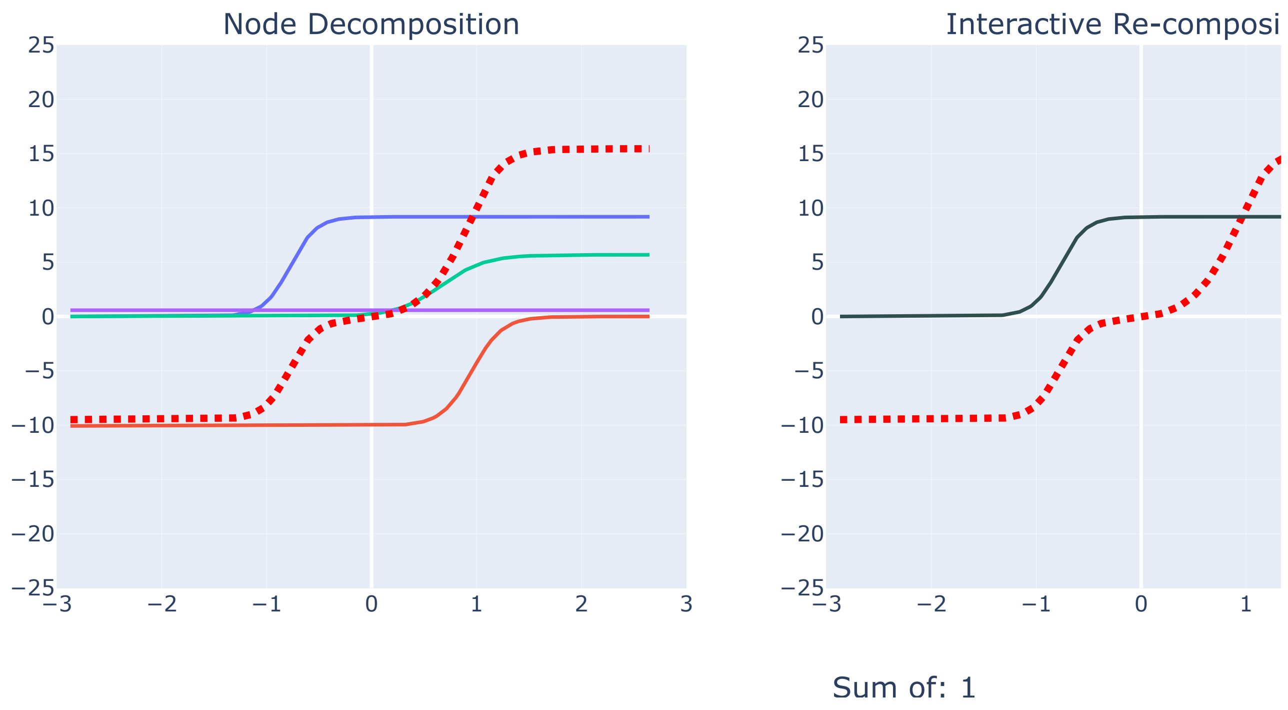
```
y_p = model(X_t).detach().numpy().squeeze()
plot_regression(X, y, y_p, y_range=[-25, 25], dy=5)
```

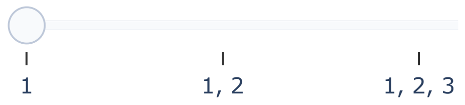




- Take a look at those non-linear predictions
- Our model is not great and we could make it better soon by adjusting the learning rate, the number of nodes, and the number of epochs
- I want you to see how **each of our hidden nodes is engineering a non-linear feature** to be used for the predictions, check it out:

```
plot_nodes(X, y_p, model)
```





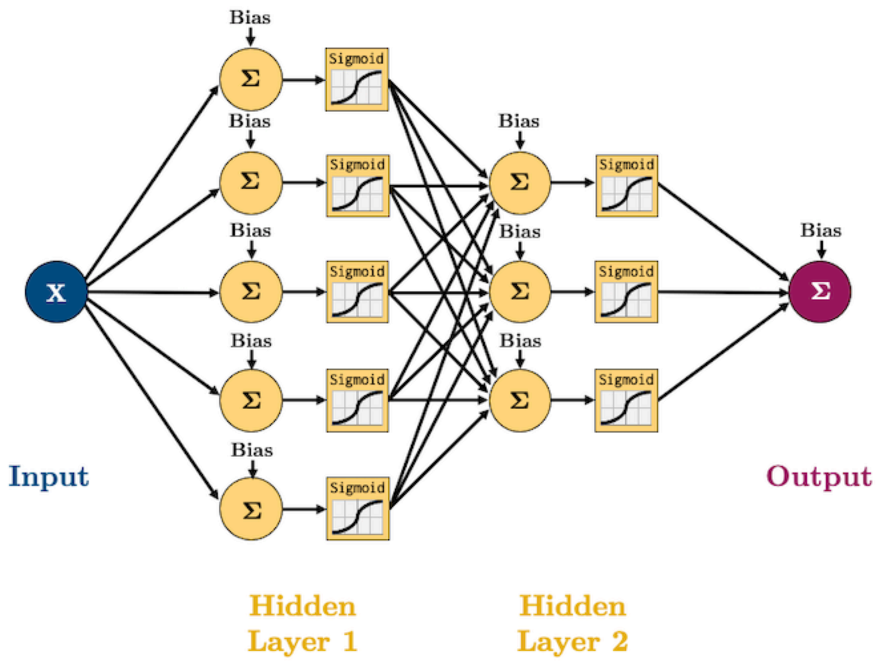
Deep Learning

You've probably heard the magic term **"deep learning"** and you're about to find out what it means

- **Deep neural network: a neural network with more than 1 hidden layer**
- On the other hand, a neural network with only 1 hidden layer is called a **shallow neural network**.

I like to think of each layer as a "feature engineer", it is trying to extract information from the layer before it

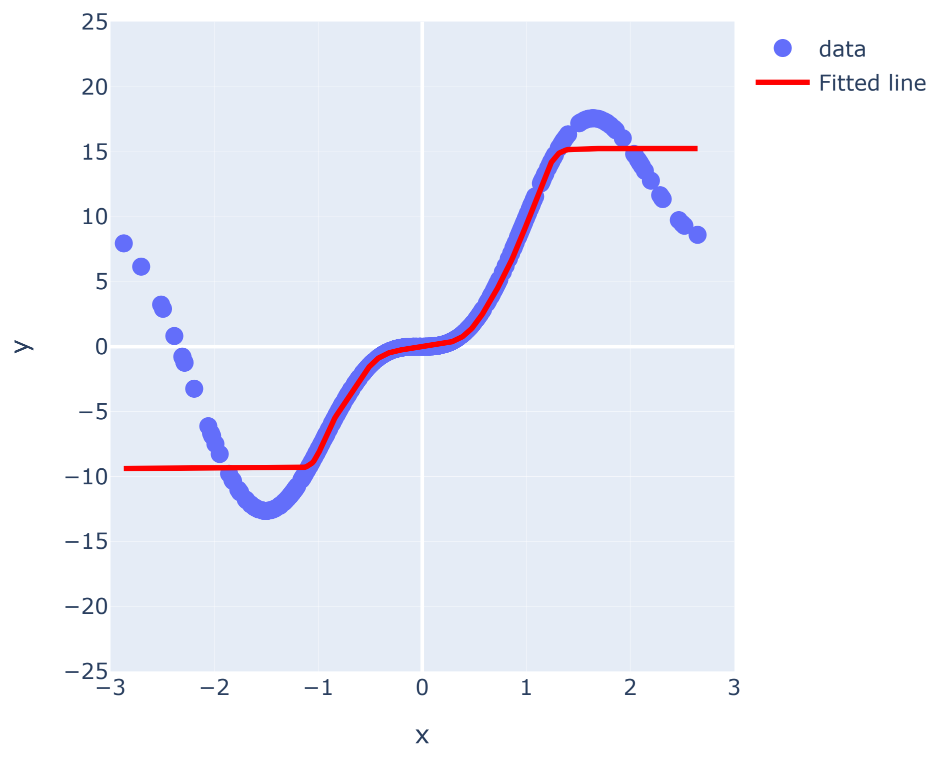
- Let's create a "deep" network of 2 layers:



```
class deepRegression(nn.Module):
    def __init__(self, input_size, hidden_size_1, hidden_size_2, output_size):
        super().__init__()
        self.main = nn.Sequential(
            nn.Linear(input_size, hidden_size_1),
            nn.Sigmoid(),
            nn.Linear(hidden_size_1, hidden_size_2),
            nn.Sigmoid(),
            nn.Linear(hidden_size_2, output_size)
        )

    def forward(self, x):
        out = self.main(x)
        return out
```

```
model = deepRegression(1, 5, 3, 1)
optimizer = torch.optim.SGD(model.parameters(), lr=0.3)
```

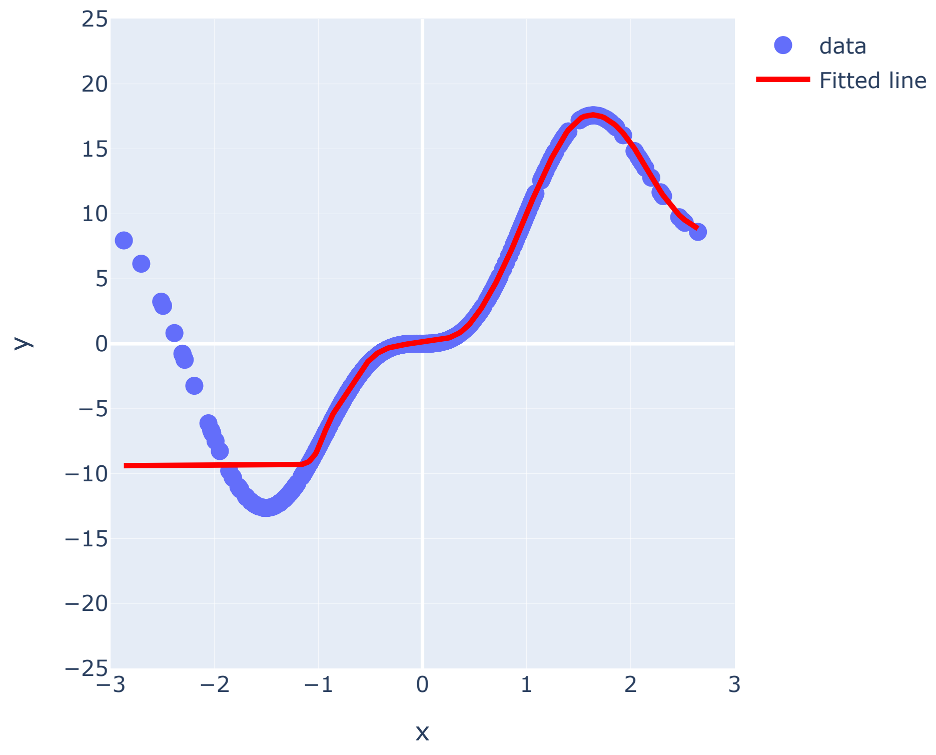
```
trainer(model, criterion, optimizer, dataloader, epochs=10**3, verbose=False)
plot_regression(X, y, model(X_t).detach(), y_range=[-25, 25], dy=5)
```



The neural network is doing a good job, but it’s still struggling to handle data points near the boundaries, but we can do better by having more neurons in our network:

```
model = deepRegression(1, 20, 20, 1)
optimizer = torch.optim.SGD(model.parameters(), lr=0.2)
```

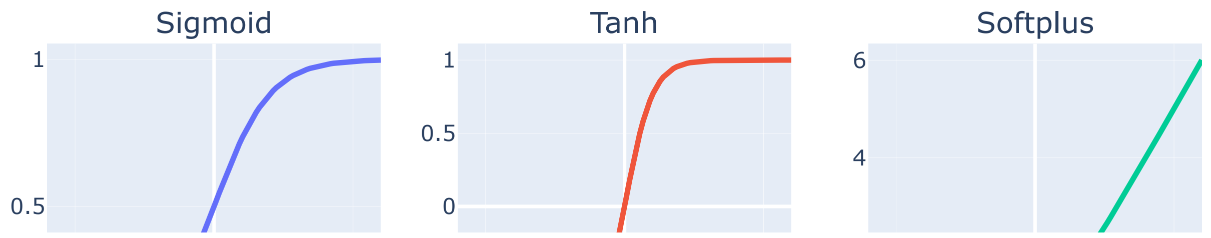
```
trainer(model, criterion, optimizer, dataloader, epochs=10**3, verbose=False)
plot_regression(X, y, model(X_t).detach(), y_range=[-25, 25], dy=5)
```

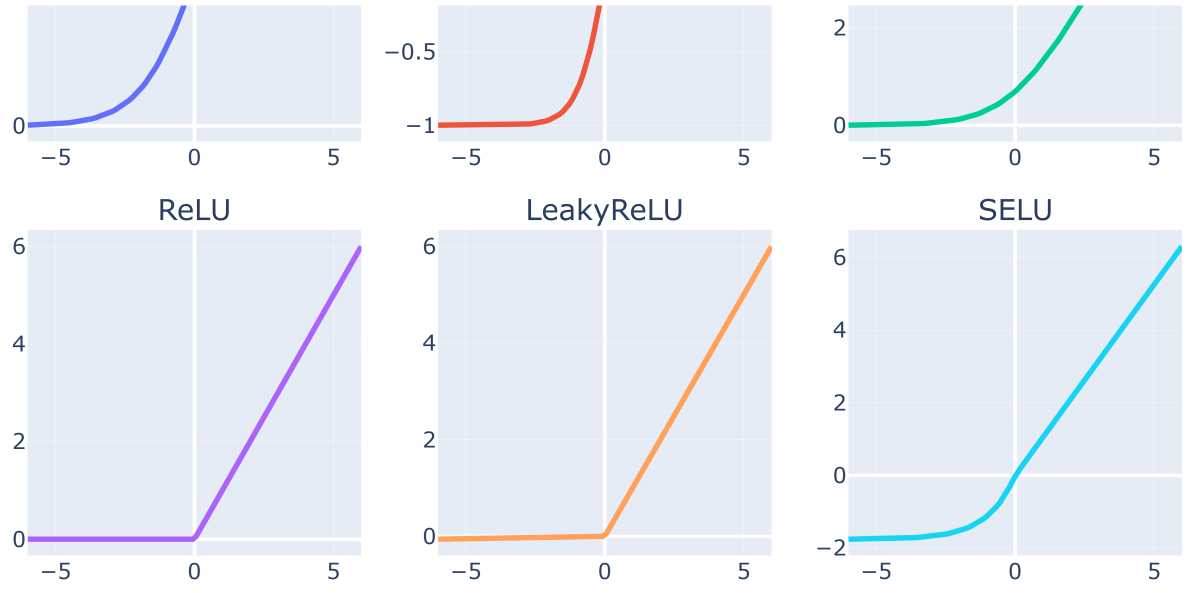


Activation Functions

- Activation functions are what allow us to model complex, non-linear functions
- There are **many** different activations functions:

```
functions = [torch.nn.Sigmoid, torch.nn.Tanh, torch.nn.Softplus, torch.nn.ReLU, torch.nr
plot_activations(torch.linspace(-6, 6, 100), functions)
```





- Activation functions should be non-linear and tend to be monotonic and continuously differentiable (smooth)
- But as you can see with the ReLU function above, that's not always the case
- I wanted to point this out because it highlights how much of an art deep learning really is.
- Here's a great quote from Yoshua Bengio (famous for his work in AI and deep learning) on his group experimenting with ReLU:

> *"…one of the biggest mistakes I made was to think, like everyone else in the 90s, that you needed smooth non-linearities in order for backpropagation to work. I thought that if we had something like rectifying non-linearities, where you have a flat part, it would be really hard to train, because the derivative would be zero in so many places. And when we started experimenting with ReLU, with deep nets around 2010, I was obsessed with the idea that, we should be careful about whether neurons won't saturate too much on the zero part.* ***But in the end, it turned out that, actually, the ReLU was working a lot better than the sigmoids and tanh, and that was a big surprise****…it turned out to work better, whereas I thought it would be harder to train!"*

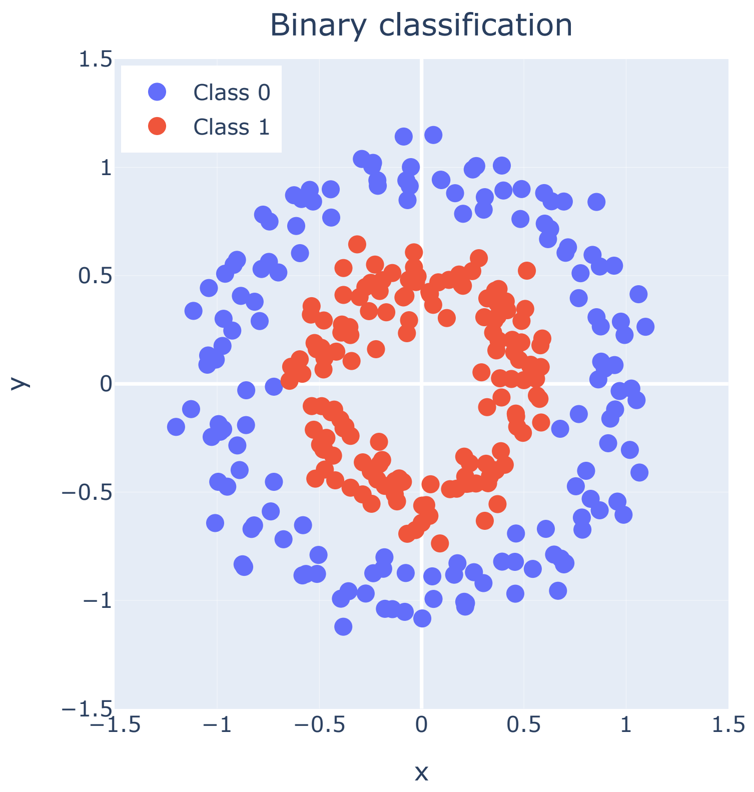
- Anyway, TL;DR **ReLU is probably the most popular these days for training deep neural nets**, but you can treat activation functions as hyper-parameters to be optimized

Neural Network Classification

Binary Classification

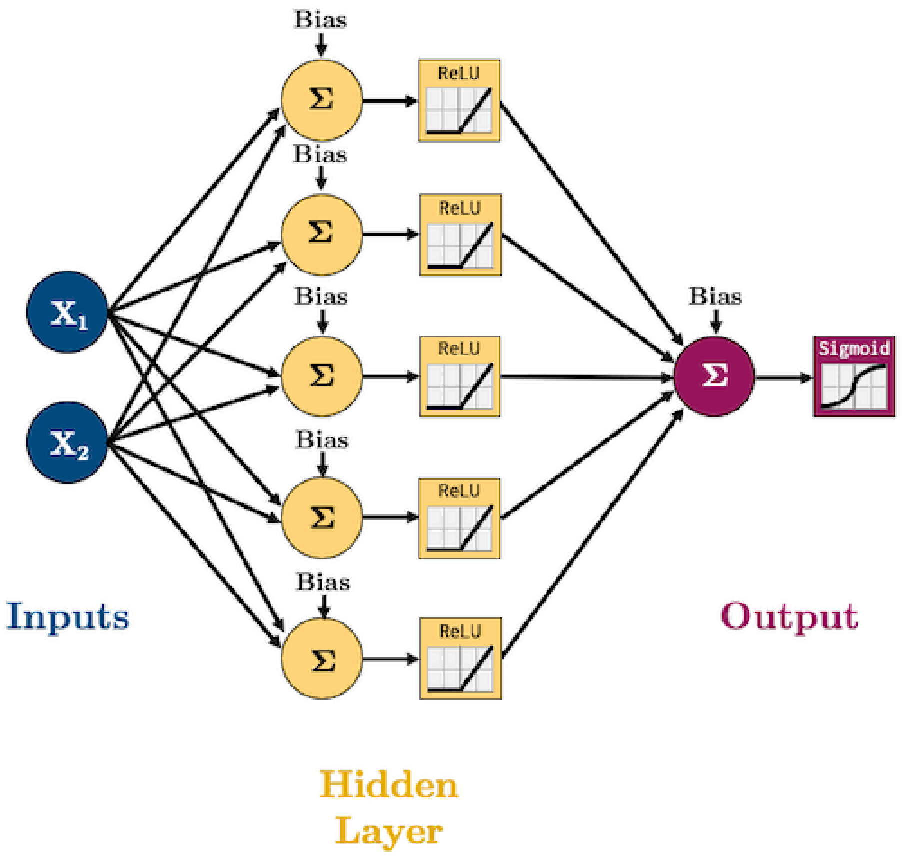
- This will actually be the easiest part of the lecture
- Up until now, we've been looking at developing networks for regression tasks, but what if we want to do binary classification?
- Well, what did we do in Logistic Regression? We just passed the output of a regression into the Sigmoid Function to get a value between 0 and 1 (a probability of an observation belonging to the positive class). Let's do the same thing here
- Let's create a toy dataset first:

```
X, y = make_circles(n_samples=300, factor=0.5, noise=0.1, random_state=2020)
X_t = torch.tensor(X, dtype=torch.float32)
y_t = torch.tensor(y, dtype=torch.float32)
plot_classification_2d(X, y)
```



```python
LEARNING_RATE = 0.1
BATCH_SIZE = 50

# Create dataloader
dataset = TensorDataset(X_t, y_t)
dataloader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True)
```

- Let's create this network to model that dataset:



- I'm going to start using `ReLU` as our activation function(s) and `Adam` as our optimizer because these are what are currently, commonly used in practice.
- We are doing classification now so we'll need to use log loss (binary cross entropy) as our loss function:

$$\mathcal{L}(w) = \sum_{x,y \in D} -y\log(\hat{y}) - (1-y)\log(1-\hat{y})$$

- In PyTorch, binary cross entropy loss criterion is `torch.nn.BCELoss`
- The formula expects a "probability" which is why we add a Sigmoid function to the end of out network.

```
class binaryClassifier(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.main = nn.Sequential(
            nn.Linear(input_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, output_size),
            nn.Sigmoid()  # <-- this will squash our output to a probability between 0 a
        )

    def forward(self, x):
        out = self.main(x)
        return out
```

- **BUT WAIT!**
- While we can do the above and then train with a `torch.nn.BCELoss` loss function, there's a better way
- We can omit the Sigmoid function and just use `torch.nn.BCEWithLogitsLoss` (which combines a Sigmoid layer and the BCELoss)
- Why would we do this? It's numerically more stable (Did you do the log-sum-exp question in Lab 1? We use it here for stability)
- From the docs:

> *"This version is more numerically stable than using a plain Sigmoid followed by a BCELoss as, by combining the operations into one layer, we take advantage of the log-sum-exp trick for numerical stability."*

- So actually, here's our model (no Sigmoid layer at the end because it's included in the loss function we'll use):

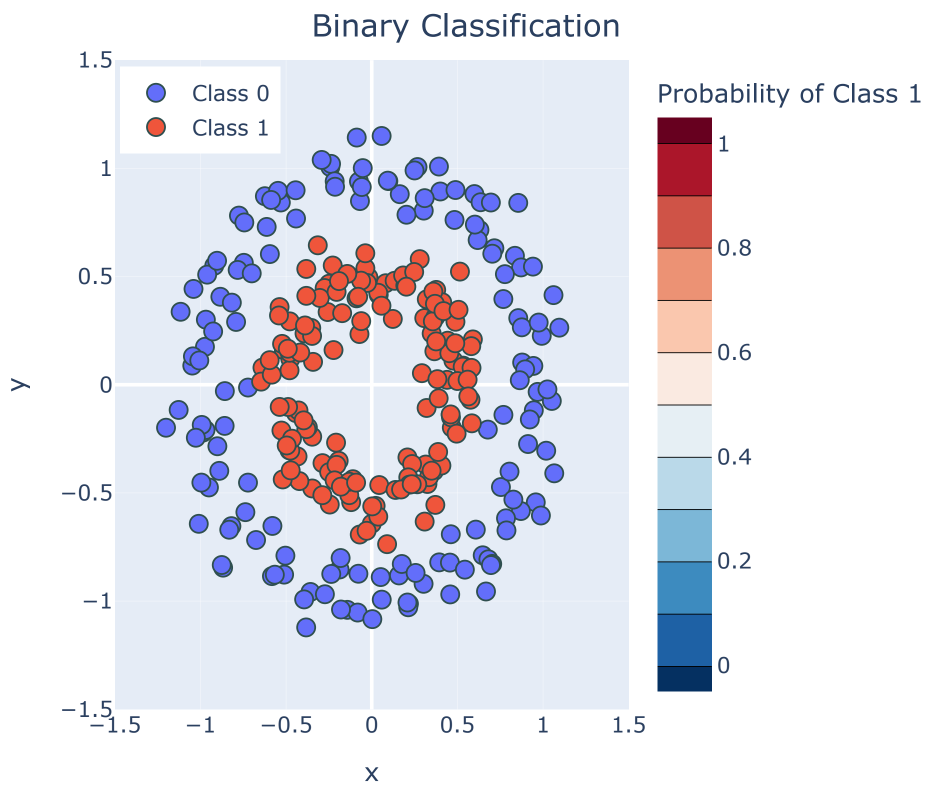
```
class binaryClassifier(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.main = nn.Sequential(
            nn.Linear(input_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, output_size)
        )

    def forward(self, x):
        out = self.main(x)
        return out
```

- Let's train the model:

```
model = binaryClassifier(2, 5, 1)
criterion = torch.nn.BCEWithLogitsLoss() # loss function
optimizer = torch.optim.Adam(model.parameters(), lr=LEARNING_RATE)  # optimization algor
trainer(model, criterion, optimizer, dataloader, epochs=20, verbose=True)
```

```
epoch: 1, loss: 0.6722
epoch: 2, loss: 0.6086
epoch: 3, loss: 0.5239
epoch: 4, loss: 0.4248
epoch: 5, loss: 0.3224
epoch: 6, loss: 0.2478
epoch: 7, loss: 0.1943
epoch: 8, loss: 0.1509
epoch: 9, loss: 0.1224
epoch: 10, loss: 0.1101
epoch: 11, loss: 0.0901
epoch: 12, loss: 0.0813
epoch: 13, loss: 0.0711
epoch: 14, loss: 0.0645
epoch: 15, loss: 0.0580
epoch: 16, loss: 0.0534
epoch: 17, loss: 0.0490
epoch: 18, loss: 0.0449
epoch: 19, loss: 0.0423
epoch: 20, loss: 0.0411
```

```
plot_classification_2d(X, y, model, transform="Sigmoid", title='Binary Classification')
```



- To be clear, our model is just outputting **logits**, which are some number between -∞ and +∞ (we aren't applying Sigmoid), so:
    - To get the probabilities we would need to pass them through a Sigmoid;
    - To get classes, we can apply some threshold (usually 0.5)
- For example, we would expect the point `(0, 0)` to have a high probability and the point `(-1,-1)` to have a low probability:

```
prediction = model(torch.tensor([[0, 0], [-1, -1]], dtype=torch.float32)).detach()
print(prediction)
```

```
tensor([[  5.0078],
        [-17.7170]])
```

```
probability = nn.Sigmoid()(prediction)
print(probability)
```

```
tensor([[9.9336e-01],
        [2.0213e-08]])
```

```
classes = np.where(probability > 0.5, 1, 0)
print(classes)
```

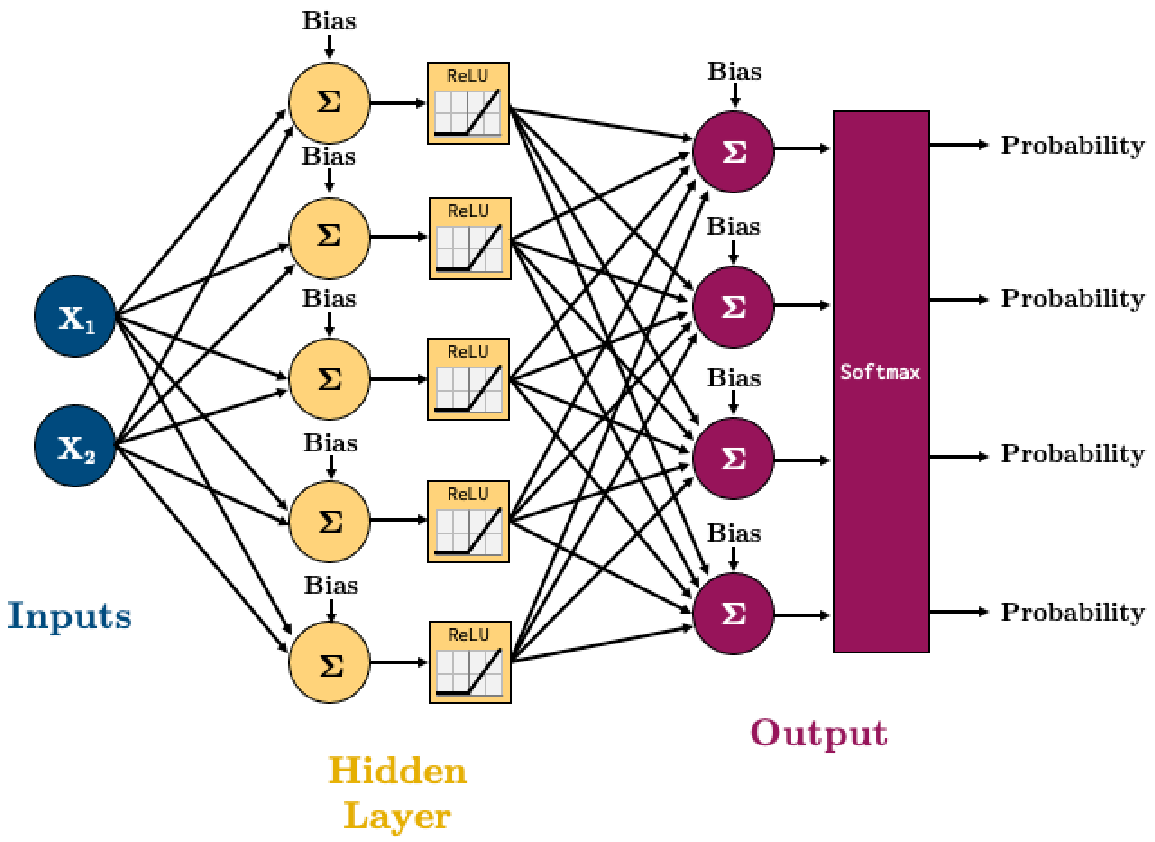
```
[[1]
 [0]]
```

Multiclass Classification

- For multiclass classification, remember softmax?

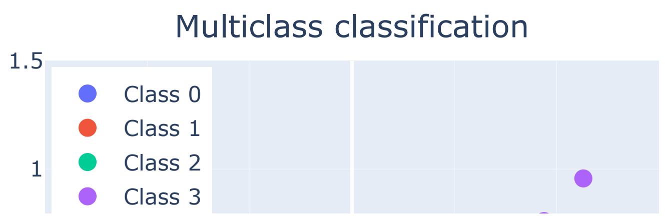
$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

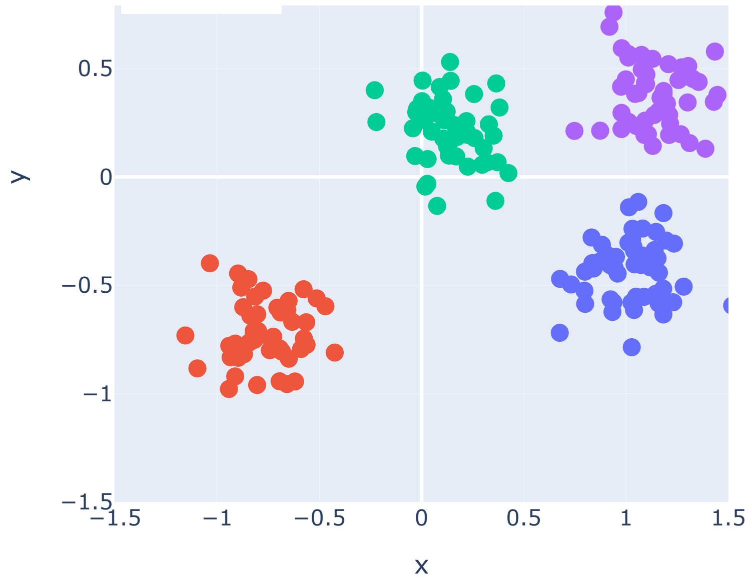
- It basically makes sure all the outputs are probabilities between 0 and 1, and that they all sum to 1.
- `torch.nn.CrossEntropyLoss` is a loss that combines a Softmax with cross entropy loss.
- Let's try a 4-class classification problem using the following network:



```python
class multiClassifier(torch.nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.main = torch.nn.Sequential(
            torch.nn.Linear(input_size, hidden_size),
            torch.nn.ReLU(),
            torch.nn.Linear(hidden_size, output_size)
        )

    def forward(self, x):
        out = self.main(x)
        return out
```

```python
X, y = make_blobs(n_samples=200, centers=4, center_box=(-1.2, 1.2), cluster_std=[0.15, 0
X_t = torch.tensor(X, dtype=torch.float32)
y_t = torch.tensor(y, dtype=torch.int64)

# Create dataloader
dataset = TensorDataset(X_t, y_t)
dataloader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True)

plot_classification_2d(X, y, title="Multiclass classification")
```



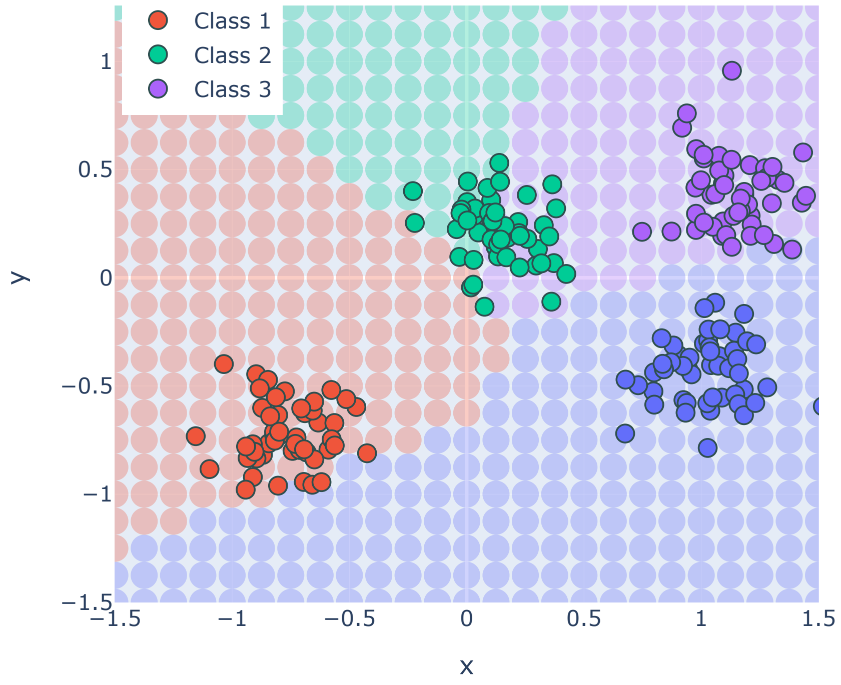


- Let’s train this model:

```
model = multiClassifier(2, 5, 4)
criterion = torch.nn.CrossEntropyLoss() # loss function
optimizer = torch.optim.Adam(model.parameters(), lr=0.2)  # optimization algorithm

for epoch in range(10):
    losses = 0
    for X_batch, y_batch in dataloader:
        optimizer.zero_grad()       # Clear gradients w.r.t. parameters
        y_hat = model(X_batch)            # Forward pass to get output
        loss = criterion(y_hat, y_batch)  # Calculate loss
        loss.backward()             # Getting gradients w.r.t. parameters
        optimizer.step()            # Update parameters
        losses += loss.item()       # Add loss for this batch to running total
    print(f"epoch: {epoch + 1}, loss: {losses / len(dataloader):.4f}")
```

```
epoch: 1, loss: 1.2422
epoch: 2, loss: 0.8272
epoch: 3, loss: 0.4796
epoch: 4, loss: 0.2644
epoch: 5, loss: 0.1331
epoch: 6, loss: 0.0752
epoch: 7, loss: 0.0510
epoch: 8, loss: 0.0348
epoch: 9, loss: 0.0245
epoch: 10, loss: 0.0235
```

```
plot_classification_2d(X, y, model, transform="Softmax", title='Multiclass Classificatic
```



- To be clear once again, our model is just outputting logits, which are some numbers between -∞ and +∞, so:
  - To get the probabilities we would need to pass them to a Softmax;
  - To get classes, we need to select the largest probability.
- For example, we would expect the point (-1, -1) to have a high probability of belonging to class 1, and the point (0,0) to have the highest probability of belonging to class 2.

```
prediction = model(torch.tensor([[-1, -1], [0, 0]], dtype=torch.float32)).detach()
print(prediction)
```

```
tensor([[-1.0275,  4.6009, -0.8710, -4.8428],
        [-9.9652, -0.1649,  4.1314, -3.0913]])
```

- Note how we get 4 predictions per data point (a prediction for each of the 4 classes)

```
probability = nn.Softmax(dim=1)(prediction)
print(probability)
```

```
tensor([[3.5664e-03, 9.9218e-01, 4.1703e-03, 7.8570e-05],
        [7.4430e-07, 1.3426e-02, 9.8585e-01, 7.1949e-04]])
```

- The predictions should now sum to 1:

```
probability.sum(dim=1, keepdim=True)
```

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

- We can get the class with maximum probability using `argmax()`:

```
classes = probability.argmax(dim=1)
print(classes)
```

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

Lecture Highlights

1. PyTorch is a neural network package that implements tensors with computation history
2. Neural Networks are simply:
    - Composed of an input layer, 1 or more hidden layers, and an output layer, each with 1 or more nodes.
    - The number of nodes in the Input/Output layers is defined by the problem/data. Hidden layers can have an arbitrary number of nodes.
    - Activation functions in the hidden layers help us model non-linear data.
    - Feed-forward neural networks are just a combination of simple linear and non-linear operations.
3. Activation functions allow the network to learn non-linear function