Neural networks

for regression

Abhinav Pradeep

October 22, 2024

Function approximation

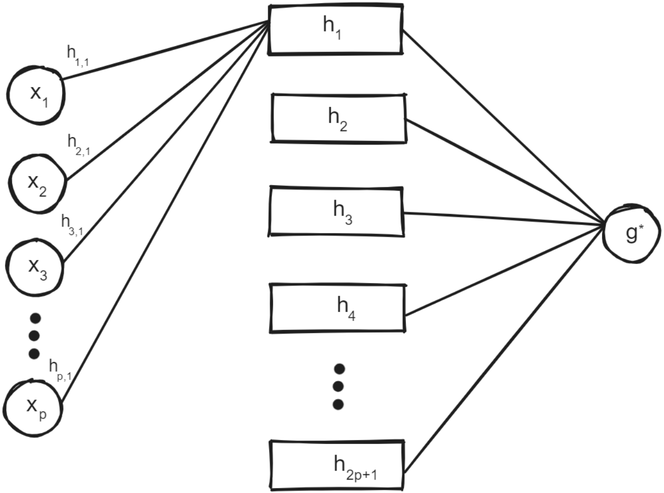
Theorem 9.1 from the textbook:

For $p \geq 2$ every continuous function $g^*(\boldsymbol{x}) : [0,1]^p \mapsto \mathbb{R}$ can be written as:

$$g^*(\boldsymbol{x}) = \sum_{j=1}^{2p+1} h_j \left(\sum_{i=1}^{p} h_{ij}(x_i) \right)$$

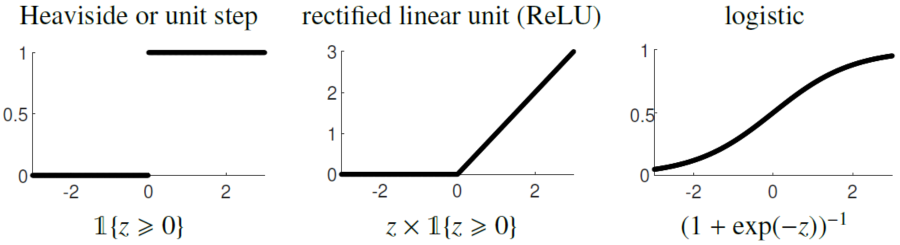
Where $\{h_j, h_{ij}\}$ are continuous univariate functions. So every continuous multivariate function can be written as a sum of $p(2p+1)$ univariate functions.

Function approximation

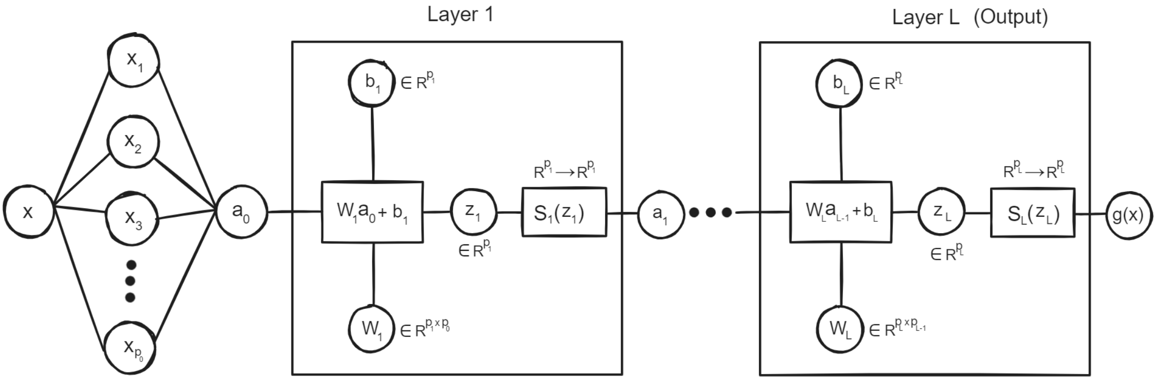


Function approximation

We do not know the set of $p(2p+1)$ functions $\{h_j, h_{ij}\}$ and they are likely non-linear. We can replace the by a larger set of functions. Call these activation functions $S(z)$. Examples are (from the textbook):

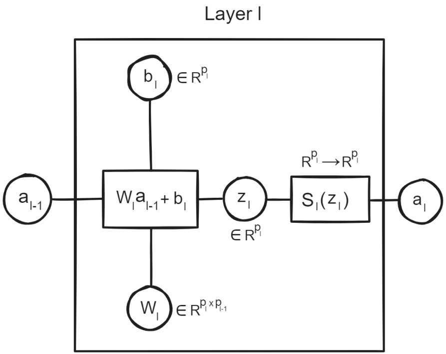


Feed-forward networks



Feed-forward networks

The activation function $\boldsymbol{a}_l = \boldsymbol{S}_l(\boldsymbol{z}_l)$ for the hidden layers $(l = 1, 2 \ldots L-1)$ is $\boldsymbol{S}_l(\boldsymbol{z}) = \begin{bmatrix} S(z_1) & S(z_2) & \ldots & S(z_{p_l}) \end{bmatrix}$. The output layers activation $\boldsymbol{S}_L$ function is more general and depends on the goal.



Feed-forward propagation

Algorithm 1 Feed-forward propagation

Input: $\boldsymbol{x} \in \mathbb{R}^{p_0}$, $\{\mathbf{W}_l \in \mathbb{R}^{p_l \times p_{l-1}}\}$, $\{\mathbf{b}_l \in \mathbb{R}^{p_l}\}$, $\{\mathbf{S}_l\}$

Output: $\boldsymbol{g}(\mathbf{x})$

```
a_0 ← x
for l = 1 to L do
    z_l ← W_l a_{l-1} + b_l
    a_l ← S_l(z_l)
end for
return g(x) ← a_L
```

$\mathbf{a}_0 \leftarrow \mathbf{x}$

for $l = 1$ to L **do**

$\quad \boldsymbol{z}_l \leftarrow \boldsymbol{W}_l \boldsymbol{a}_{l-1} + \boldsymbol{b}_l$

$\quad \boldsymbol{a}_l \leftarrow \boldsymbol{S}_l(\boldsymbol{z}_l)$

end for

return $\boldsymbol{g}(\boldsymbol{x}) \leftarrow \boldsymbol{a}_L$

Back propagation

For steepest descent type algorithms we wish to compute the gradient of training loss. First we gather all the parameters into a vector as:

$$\boldsymbol{\theta} = \{\boldsymbol{W}_l, \boldsymbol{b}_l\}$$

Where $\dim \boldsymbol{\theta} = \sum_{l=1}^{L} p_{l-1} p_l + \sum_{l=1}^{L} p_l$

From here we have that training loss for $\tau = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^{n}$ is:

$$\ell_\tau(g(\cdot \mid \boldsymbol{\theta})) := \frac{1}{n} \sum_{i=1}^{n} \text{Loss}(\boldsymbol{y}_i, g(\boldsymbol{x}_i \mid \boldsymbol{\theta}))$$

Back propagation

To make notation simpler take:

$$C_i(\boldsymbol{\theta}) = \text{Loss}(\boldsymbol{y}_i, g(\boldsymbol{x}_i \mid \boldsymbol{\theta}))$$

Hence we have:

$$\ell_\tau(g(\cdot \mid \boldsymbol{\theta})) = \frac{1}{n}\sum_{i=1}^{n} C_i(\boldsymbol{\theta})$$

We now wish to find $\nabla_{\boldsymbol{\theta}}\ell_\tau(g(\cdot \mid \boldsymbol{\theta}))$. Denote $\boldsymbol{S}_l'(\boldsymbol{z}_l) = \begin{bmatrix} S'(z_1) & S'(z_2) & \dots & S'(z_{p_l}) \end{bmatrix}$. Define:

$$\boldsymbol{D}_l := \text{Diag}\left(\boldsymbol{S}_l'(\boldsymbol{z}_l)\right)$$

We then can prove that:

$$\frac{\partial C}{\partial \boldsymbol{W}_l} = \boldsymbol{\delta}_l \boldsymbol{a}_{l-1}^T$$

and

$$\nabla_{\boldsymbol{b}_l} C = \boldsymbol{\delta}_l$$

Where, we have the recursion:

$$\boldsymbol{\delta}_{l-1} = \boldsymbol{D}_{l-1} \boldsymbol{W}_l^T \boldsymbol{\delta}_l \implies \boldsymbol{\delta}_{l-1} = \mathrm{Diag}\left(\boldsymbol{S}_l'(\boldsymbol{z}_l)\right) \odot \boldsymbol{W}_l^T \boldsymbol{\delta}_l$$

With base case:

$$\boldsymbol{\delta}_L = \frac{\partial \boldsymbol{S}_L}{\partial \boldsymbol{z}_L} \nabla_{\boldsymbol{g}} C$$

Back propagation

Hence we can now find $\nabla_{\boldsymbol{\theta}} C$ as:

Algorithm 2 Computing gradient of cost

Input: $\boldsymbol{x} \in \mathbb{R}^{p_0}$, corresponding $\boldsymbol{y}$, $\{\mathbf{W}_l \in \mathbb{R}^{p_l \times p_{l-1}}\}$, $\{\mathbf{b}_l \in \mathbb{R}^{p_l}\}$, $\{\mathbf{S}_l\}$
Output: $\nabla_{\boldsymbol{\theta}} C$

1: $\{\boldsymbol{a}_l\}, \{\boldsymbol{z}_l\} \leftarrow$ Algorithm 1 ▷ Forward propagation
2: $\boldsymbol{\delta}_L \leftarrow \frac{\partial \boldsymbol{S}_L}{\partial \boldsymbol{z}_L} \nabla_{\boldsymbol{g}} C$ ▷ Backward propagation
3: $\boldsymbol{z}_0 \leftarrow \mathbf{0}$
4: **for** $l = L$ bacwards to 1 **do**
5: $\quad \nabla_{\boldsymbol{b}_l} C \leftarrow \boldsymbol{\delta}_l$
6: $\quad \frac{\partial C}{\partial \boldsymbol{W}_l} \leftarrow \boldsymbol{\delta}_l \boldsymbol{a}_{l-1}^T$
7: $\quad \boldsymbol{\delta}_{l-1} \leftarrow \text{Diag}\left(\boldsymbol{S}_l'(\boldsymbol{z}_l)\right) \odot \boldsymbol{W}_l^T \boldsymbol{\delta}_l$
8: **end for**
9: Collect $\{\nabla_{\boldsymbol{b}_l} C\}$ and $\{\frac{\partial C}{\partial \boldsymbol{W}_l}\}$ into $\nabla_{\boldsymbol{\theta}} C$
10: **return** $\nabla_{\boldsymbol{\theta}} C$

Training

Hence we can find $\nabla_{\boldsymbol{\theta}}\ell_\tau(g(\cdot \mid \boldsymbol{\theta}))$ by simply looping over Algorithm 2 for each pair in $\tau = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^n$ adding up the obtained derivatives for each C_i. Now we can work on finding $\boldsymbol{\theta}^* := \mathrm{argmin}_{\boldsymbol{\theta}}\, \ell_\tau(g(\cdot \mid \boldsymbol{\theta}))$. For brevity $\nabla_{\boldsymbol{\theta}}\ell_\tau := \nabla_{\boldsymbol{\theta}}\ell_\tau(g(\cdot \mid \boldsymbol{\theta}))$. We take a steepest descent method of the form:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha_t \nabla_{\boldsymbol{\theta}_t}\ell_\tau$$

We wish to use α_t to control step size: higher curvature means we should be have slower rate of descent whereas low curvature means we should have higher rate of descent. Normally we would take: $\alpha_t = (H_{\boldsymbol{\theta}}\ell_\tau)^{-1}$. However, this is computationally expensive. Instead we take the short Barzilai-Borwein formula ($\boldsymbol{\delta} = \boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t$ and $\boldsymbol{s} = \nabla_{\boldsymbol{\theta}_{t+1}}\ell_\tau - \nabla_{\boldsymbol{\theta}_t}\ell_\tau$):

$$\alpha_t = \frac{\boldsymbol{\delta}^T \boldsymbol{s}}{\|\boldsymbol{s}\|}$$

Training

Algorithm 3 Gradient descent training

Input: $\mathcal{T} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^{n}$, initial guess θ_0 in terms of $\{\mathbf{W}_l \in \mathbb{R}^{p_l \times p_{l-1}}\}$ and $\{\mathbf{b}_l \in \mathbb{R}^{p_l}\}$, $\{\mathbf{S}_l\}$

Output: $\text{argmin}_{\boldsymbol{\theta}} \, \ell_{\mathcal{T}}(g(\cdot \mid \boldsymbol{\theta}))$

$t \leftarrow 1 \quad \boldsymbol{\delta} \leftarrow 0.1 \cdot \mathbf{1} \quad \nabla_{\boldsymbol{\theta}_{t-1}} \ell_{\mathcal{T}} \leftarrow \mathbf{0} \quad \alpha \leftarrow 0.1$ ▷ Initialise parameters suitably

while stopping condition is not met **do**

  $\nabla_{\boldsymbol{\theta}_t} \ell_{\mathcal{T}} \leftarrow$ BackPropogation

  $\boldsymbol{s} \leftarrow \nabla_{\boldsymbol{\theta}_t} \ell_{\mathcal{T}} - \nabla_{\boldsymbol{\theta}_{t-1}} \ell_{\mathcal{T}}$

  if $\boldsymbol{\delta}^T \boldsymbol{s} > 0$ **then**

    $\alpha \leftarrow \frac{\boldsymbol{\delta}^T \boldsymbol{s}}{\|\boldsymbol{s}\|}$

  else

    $\alpha \leftarrow \alpha \cdot 2$

  end if

  $\boldsymbol{\delta} \leftarrow -\alpha \nabla_{\boldsymbol{\theta}_t} \ell_{\mathcal{T}} \quad \boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \boldsymbol{\delta} \quad t \leftarrow t + 1$

end while

return $\boldsymbol{\theta}_t$

Training

Now we look at stochastic gradient descent. Consider the case that the training set $\tau = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^n$. In the case that n is very large, the training will be expensive. Now consider a discrete R.V K such that $\forall k = 1, 2, \ldots n$

$$\mathbb{P}\left[K = k\right] = \frac{1}{n}$$

We hence have that, by definition,

$$\ell_\tau(g(\cdot \mid \boldsymbol{\theta})) := \frac{1}{n}\sum_{i=1}^{n} \text{Loss}(\boldsymbol{y}_i, g(\boldsymbol{x}_i \mid \boldsymbol{\theta})) = \mathbb{E}\left[\text{Loss}(\boldsymbol{y}_K, g(\boldsymbol{x}_K \mid \boldsymbol{\theta}))\right]$$

Hence, for some $N \ll n$ i.i.d $K_i \sim K$ we have the Monte Carlo estimate:

$$\hat{\ell}_\tau(g(\cdot \mid \boldsymbol{\theta})) := \frac{1}{N}\sum_{i=1}^{N} \text{Loss}(\boldsymbol{y}_{K_i}, g(\boldsymbol{x}_{K_i} \mid \boldsymbol{\theta}))$$

Polynomial Regression

We consider an architecture of:

$$\begin{bmatrix} p_0 & p_1 & p_2 & p_3 \end{bmatrix} = \begin{bmatrix} 1 & 20 & 20 & 1 \end{bmatrix}$$

This results in:

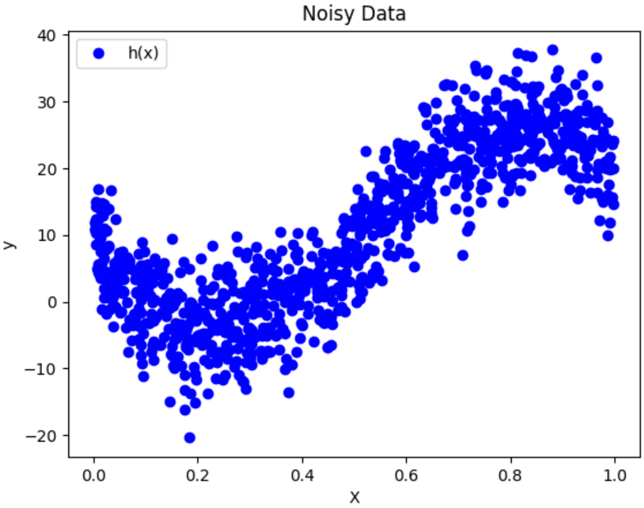
Parameters between p_0 and p_1: $p_0 \times p_1 = 20$ weights and $p_1 = 20$ biases.
Parameters between p_1 and p_2: $p_1 \times p_2 = 400$ weights and $p_2 = 20$ biases.
Parameters between p_2 and p_3: $p_2 \times p_3 = 20$ weights and $p_3 = 1$ bias.

Hence, $\dim \boldsymbol{\theta} = 481$. Moreover, we can use a ReLU activation function.

First we generate data:

```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(10)

def GenerateData(NumberOfPoints : int):
    X = np.random.uniform(0, 1, NumberOfPoints)
    h = lambda x: 10 - 140*x + 400*(x**2) - 250*(x**3)
    y = np.random.normal(h(X),np.sqrt(25))
    return (X,y)

(X,y) = GenerateData(1000)

X = X.reshape(-1,1)
y = y.reshape(-1,1)
```

Polynomial Regression



Polynomial Regression

Now we can start writing the neural network. Consider the activation function:

```
def ReLU(z, l : int):
    # If last layer, dont apply ReLU
    if l == 3:
        return (z, np.ones_like(z))
    # Returns (S(z), S'(z)) = (z 1_{z >= 0}, 1_{z >= 0}) element-wise
    else:
        # Converted to bool we have True = 1.0 and False = 0.0.
        # Derivative is diagonal rep.
        return (np.maximum(0,z), np.array(z > 0, dtype=float))
```

Polynomial Regression

Now we initialize the weights as:

```
def Initialise(p):
    W = [None] * len(p)
    b = [None] * len(p)

    for l in range(1,len(p)):
        # Weight matrix has to be of dimension p[l] \times p[l-1]
        W[l] = np.random.randn(p[l], p[l-1])
        # Bias b is a vector of size p[l]
        b[l] = np.random.randn(p[l], 1)
    return W, b
```

We can write feed-forward propagation as:

```python
def FeedForward(x,W,b):
    # The Jacobian is again stored as diagonal matrix here
    a, z, DdS_dz = [None] * 4, [None] * 4, [None] * 4
    a[0] = x.reshape(-1,1)
    # Processing like below needs to be done from layer 1 to 3
    for l in range(1,4):
        z[l] = W[l] @ a[l-1] + b[l]
        a[l], DdS_dz[l] = ReLU(z[l],l)
    return a, z, DdS_dz
```

Now we can focus on training. Taking squared error loss we write the function as:

```python
def Loss(y,g):
    return (g - y)**2, 2 * (g - y)
```

Like with ReLU this also returns the derivative.

Polynomial Regerssion

```python
def BackPropagation(W,b,X,y):
    n = len(y)
    Delta = [None]*4
    dC_db, dC_dW = [None]*4, [None]*4
    LossIncurred = 0
    # For each (x,y) in the training
    # set:

    for i in range(n):
        a, z, DdS_dz =
            FeedForward(X[i,:].T, W, b)
        # Cost and its gradient at last
        # layer
        C , dC_dg = Loss(y[i], a[3])
        # Update loss incurred
        LossIncurred += C/n
        # Is dot product here and is
        # equivalent to full mmult as
        # dS_dz is diagonal
        Delta[3] = DdS_dz @ dC_dg
```

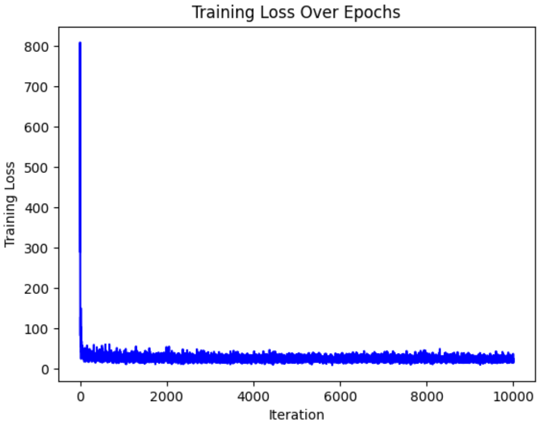
```python
        # Backwards propagation
        for l in range(3,0,-1):
            dCi_dbl = Delta[l]
            dCi_dWl = Delta[l] @ a[l-1].T
            # Accumulate
            if dC_db[l] is None:
                dC_db[l] = dCi_dbl / n
            else:
                dC_db[l] += dCi_dbl / n
            if dC_dW[l] is None:
                dC_dW[l] = dCi_dWl / n
            else:
                dC_dW[l] += dCi_dWl / n
            # z[0] and DdS_dz[0] are None
            # (FeedForward dosent calculate)
            if l > 1:
                Delta[l-1] = DdS_dz[l-1] *
                    (W[l].T @ Delta[l])

        return dC_dW , dC_db , LossIncurred
```

Hence we now have everything to begin training. As done in the textbook, we can now do gradient descent. Here we do:

```
for epoch in range(1, num_epochs + 1):
    batch_idx = np.random.choice(n, batch_size)
    batch_X = X[batch_idx].reshape(-1, 1)
    batch_y = y[batch_idx].reshape(-1, 1)

    dc_dW, dc_db, batch_loss = BackPropagation(W, b, batch_X, batch_y)
    loss_arr.append(batch_loss.flatten()[0])

    d_beta = list2vec(dc_dW, dc_db)
    beta = list2vec(W, b)

    beta = beta - lr * d_beta

    W, b = vec2list(beta, p)
```

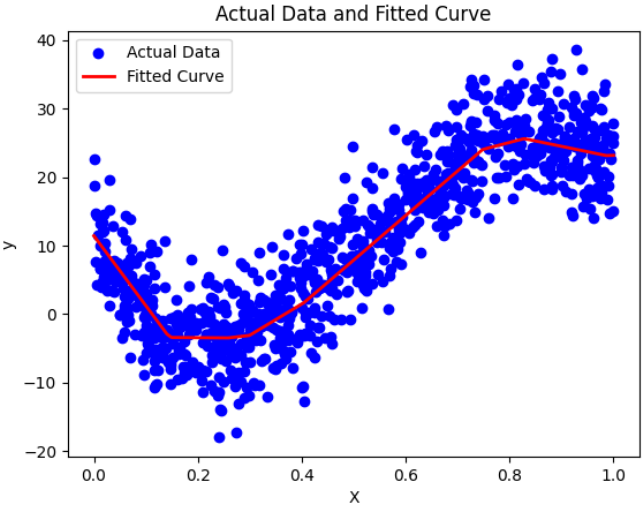
Taking batch˙size = 50

Result




```
epoch | batch loss
-------------------------
1: 291.0456945346262
1000: 29.10653060335138
2000: 27.90970112756271б
3000: 19.16270342971836б
4000: 24.478730987696775
5000: 20.77197762268555
6000: 17.870963762181447
7000: 21.59237202881953
8000: 18.538087981621157
9000: 17.72078710219764
10000: 22.82676949209595
```

Result



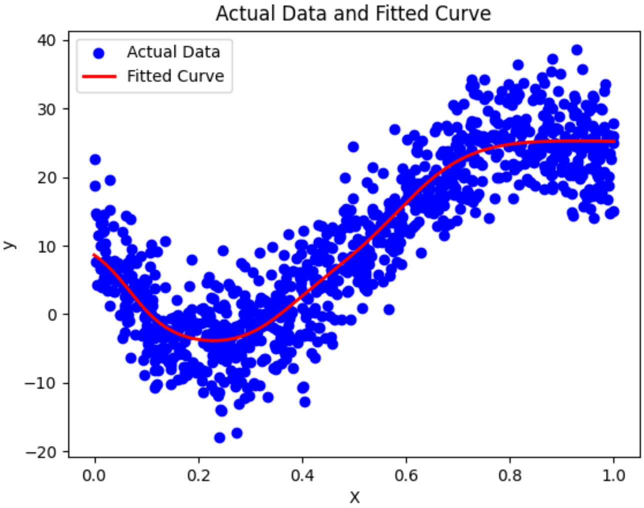
Actual Data and Fitted Curve
Actual Data
Fitted Curve
y
X
40
30
20
10
0
−10
−20
0.0
0.2
0.4
0.6
0.8
1.0

Bonus: Using sigmoid activation

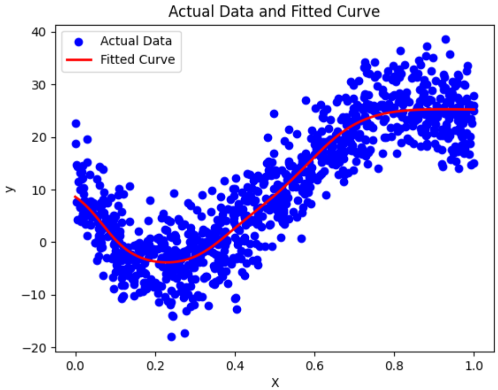
To do so you can just use this function instead:

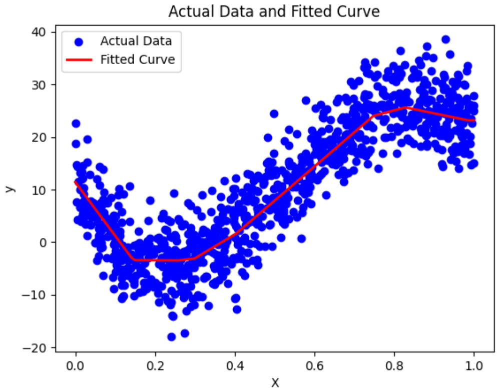
```
def Sigmoid(z, l : int):
    if l == 3:
        return (z, np.ones_like(z))
    else:
        S = lambda z: 1/(1+np.exp(-z))
        Sprime = lambda z: (np.exp(z))/((1+np.exp(z))**2)
        return (S(z), Sprime(z))
```

Bonus: Using sigmoid activation



Bonus: Using sigmoid activation



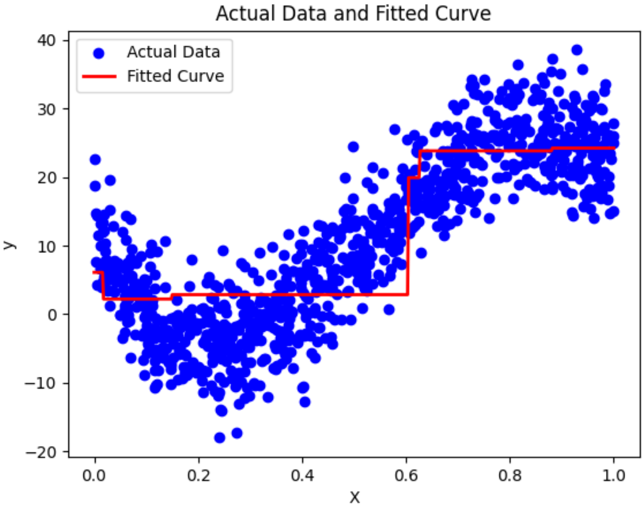


Bonus: Using indicator activation

To do so you can just use this function instead:

```python
def Indicator(z, l : int):
    if l == 3:
        return (z, np.ones_like(z))
    else:
        return (np.array(z > 0, dtype=float), np.zeros_like(z))
```

Bonus: Using indicator activation



Bonus: Using indicator activation