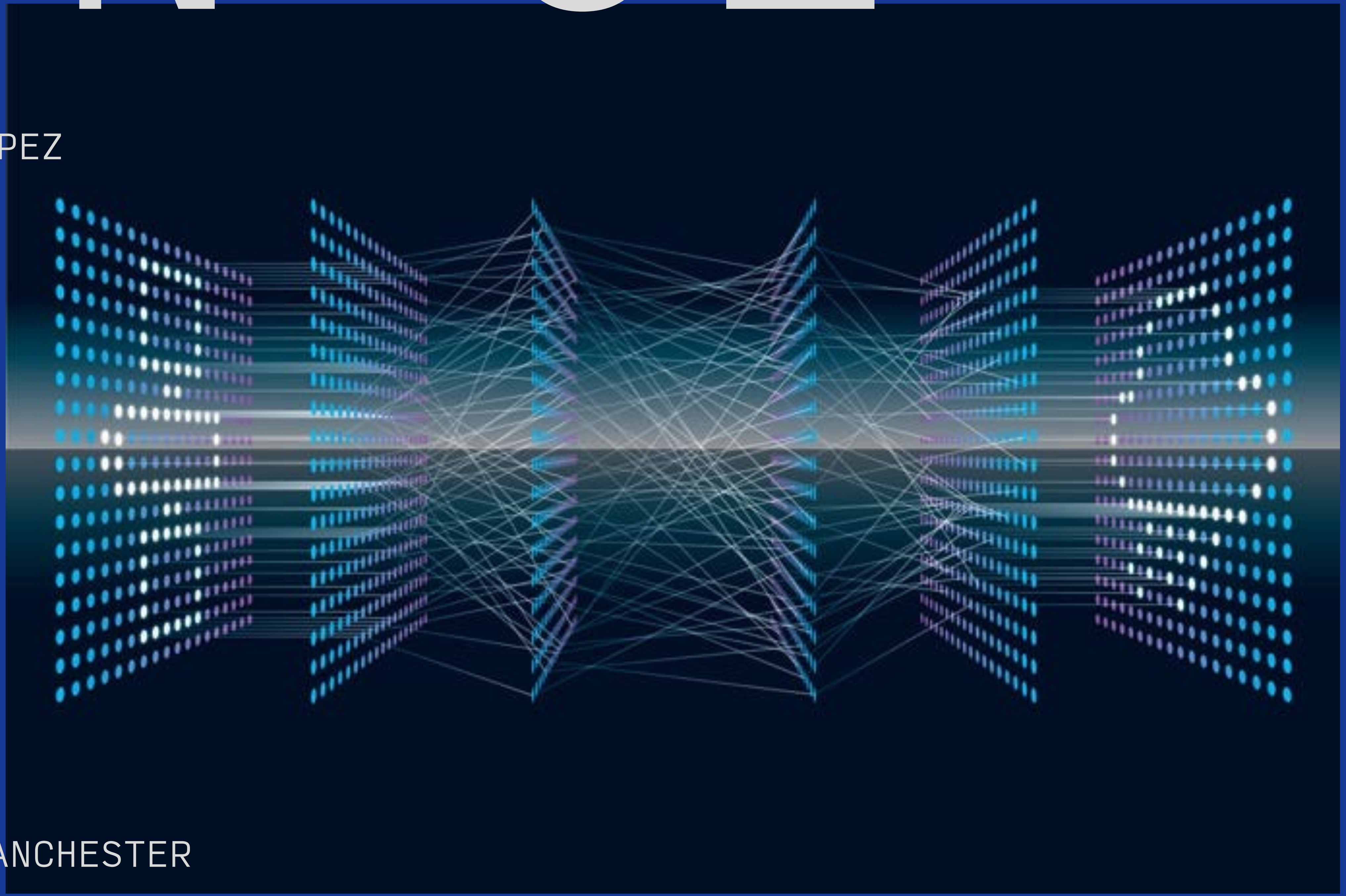
SESSION 02

WORKSHOP

FEBRUARY 2026

BY DR CORONA-LOPEZ

ARTIFICIAL NEURAL NETWORKS

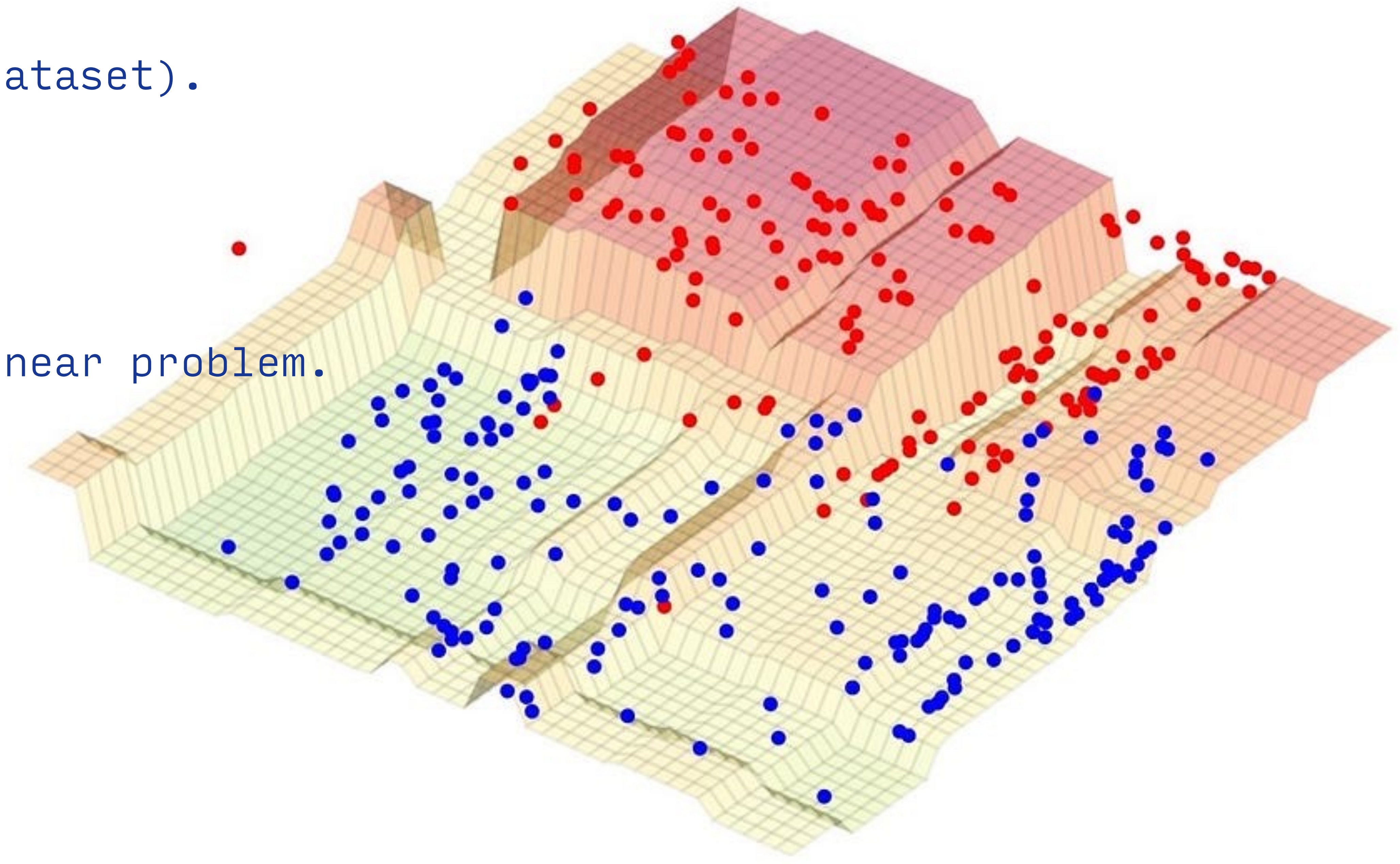


FACULTY OF SCIENCE AND ENGINEERING

UNIVERSITY OF MANCHESTER

AGENDA

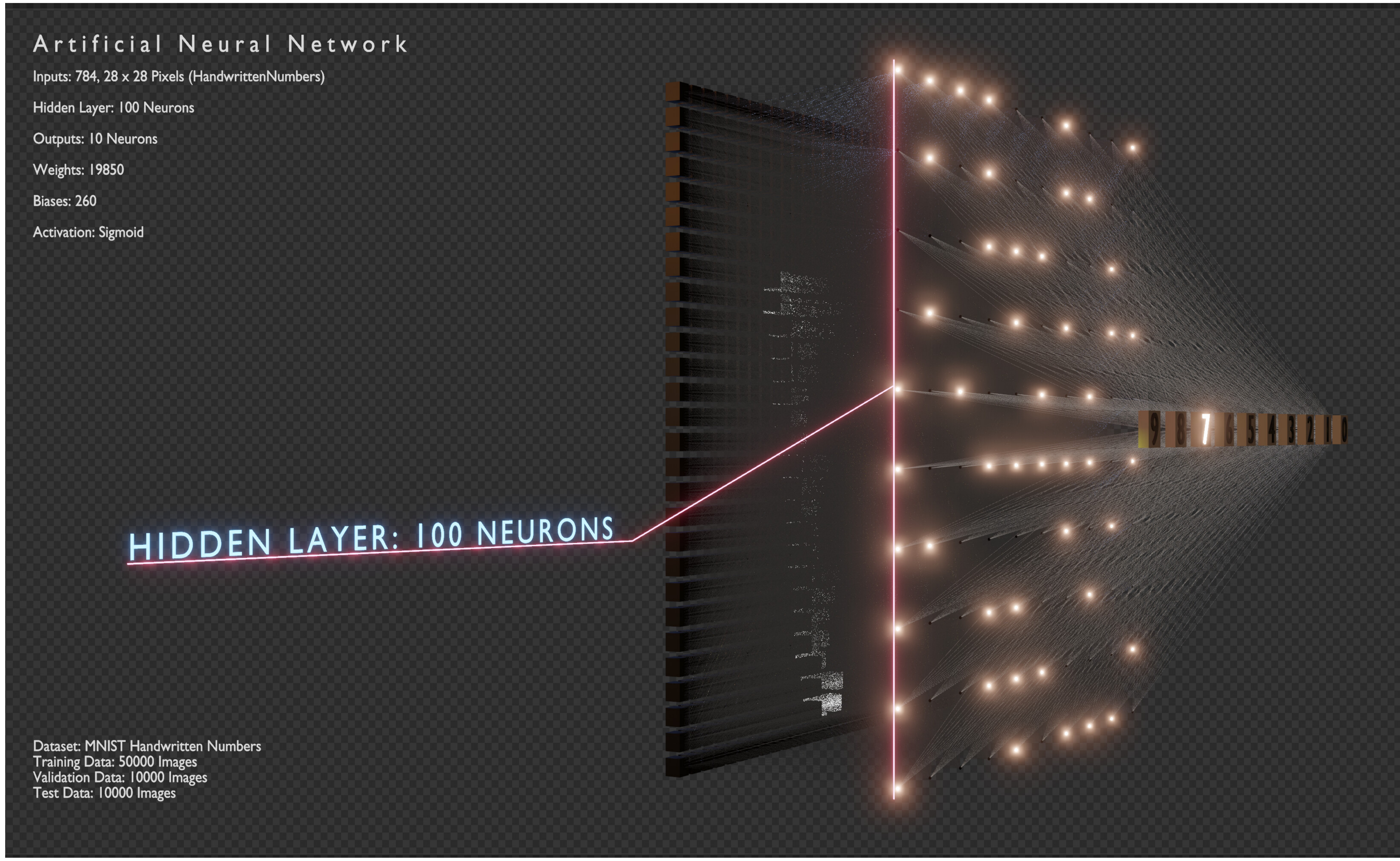
- **anatomy of a neuron:** linear classification and manual forward pass.
- **learning via gradient descent:** binary cross-entropy and the pytorch standard recipe.
- **the non-linear wall:** why lines fail (the circles dataset).
- **multi-layer perceptrons (mlps):** combining neurons and non-linear activations.
- **visualizing probability fields:** solving the non-linear problem.



neural networks are computational models inspired by the human brain's neural structure.

they consist of interconnected neurons organized in layers that transform input data into meaningful outputs.

each neuron performs weighted calculations on its inputs and applies an activation function to introduce non-linearity.


Artificial Neural Network
Inputs: 784, 28 x 28 Pixels (HandwrittenNumbers)
Hidden Layer: 100 Neurons
Outputs: 10 Neurons
Weights: 19850
Biases: 260
Activation: Sigmoid
HIDDEN LAYER: 100 NEURONS
Dataset: MNIST Handwritten Numbers
Training Data: 50000 Images
Validation Data: 10000 Images
Test Data: 10000 Images

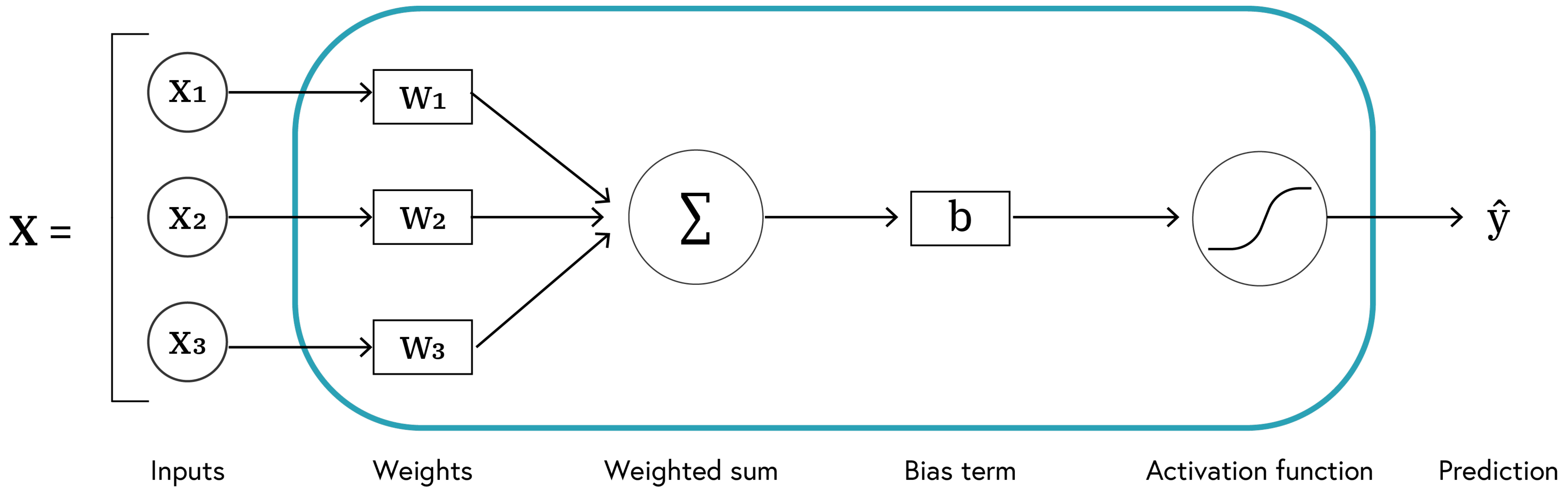
NEURON

basic computational units of neural networks

$$\hat{y} = f(W \cdot X + b)$$

each neuron:

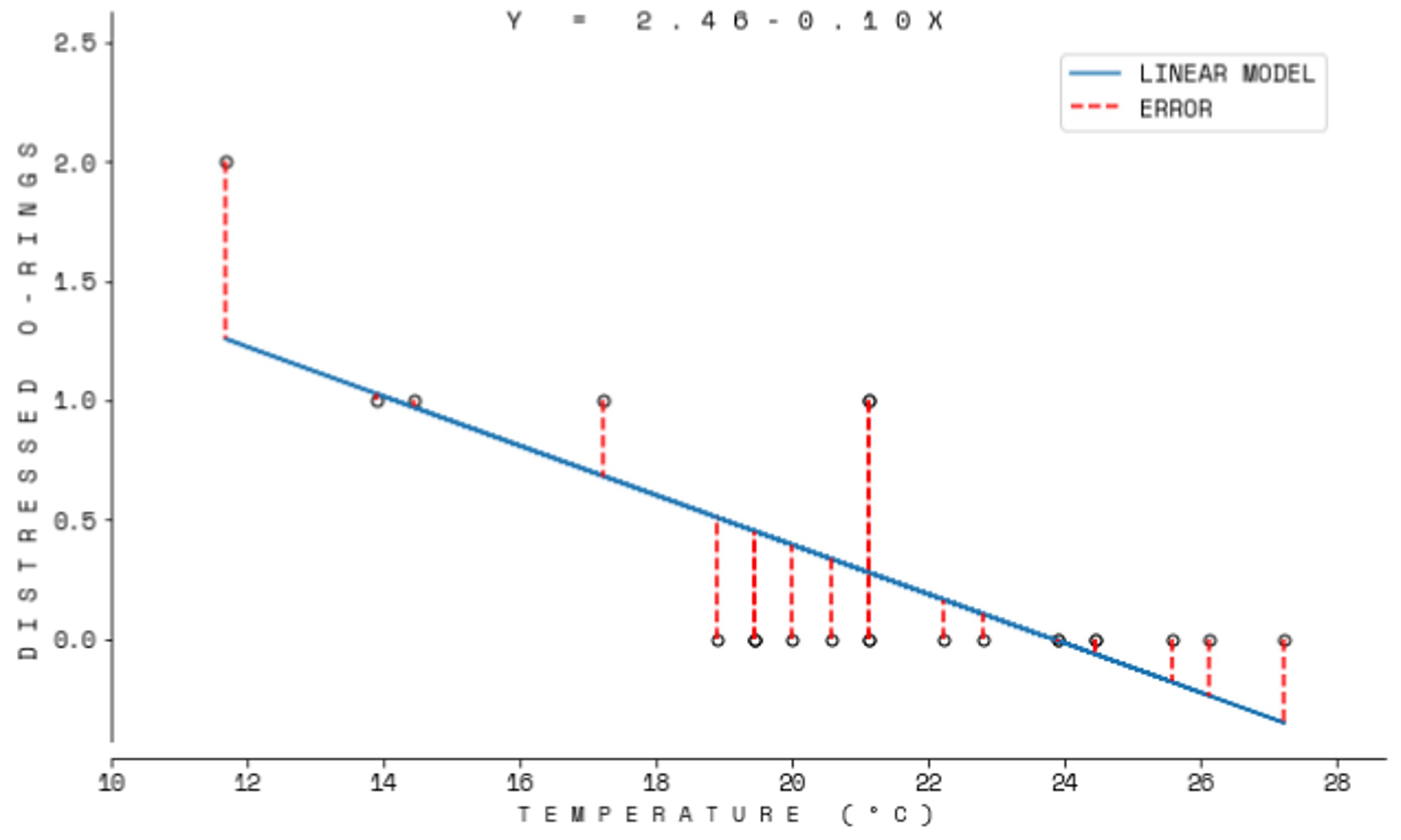
- receives inputs
- applies weights and bias
- processes through activation function
- produces output



TRAINING BASICS

1. **forward pass:** feed data through network to get predictions
2. **loss calculation:** measure error between predictions and targets
3. **backward pass:** compute gradients to adjust parameters
4. **parameter update:** improve model using gradient descent

$$\hat{y} = \underset{x}{\operatorname{argmin}} ||W x - b||_2$$

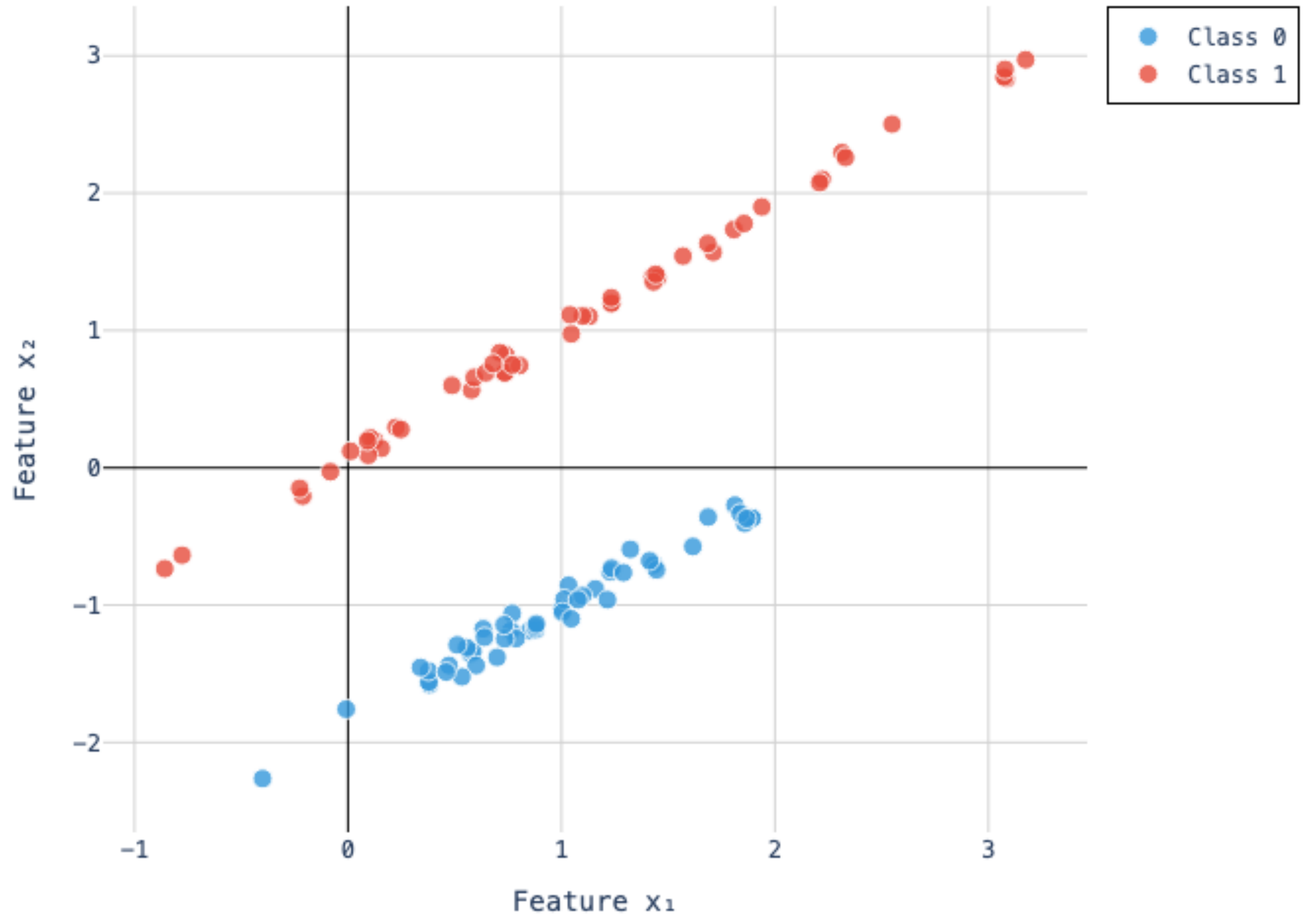


FORWARD PASS

forward pass formula:

$$Z = XW^T + b = \mathit{Logits} \rightarrow A = \sigma(Z) = y \rightarrow \mathit{Probabilities}$$

```
torch.matmul(X, weights) + bias
```



LOSS FUNCTION

the goal:

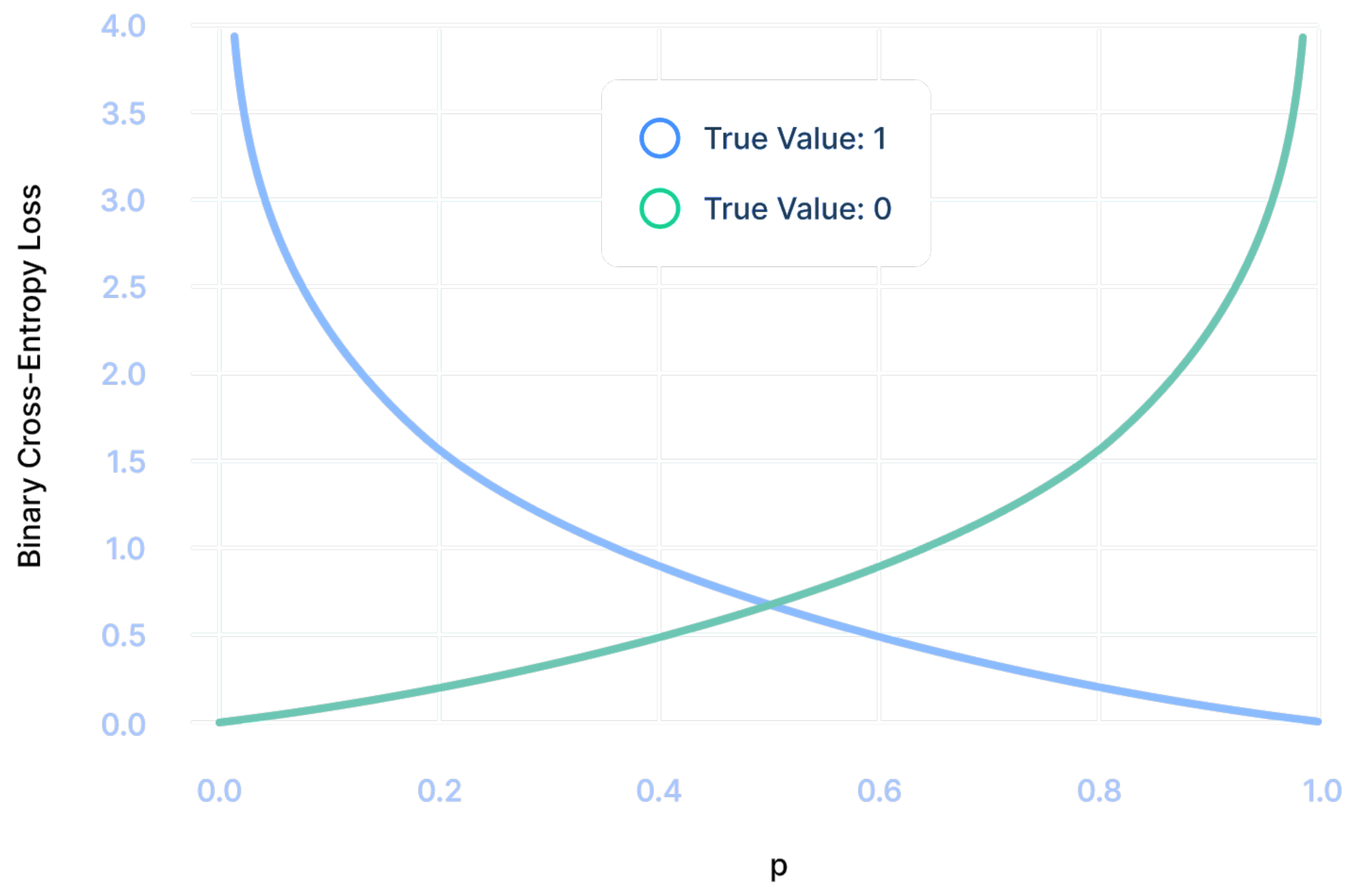
iteratively update random initial parameters to draw an accurate decision boundary.

the loss function (binary cross-entropy):

instead of mean squared error (MSE), binary classification uses BCE.

it measures the "distance" between predicted probabilities ($\hat{y}$) and true labels (y).

$$L = -\frac{1}{N}\sum[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$



LEARNING ALGORITHM

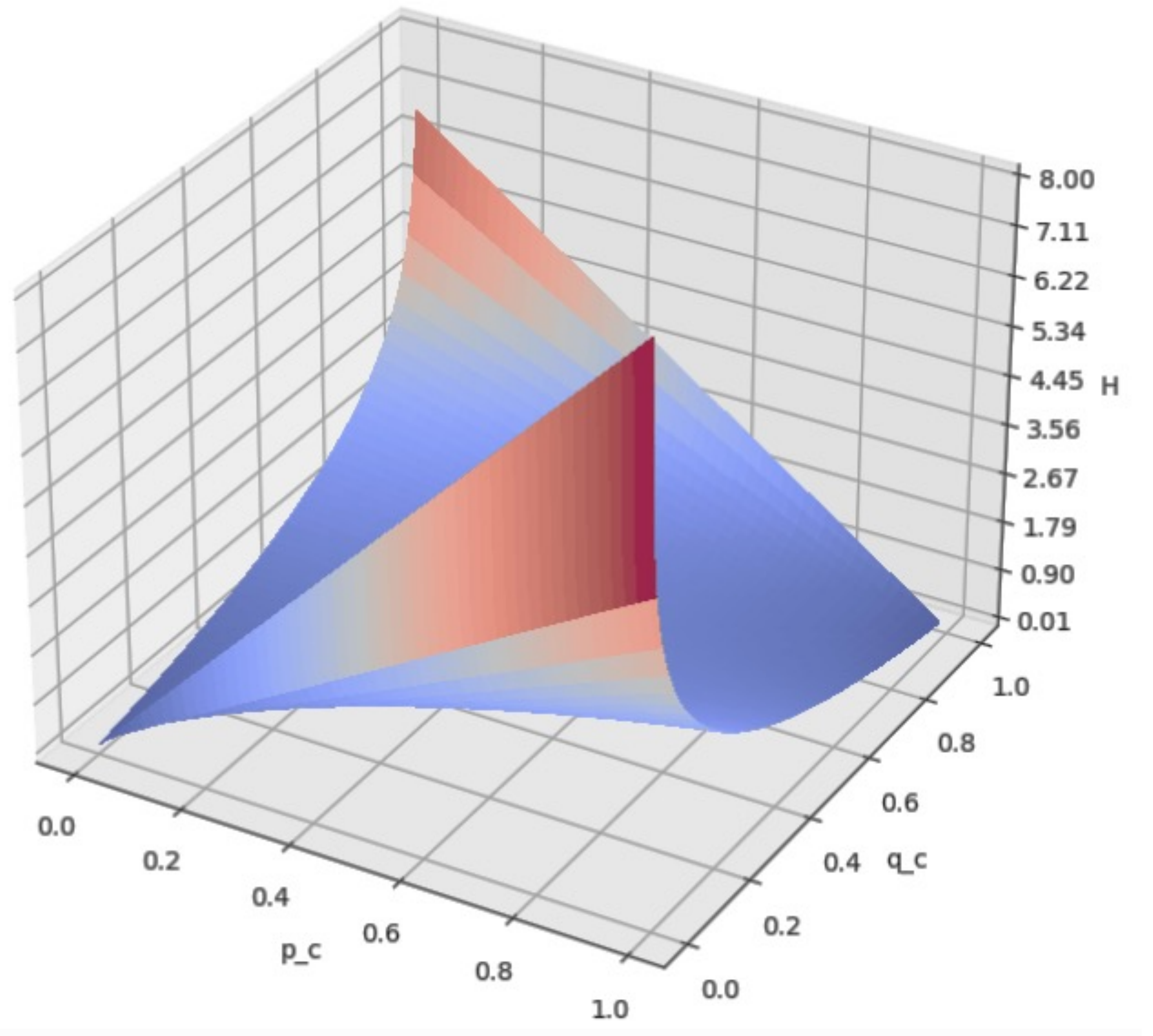
gradient descent:

basically a form of sensitivity analysis. we are asking the math:

how much does the output change if i nudge this parameter?

updates parameters in the direction that most reduces the loss function (where η is the learning rate).

$$w_{new} = w_{old} - \eta \frac{\partial L}{\partial w} \qquad b_{new} = b_{old} - \eta \frac{\partial L}{\partial b}$$



LEARNING ALGORITHM

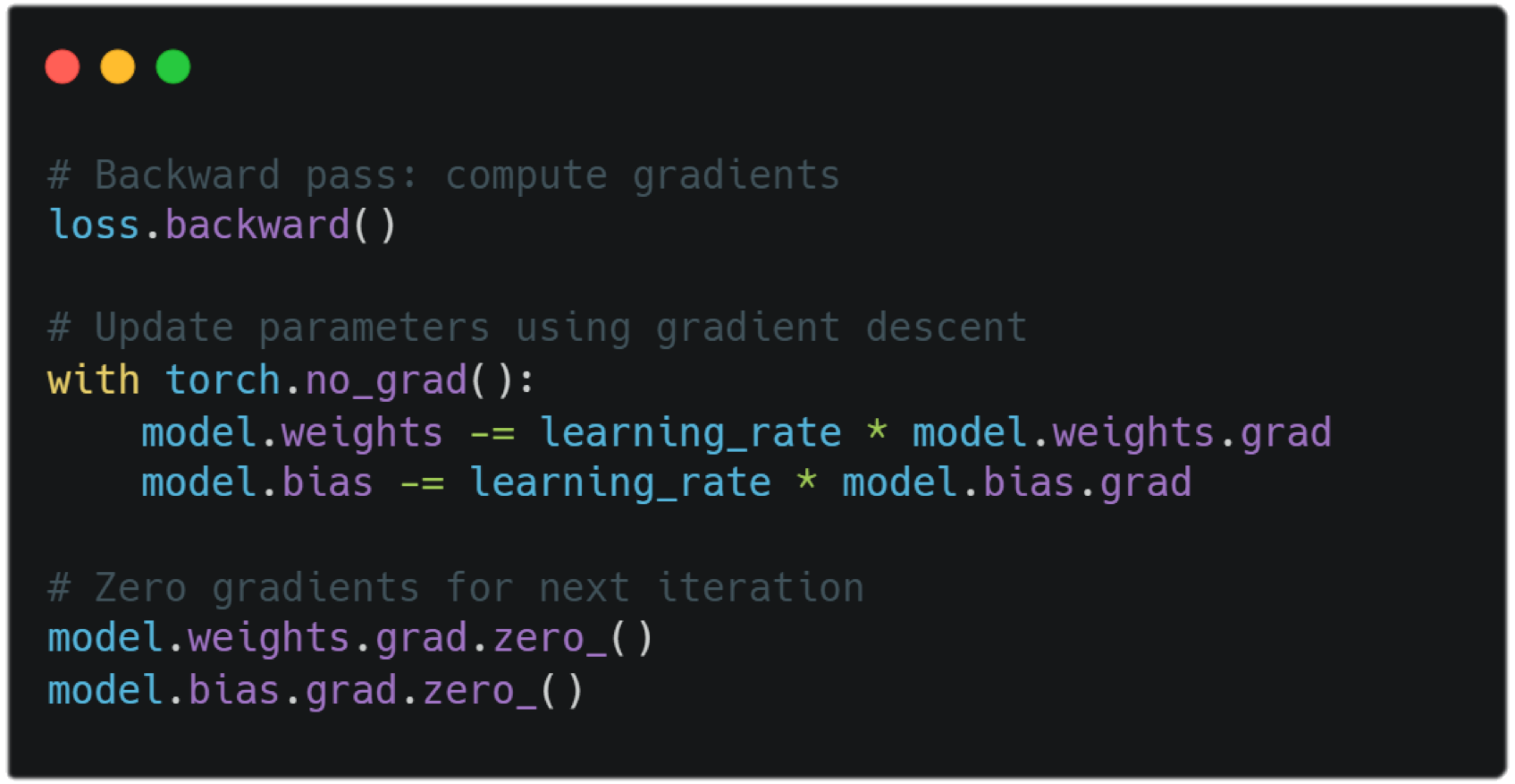
gradient descent:

basically a form of sensitivity analysis. we are asking the math:

how much does the output change if i nudge this parameter?

updates parameters in the direction that most reduces the loss function (where η is the learning rate).

$$w_{new} = w_{old} - \eta \frac{\partial L}{\partial w} \qquad b_{new} = b_{old} - \eta \frac{\partial L}{\partial b}$$



```python
# Backward pass: compute gradients
loss.backward()

# Update parameters using gradient descent
with torch.no_grad():
    model.weights -= learning_rate * model.weights.grad
    model.bias -= learning_rate * model.bias.grad

# Zero gradients for next iteration
model.weights.grad.zero_()
model.bias.grad.zero_()
```

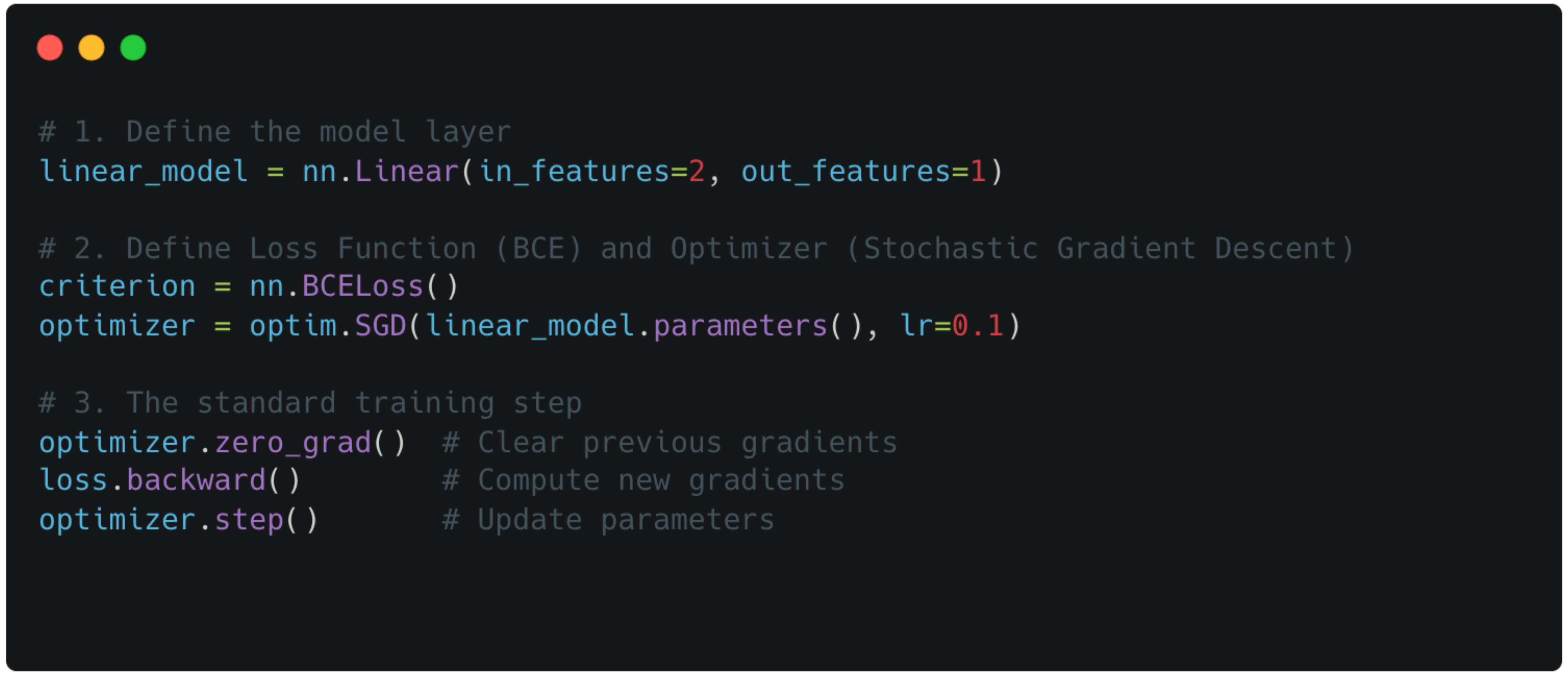
PYTORCH RECIPE

moving beyond manual math:

`torch.nn.Linear(in_features, out_features)` replaces manual matrix multiplication. it implements the linear transformation and manages the learnable weights and biases automatically.

benefits of using pytorch modules:

1. **automatic parameter management** (no need to manually initialize matrices)
2. **gpu compatibility** (seamlessly move data to gpu with `.to('cuda')`)
3. **integration with optimizers** (works perfectly with `torch.optim`)
4. **standard interface** (follows the `nn.module` pattern used in all ai research)



```
# 1. Define the model layer
linear_model = nn.Linear(in_features=2, out_features=1)

# 2. Define Loss Function (BCE) and Optimizer (Stochastic Gradient Descent)
criterion = nn.BCELoss()
optimizer = optim.SGD(linear_model.parameters(), lr=0.1)

# 3. The standard training step
optimizer.zero_grad()  # Clear previous gradients
loss.backward()        # Compute new gradients
optimizer.step()       # Update parameters
```

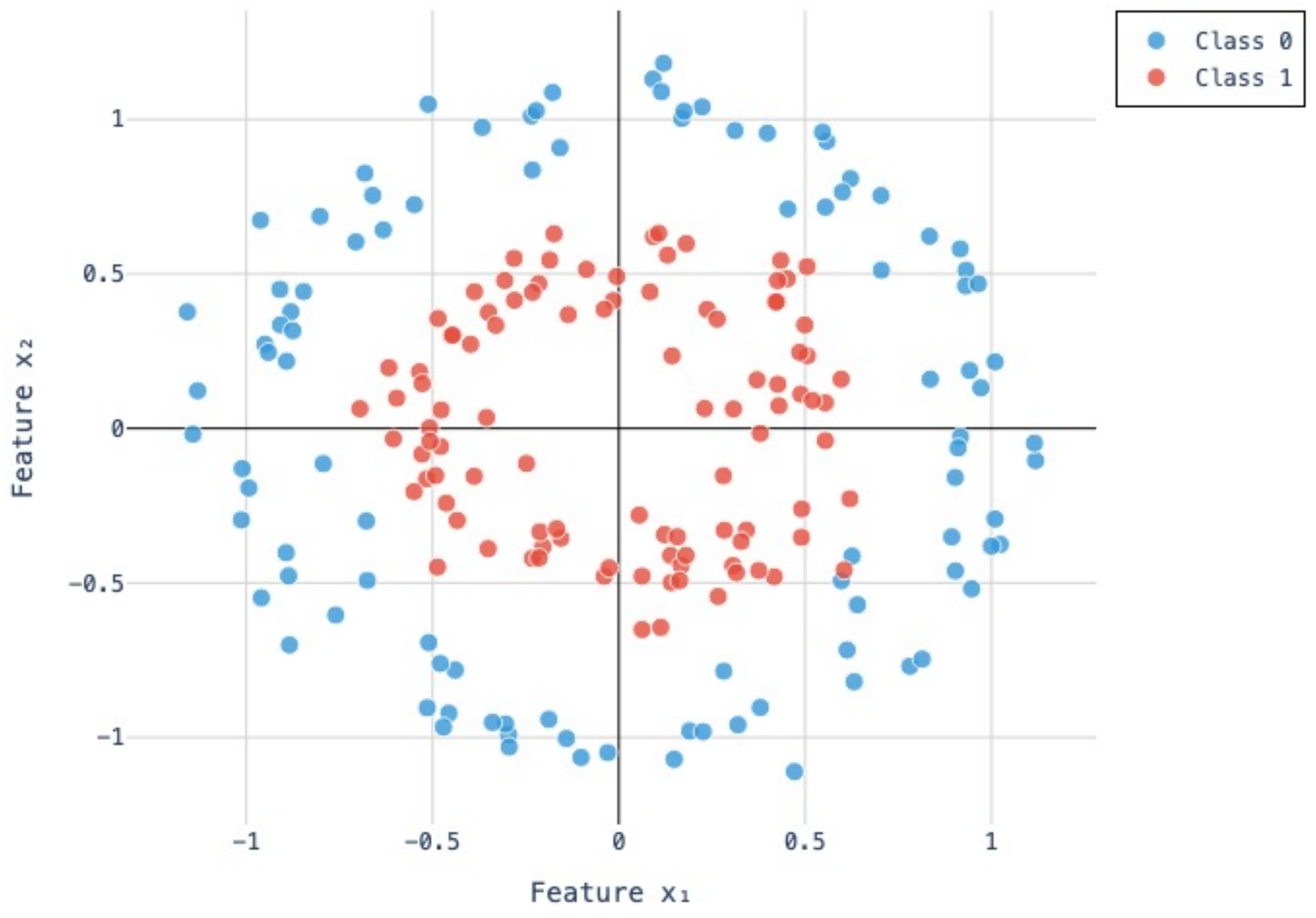
NON-LINEAR WALL

the limitation of linear models: a single neuron (or standard logistic regression) can only ever draw a single, straight hyperplane to separate data.

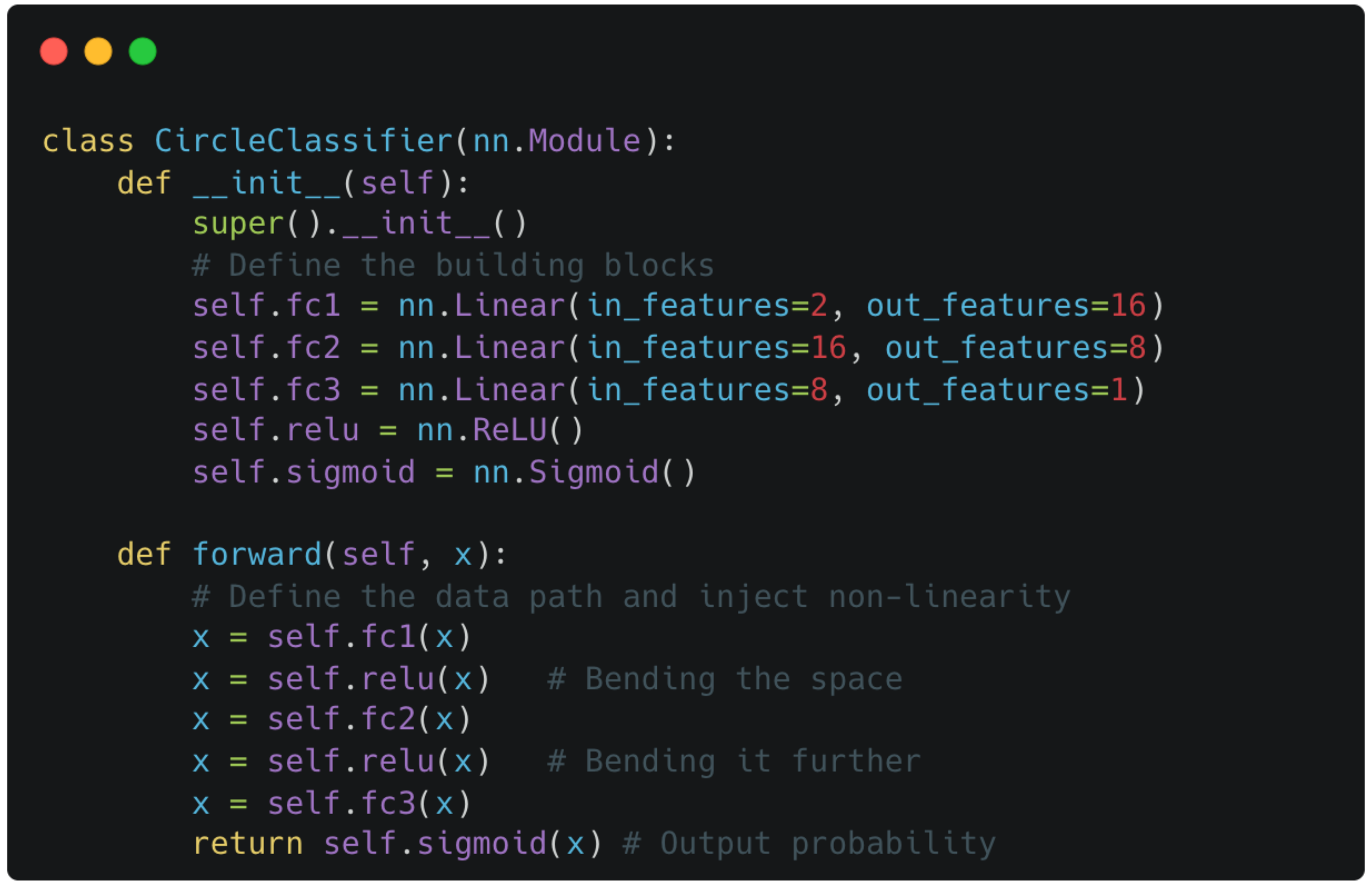
the circles problem: what happens when our data represents a complex, non-linear relationship?

example: a core system state (inner circle) surrounded by an failure state (outer ring).

the result: if we train our previous linear model on this dataset, it completely fails. it tries to draw a straight line through a circle, resulting in approximately 50% accuracy (no better than a random coin flip).



OOP PYTORCH



```python
class CircleClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        # Define the building blocks
        self.fc1 = nn.Linear(in_features=2, out_features=16)
        self.fc2 = nn.Linear(in_features=16, out_features=8)
        self.fc3 = nn.Linear(in_features=8, out_features=1)
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        # Define the data path and inject non-linearity
        x = self.fc1(x)
        x = self.relu(x)   # Bending the space
        x = self.fc2(x)
        x = self.relu(x)   # Bending it further
        x = self.fc3(x)
        return self.sigmoid(x) # Output probability
```

nn.Module pattern: we build neural networks by creating a class that inherits from torch.nn.Module.

two essential methods:

- __init__(self): this is where we define the architectural building blocks (our layers).
- forward(self, x): this is where we define the "flow" of data. we explicitly apply our relu activation functions between the linear layers here.

notice how we don't apply relu after the final layer. we use sigmoid there because we specifically want our final output to be a probability between 0 and 1.

TRAINING

upgrading the optimizer:

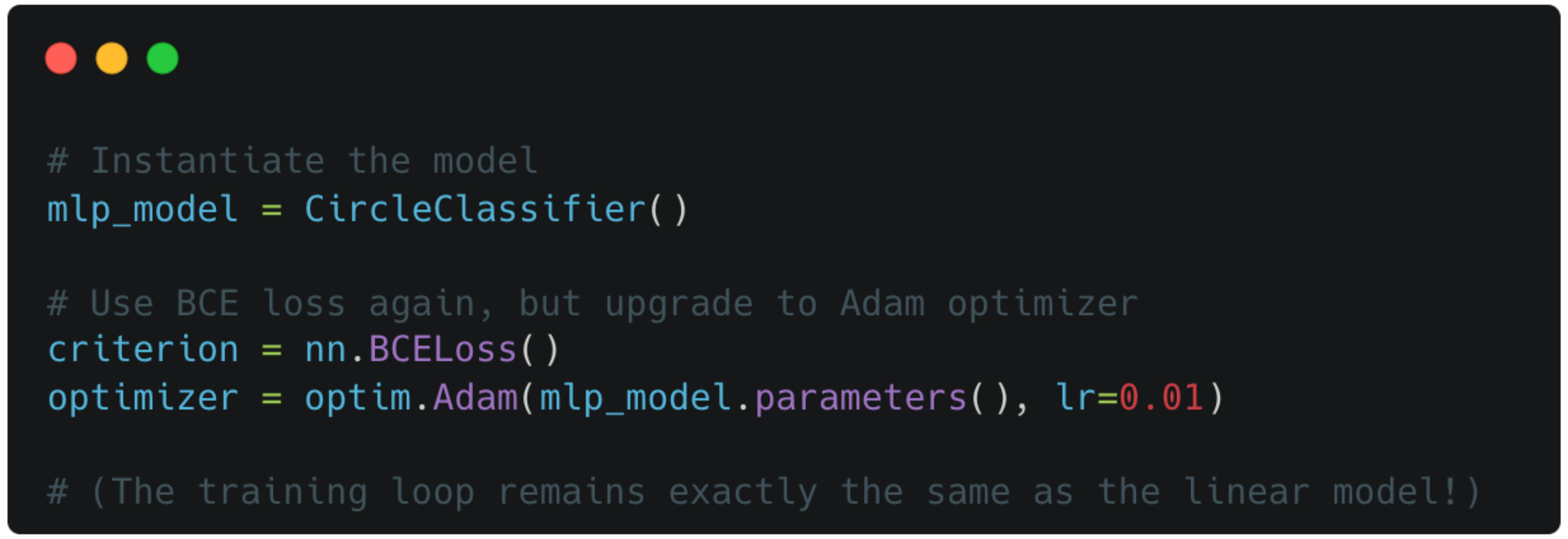
while stochastic gradient descent (sgd) works fine for single-neuron linear models, deep networks with hidden layers often require more sophisticated navigation of the loss landscape.

adam (optim.Adam):

adam dynamically adjusts the learning rate for each individual parameter. it's the industry-standard default for training deep neural networks.

the results:

by combining our mlp architecture with the adam optimizer, the model successfully learns the complex circular decision boundary.



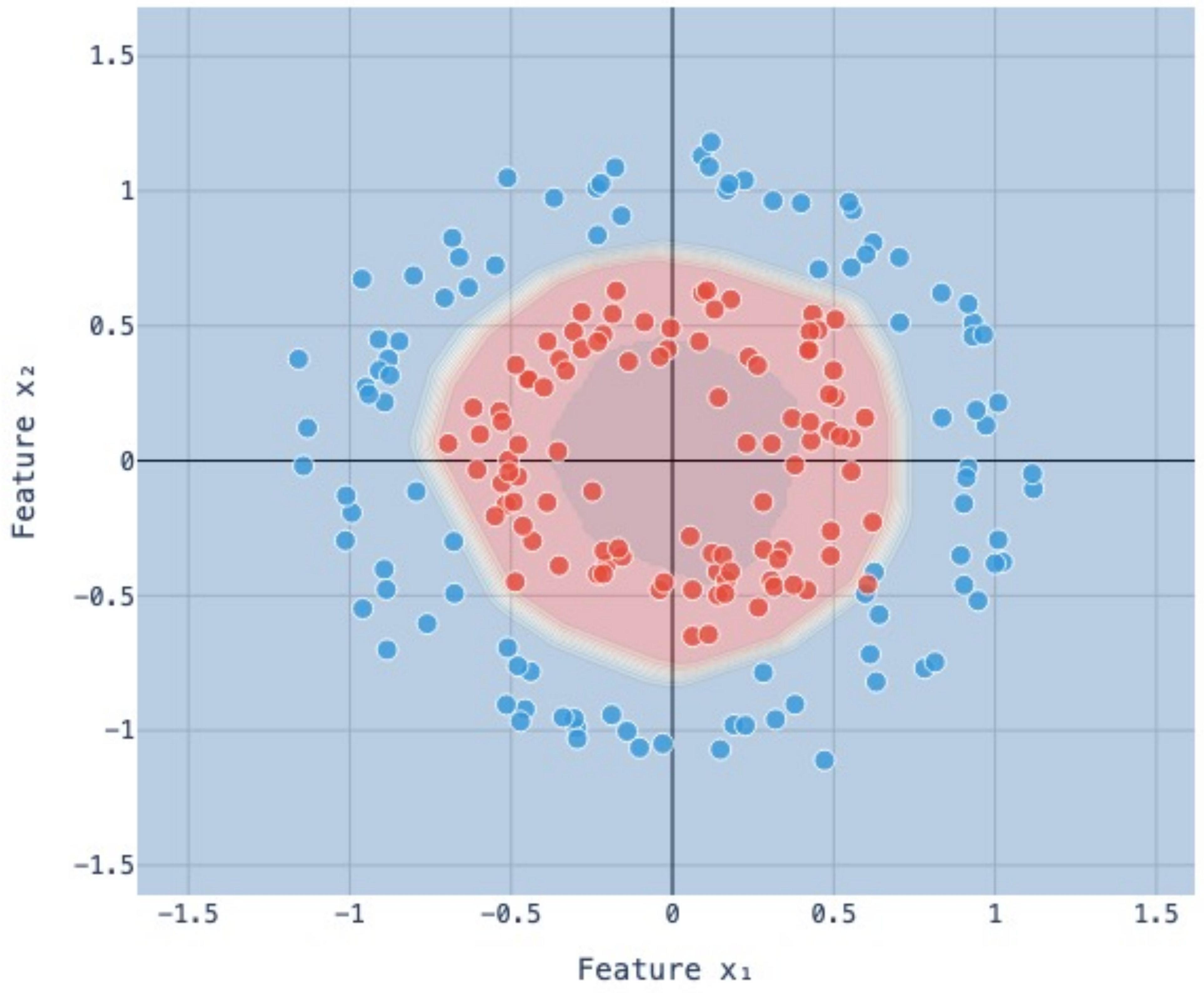
```python
# Instantiate the model
mlp_model = CircleClassifier()

# Use BCE loss again, but upgrade to Adam optimizer
criterion = nn.BCELoss()
optimizer = optim.Adam(mlp_model.parameters(), lr=0.01)

# (The training loop remains exactly the same as the linear model!)
```

DECISION BOUNDARY

because we used hidden layers and relu activations, our model is no longer drawing a rigid 1d line across our 2d data.

the probability field: the model has learned a continuous, non-linear probability field.

- **deep blue areas:** the model is highly confident the data belongs to the inner circle
- **deep red areas:** the model is highly confident the data belongs to the outer ring
- **the "coastline" (white/light colors):** this is the actual decision boundary where the model crosses the 50% probability threshold.


Feature x₂
Feature x₁