PPGEEC2318

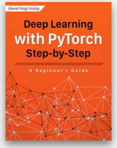
Machine Learning

Rethinking the training loop: a simple classification problem

Part 02

Ivanovitch Silva

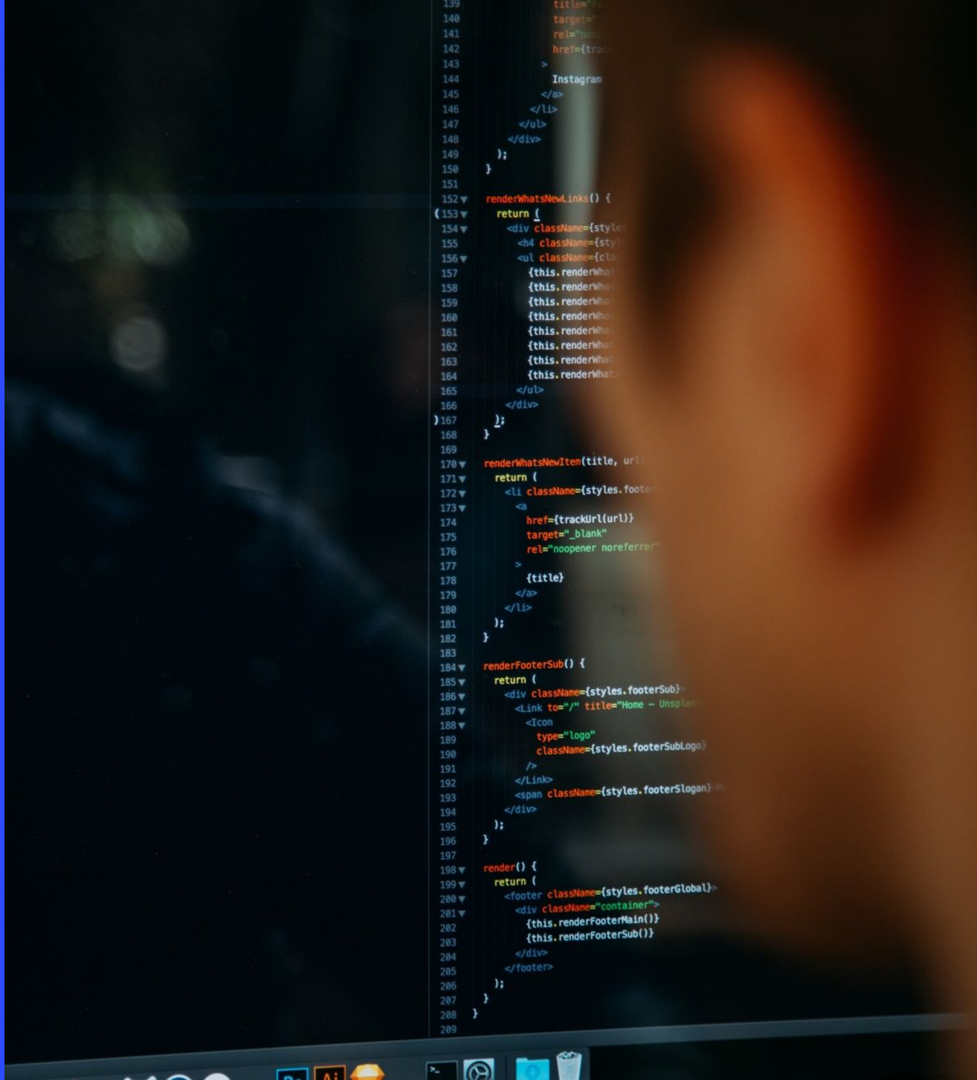
ivanovitch.silva@ufrn.br


Deep Learning
with PyTorch
Step-by-Step

Going Classy

Object-Oriented Programming - OOP

week04a.ipynb, week04b.ipynb

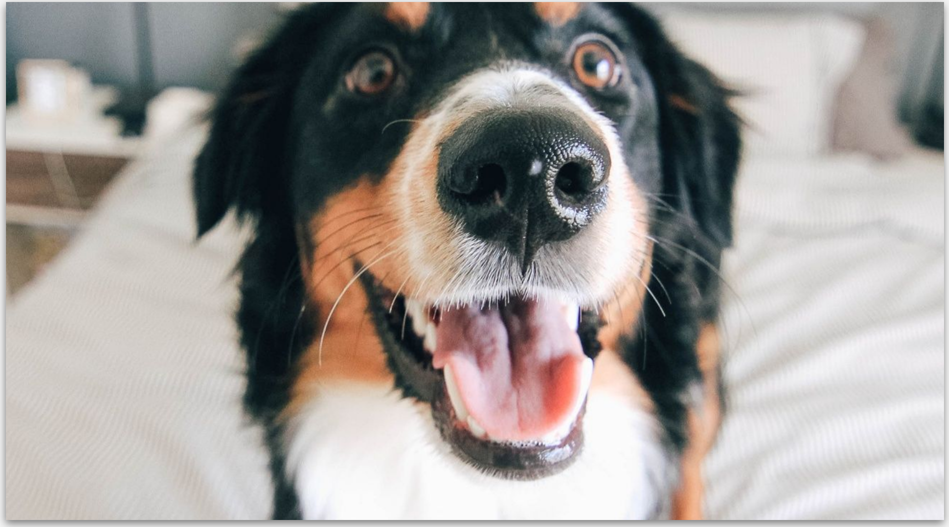


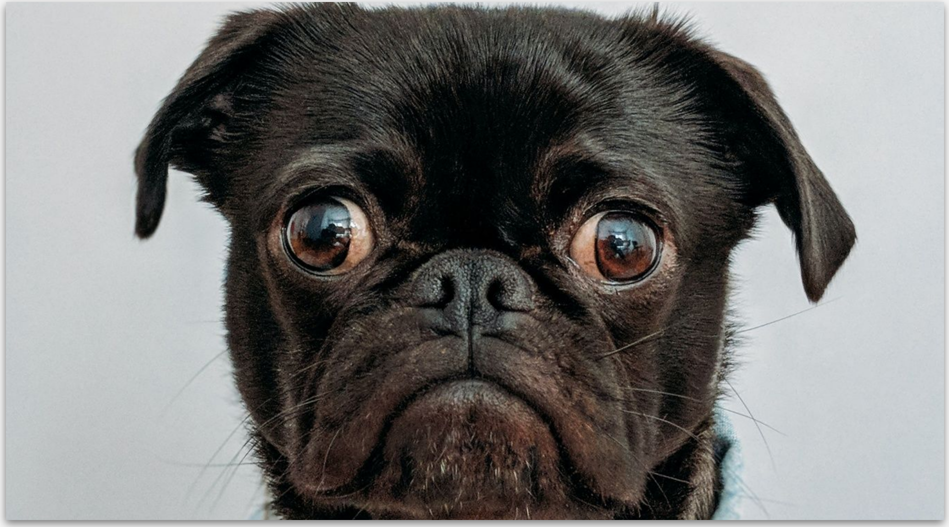
Attribute	Description
`model`	Neural network model (PyTorch object)
`loss_fn`	Loss function
`optimizer`	Optimizer (e.g., SGD, Adam)
`device`	Execution device (`cuda` or `cpu`)
`train_loader`	DataLoader for the training dataset
`val_loader`	DataLoader for the validation dataset
`losses`	List of training losses per epoch
`val_losses`	List of validation losses per epoch
`total_epochs`	Total number of completed training epochs
`train_step_fn`	Internal function to execute a training step
`val_step_fn`	Internal function to execute a validation step

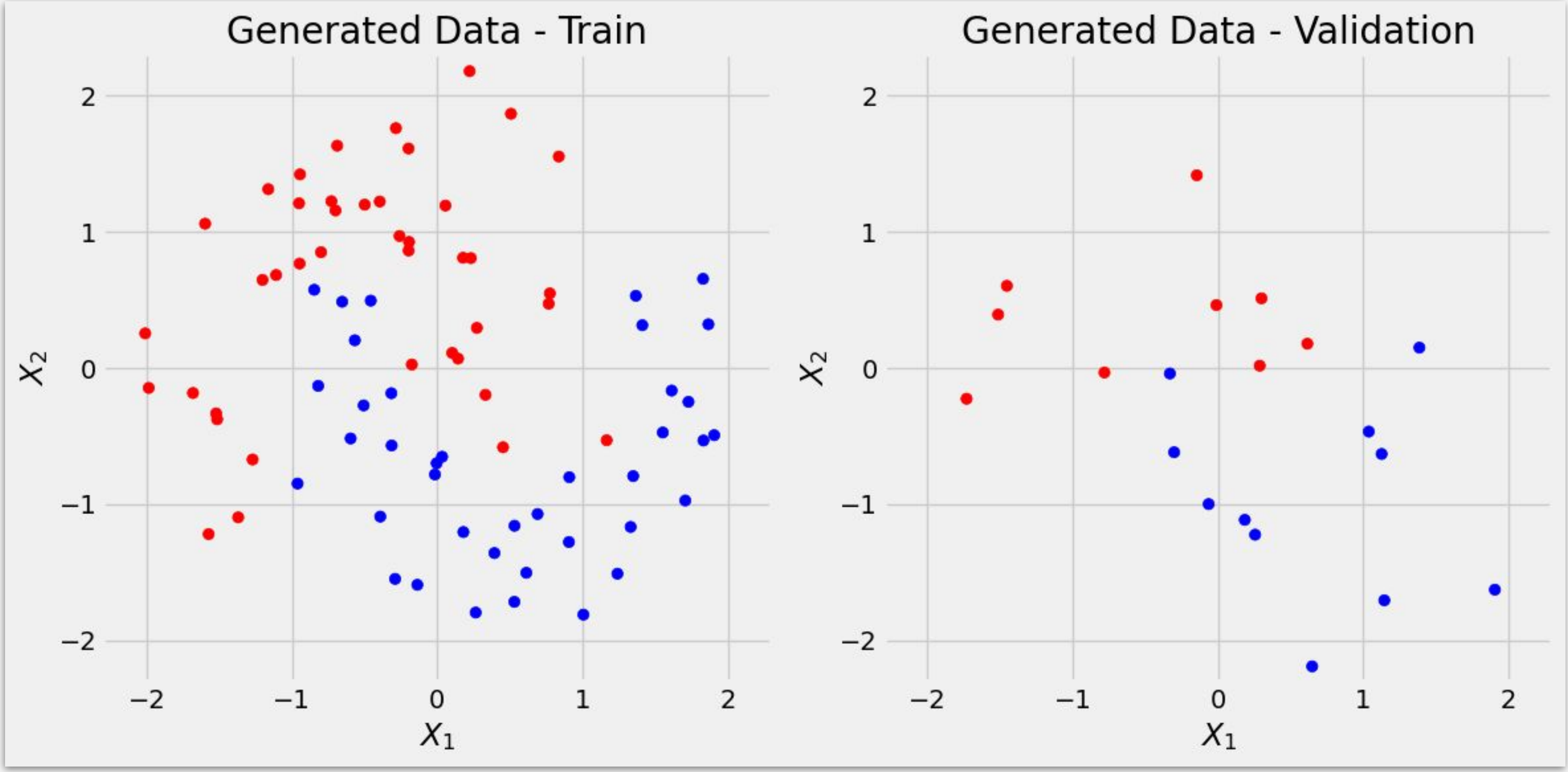
Method	Description
`__init__(model, loss_fn, optimizer)`	Constructor. Initializes all attributes
`to(device)`	Sets the device and moves the model to it
`set_loaders(train_loader, val_loader=None)`	Assigns train and validation DataLoaders
`_make_train_step_fn()`	Creates the training step function
`_make_val_step_fn()`	Creates the validation step function
`_mini_batch(validation=False)`	Executes one mini-batch loop for training or validation
`set_seed(seed=42)`	Sets random seeds for reproducibility
`train(n_epochs, seed=42)`	Executes the training loop for `n_epochs`
`save_checkpoint(filename)`	Saves the model and optimizer state to a file
`load_checkpoint(filename)`	Loads model and optimizer state from a file
`predict(x)`	Predicts the output for input `x`
`plot_losses()`	Plots the training and validation loss curves

A simple Classification Problem

week05c.ipynb

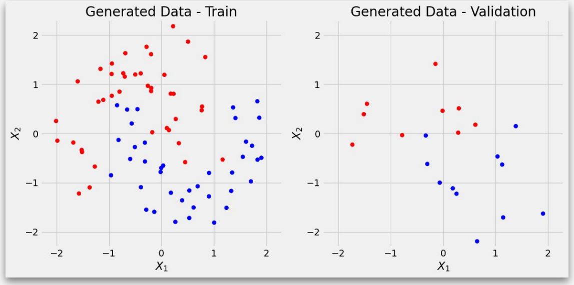






Generated Data - Train
Generated Data - Validation
X_1
X_2
−2
−1
0
1
2

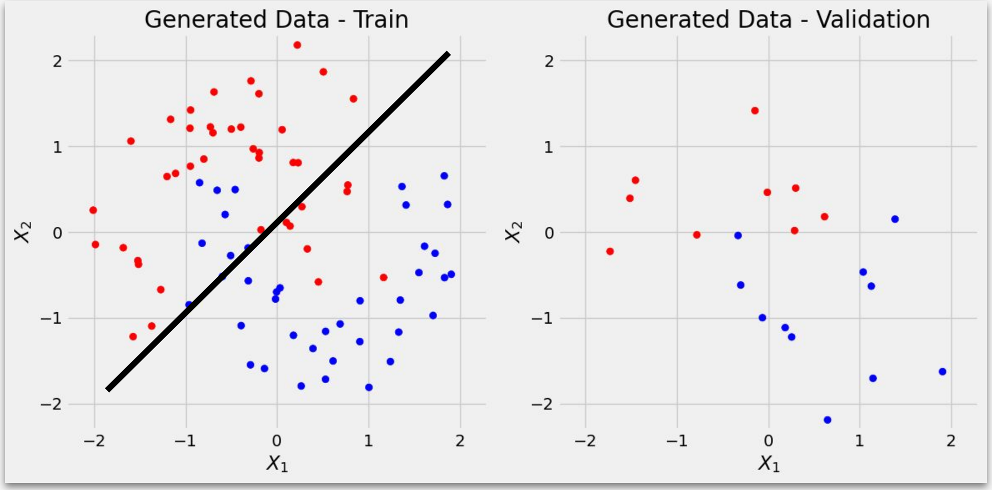
```python
# Generate a synthetic two-dimensional dataset with 100 samples
# 'make_moons' is used for binary classification problems
X, y = make_moons(n_samples=100, noise=0.3, random_state=0)

# Split the dataset into training and validation sets
# 20% of the data is used for validation
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=.2, random_state=13)

# Initialize a StandardScaler instance
sc = StandardScaler()

# Fit the scaler only on the training data
# This computes the mean and standard deviation to be used for later scaling
sc.fit(X_train)

# Transform both training and validation sets
# Scale the training data
X_train = sc.transform(X_train)

# Apply the same transformation to the validation data
X_val = sc.transform(X_val)
```

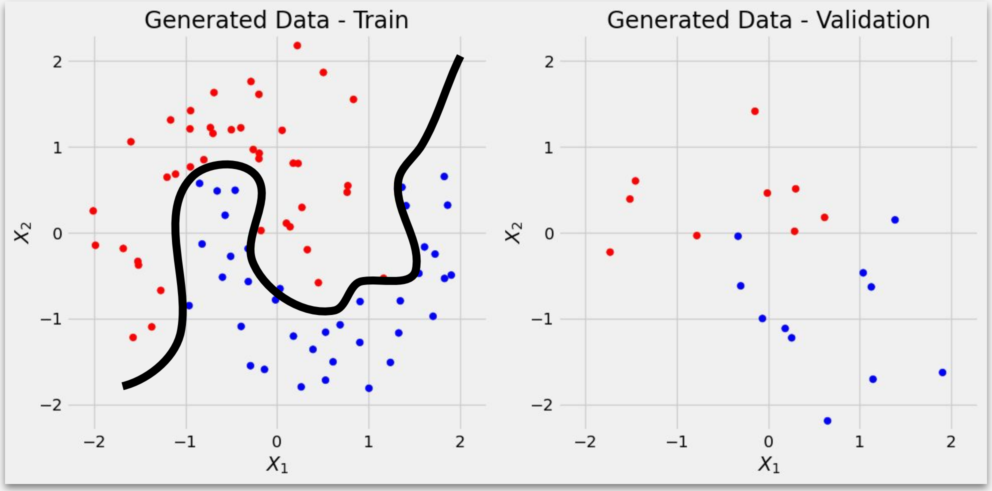


Linear Regression vs Logistic Regression

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

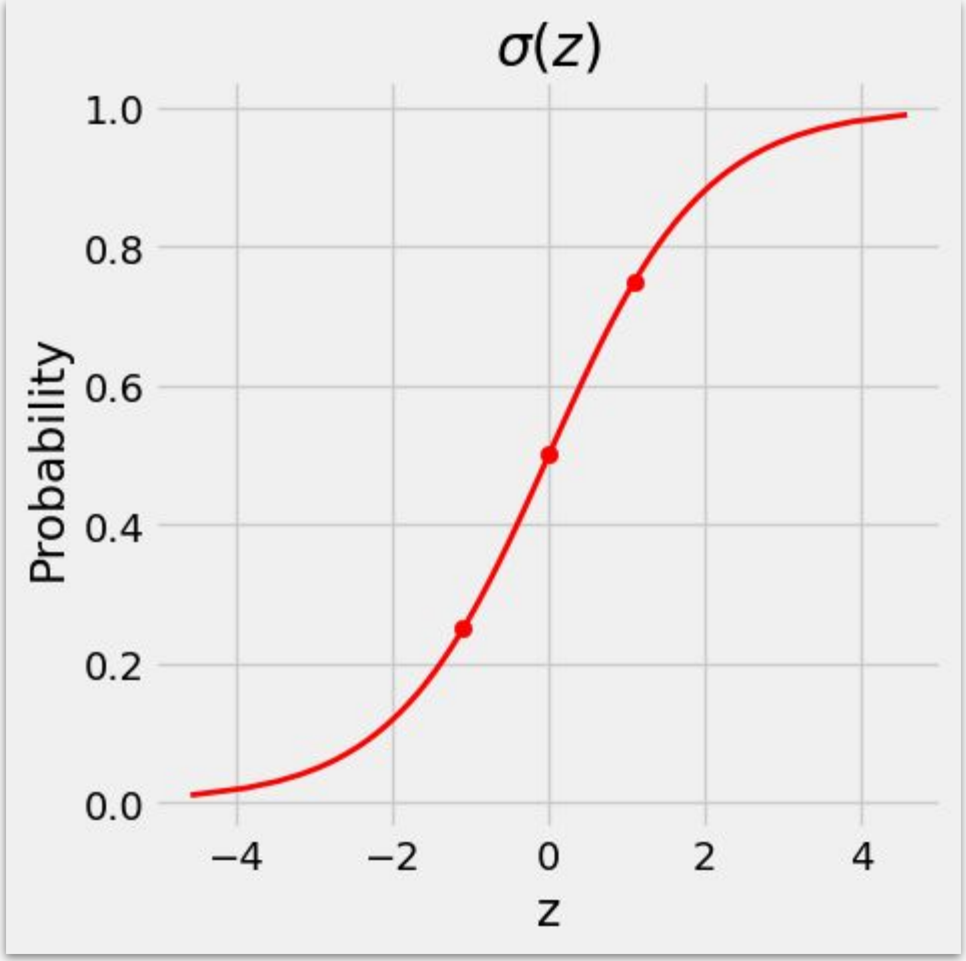




$$y = b + w_1 x_1 + w_2 x_2 + \epsilon$$

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

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



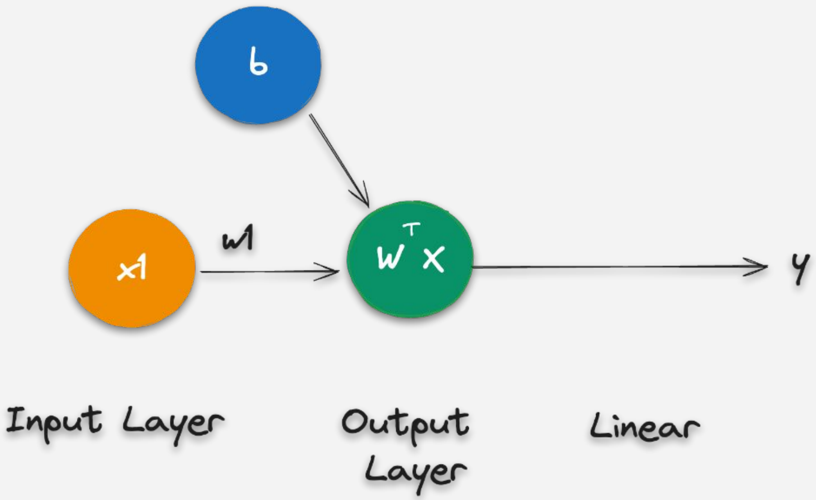
Only apply linear regression is not enough. The goal of the logistic regression is to estimate the **probability of occurrence**.

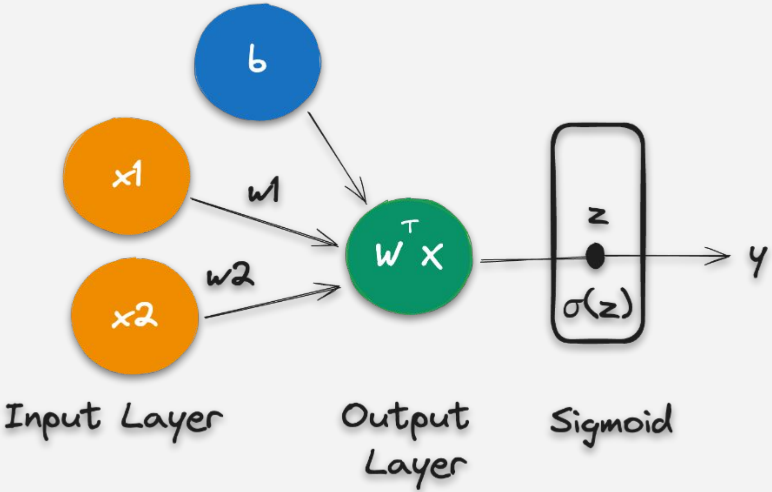
Logits

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

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

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



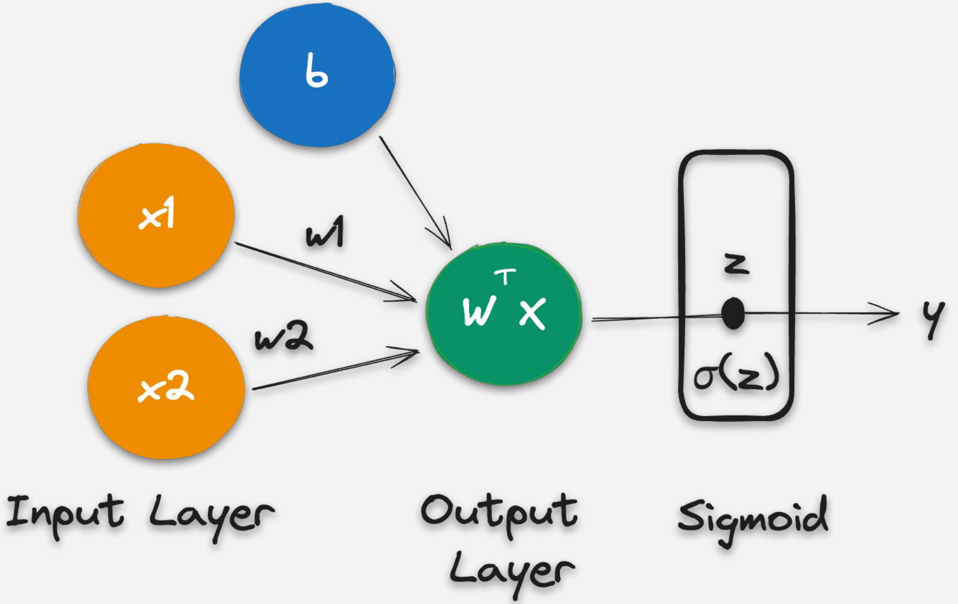


Linear Regression vs Logistic Regression

Logistic Regression

A Note on Notation

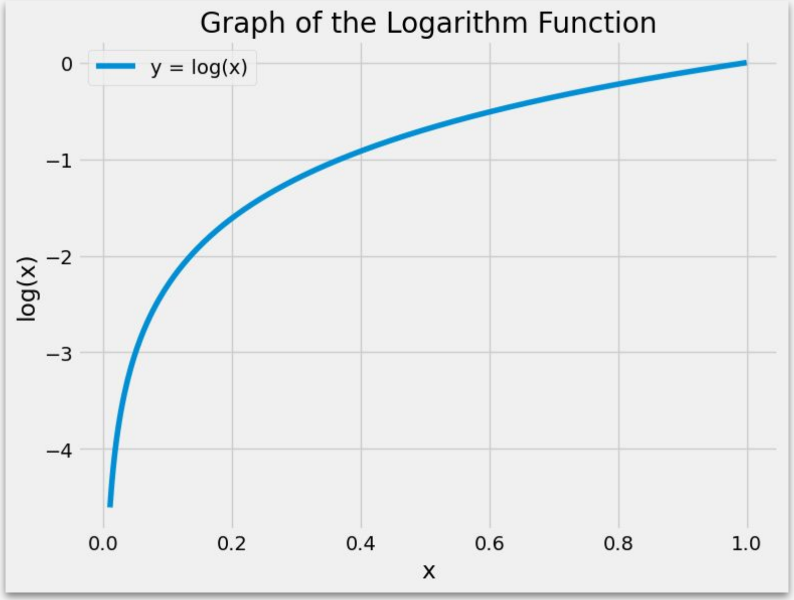
$$W = \underset{(3\times1)}{\begin{bmatrix} b \\ w_1 \\ w_2 \end{bmatrix}} ; X = \underset{(3\times1)}{\begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}}$$



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

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

```
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]))])
```



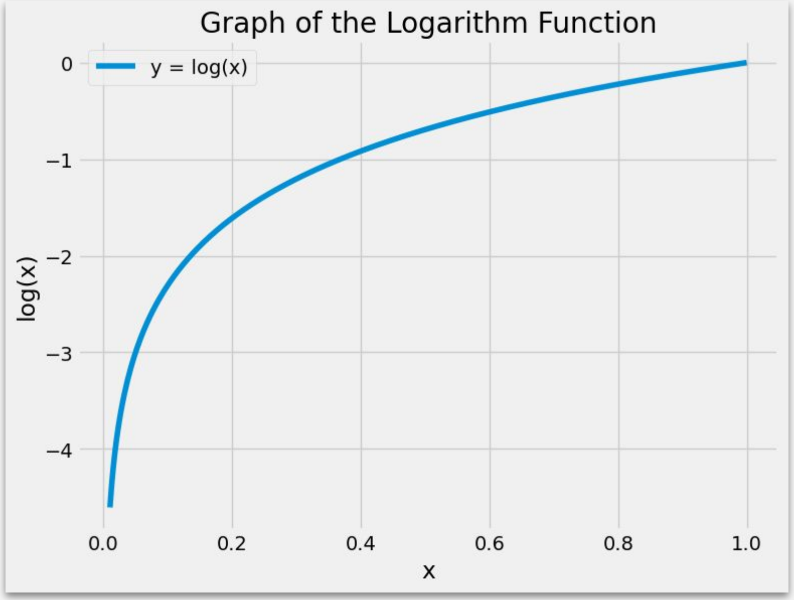
#intuitive-way

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

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

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

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

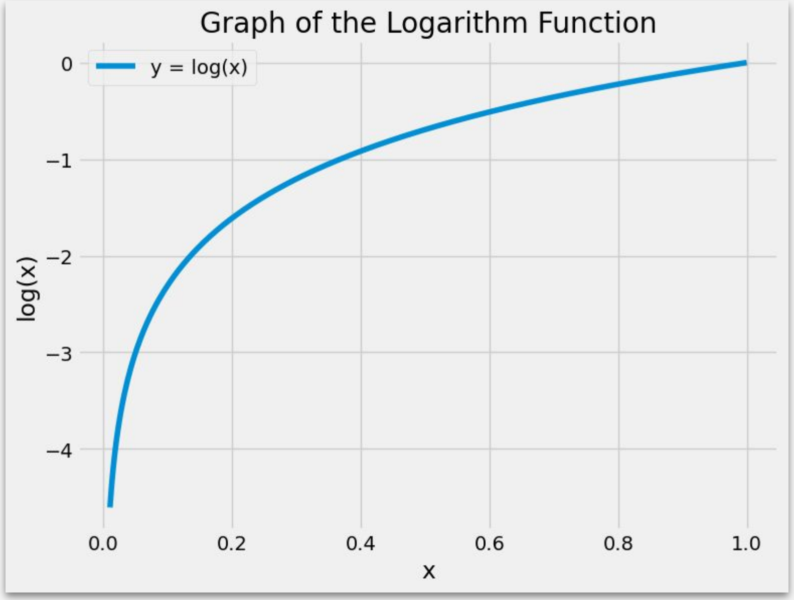


#intuitive-way

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)

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

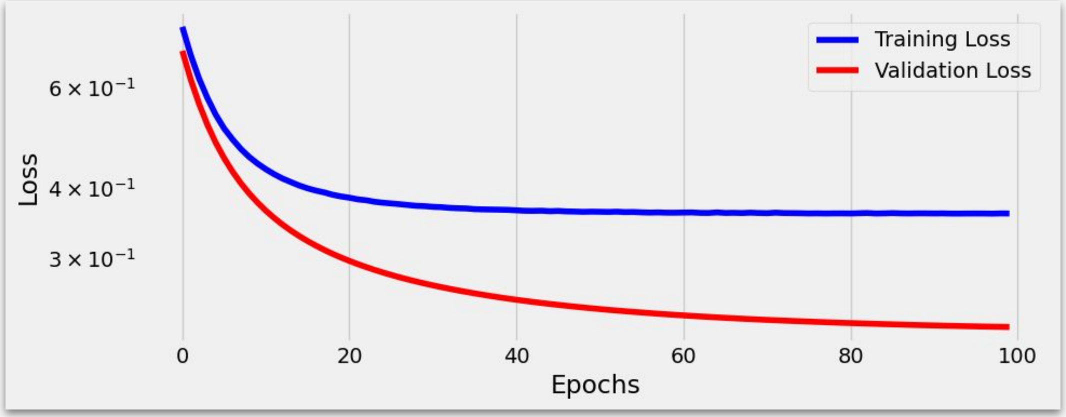
```
loss_fn = nn.BCELoss(reduction='mean')

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)
```

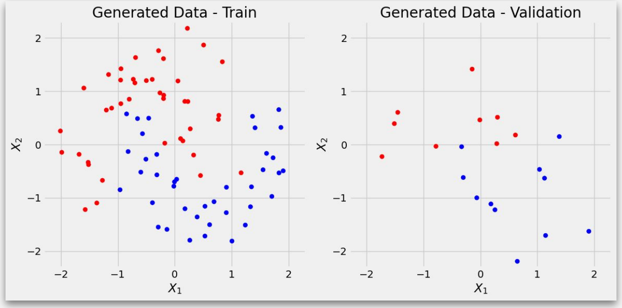
Model Configuration

```python
# 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

```python
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

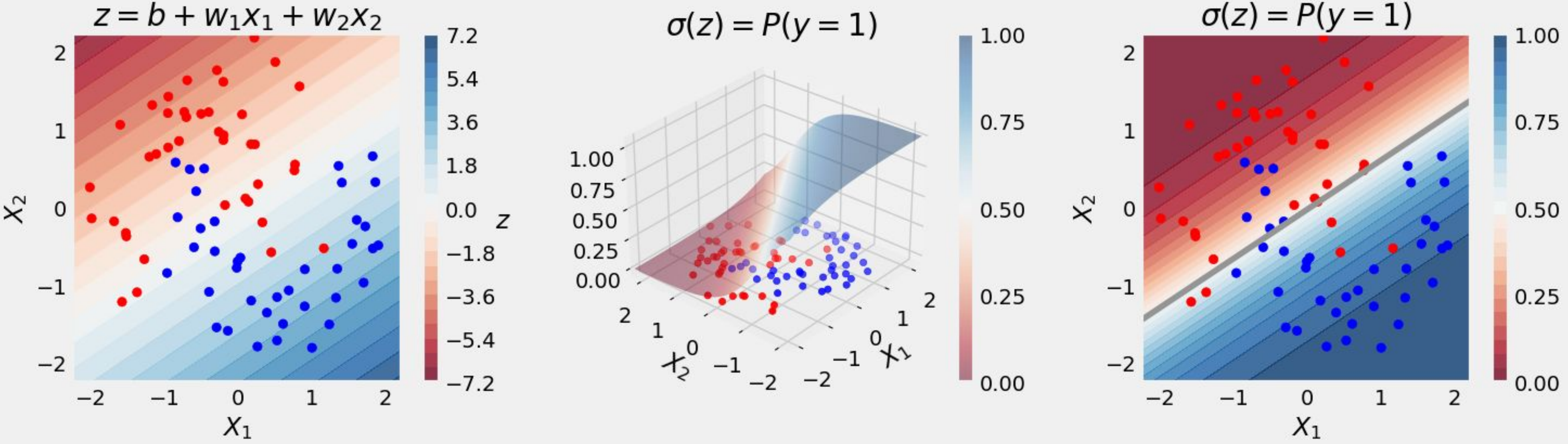
$$\begin{aligned} z = 0 &= b + w_1x_1 + w_2x_2 \\ -w_2x_2 &= b + w_1x_1 \\ x_2 &= -\frac{b}{w_2} - \frac{w_1}{w_2}x_1 \end{aligned}$$

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

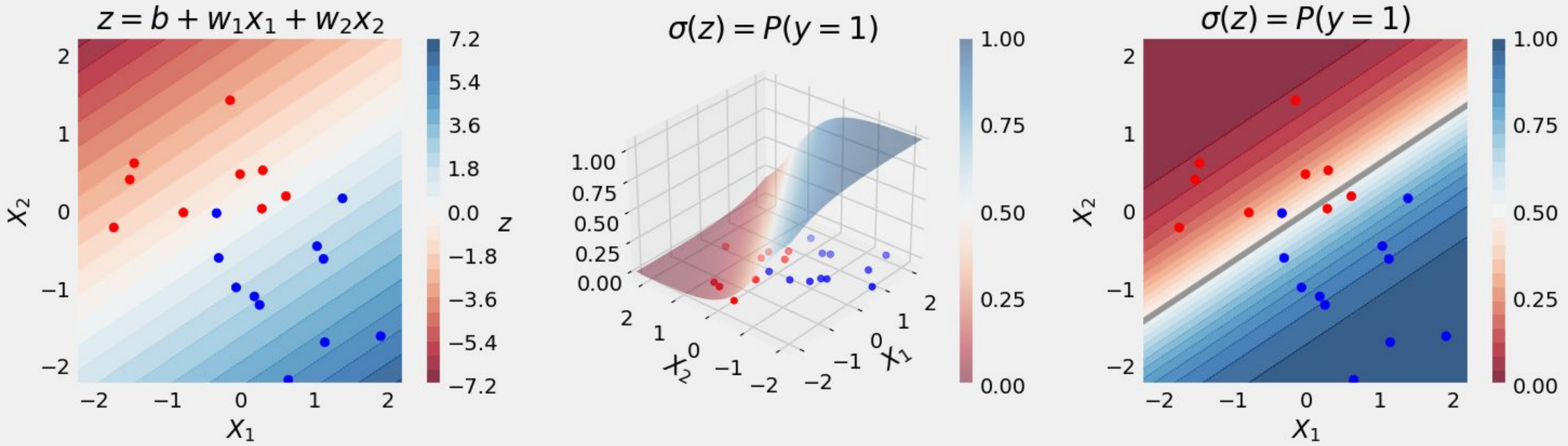
```
print(model.state_dict())
OrderedDict([('linear.weight', tensor([[ 1.1806, -1.8693]])), ('linear.bias', tensor([-0.0591]))])
```

$$\begin{aligned} z &= b + w_1x_1 + w_2x_2 \\ z &= -0.0591 + 1.1806x_1 - 1.8693x_2 \end{aligned}$$

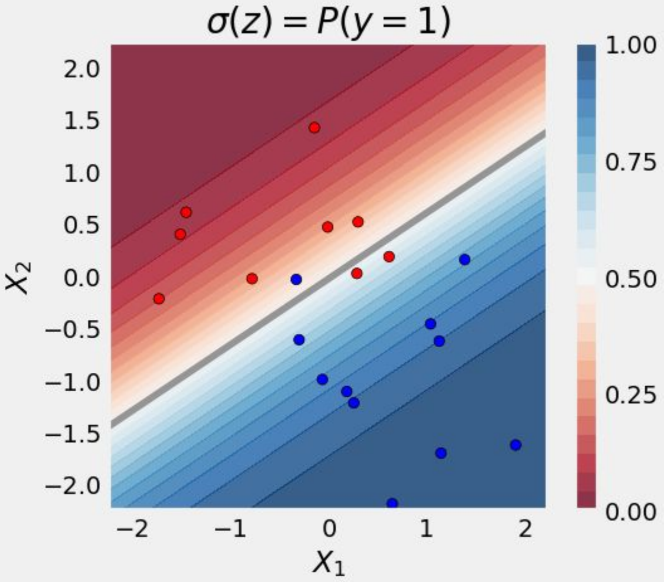
Decision Boundary (training data)

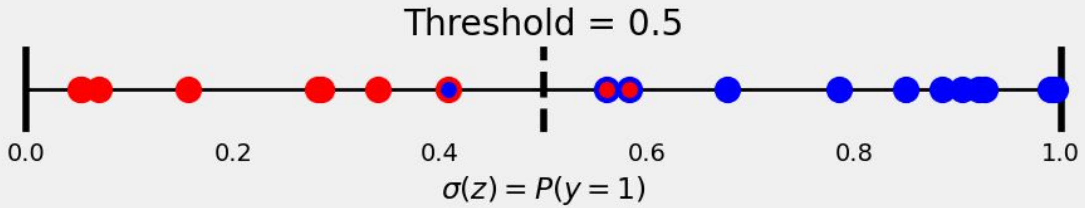


Decision Boundary (validation data)

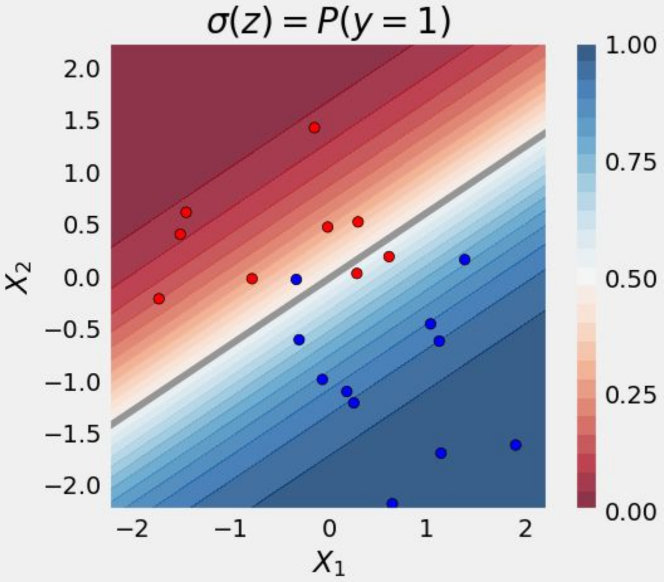


Classification Threshold

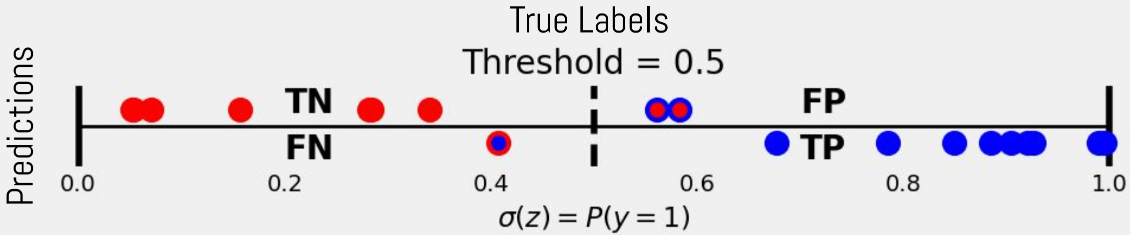




Confusion Matrix



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



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

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