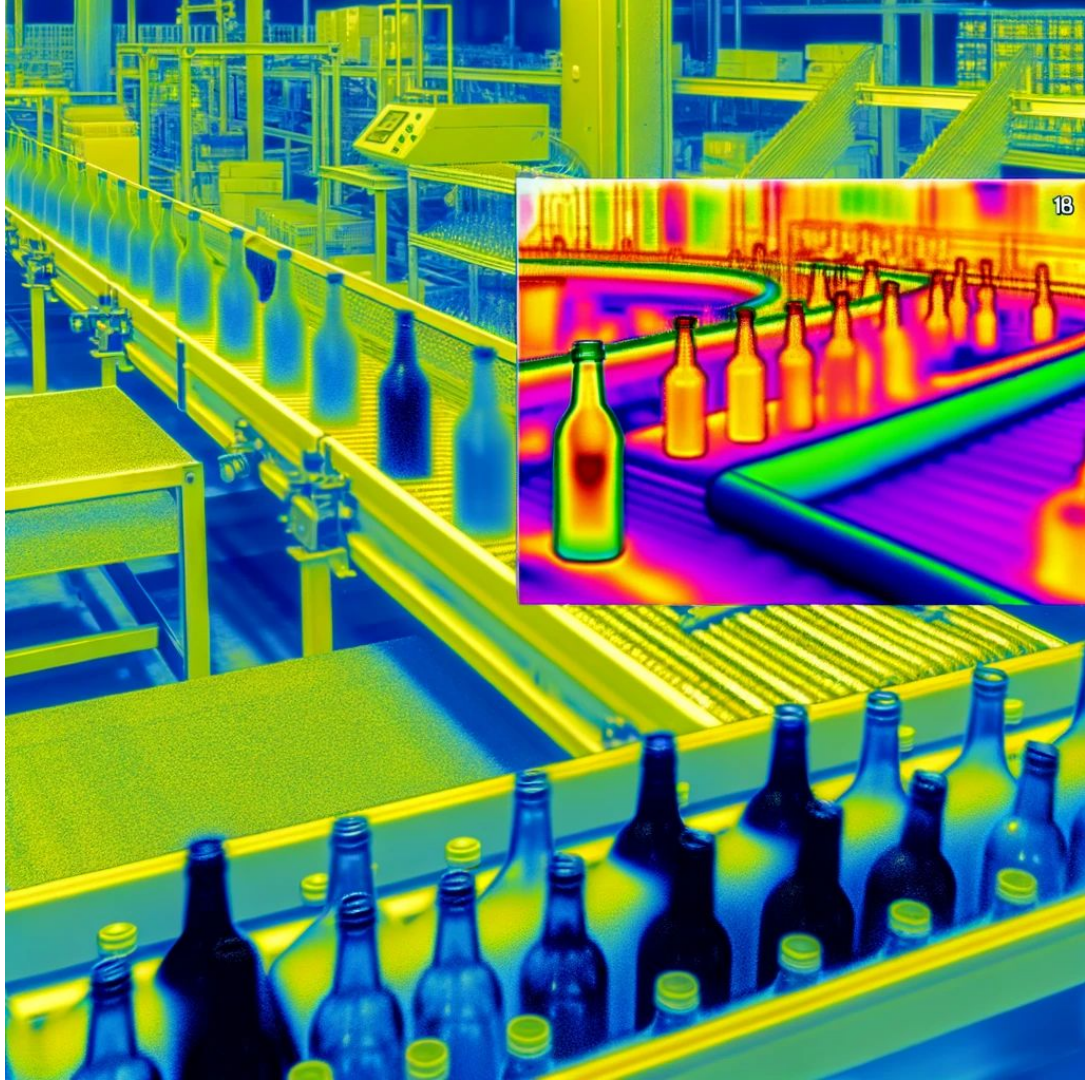
PPGEEC2318

Machine Learning

Rethinking the training loop: a simple classification problem

Ivanovitch Silva

ivanovitch.silva@ufrn.br



Run

Data Generation

Data Preparation

Model Configuration

v0	v0	v0	v0	1
v0	v0	v1	v1	2
v0	v1	v1	v2	3
v0	v1	v1	v3	4
v0	v2	v2	v4	5

Data Generation

- v0: Numpy manual split

Data Preparation

- v0: x_train y_train as a tensor (GPU)
- v1: TensorDataset DataLoader
- v2: TensorDataset DataLoader Random Split

Model Configuration

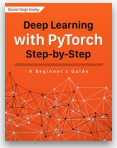
- v0: sequential optimizer loss
- v1: sequential optimizer loss make_train_step_fn
- v2: sequential optimizer loss make_train_step_fn make_val_step_fn

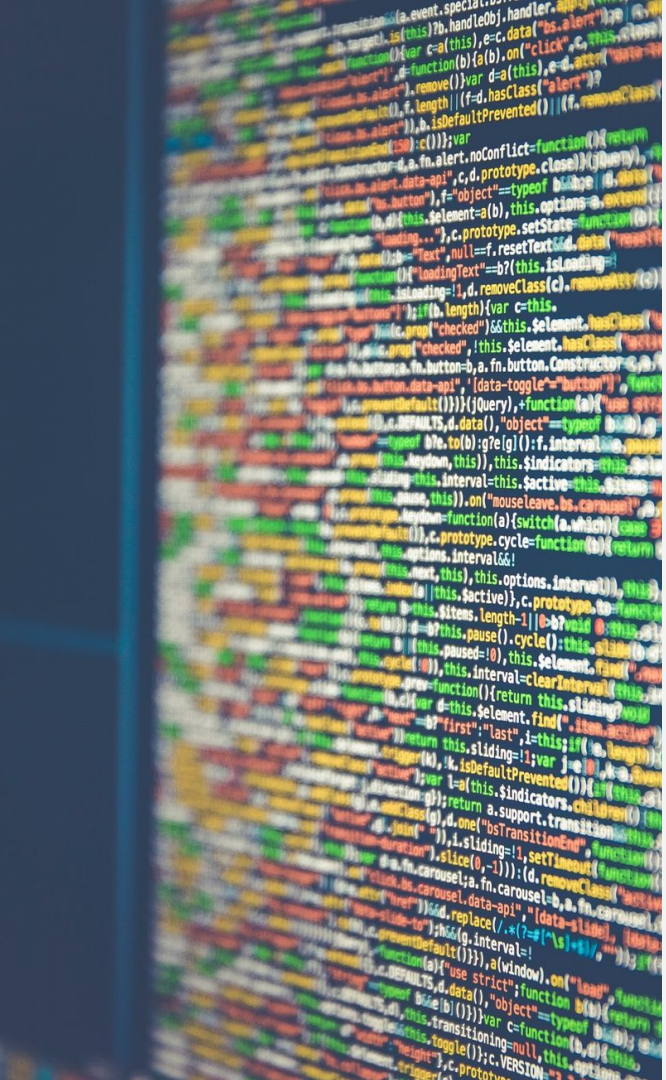
Model Training

- v0: model.train() yhat loss backward optimizer update
- v1: high-order function build a training step function fn(x_train_tensor, y_train_tensor)
- v2: build a trainning step function mini-batch dataloader
- v3: build a trainning step function mini-batch function dataloader
- v4: build a trainning step function mini-batch function dataloader validation function

Saving and Loading Models

Deploy and Make Predictions



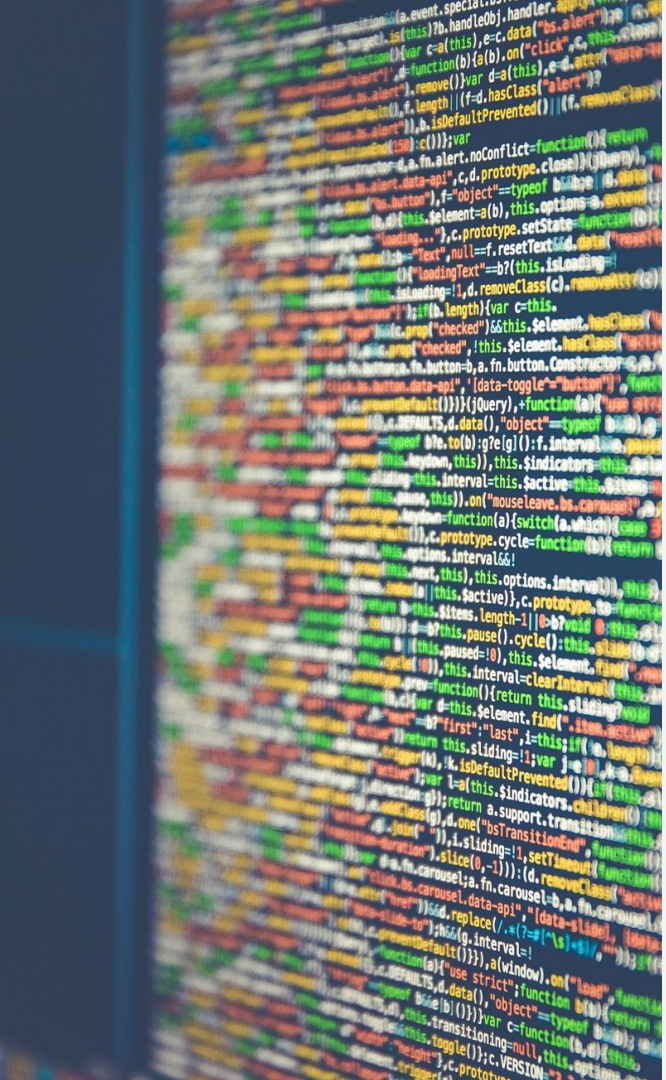


High-Order Functions

Higher-order functions in Python are functions that either **take other functions** as arguments or **return functions** as their results. This concept is key in functional programming and enhances flexibility and expressivity in your code.

Functions Accepting Functions as Arguments

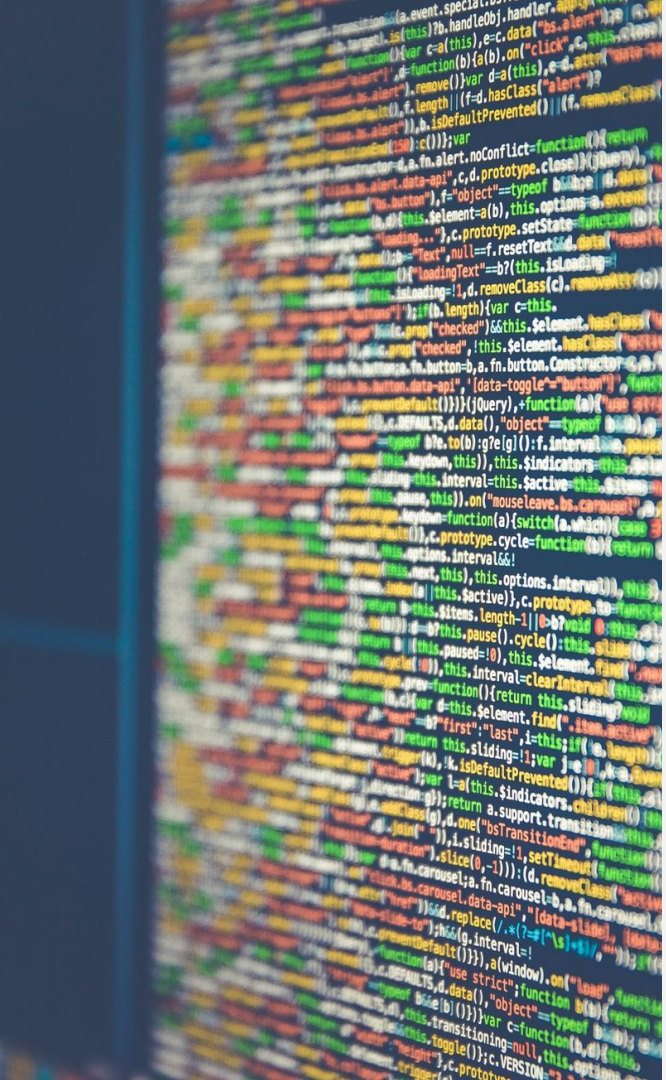
```
def square(x):
    return x * x

numbers = [1, 2, 3, 4, 5]
squares = map(square, numbers)
print(list(squares))  # Output: [1, 4, 9, 16, 25]
```



High-Order Functions

Functions Returning Functions

```
def multiplier(n):
    def inner(x):
        return x * n
    return inner

double = multiplier(2)
triple = multiplier(3)

print(double(5))  # Output: 10
print(triple(5))  # Output: 15
```



Why Use High-Order Functions?

- **Code Reusability:** Using higher-order functions, you can write more generic and reusable code. For instance, you can write a general-purpose filter function that can be applied with different criterion functions.
- **Functional Programming:** Higher-order functions are a key component of functional programming, a paradigm emphasizing immutability and the use of functions to transform data.
- **Readability and Expressiveness:** They allow developers to express complex ideas more clearly and concisely.

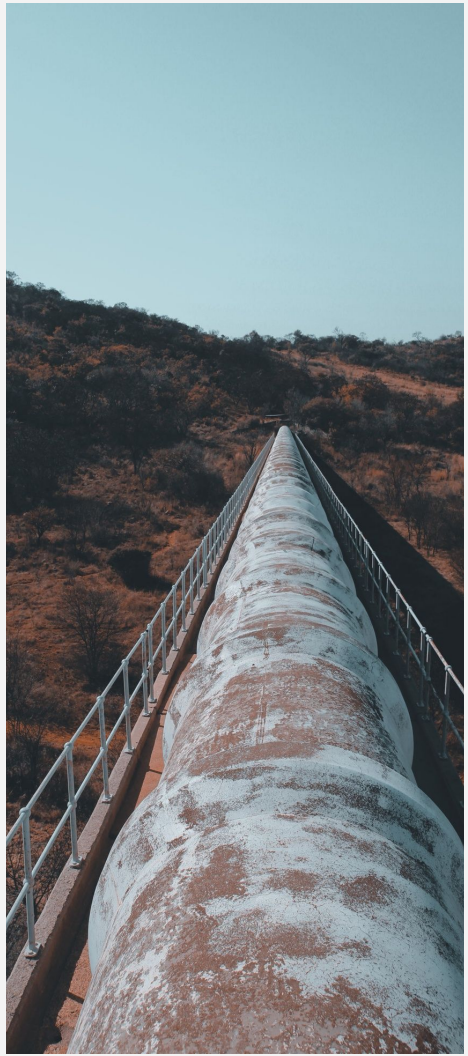

Model lives
on GPU

Dataset

```
from torch.utils.data import Dataset

class CustomDataset(Dataset):
    def __init__(self, x_tensor, y_tensor):
        self.x = x_tensor
        self.y = y_tensor

    def __getitem__(self, index):
        return (self.x[index], self.y[index])

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

# Wait, is this a CPU tensor now? Why? Where is .to(device)?
x_train_tensor = torch.from_numpy(x_train).float()
y_train_tensor = torch.from_numpy(y_train).float()

train_data = CustomDataset(x_train_tensor, y_train_tensor)
print(train_data[0])
```

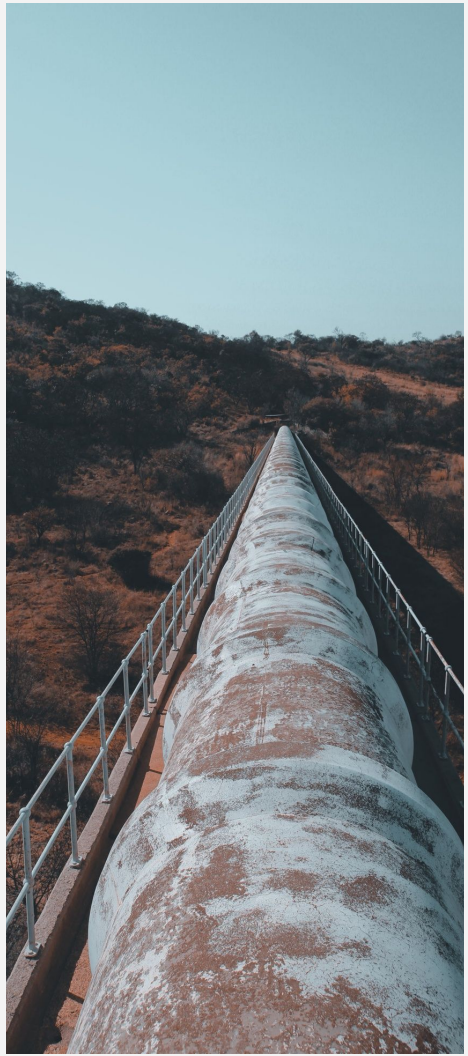
TensorDataset & DataLoader

```
# Our data was in Numpy arrays, but we need to transform them into
# PyTorch's Tensors
x_train_tensor = torch.from_numpy(x_train).float()
y_train_tensor = torch.from_numpy(y_train).float()

# Builds Dataset
train_data = TensorDataset(x_train_tensor, y_train_tensor)

# Builds DataLoader
train_loader = DataLoader(dataset=train_data,
                          batch_size=16,
                          shuffle=True)
```



TensorDataset & DataLoader

```
# Builds tensors from numpy arrays BEFORE split
x_tensor = torch.from_numpy(x).float()
y_tensor = torch.from_numpy(y).float()

# Builds dataset containing ALL data points
dataset = TensorDataset(x_tensor, y_tensor)

# Performs the split
ratio = .8
n_total = len(dataset)
n_train = int(n_total * ratio)
n_val = n_total - n_train

train_data, val_data = random_split(dataset, [n_train, n_val])

# Builds a loader of each set
train_loader = DataLoader(dataset=train_data,
                          batch_size=16, shuffle=True)
val_loader = DataLoader(dataset=val_data, batch_size=16)
```



Model Configuration

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'

# Sets learning rate - this is "eta" ~ the "n" like Greek letter
lr = 0.1

torch.manual_seed(42)
# Now we can create a model and send it at once to the device
model = nn.Sequential(nn.Linear(1, 1)).to(device)

# Defines a SGD optimizer to update the parameters (now retrieved directly from
the model)
optimizer = optim.SGD(model.parameters(), lr=lr)

# Defines a MSE loss function
loss_fn = nn.MSELoss(reduction='mean')

# Creates the train_step function for our model, loss function and optimizer
train_step_fn = make_train_step_fn(model, loss_fn, optimizer)

# Creates the val_step function for our model and loss function
val_step_fn = make_val_step_fn(model, loss_fn)
```

Model Configuration

```
def make_train_step_fn(model, loss_fn, optimizer):
    # Builds function that performs a step in the train loop
    def perform_train_step_fn(x, y):
        # Sets model to TRAIN mode
        model.train()

        # Step 1 - Computes our model's predicted output - forward pass
        yhat = model(x)
        # Step 2 - Computes the loss
        loss = loss_fn(yhat, y)
        # Step 3 - Computes gradients for both "a" and "b" parameters
        loss.backward()
        # Step 4 - Updates parameters using gradients and the learning rate
        optimizer.step()
        optimizer.zero_grad()

        # Returns the loss
        return loss.item()

    # Returns the function that will be called inside the train loop
    return perform_train_step_fn
```

Model Configuration

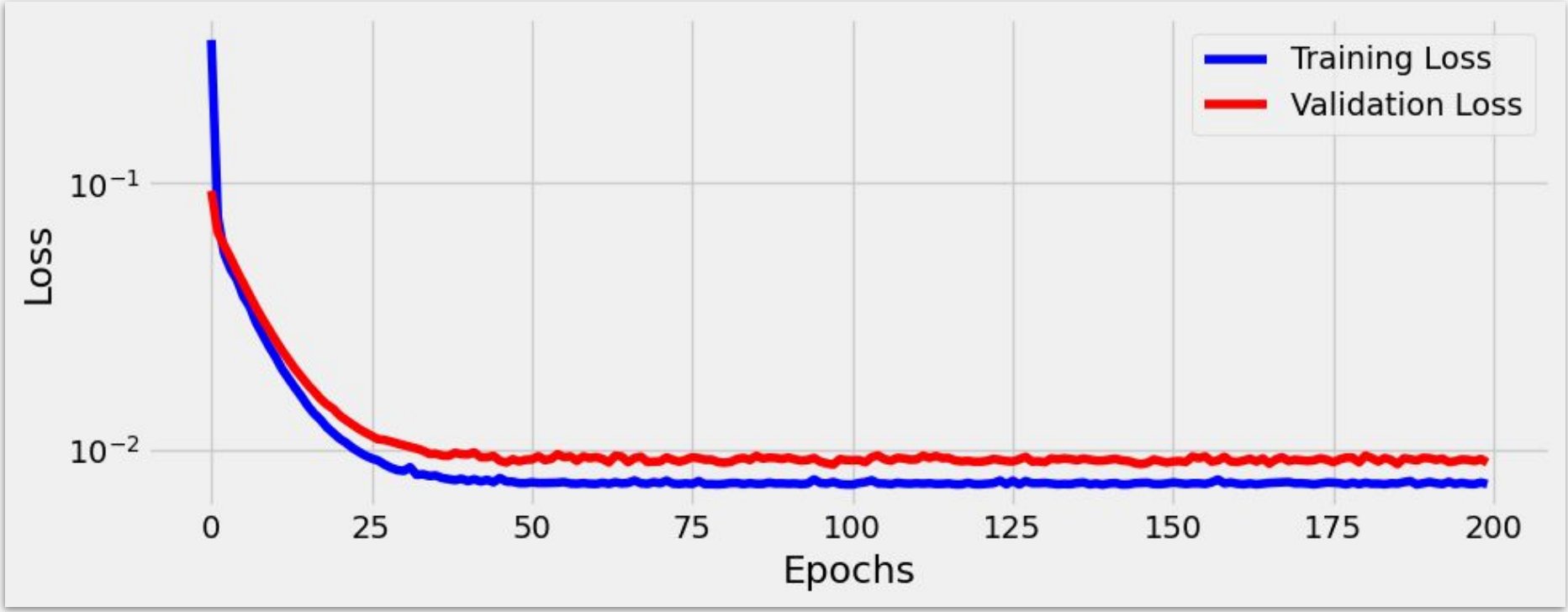
```
def make_val_step_fn(model, loss_fn):
    # Builds function that performs a step in the validation loop
    def perform_val_step_fn(x, y):
        # Sets model to EVAL mode
        model.eval()

        # Step 1 - Computes our model's predicted output - forward pass
        yhat = model(x)
        # Step 2 - Computes the loss
        loss = loss_fn(yhat, y)
        # There is no need to compute Steps 3 and 4, since we don't update
parameters during evaluation
        return loss.item()

    return perform_val_step_fn
```

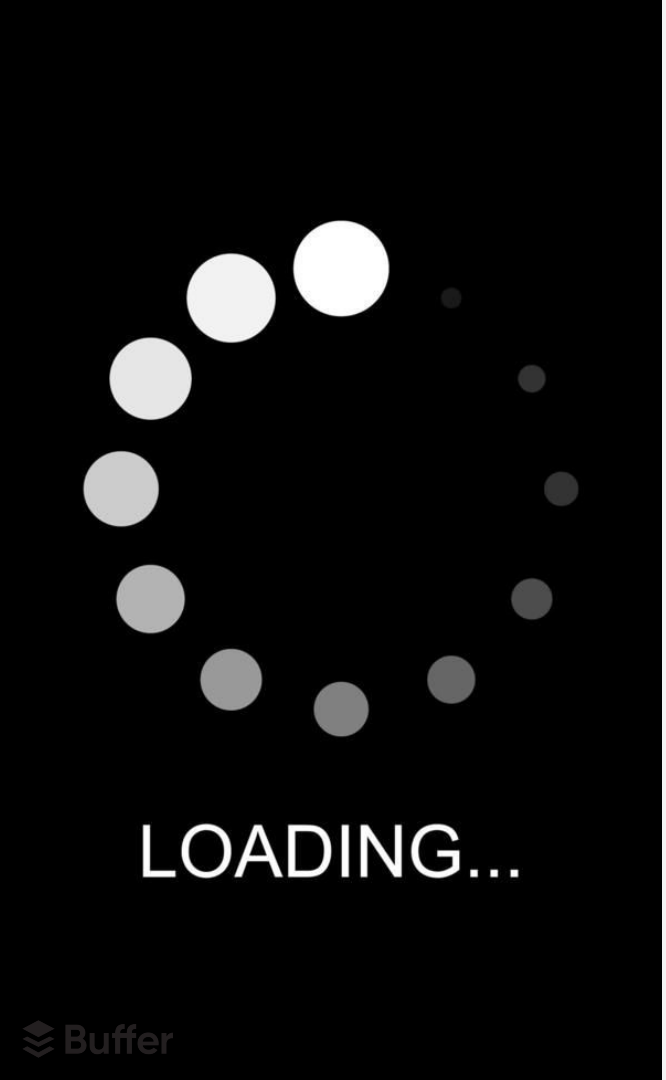
Training

```
# Defines number of epochs
n_epochs = 200

losses = []
val_losses = []

for epoch in range(n_epochs):
   # inner loop
   loss = mini_batch(device, train_loader, train_step_fn)
   losses.append(loss)

   # VALIDATION
   # no gradients in validation!
   with torch.no_grad():
       val_loss = mini_batch(device, val_loader, val_step_fn)
       val_losses.append(val_loss)
```

Training

```
def mini_batch(device, data_loader, step_fn):
    mini_batch_losses = []
    for x_batch, y_batch in data_loader:
        x_batch = x_batch.to(device)
        y_batch = y_batch.to(device)

        mini_batch_loss = step_fn(x_batch, y_batch)
        mini_batch_losses.append(mini_batch_loss)

    loss = np.mean(mini_batch_losses)
    return loss
```


Training Loss
Validation Loss
Loss
10^{-1}
10^{-2}
0
25
50
75
100
125
150
175
200
Epochs



Saving and Loading Models

```
checkpoint = {'epoch': n_epochs,
              'model_state_dict': model.state_dict(),
              'optimizer_state_dict': optimizer.state_dict(),
              'loss': losses,
              'val_loss': val_losses}

torch.save(checkpoint, 'model_checkpoint.pth')
```

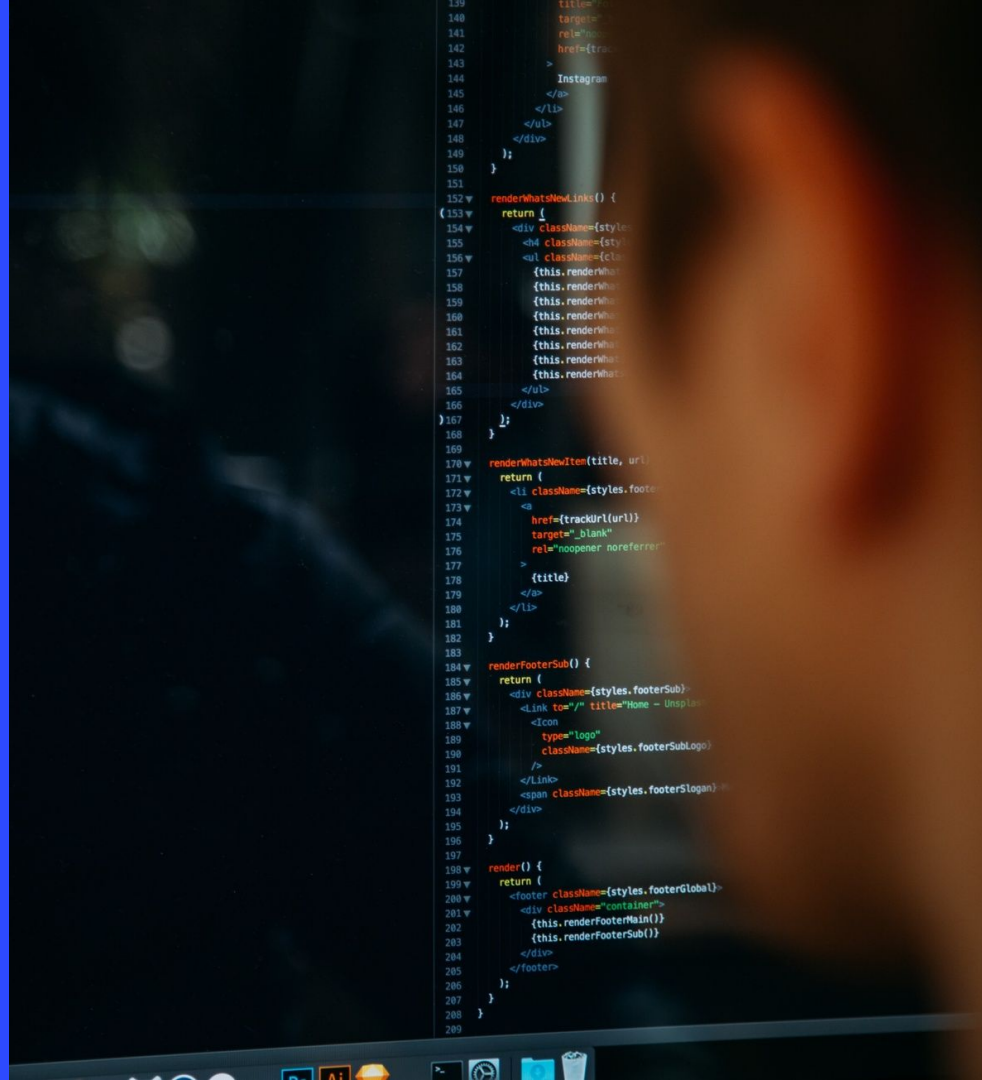
```
checkpoint = torch.load('model_checkpoint.pth')

model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])

saved_epoch = checkpoint['epoch']
saved_losses = checkpoint['loss']
saved_val_losses = checkpoint['val_loss']

model.train() # always use TRAIN for resuming training
```

Make Predictions

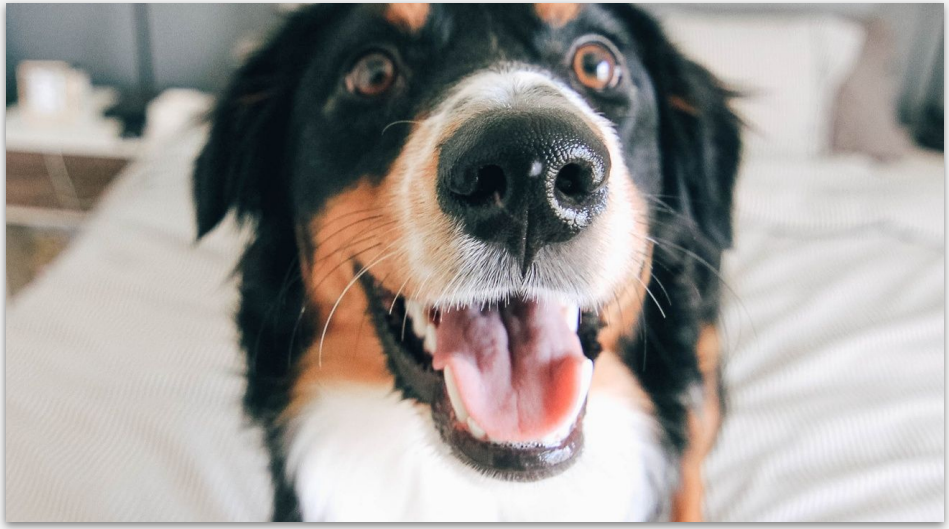
```
checkpoint = torch.load('model_checkpoint.pth')

model.load_state_dict(checkpoint['model_state_dict'])

new_inputs = torch.tensor([[.20], [.34], [.57]])

model.eval() # always use EVAL for fully trained models!
model(new_inputs.to(device))

tensor([[1.4174],
        [1.6896],
        [2.1366]], device='cuda:0')
```

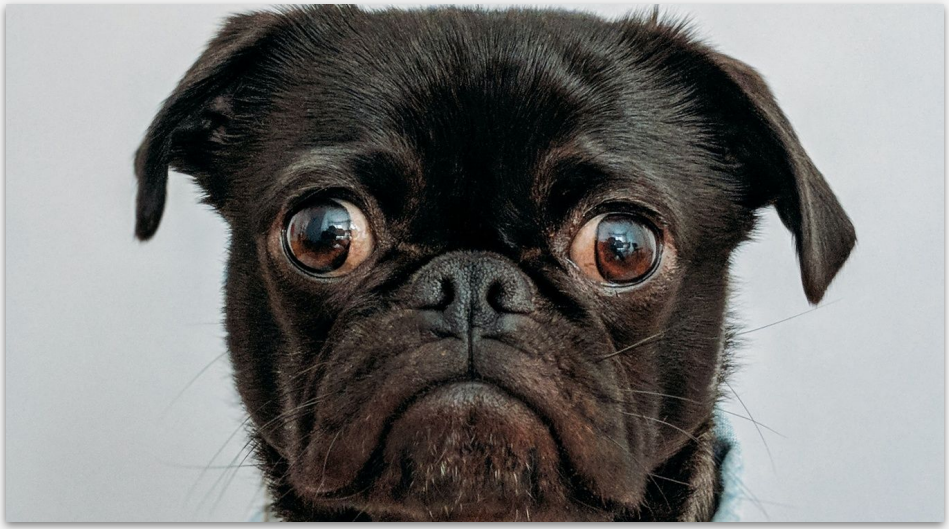
Going Classy

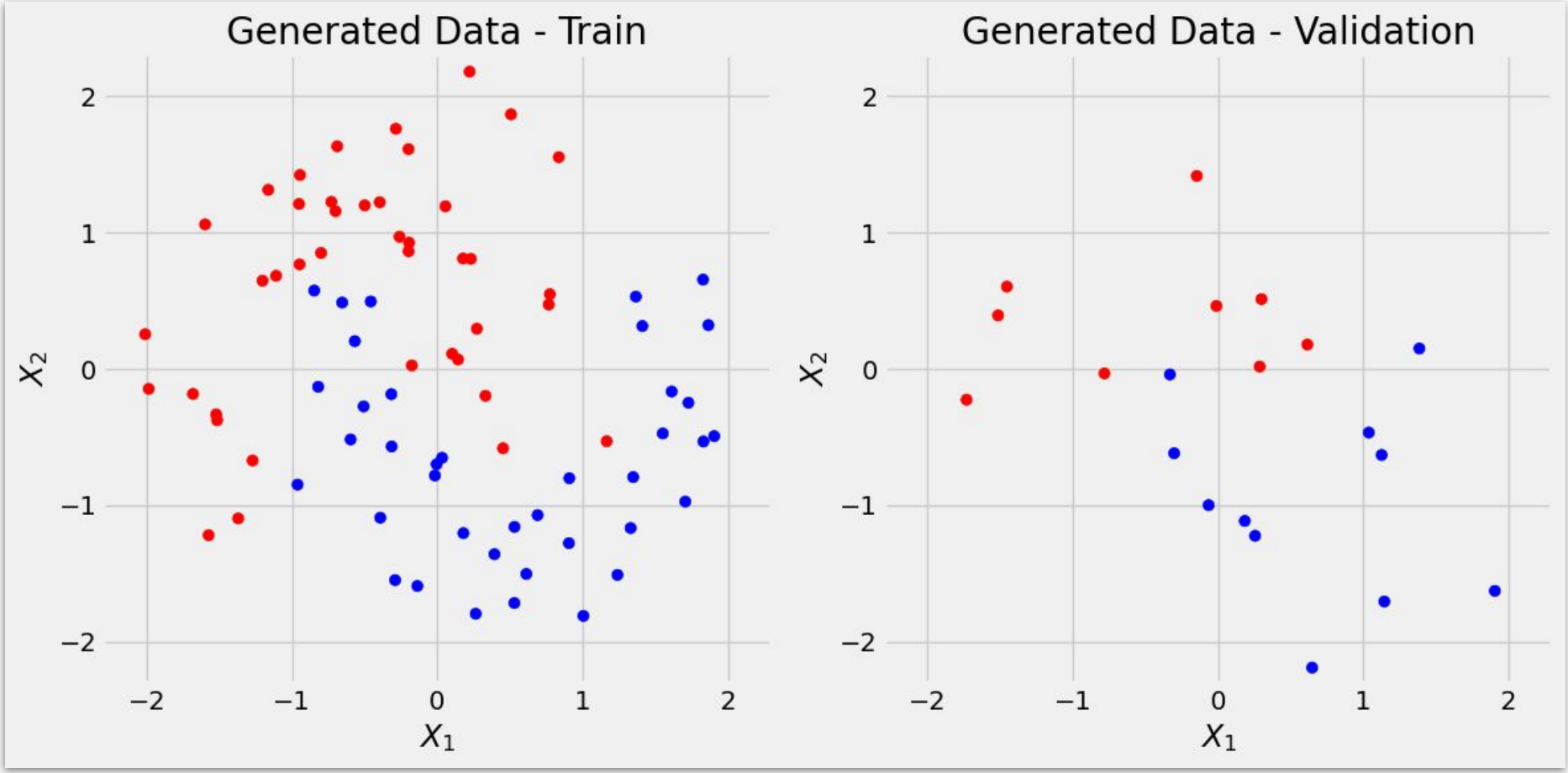
Object-Oriented Programming - OOP

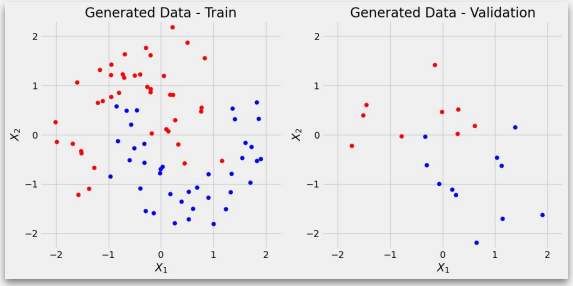


A simple Classification Problem





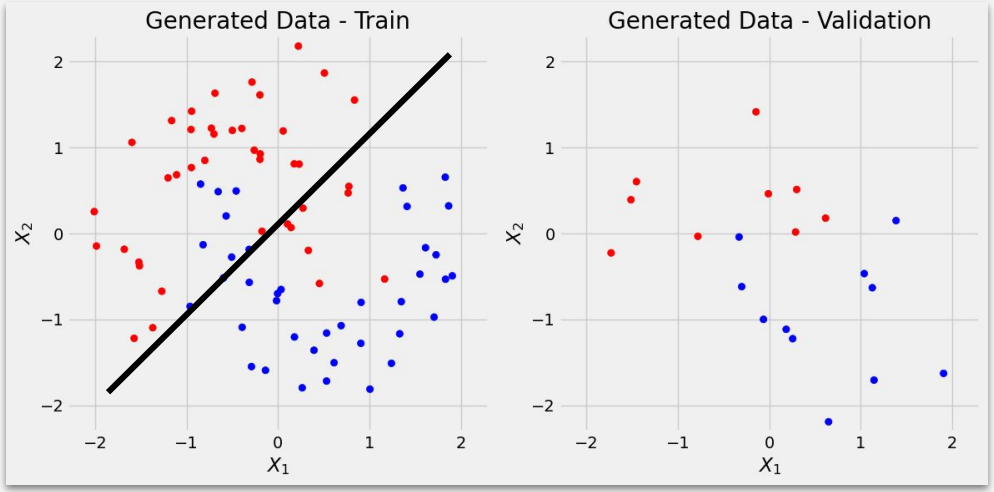

Generated Data - Train
Generated Data - Validation
X_1
X_2
2
1
0
−1
−2

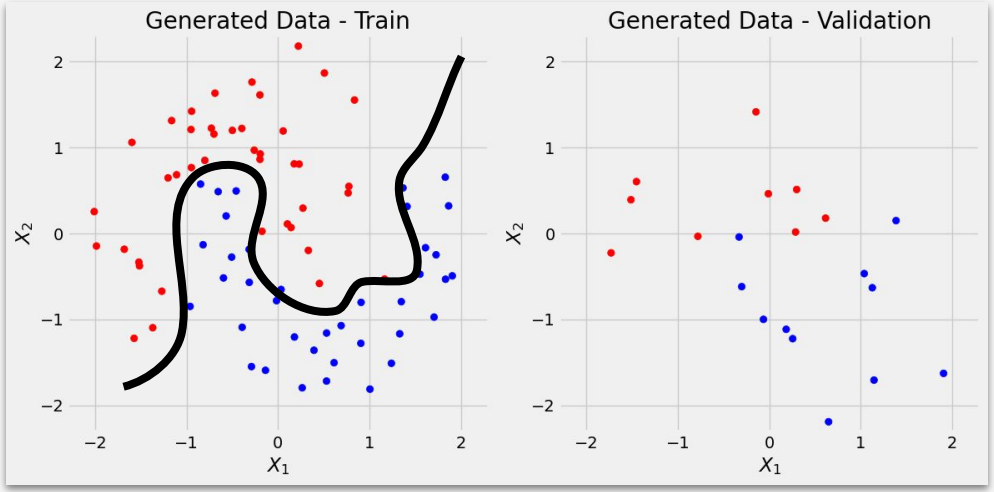


Linear Regression vs Logistic Regression

The **dependent variable** is a metric variable: eg. height, speed, salary, grade, etc

The **dependent variable** is a dichotomous variable: eg. red or blue, happy or sad, 0 or 1, gender, etc

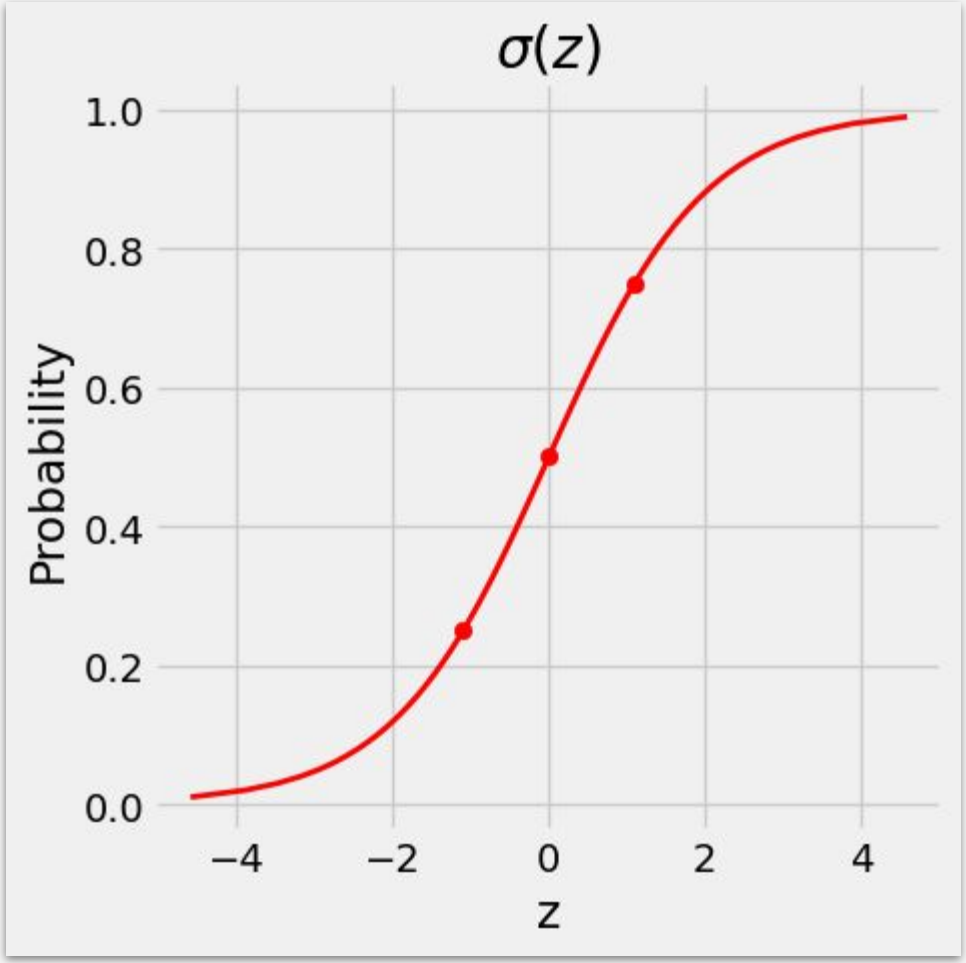




$$y = b + w_1 x_1 + w_2 x_2 + \epsilon$$

$$y = \begin{cases} 1, & \text{if } b + w_1 x_1 + w_2 x_2 \geq 0 \\ 0, & \text{if } b + w_1 x_1 + w_2 x_2 < 0 \end{cases}$$

Mapping a linear regression model to **discrete labels**.



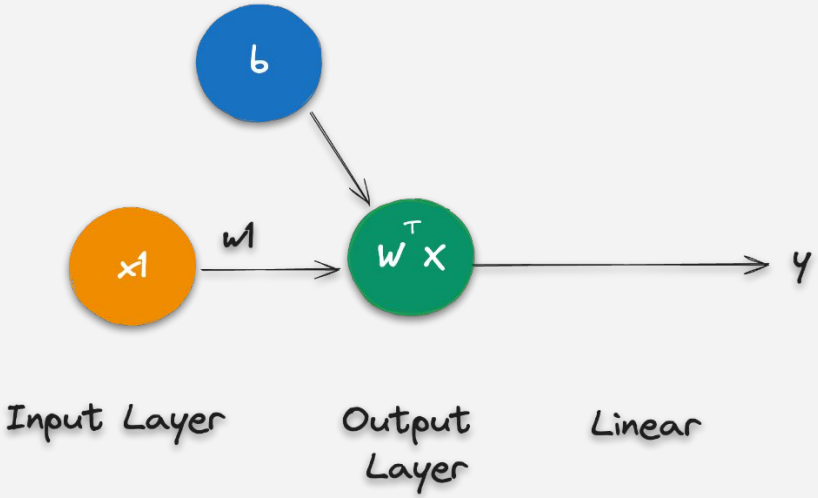
Only apply linear regression is not enough. The goal of the logistic regression is to estimate the **probability of occurrence**.

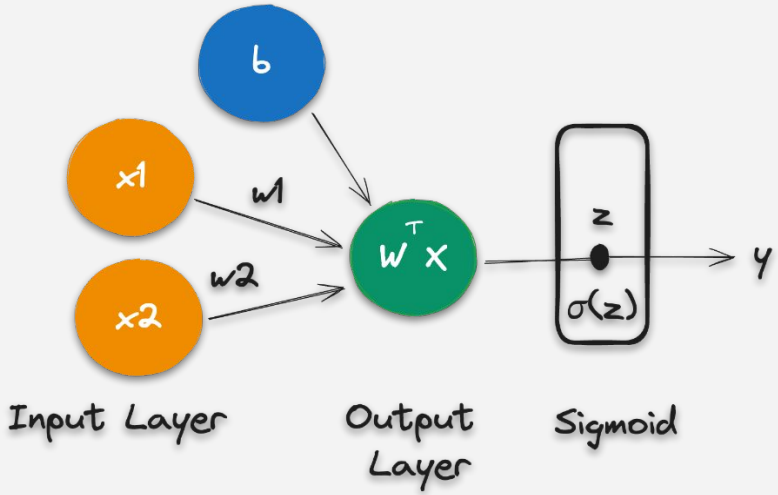
Logits

$$z = b + w_1 x_1 + w_2 x_2$$

$$\mathrm{P}(y = 1) \approx 1.0, \quad \text{if } z \gg 0$$
$$\mathrm{P}(y = 1) = 0.5, \quad \text{if } z = 0$$
$$\mathrm{P}(y = 1) \approx 0.0, \quad \text{if } z \ll 0$$

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



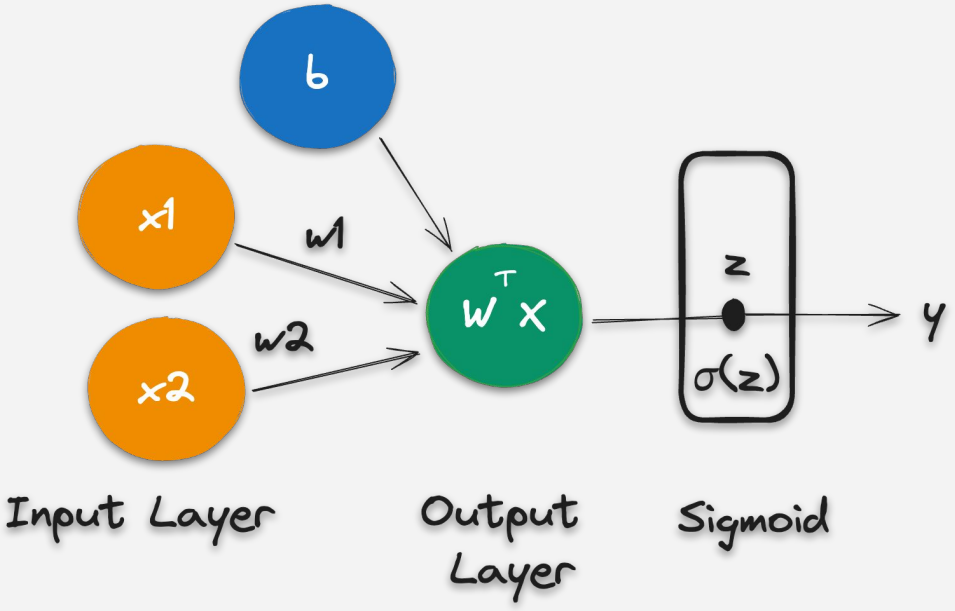


Linear Regression vs Logistic Regression

Logistic Regression

A Note on Notation

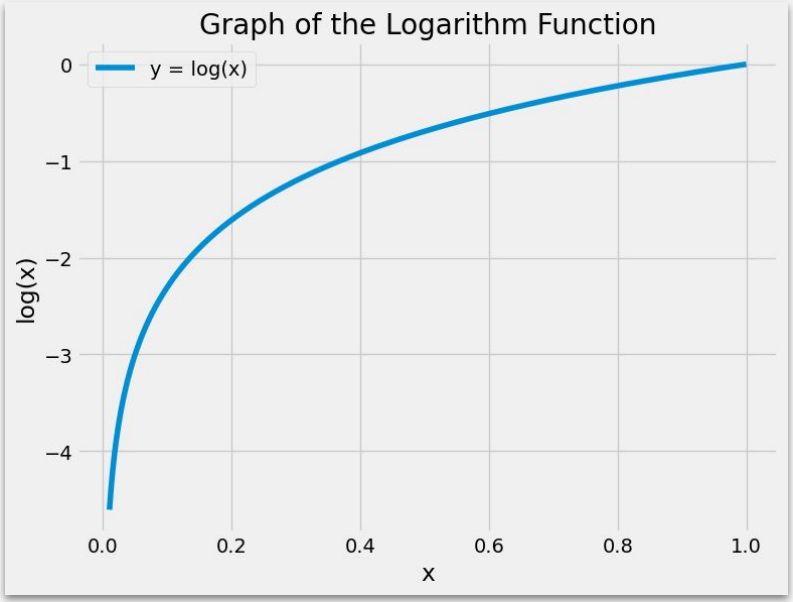
$$W = \underset{(3\times1)}{\begin{bmatrix} b \\ w_1 \\ w_2 \end{bmatrix}}; X = \underset{(3\times1)}{\begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}}$$



$$z = W^T X = \underset{(1\times3)}{\begin{bmatrix} - & w^T & - \end{bmatrix}} \underset{(3\times1)}{\begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}} = \underset{(1\times3)}{\begin{bmatrix} b & w_1 & w_2 \end{bmatrix}} \underset{(3\times1)}{\begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}}$$

$$= b + w_1 x_1 + w_2 x_2$$

```
torch.manual_seed(42)
model = nn.Sequential()
model.add_module('linear', nn.Linear(2, 1))
model.add_module('sigmoid', nn.Sigmoid())
print(model.state_dict())

OrderedDict([('linear.weight', tensor([[0.5406,
0.5869]])), ('linear.bias', tensor([-0.1657]))])
```



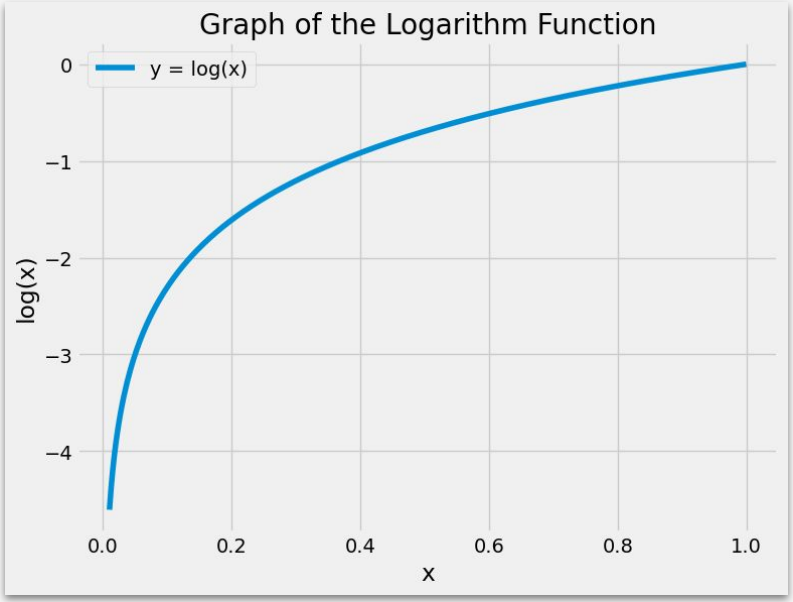
#intuitive-way

We already have a model, and now we need to define an appropriate **loss** for the logistic regression.

$$y_i = 1 \Rightarrow error_i = \log\left(P(y_i = 1)\right)$$

$$P(y_i = 0) = 1 - P(y_i = 1)$$

$$y_i = 0 \Rightarrow error_i = \log\left(1 - P(y_i = 1)\right)$$

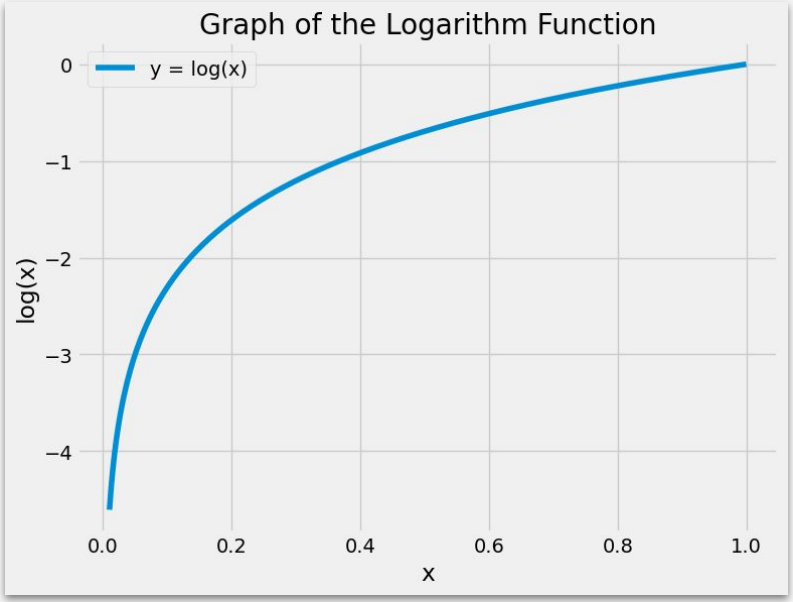


#intuitive-way

We already have a model, and now we need to define an appropriate **loss** for the logistic regression.

Binary Cross-Entropy (BCE)

$$BCE(y) = -\frac{1}{(N_{\text{pos}} + N_{\text{neg}})}\left[\sum_{i=1}^{N_{pos}} \log\left(P(y_i = 1)\right) + \sum_{i=1}^{N_{neg}} \log\left(1 - P(y_i = 1)\right)\right]$$



#intuitive-way

We already have a model, and now we need to define an appropriate **loss** for the logistic regression.

Binary Cross-Entropy (BCE)

$$BCE(y) = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log\left(P(y_i = 1)\right) + (1 - y_i)\log\left(1 - P(y_i = 1)\right)\right]$$

Binary Cross-Entropy (BCE)

$$BCE(y) = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log\left(P(y_i = 1)\right) + (1 - y_i)\log\left(1 - P(y_i = 1)\right)\right]$$

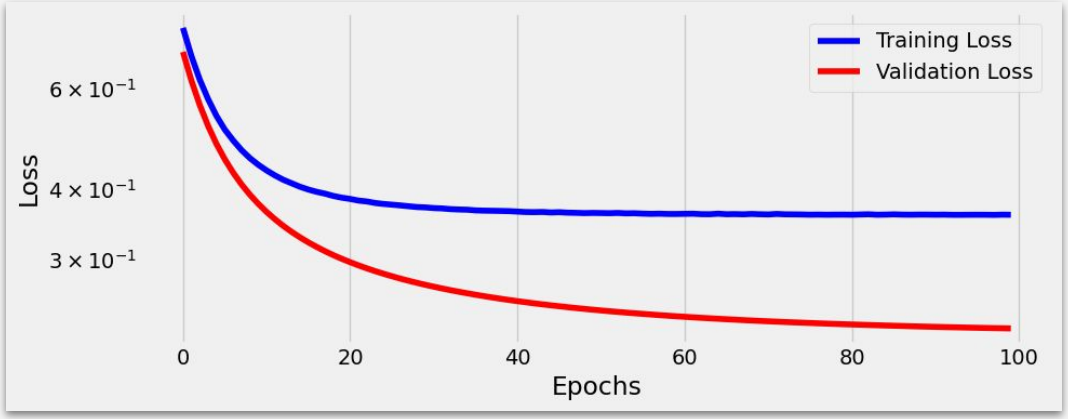
```
loss_fn = nn.BCELoss(reduction='mean')

dummy_labels = torch.tensor([1.0, 0.0])
dummy_predictions = torch.tensor([.9, .2])

# RIGHT
right_loss = loss_fn(dummy_predictions, dummy_labels)

# WRONG
wrong_loss = loss_fn(dummy_labels, dummy_predictions)

print(right_loss, wrong_loss)
tensor(0.1643) tensor(15.0000)
```

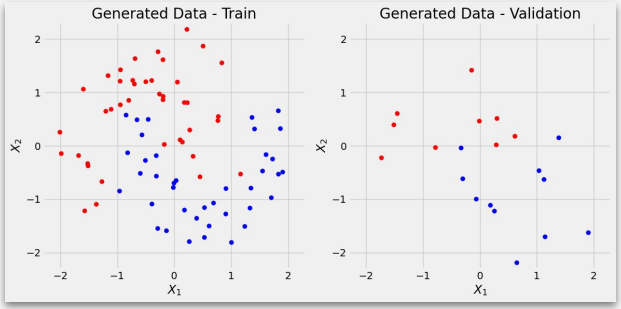
Model Configuration

```
# Sets learning rate - this is "eta" ~ the "n" like Greek letter
lr = 0.1

torch.manual_seed(42)
model = nn.Sequential()
model.add_module('linear', nn.Linear(2, 1))

# Defines a SGD optimizer to update the parameters
optimizer = optim.SGD(model.parameters(), lr=lr)

# Defines a BCE loss function
loss_fn = nn.BCEWithLogitsLoss()
```

Training

```
n_epochs = 100

arch = Architecture(model, loss_fn, optimizer)
arch.set_loaders(train_loader, val_loader)
arch.set_seed(42)
arch.train(n_epochs)
```





Making Predictions

$$y = \begin{cases} 1, & \text{if } z \geq 0 \\ 0, & \text{if } z < 0 \end{cases}$$

$$y = \begin{cases} 1, & \text{if } \mathrm{P}(y = 1) \geq 0.5 \\ 0, & \text{if } \mathrm{P}(y = 1) < 0.5 \end{cases}$$

```
# prediction logits (z)
logits_val = arch.predict(X_val[:4])
logits_val

array([[-0.37522304],
       [ 0.7390274 ],
       [-2.5800889 ],
       [-0.93623203]], dtype=float32)
```

```
# prediction probabilities
prob_val = torch.sigmoid(torch.as_tensor(logits_val[:4]).float())
prob_val

tensor([[0.4073],
        [0.6768],
        [0.0704],
        [0.2817]])
```

Decision Boundary

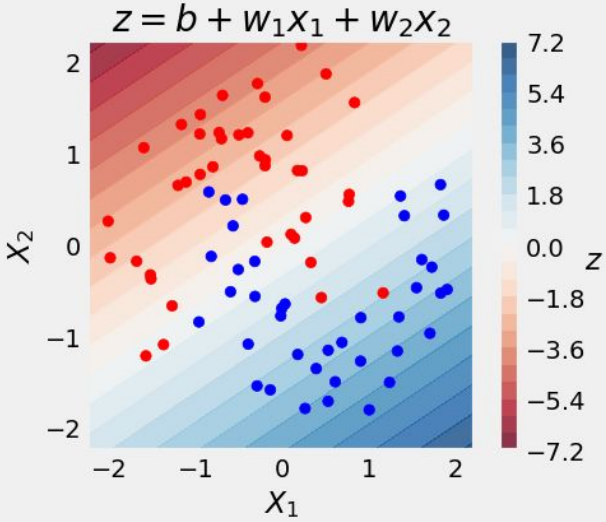
$$
\begin{aligned}
z = 0 &= b + w_1 x_1 + w_2 x_2 \\
-w_2 x_2 &= b + w_1 x_1 \\
x_2 &= -\frac{b}{w_2} - \frac{w_1}{w_2} x_1
\end{aligned}
$$

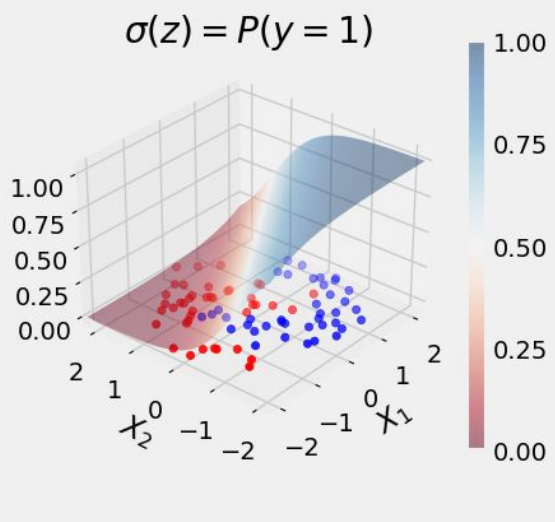
$$
\begin{aligned}
x_2 &= -\frac{0.0591}{1.8693} + \frac{1.1806}{1.8693} x_1 \\
x_2 &= -0.0316 + 0.6315 x_1
\end{aligned}
$$

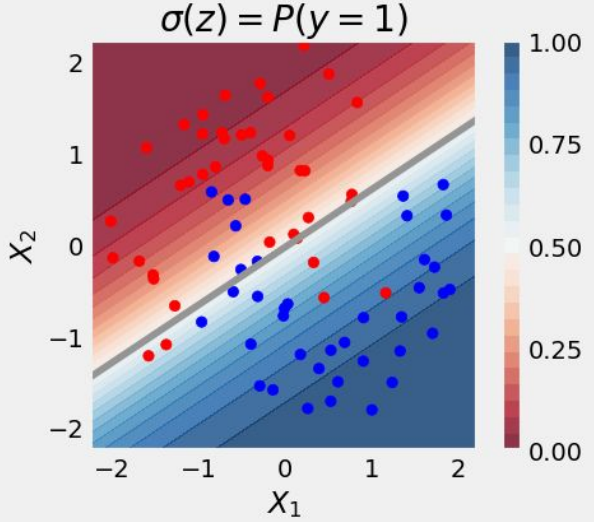
```
print(model.state_dict())
OrderedDict([('linear.weight', tensor([[ 1.1806, -1.8693]])), ('linear.bias', tensor([-0.0591]))])
```

$$
\begin{aligned}
z &= b + w_1 x_1 + w_2 x_2 \\
z &= -0.0591 + 1.1806 x_1 - 1.8693 x_2
\end{aligned}
$$

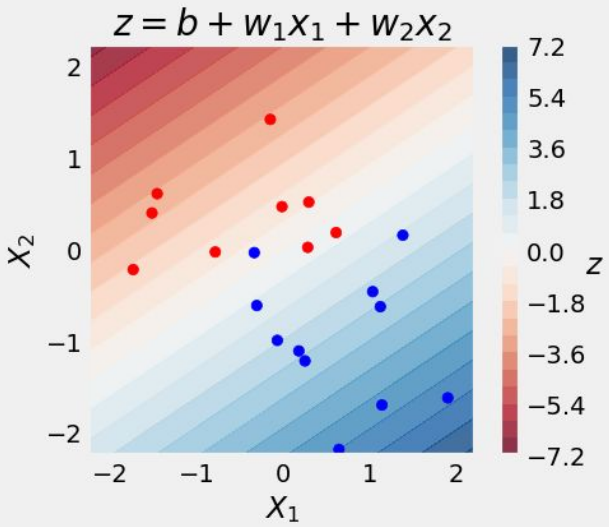
Decision Boundary (training data)

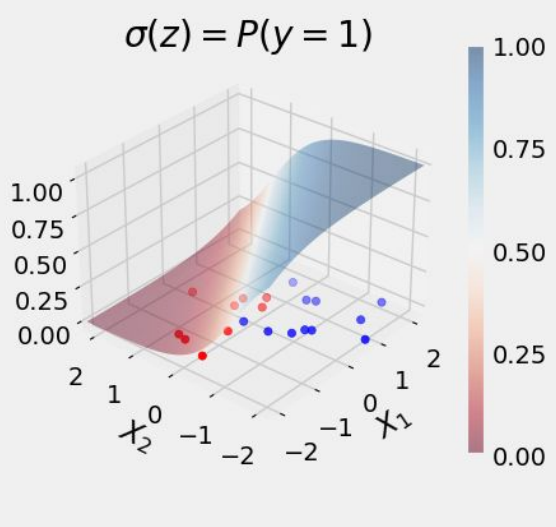


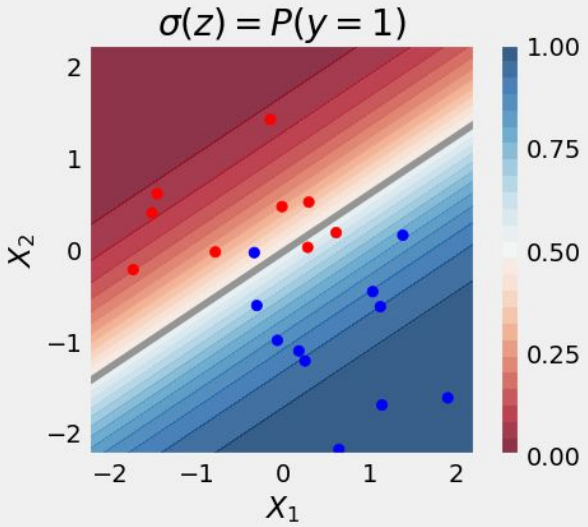




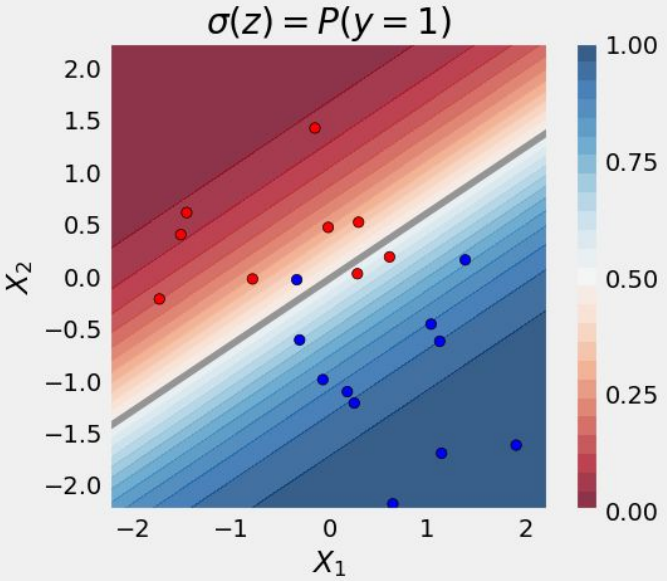
Decision Boundary (validation data)

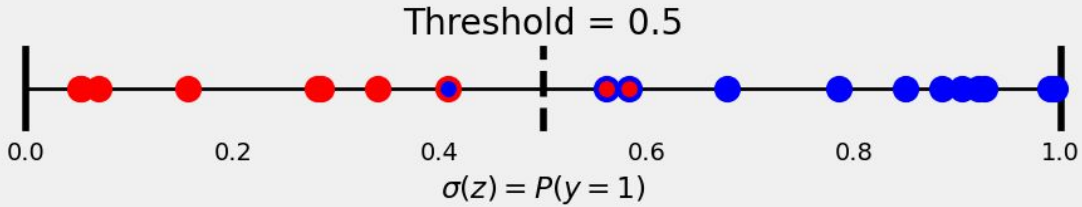




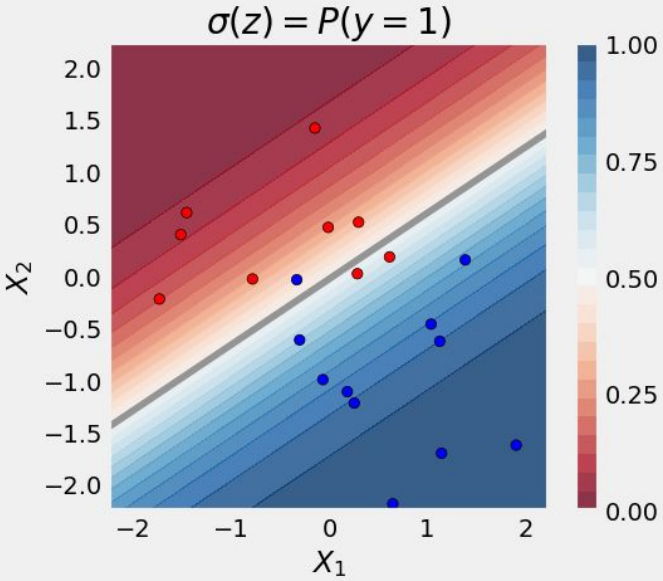


Classification Threshold

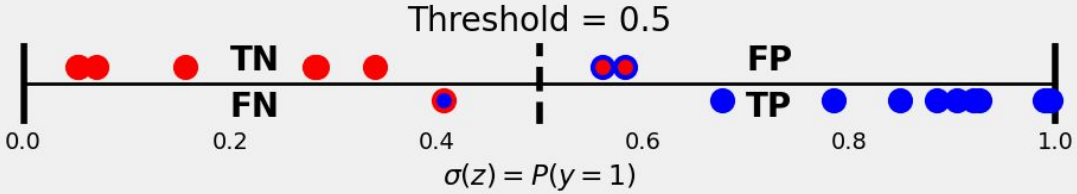




Confusion Matrix



$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}} \qquad \text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}}$$



$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \qquad \text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

$$\text{Accuracy} = \frac{\text{TP+TN}}{\text{TP+TN+FP+FN}}$$