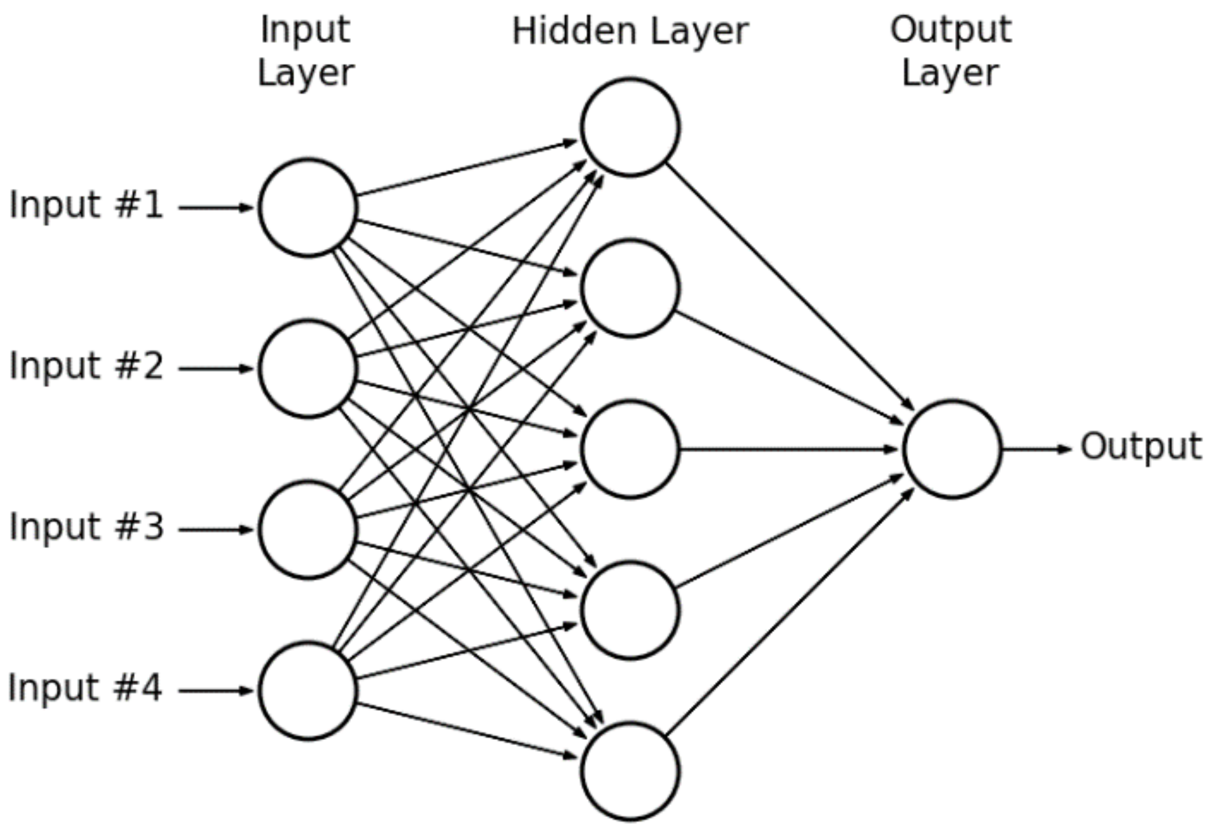
The Multilayer Perceptron

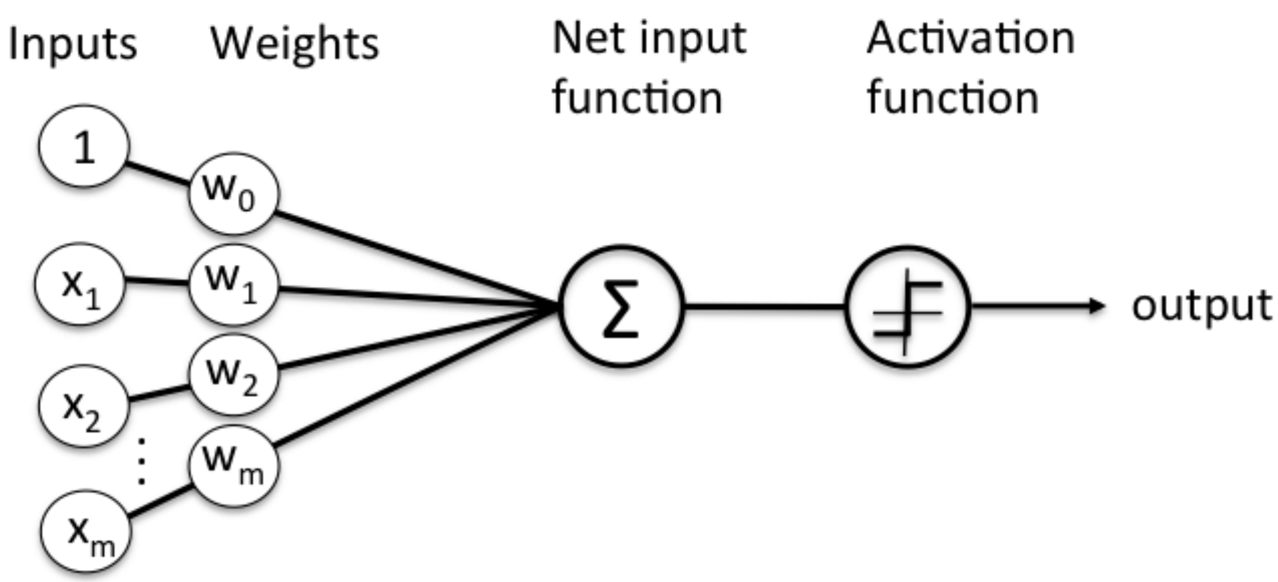
Contents



- Introduction
- Regression
- Classification
- Our own MLP framework

Introduction

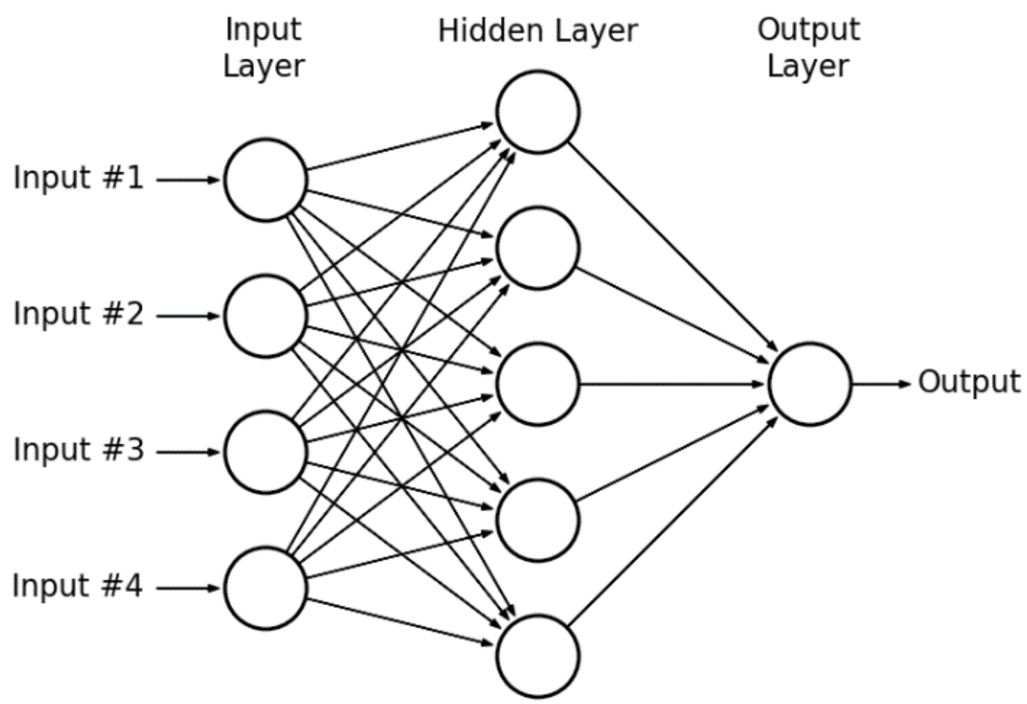
Perceptron



The Perceptron computes a weighted sum of its inputs and then applies an activation function.

$$\hat{y} = f\left(w_0 + w_1 x_1 + \ldots + w_m x_m\right)$$

Multilayer Perceptron



The Multilayer Perceptron (MLP) stacks Perceptrons in sequential layers, feeding the outputs of one layer to the inputs of the next layer. In the case of a 2 layer MLP,

$$\hat{y} = f_2\left(w_0^2 + w_1^2 h_1 + \ldots + w_{m_h}^2 h_{m_h}\right)$$

$$h = f_1\left(w_0^1 + w_1^1 x_1 + \ldots + w_m^1 x_m\right)$$

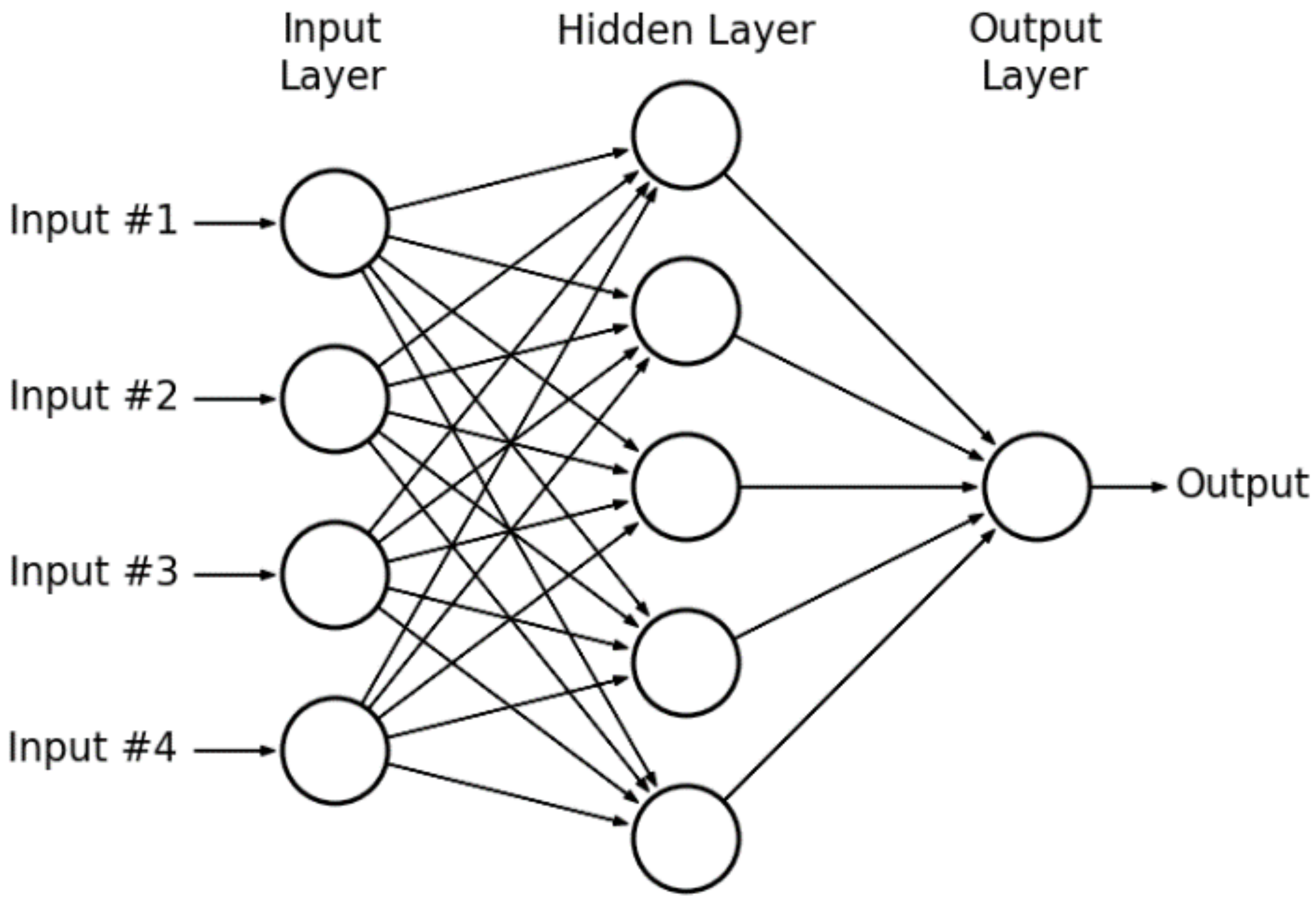
Training an MLP

Gradient descent on the perceptron

- Compute the output, $\hat{y}$
- Compute the gradient of the loss function wrt the parameters, $\dfrac{\partial l}{\partial w}$
- Update parameters, $w \leftarrow w - \eta \dfrac{\partial l}{\partial w}$
- Repeat until convergence

Training an MLP

Backpropagation

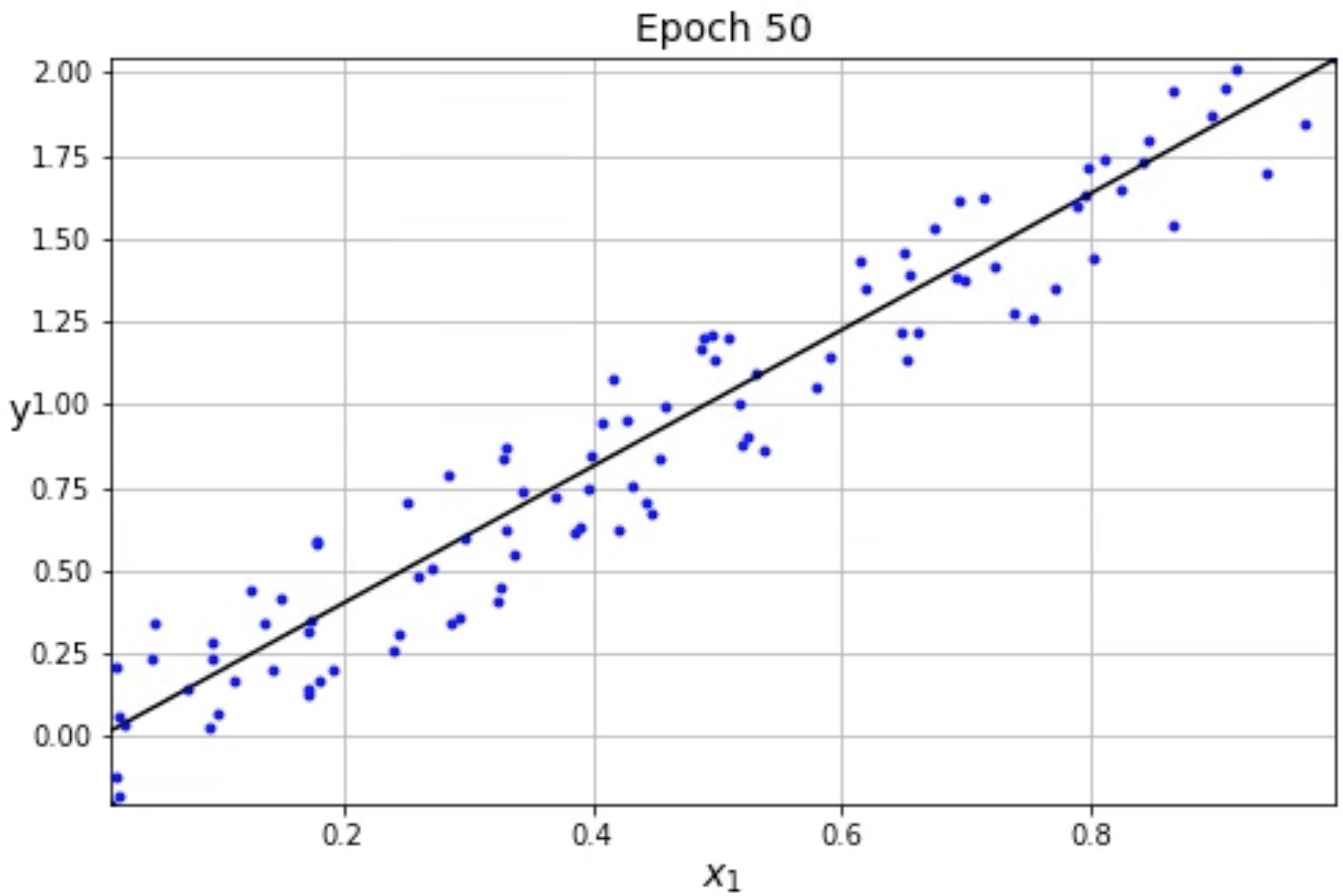


Regression

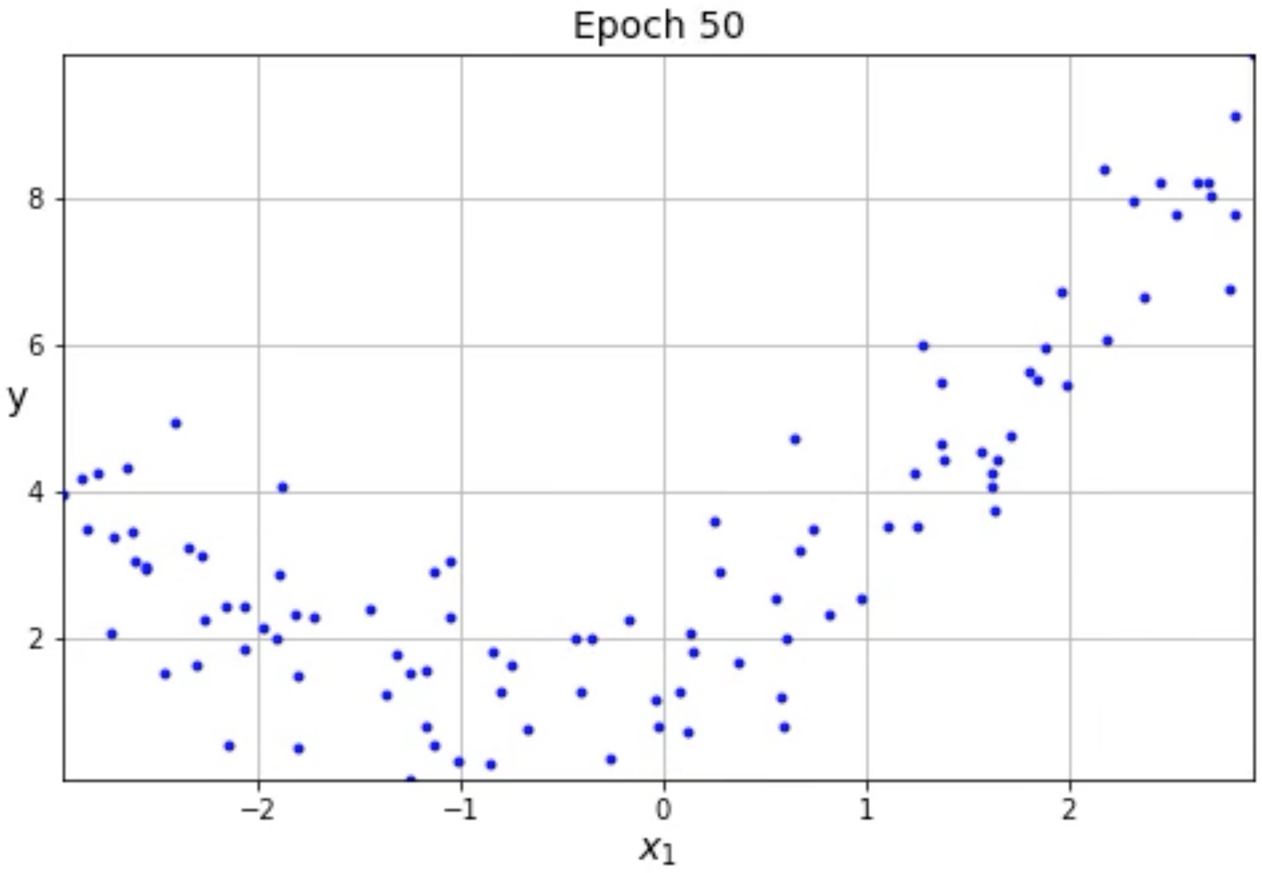
```python
def mse(output, target):
    return 0.5*(output - target)**2

def grad_mse(output, target):
    return (output - target)
```

```python
class MLP():
  def __init__(self, D_in, H, D_out):
    self.w1, self.b1 = np.random.normal(loc=0.0,
                              scale=np.sqrt(2/(D_in+H)),
                              size=(D_in, H)), np.zeros(H)
    self.w2, self.b2 = np.random.normal(loc=0.0,
                              scale=np.sqrt(2/(H+D_out)),
                              size=(H, D_out)), np.zeros(D_out)

    self.loss = mse
    self.grad_loss = grad_mse

  def __call__(self, x):
    self.h = np.dot(x, self.w1) + self.b1
    y_hat = np.dot(self.h, self.w2)  + self.b2
    return self.final_activation(y_hat)

  def final_activation(self, x):
    return x
```

```python
def fit(self, X, Y, epochs = 100, lr = 0.001):
  for e in range(epochs):
   for x, y in zip(X, Y):
      # add batch dimension
      x = x[None,:]
      y_pred = self(x)
      # loss function
      loss = self.loss(y_pred, y).mean()
      # Backprop
      # dl/dy
      dldy = self.grad_loss(y_pred, y)
      # dl/dw2 = dl/dy * dy/dw2
      grad_w2 = np.dot(self.h.T, dldy)
      grad_b2 = dldy.mean(axis=0)*self.h.shape[0]
      # dl/dh = dl/dy * dy/dh
      dldh = np.dot(dldy, self.w2.T)*self.h
      # dl/dw1 = dl/dy * dy/dh * dh/dw1
      grad_w1 = np.dot(x.T, dldh)
      grad_b1 = dldh.mean(axis=0)*x.shape[0]
      # Update (GD)
      self.w1 = self.w1 - lr * grad_w1
      self.b1 = self.b1 - lr * grad_b1
      self.w2 = self.w2 - lr * grad_w2
      self.b2 = self.b2 - lr * grad_b2
```

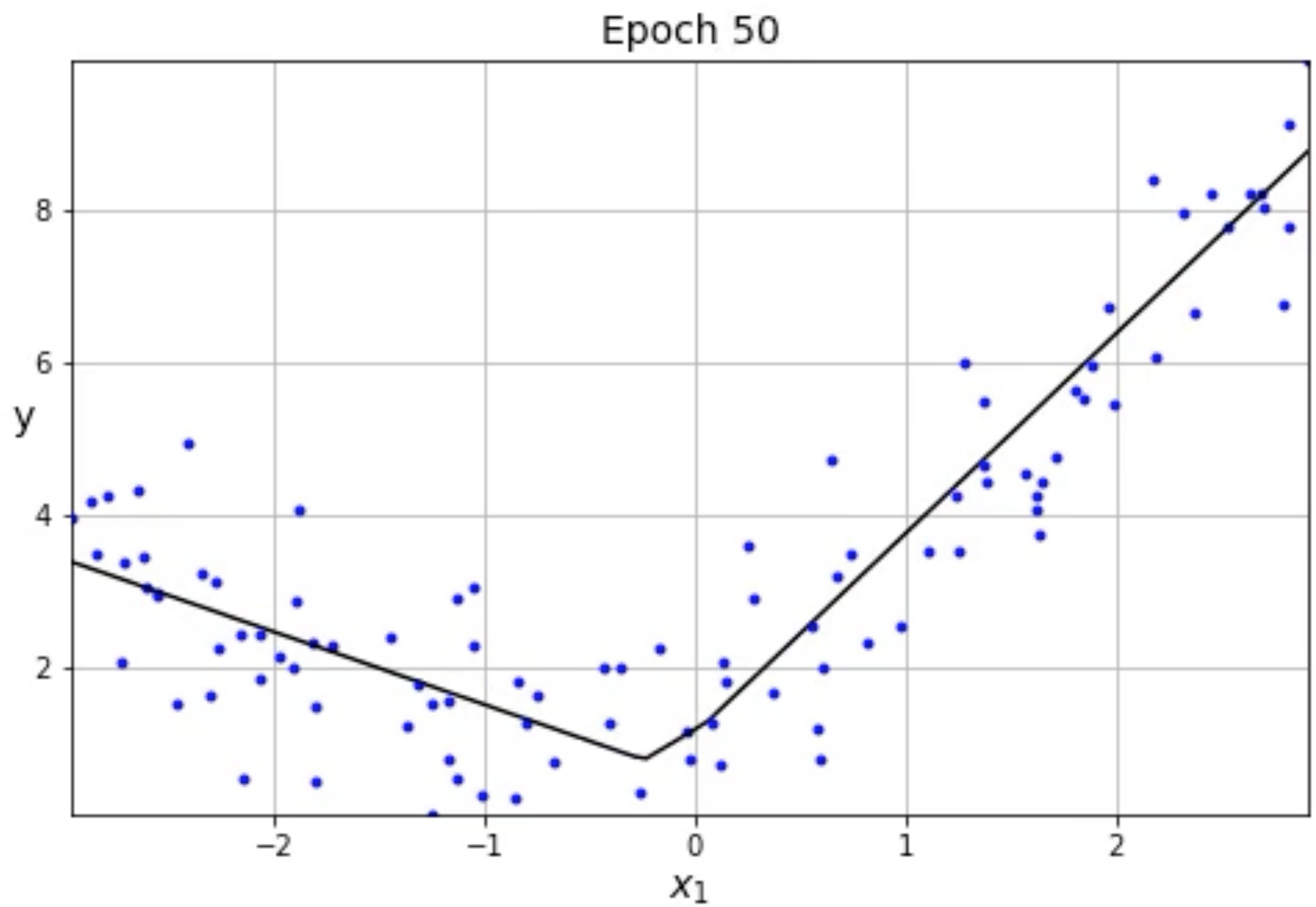

Epoch 50
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
y
0.2
0.4
0.6
0.8
x_1

Polynomial Regression



A linear combination of linear functions is a linear function. We need non linearity !

```
def relu(x):
  return np.maximum(0, x)

def reluPrime(x):
  return x > 0

class MLPrelu(MLP):
  def __call__(self, x):
    self.h = relu(np.dot(x, self.w1) + self.b1)
    y_hat = np.dot(self.h, self.w2) + self.b2
    return self.final_activation(y_hat)

  def fit(self, X, Y, epochs = 100, lr = 0.001):
    for e in range(epochs):
     for x, y in zip(X, Y):
        x = x[None,:]
        y_pred = self(x)
        loss = self.loss(y_pred, y).mean()
        # Backprop
        dldy = self.grad_loss(y_pred, y)
        grad_w2 = np.dot(self.h.T, dldy)
        grad_b2 = dldy
        dldh = np.dot(dldy, self.w2.T)*reluPrime(self.h)
        grad_w1 = np.dot(x.T, dldh)
        grad_b1 = dldh
        # Update (GD)
        self.w1 = self.w1 - lr * grad_w1
        self.b1 = self.b1 - lr * grad_b1
        self.w2 = self.w2 - lr * grad_w2
        self.b2 = self.b2 - lr * grad_b2
```


Epoch 50
y
8
6
4
2
−2
−1
0
1
2
x_1

Binary Classification