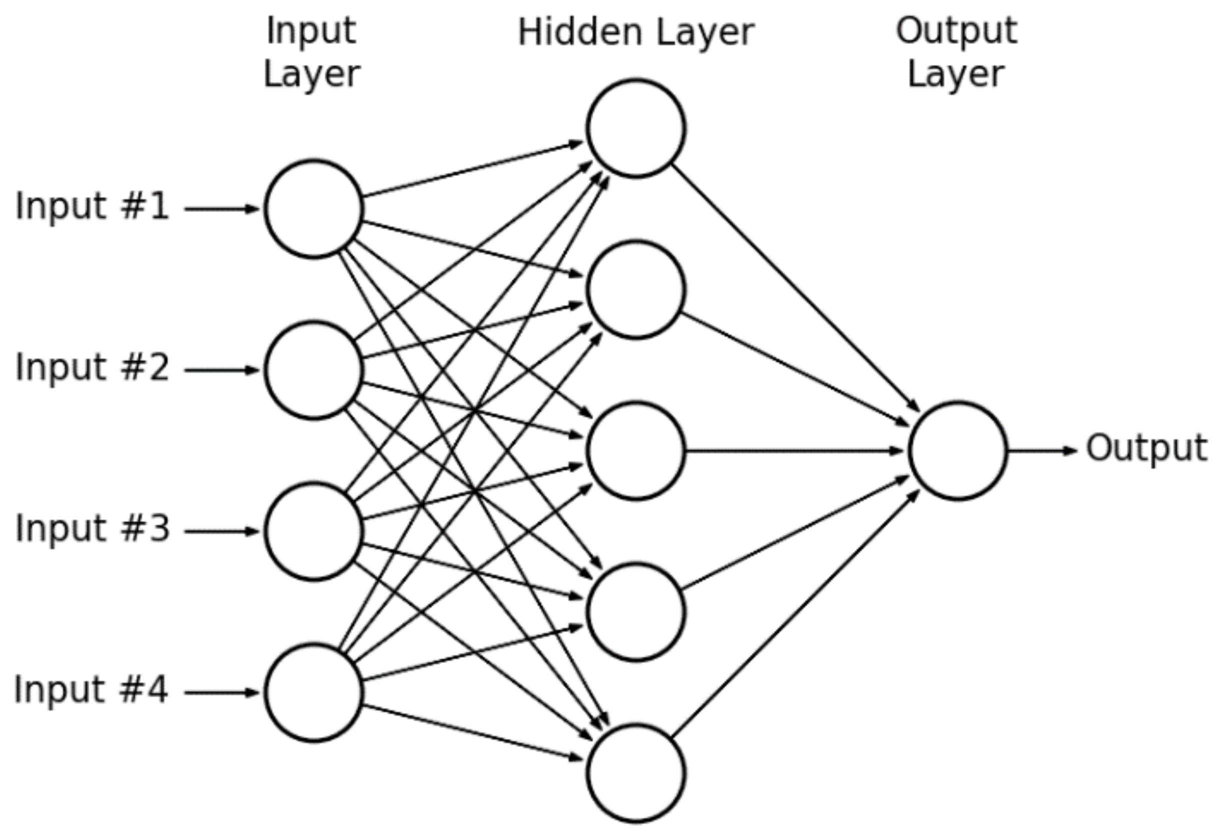
The Multilayer Perceptron

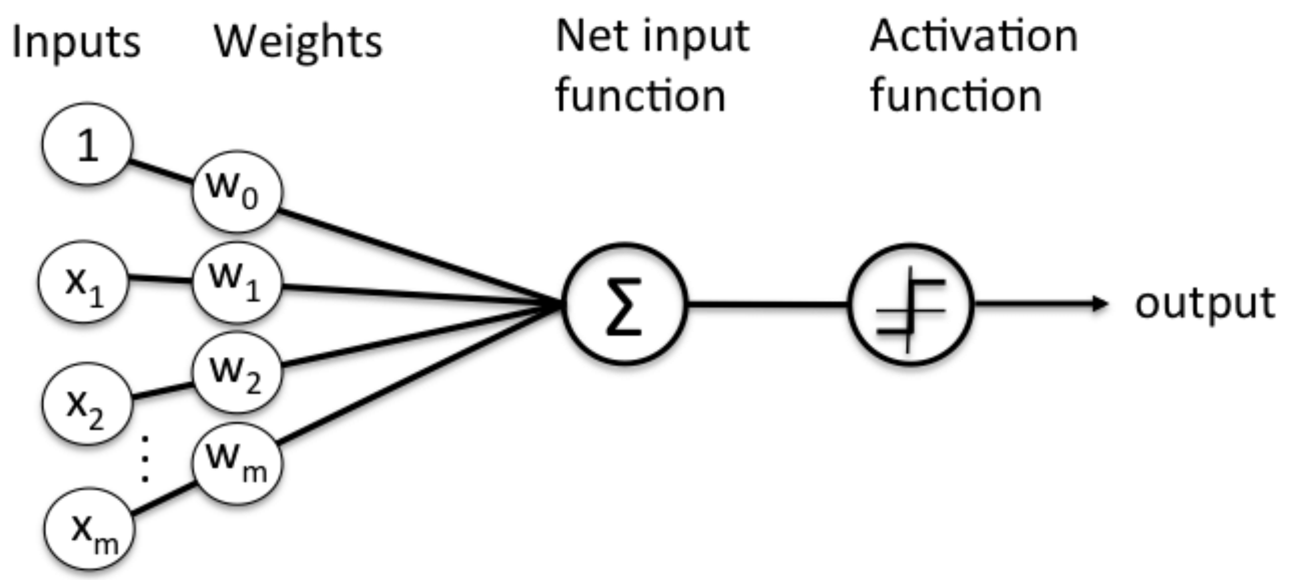
Contents

- Introduction
- Regression
- Classification
- Our own MLP framework



Introduction

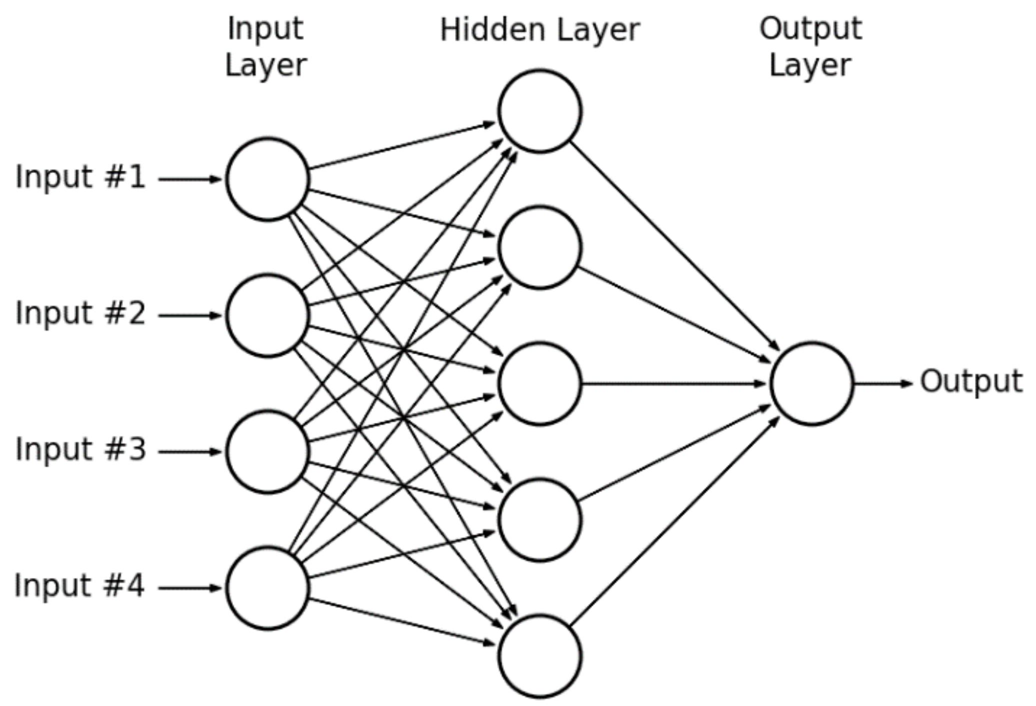
Perceptron



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,

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

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

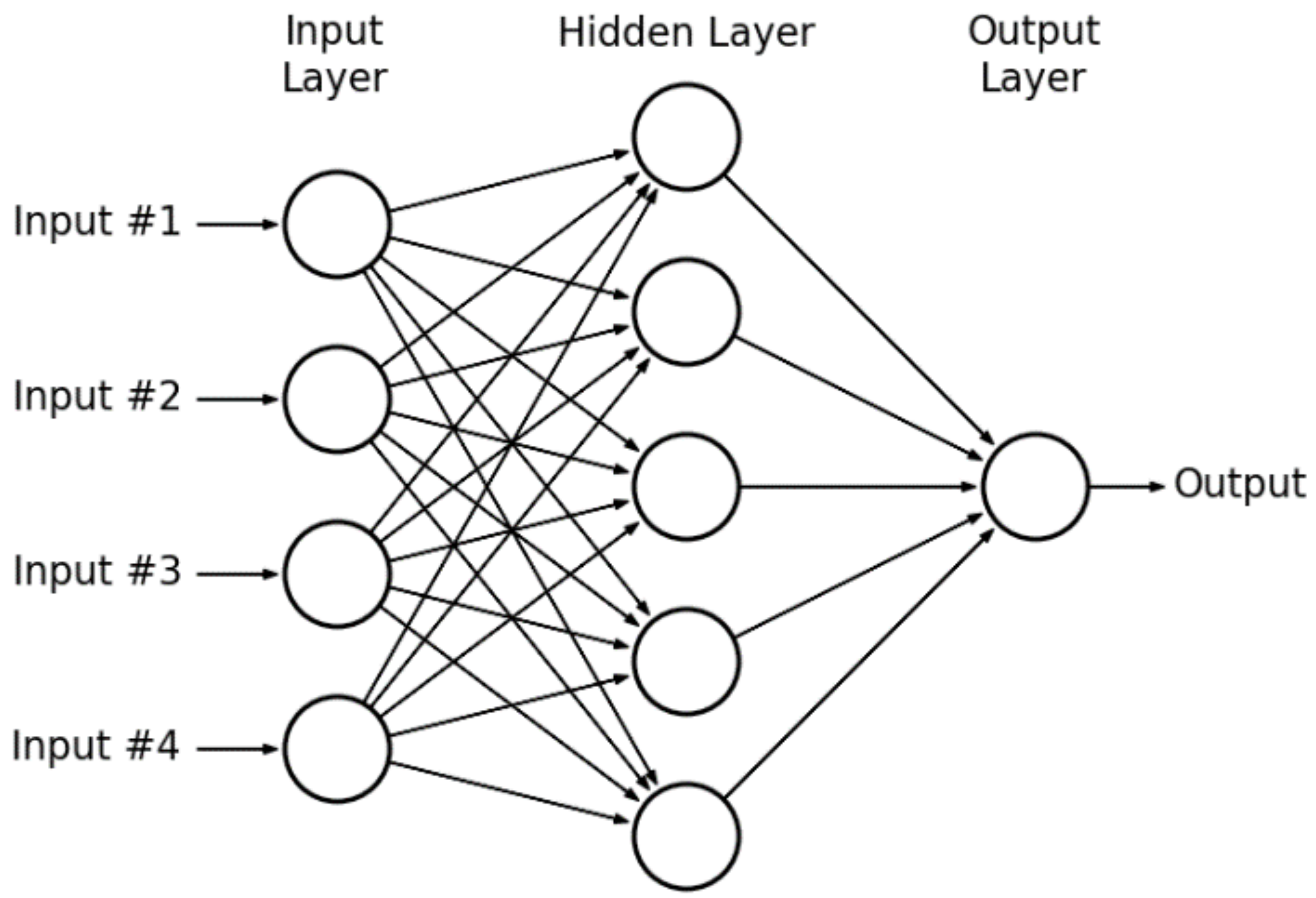
Training an MLP

Gradient descent on the perceptron

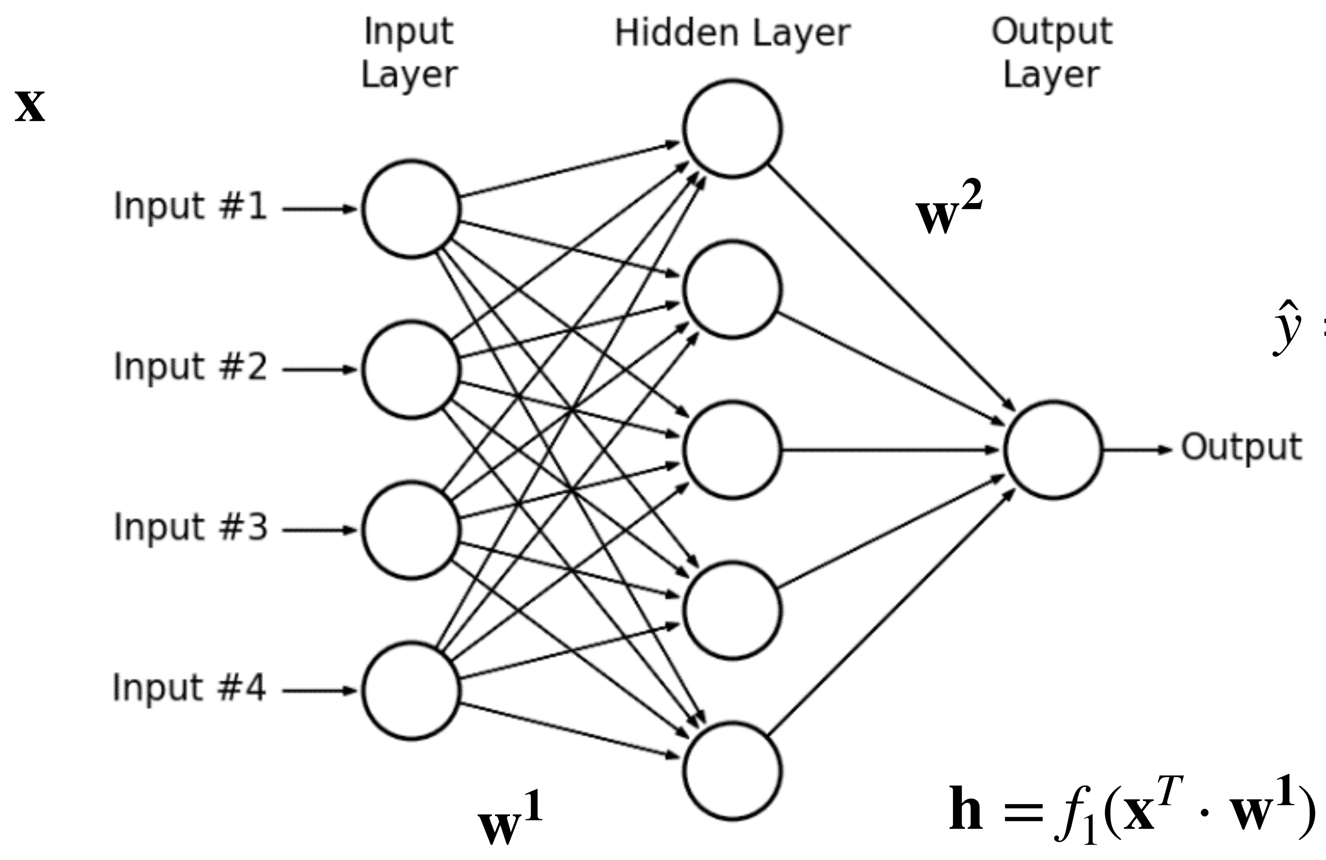
- Compute the output, $\hat{y}$
- Compute the gradient of the loss function w.r.t the parameters, $\partial l / \partial w$
- Update parameters, $w \leftarrow w - \eta \dfrac{\partial l}{\partial w}$
- Repeat until convergence

Training an MLP

Backpropagation



Backpropagation



$$l = \frac{1}{2}(\hat{y} - y)^2$$

$$\hat{y} = \mathbf{h}^T \cdot \mathbf{w^2}$$

$$\mathbf{h} = f_1(\mathbf{x}^T \cdot \mathbf{w^1})$$

Assuming MSE loss function and linear activation function at the output

$$\frac{\partial l}{\partial w^2} = \frac{\partial l}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w^2} = (\hat{y} - y)h$$

$$\frac{\partial l}{\partial w^1} = \frac{\partial l}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h} \frac{\partial h}{\partial w^1} = (\hat{y} - y)w^2 f_1'(w^1 x)x$$

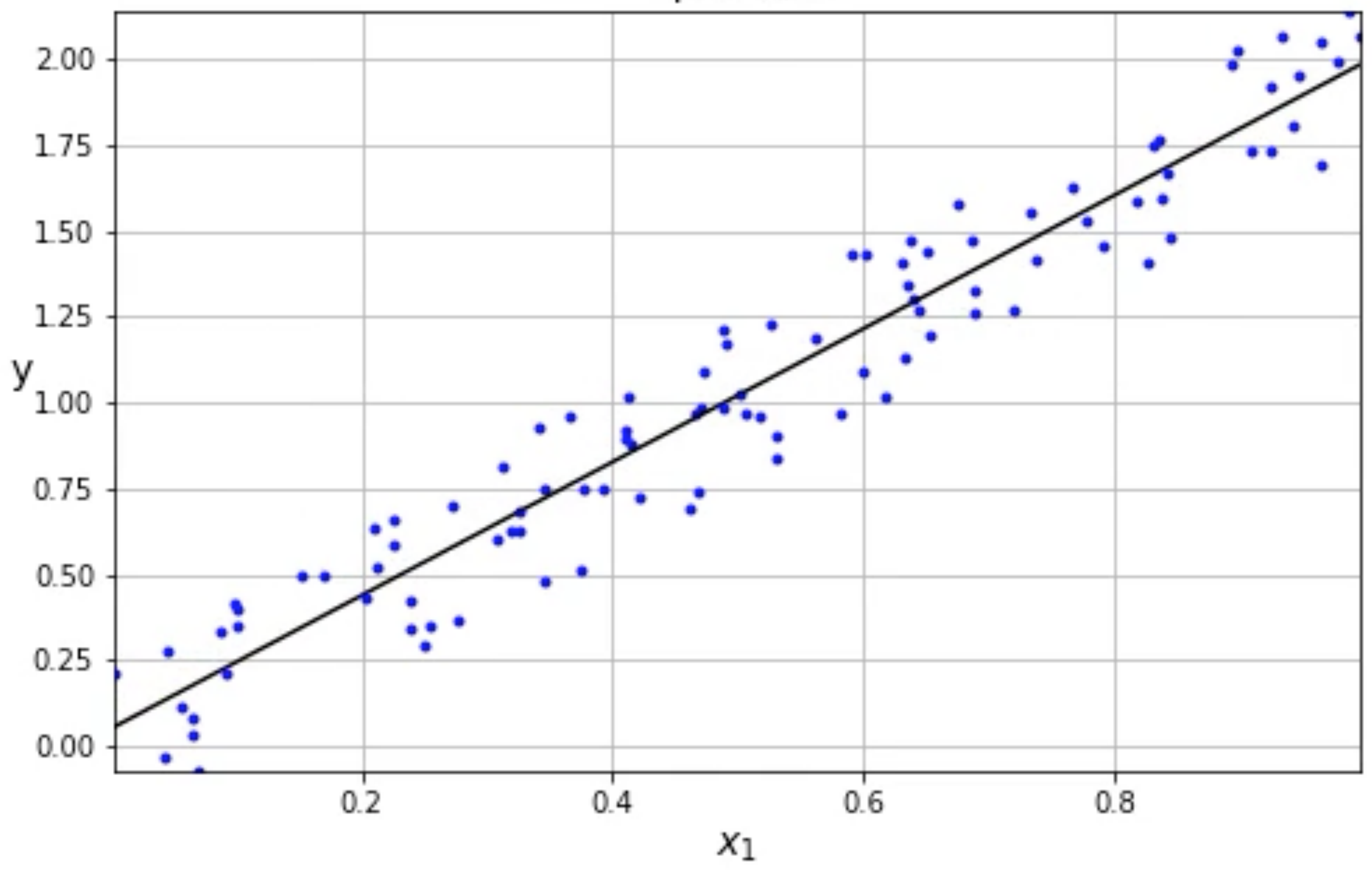
Regression

```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
```

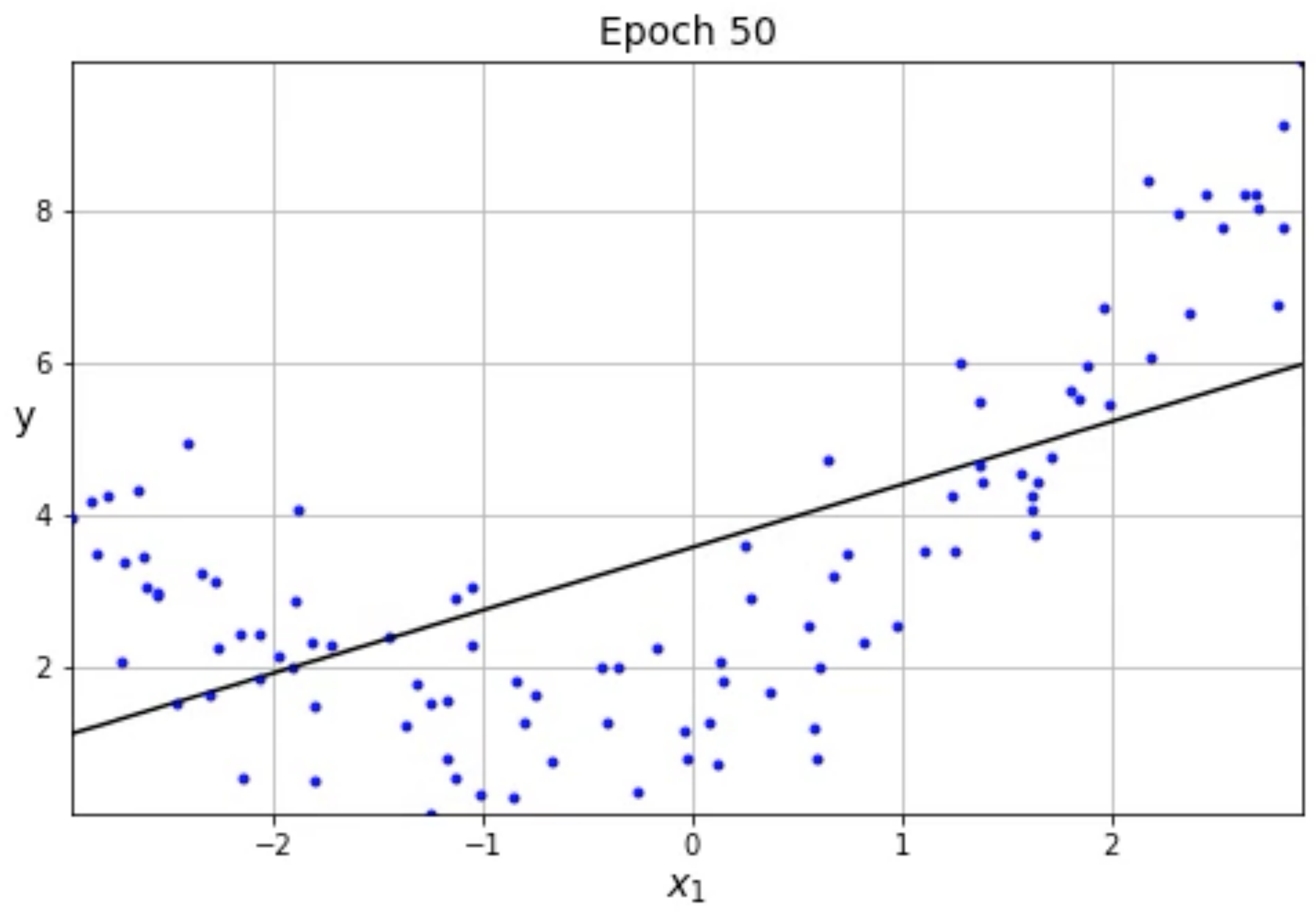
```python
def mse(output, target):
    return 0.5*(output - target)**2

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

```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
      # dl/dh = dl/dy * dy/dh
      dldh = np.dot(dldy, self.w2.T)
      # dl/dw1 = dl/dy * dy/dh * dh/dw1
      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



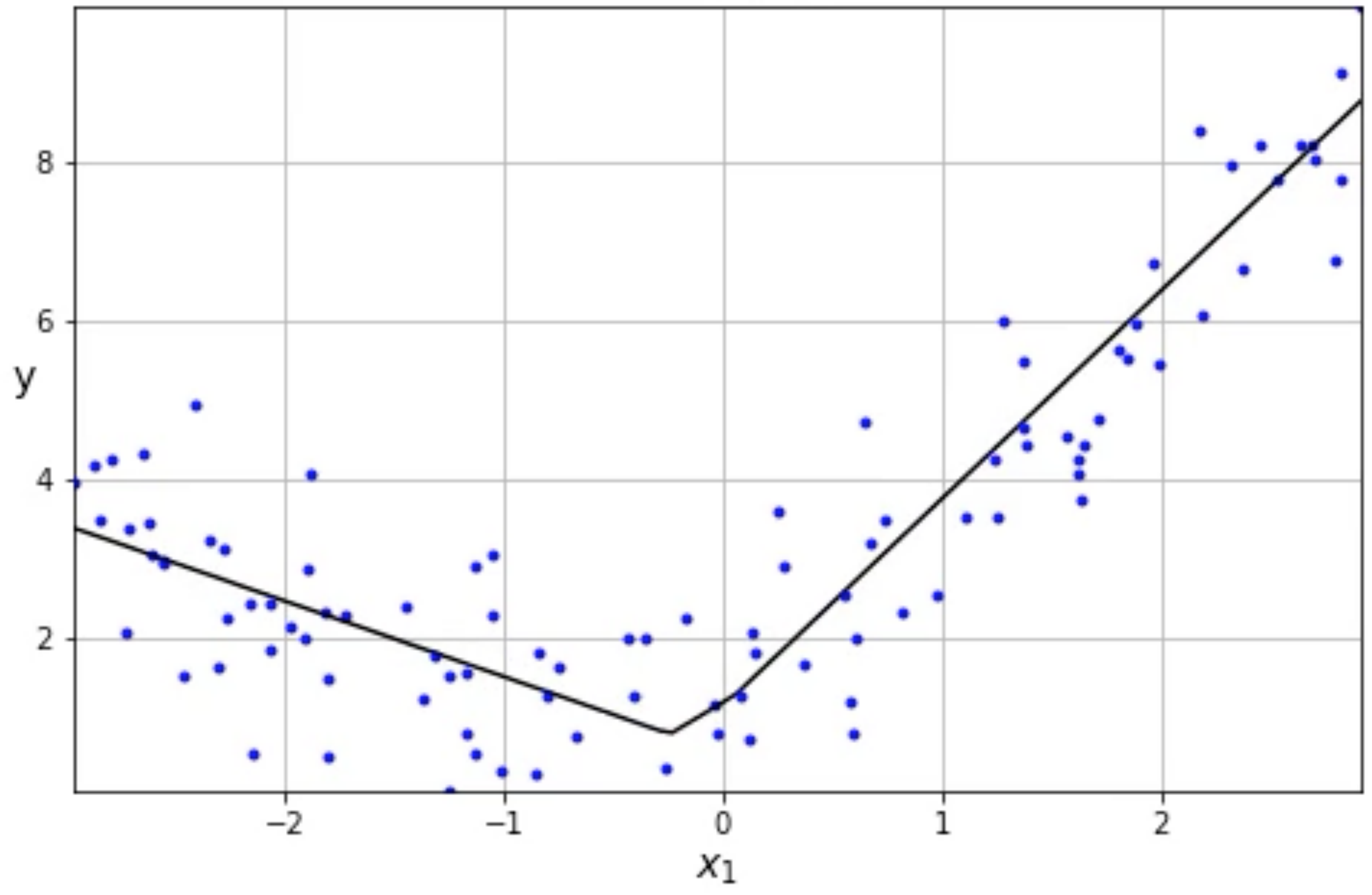
Polynomial Regression



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
```

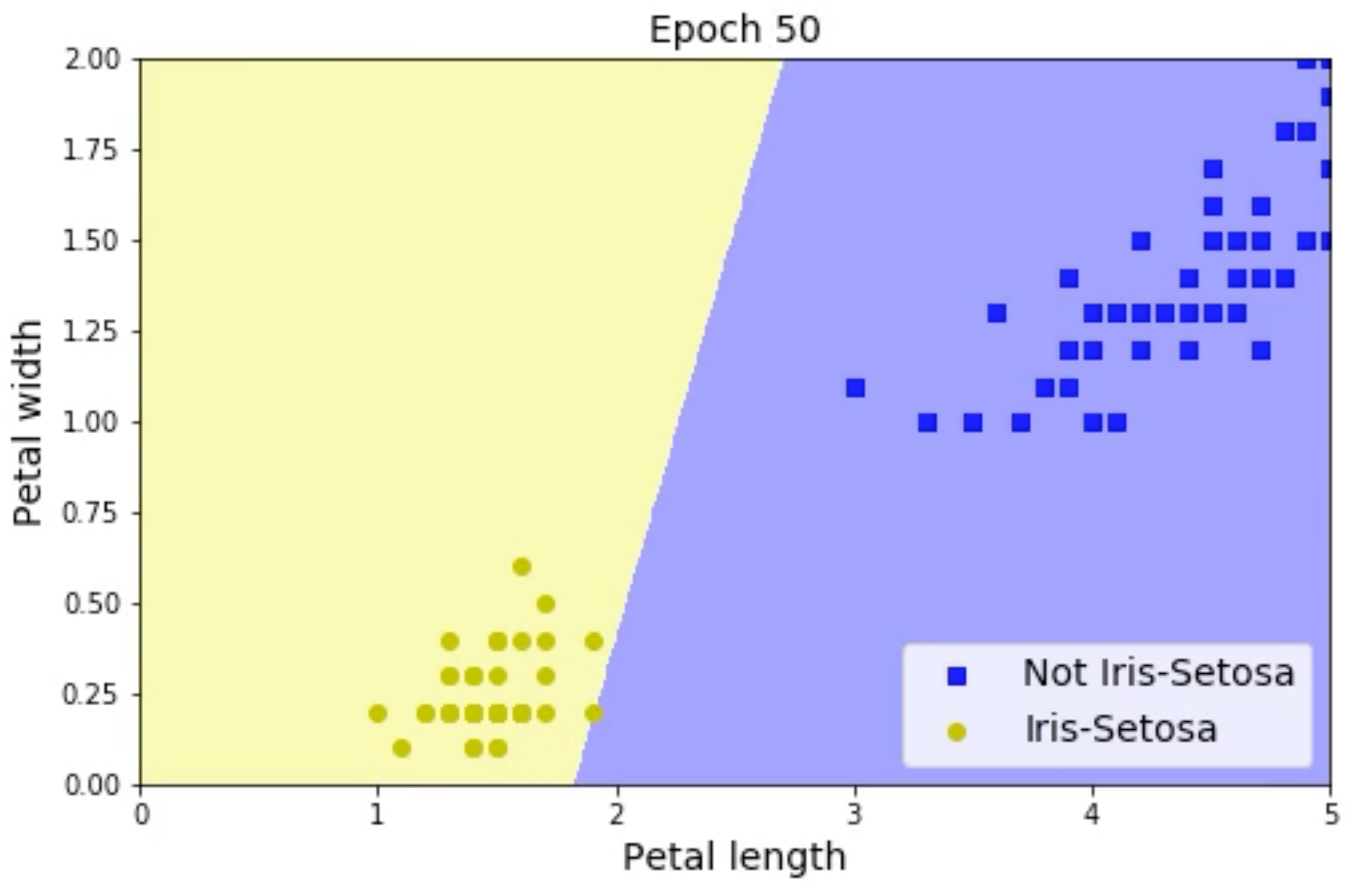
Epoch 50

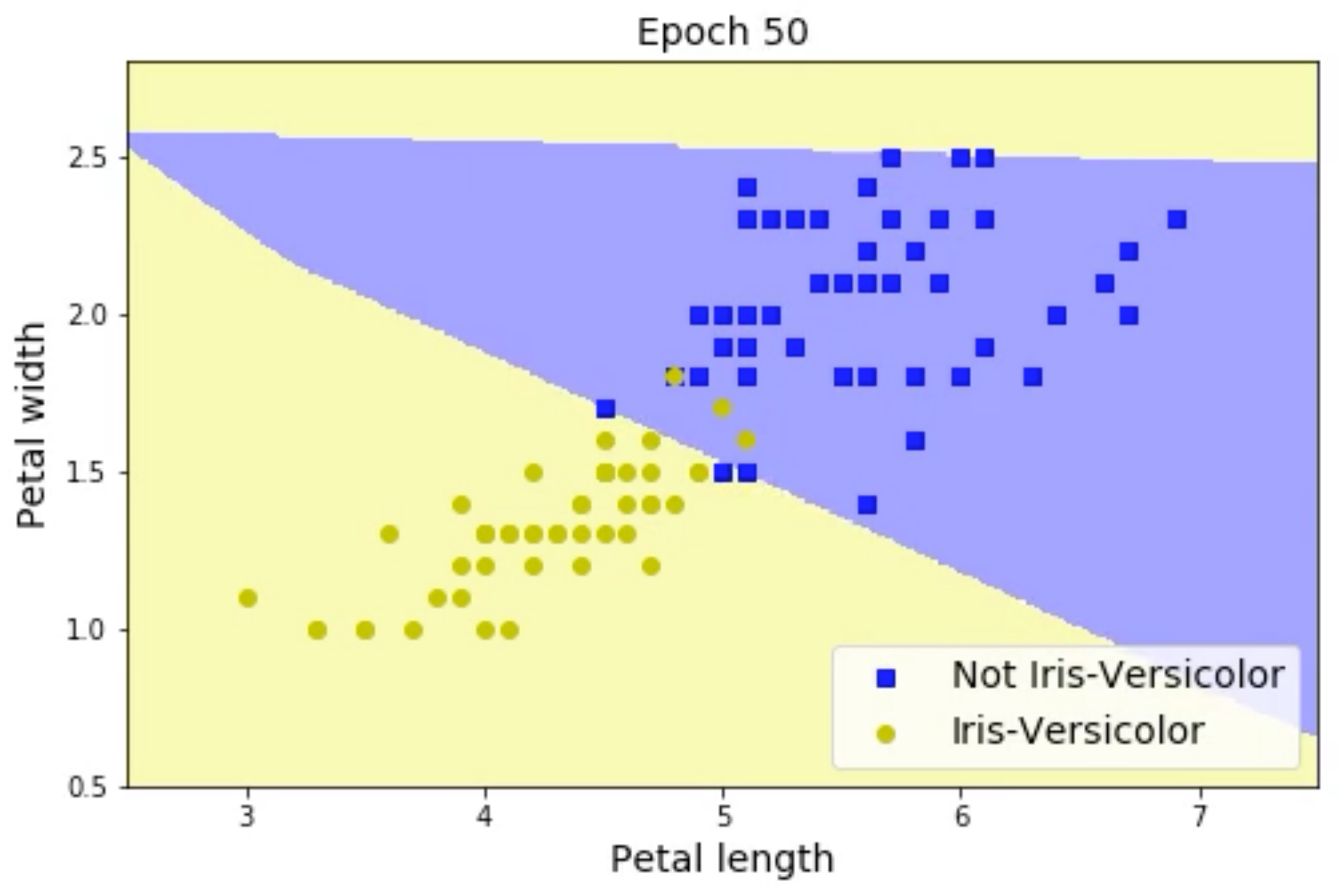


Binary Classification

```python
class MLPreluBinaryClass(MLPrelu):
  def final_activation(self, x):
    return x > 0
```

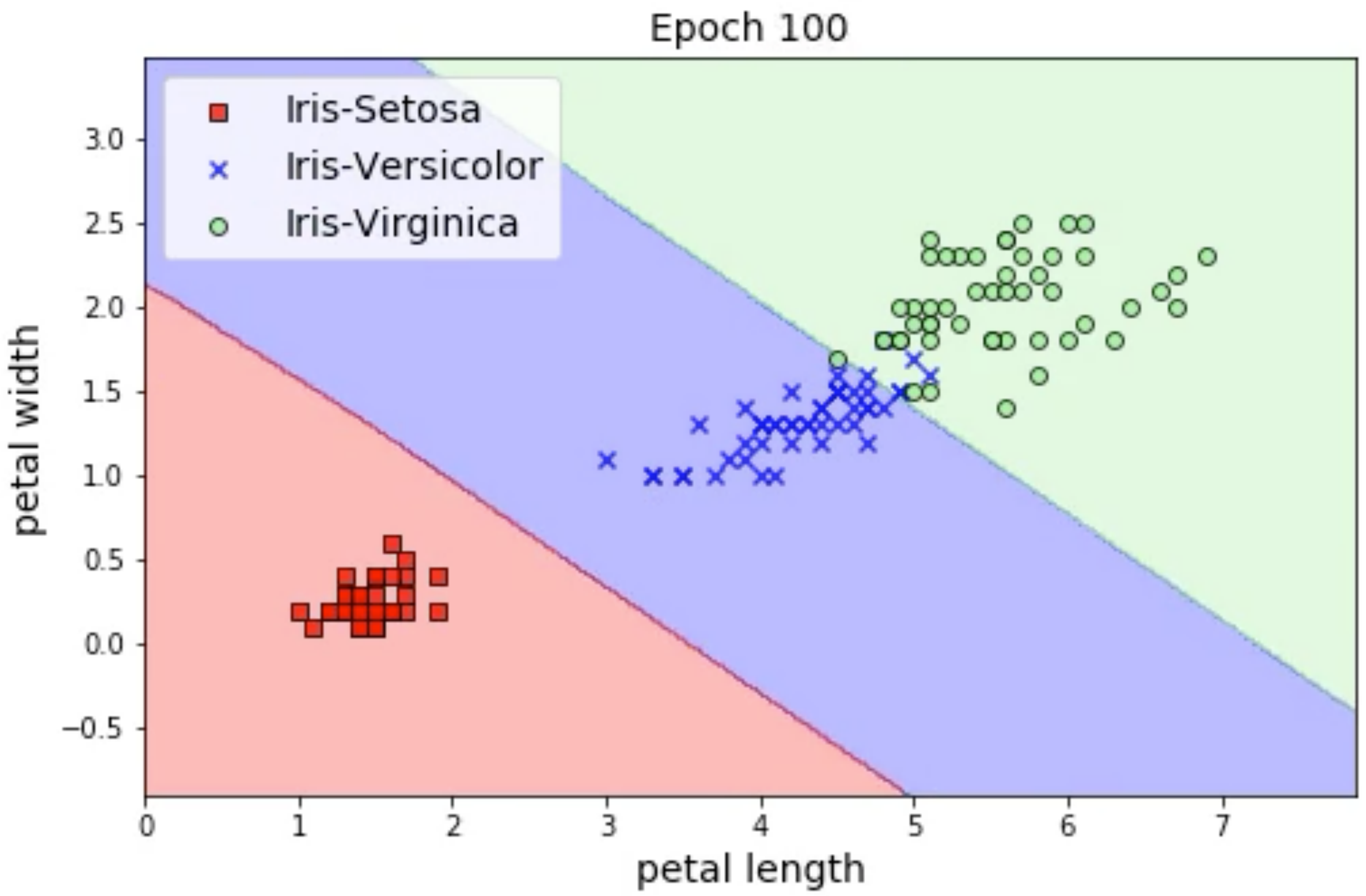
```python
class MLPrelu(MLP):
  def __call__(self, x):
    self.h = relu(np.dot(x, self.w1) + self.b1)
    y_hat = np.dot(self.h, self.w2) + self.b2
    return self.final_activation(y_hat)

  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
Petal width
Petal length
Not Iris-Setosa
Iris-Setosa
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0
1
2
3
4
5


Epoch 50
Petal width
Petal length
Not Iris-Versicolor
Iris-Versicolor
2.5
2.0
1.5
1.0
0.5
3
4
5
6
7

Multiclass Classification

```python
def crossentropy(output, target):
    logits = output[np.arange(len(output)),target]
    entropy = - logits + np.log(np.sum(np.exp(output),axis=-1))
    return entropy

def softmax(x):
    return np.exp(x) / np.exp(x).sum(axis=-1,keepdims=True)

def grad_crossentropy(output, target):
    answers = np.zeros_like(output)
    answers[np.arange(len(output)),target] = 1
    return (- answers + softmax(output)) / output.shape[0]

class MLPreluClass(MLPrelu):
  def __init__(self, D_in, H, D_out):
    super().__init__(D_in, H, D_out)
    self.loss = crossentropy
    self.grad_loss = grad_crossentropy
```


Epoch 100
Iris-Setosa
Iris-Versicolor
Iris-Virginica
petal width
petal length
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
0
1
2
3
4
5
6
7

What if now … ?

- We want to use a different activation function (not relu) ?
- We want to use more than 2 layers (3, 4, 5…) ?
- We want to use a different optimization algorithm (not SGD) ?
- We need more flexibility… we need an MLP framework !

Building an MLP framework

Pytorch-like API

```
D_in, H, D_out = 2, 3, 2

mlp = MLP([
   Linear(D_in, H),
   ReLU(),
   Linear(H, D_out)
])

optimizer = SGD(mlp, lr=0.1)
loss = CrossEntropy(mlp)

epochs = 100
for e in range(epochs):
  for x, y in zip(X, Y):
    y_pred = mlp(x)
    loss(y_pred, y)
    loss.backward()
    optimizer.update()
```

MLP

```python
class MLP:
    def __init__(self, layers):
        self.layers = layers

    def __call__(self, x):
        for layer in self.layers:
            x = layer(x)
        return x
```

Layers

```python
class Layer():
    def __init__(self):
        self.params = []
        self.grads = []

    def __call__(self, x):
        return x

    def backward(self, grad):
        return grad

    def update(self, params):
        return
```

```python
class Linear(Layer):
    def __init__(self, d_in, d_out):
        self.w = np.random.normal(loc=0.0,
                                  scale=np.sqrt(2/(d_in+d_out)),
                                  size=(d_in, d_out))
        self.b = np.zeros(d_out)

    def __call__(self, x):
        self.x = x
        self.params = [self.w, self.b]
        return np.dot(x, self.w) + self.b

    def backward(self, grad_output):
        grad = np.dot(grad_output, self.w.T)
        self.grad_w = np.dot(self.x.T, grad_output)
        self.grad_b = grad_output.mean(axis=0)*self.x.shape[0]
        self.grads = [self.grad_w, self.grad_b]
        return grad

    def update(self, params):
        self.w = params[0]
        self.b = params[1]
```

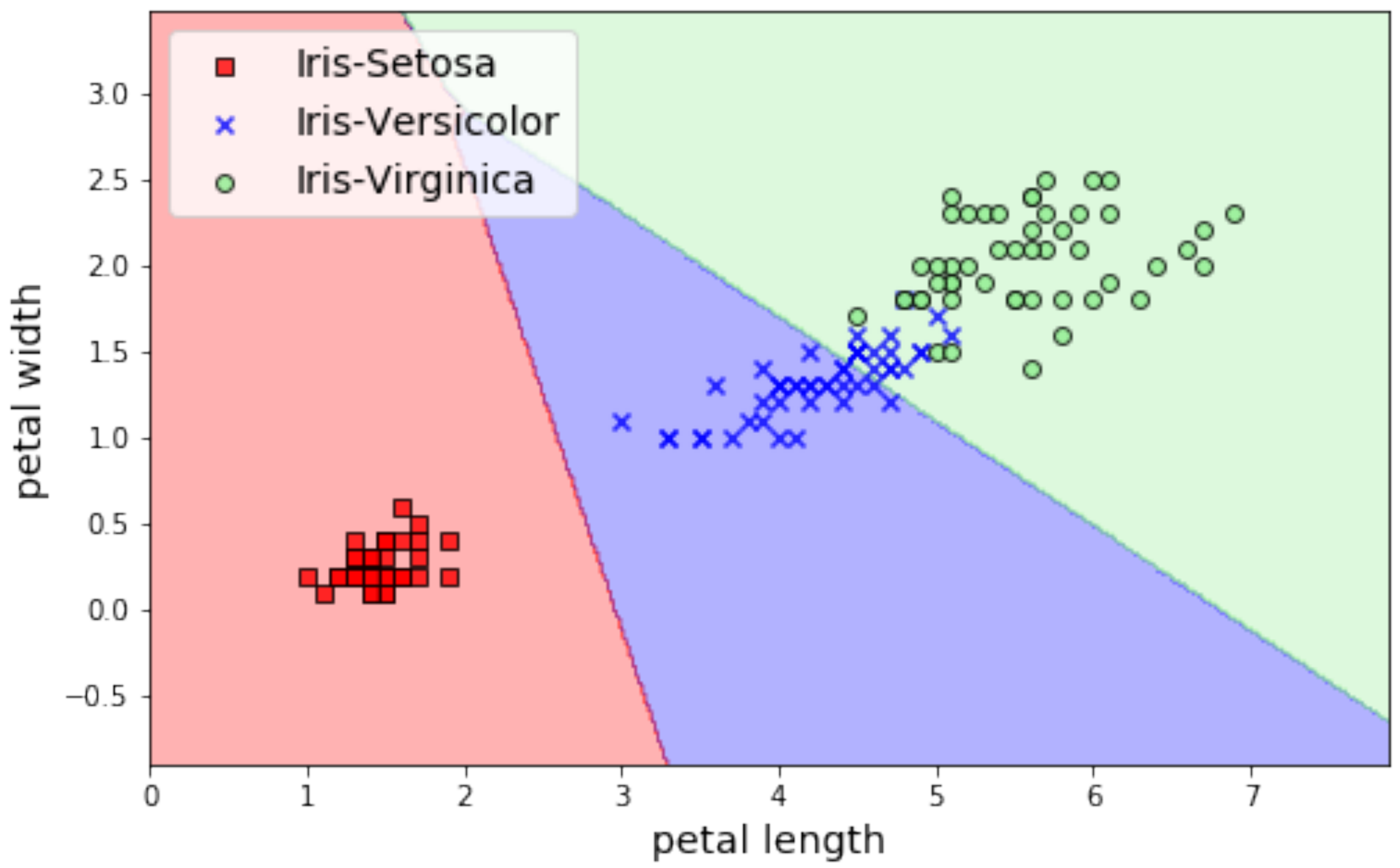
```python
class ReLU(Layer):
    def __call__(self, x):
        self.x = x
        return np.maximum(0, x)

    def backward(self, grad_output):
        grad = self.x > 0
        return grad_output*grad
```

Optimizers

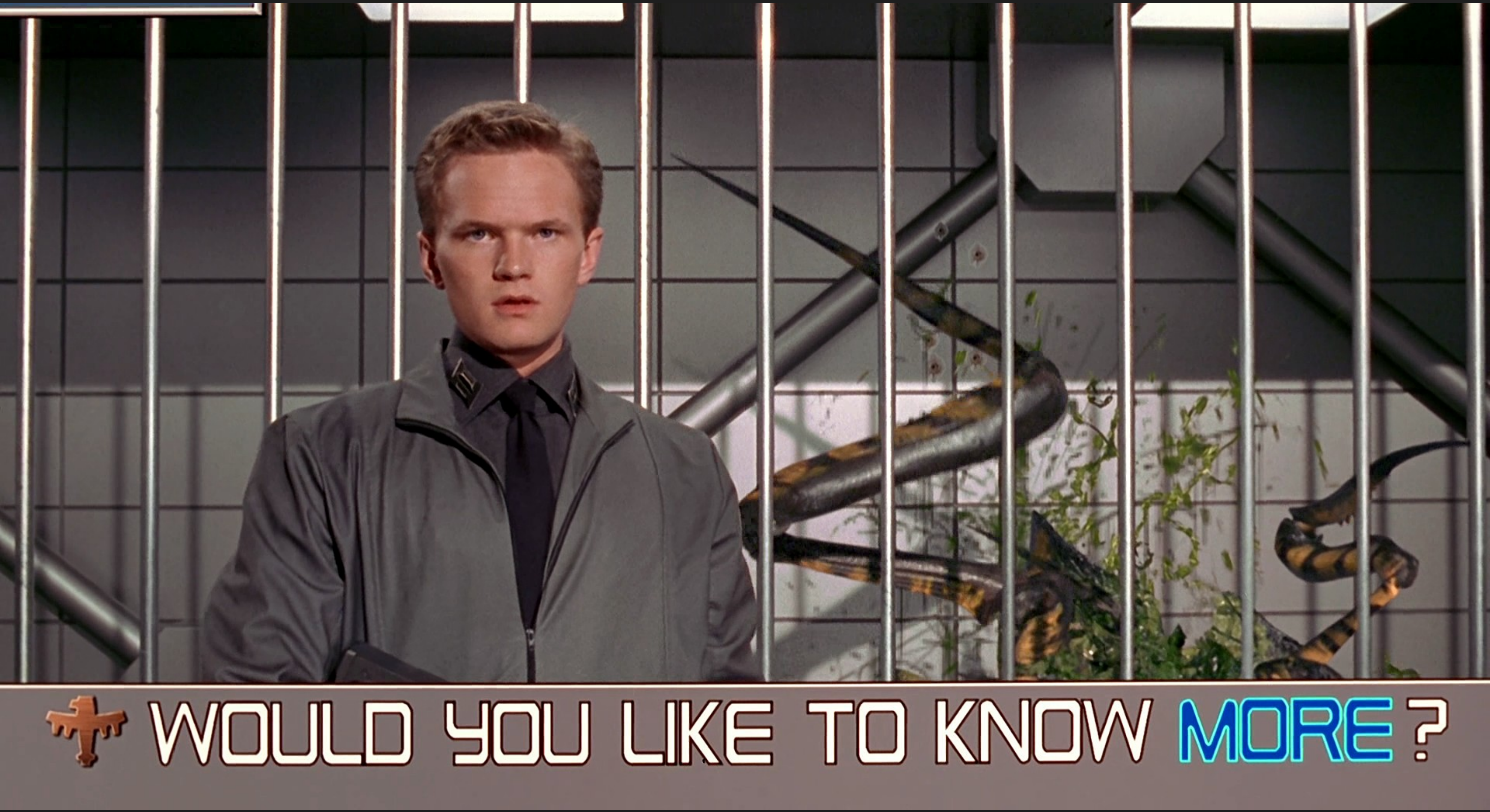
```python
class SGD():
    def __init__(self, net, lr):
        self.net = net
        self.lr = lr

    def update(self):
        for layer in self.net.layers:
            layer.update([
                params - self.lr*grads
                for params, grads in zip(layer.params, layer.grads)
            ])
```

Losses

```python
class CrossEntropy():
    def __init__(self, net):
        self.net = net

    def __call__(self, output, target):
        self.output, self.target = output, target
        logits = output[np.arange(len(output)), target]
        loss = - logits + np.log(np.sum(np.exp(output), axis=-1))
        loss = loss.mean()
        return loss

    def grad_crossentropy(self):
        answers = np.zeros_like(self.output)
        answers[np.arange(len(self.output)), self.target] = 1
        return (- answers + softmax(self.output)) / self.output.shape[0]

    def backward(self):
        grad = self.grad_crossentropy()
        for layer in reversed(self.net.layers):
            grad = layer.backward(grad)
```


Iris-Setosa
Iris-Versicolor
Iris-Virginica
petal width
petal length
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
0
1
2
3
4
5
6
7

https://colab.research.google.com/github/sensioai/nbs/blob/master/mlp/mlp.ipynb



https://playground.tensorflow.org/

Let’s code !

https://colab.research.google.com/github/sensioai/nbs/blob/master/mlp/exercise.ipynb

The Multilayer Perceptron