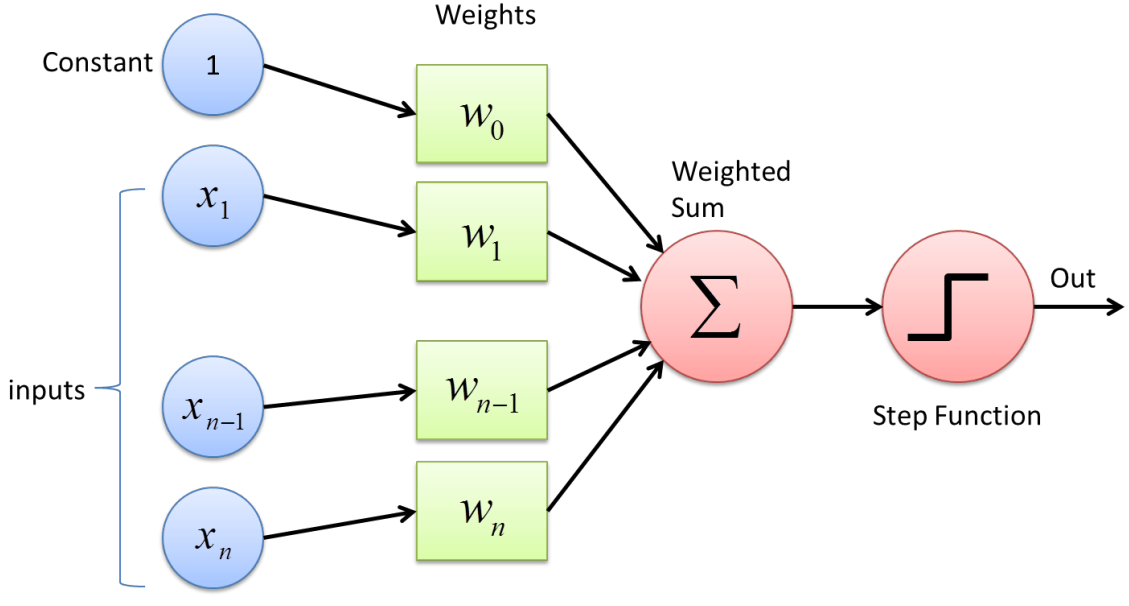
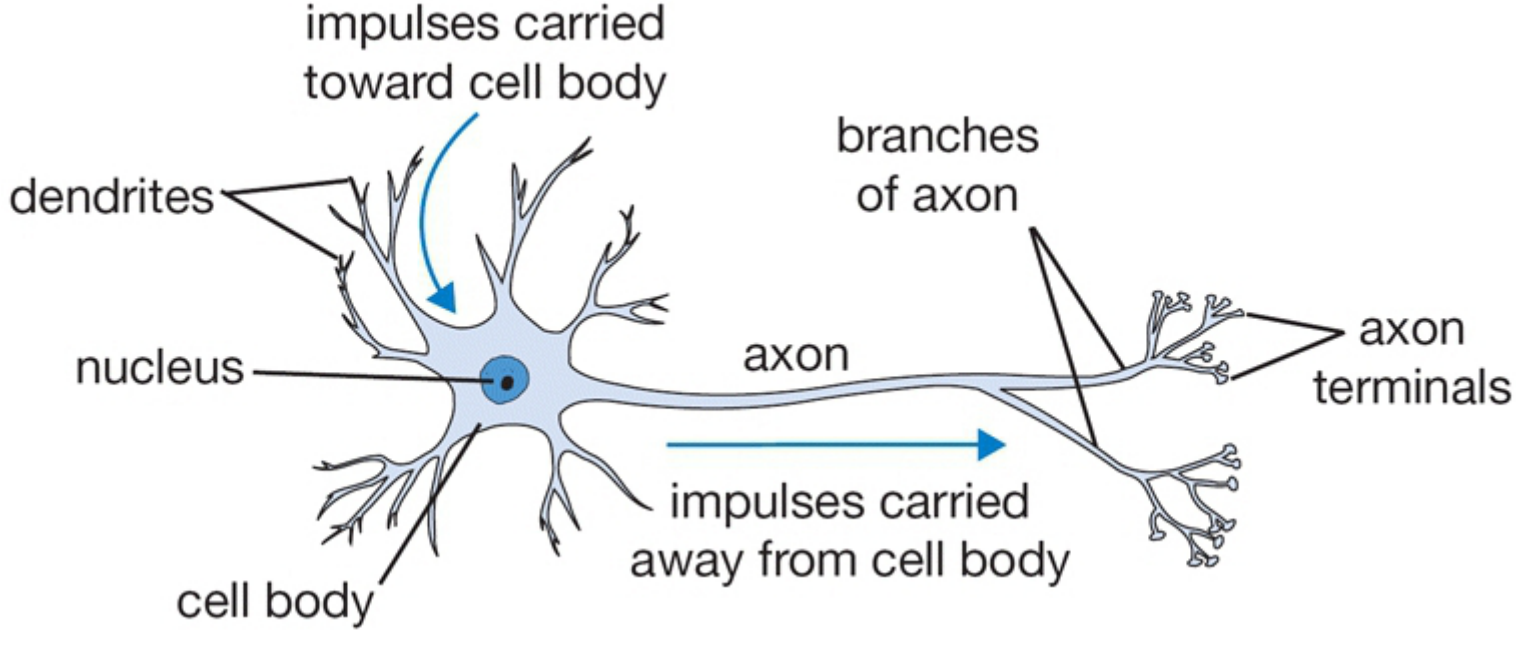
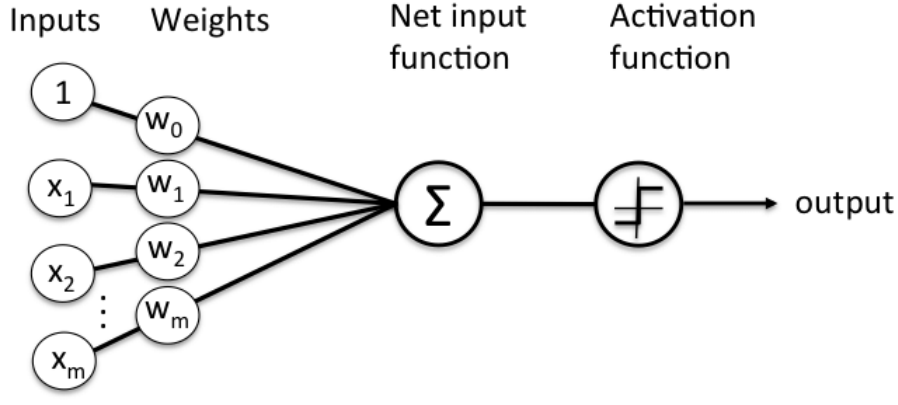
PERCEPTRON


Weights
Constant
1
w_0
x_1
w_1
Weighted
Sum
Σ
Out
inputs
x_{n-1}
w_{n-1}
x_n
w_n
Step Function

Artificial neural networks are inspired in biological neural networks



The perceptron computes a weighted sum of its inputs, then applies an activation function.

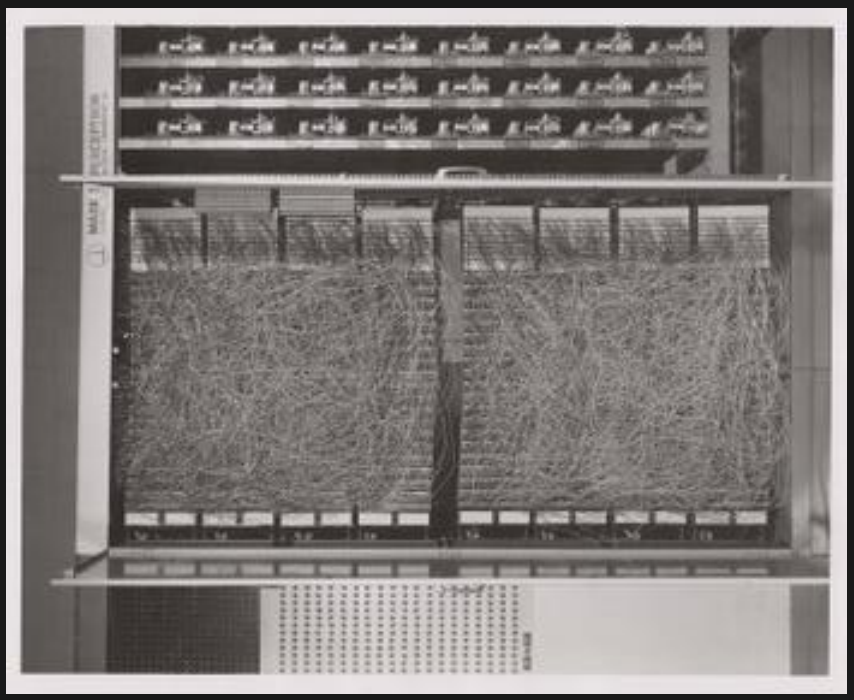


$$\hat{y} = f(\mathbf{x}^T \mathbf{w}) = f(w_0 + w_1 x_1 + \ldots + w_m x_m)$$

where $\hat{y}$ is the output, $\mathbf{w}$ are the perceptron parameters (also called weights), $\mathbf{x}$ are the data input features and the f is the activation function.

HISTORY

- Invented in 1958 at the Cornell Aeronautical Laboratory by Frank Rosenblat
- The perceptron was intended to be a machine, rather than a program
- This machine was designed for image recognition: it had an array of 400 photocells, randomly connected to the "neurons". Weights were encoded in potentiometers, and weight updates during learning were performed by electric motors



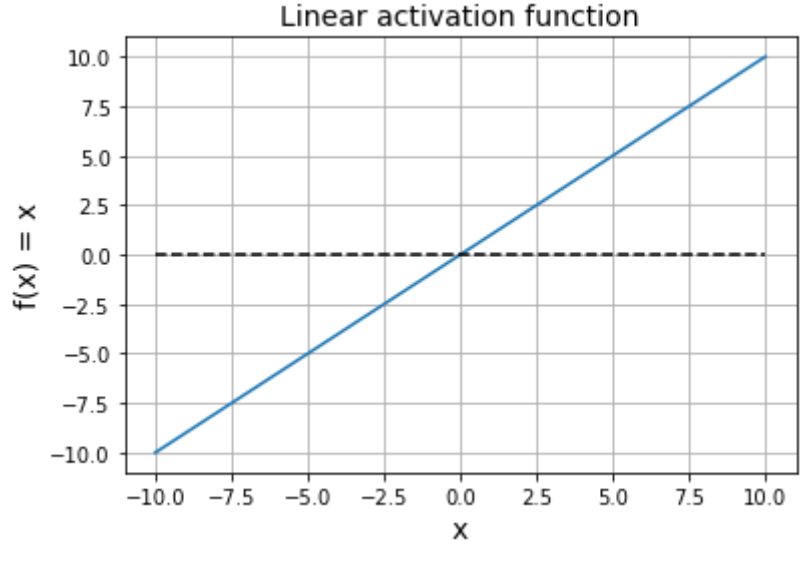
HISTORY

- Although the perceptron initially seemed promising, it was quickly proved that perceptrons could not be trained to recognise many classes of patterns.
- First AI Winter until the 1980s
- The perceptron is used today as the basic unit of feedforward neural network with two or more layers (also called multilayer perceptrons) with greater processing power

LINEAR REGRESSION

If we use a linear activation function we can use the preceptron for linear regression.

$$\hat{y} = w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n$$



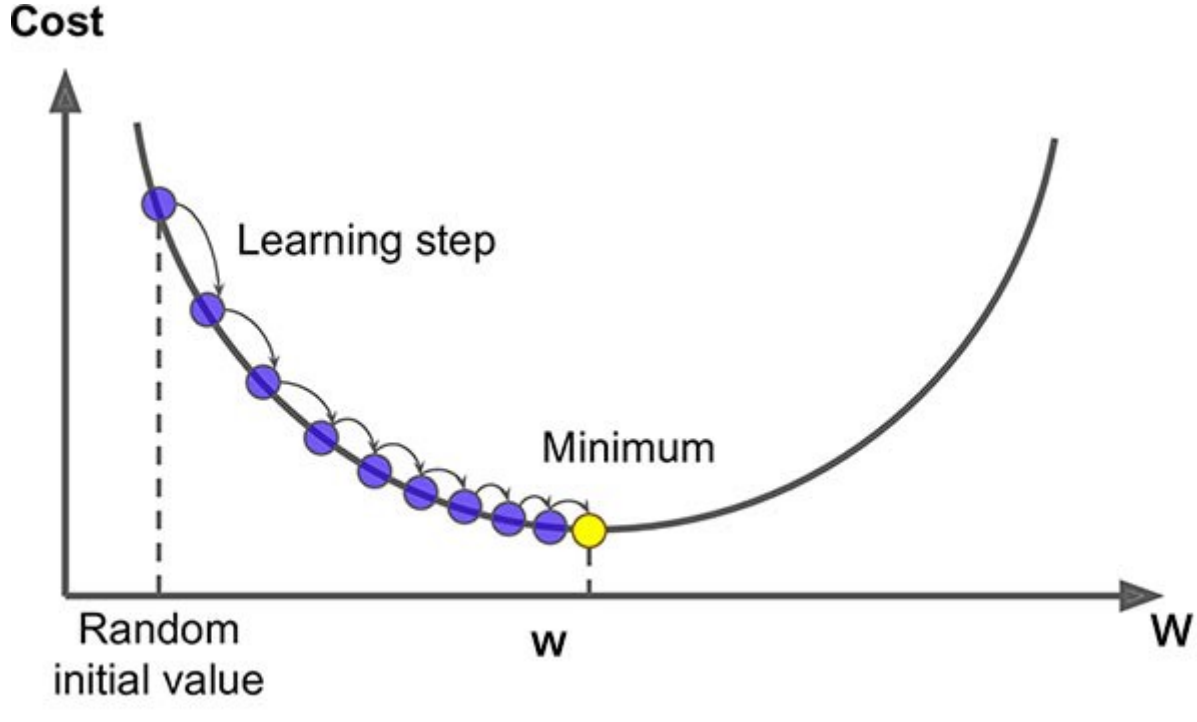
TRAINING A PERCEPTRON

Our objective is to find the values of $\mathbf{w}$ that minimize a cost function, for regression we usually use the Mean Square Error (MSE).

$$MSE(\hat{y}, y) = \frac{1}{N} \sum_{j=1}^{N} (\hat{y}^{(j)} - y^{(j)})^2$$

where N is the number of samples and $y^{(i)}$ is the ground truth of the j^{th} sample in the dataset.

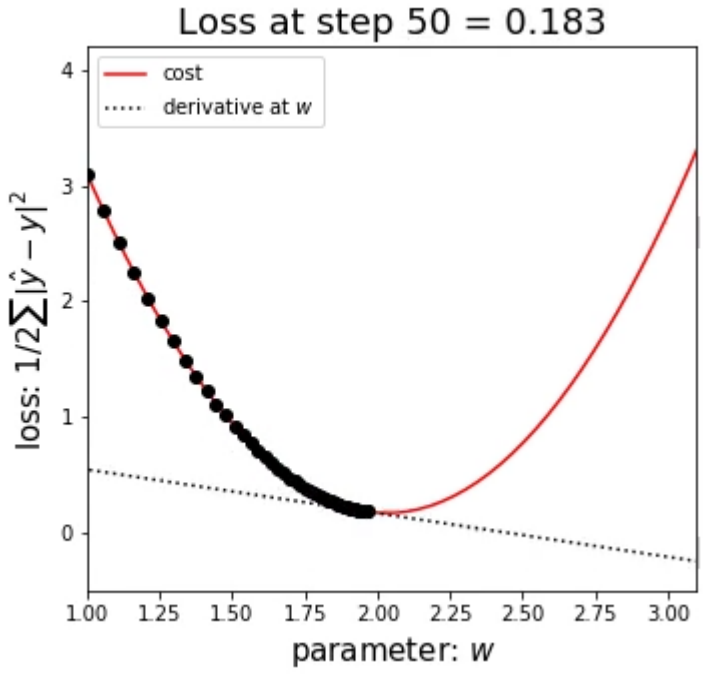
GRADIENT DESCENT

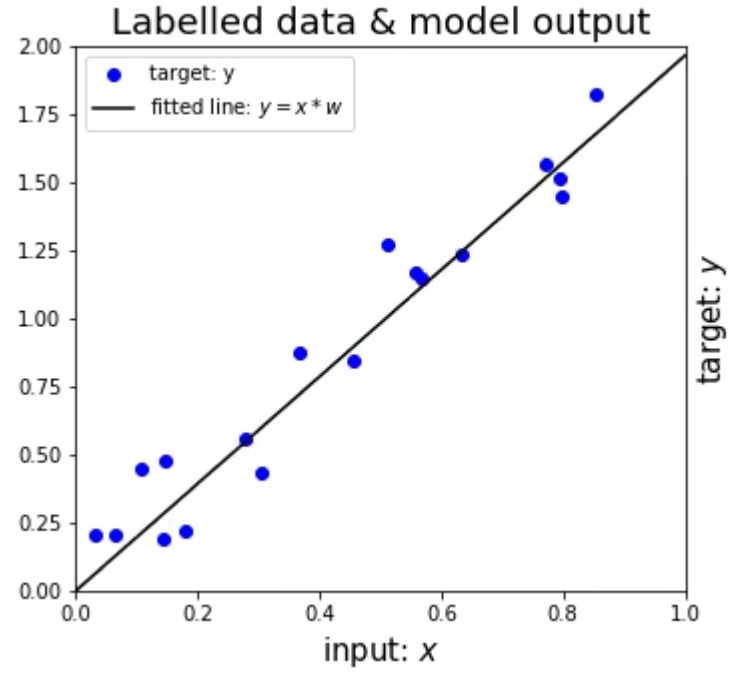


GRADIENT DESCENT

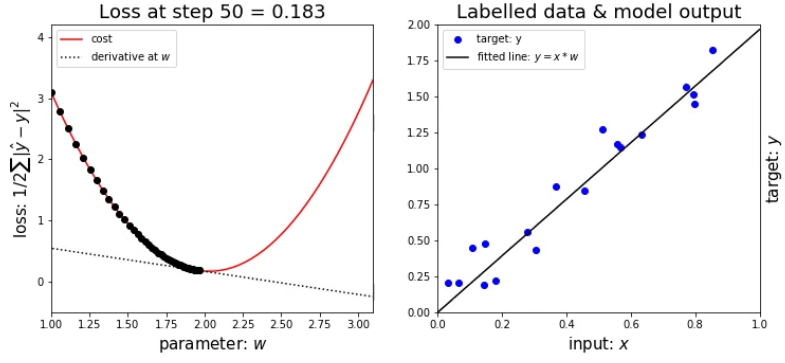
The algorithm goes as follows:

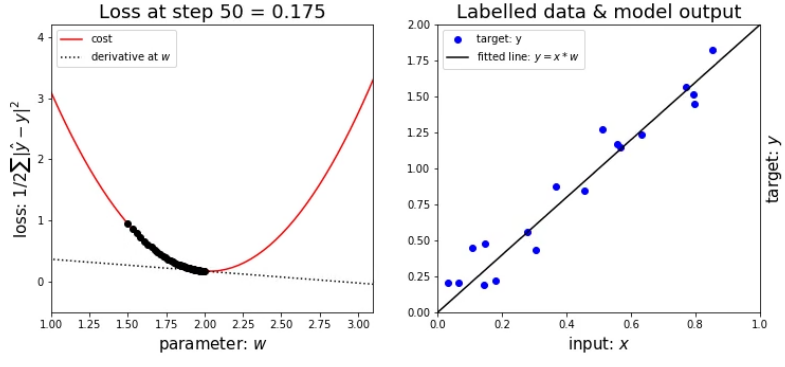
- Compute the output of the model, $\hat{y}$.
- Compute the gradient of the loss function with respect to the parameters, $\frac{\partial MSE}{\partial w} = \frac{2}{N}\frac{\partial \hat{y}}{\partial w}(\hat{y} - y)$ where $\frac{\partial \hat{y}}{\partial w} = x$.
- Update the parameters, $w \leftarrow w - \eta \frac{\partial MSE}{\partial w}$, where η is the learning rate.
- Repeat until convergence.

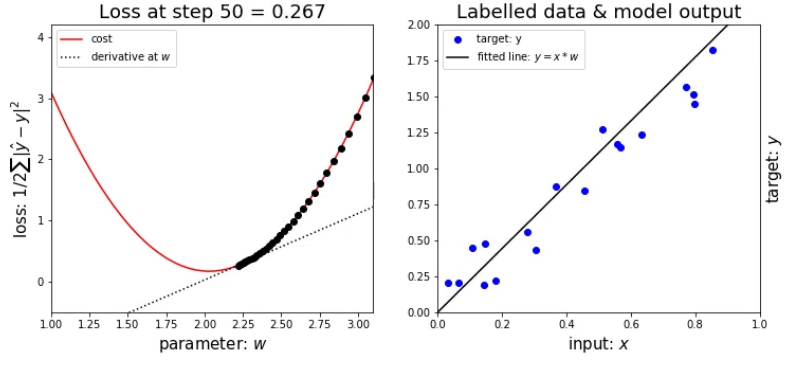

Loss at step 50 = 0.183
cost
derivative at w
loss: 1/2∑|ŷ − y|²
parameter: w
1.00
1.25
1.50
1.75
2.00
2.25
2.50
2.75
3.00
0
1
2
3
4


Labelled data & model output
target: y
fitted line: y = x * w
input: x
target: y
0.0
0.2
0.4
0.6
0.8
1.0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00

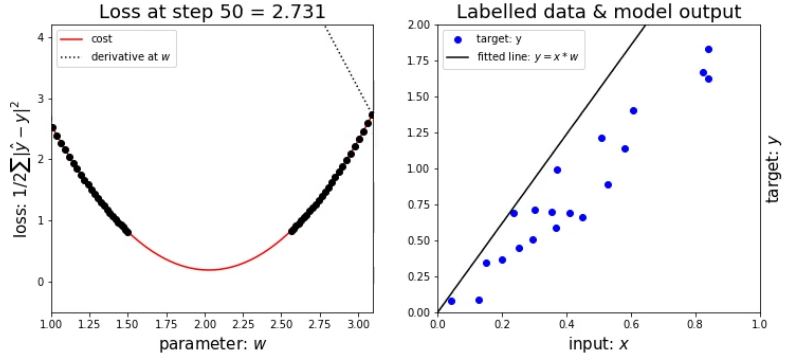
Effect of the initialization

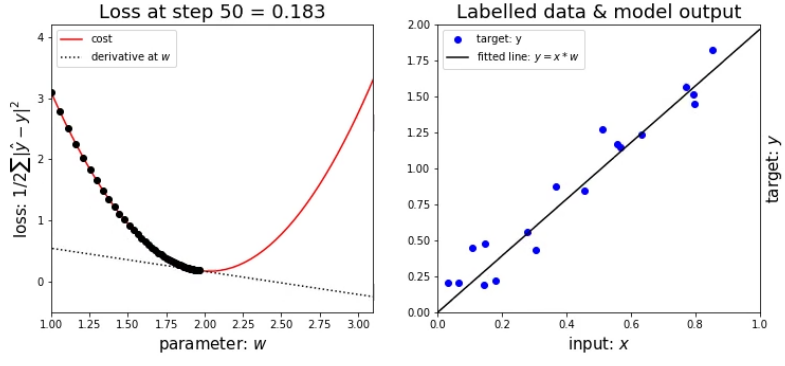


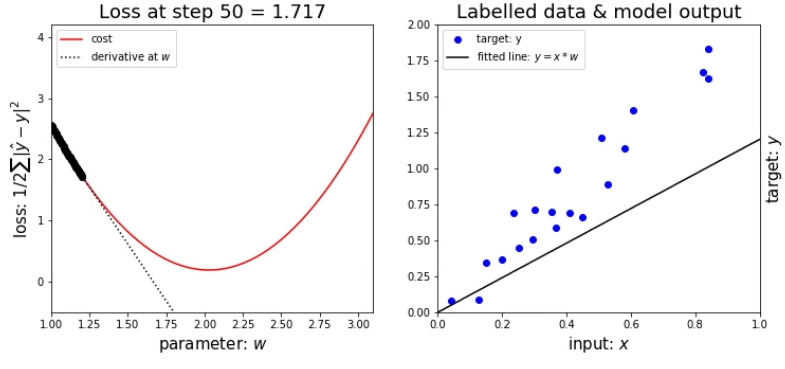




Effect of the learning rate



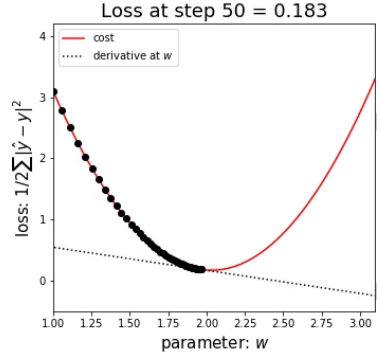


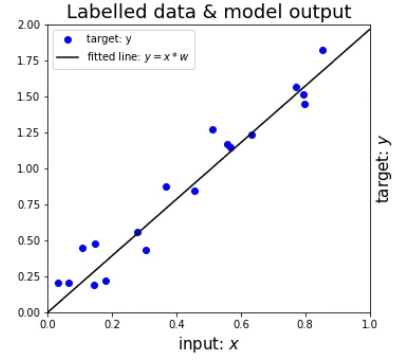


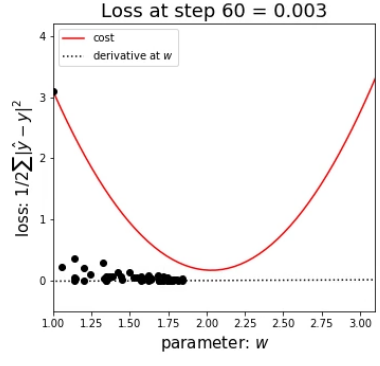
GRADIENT DESCENT FLAVOURS

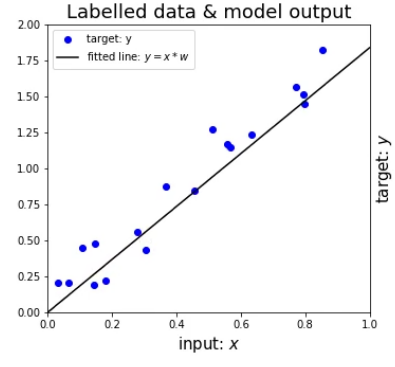
Depending on the data used to evaluate the gradients we distinguish between:

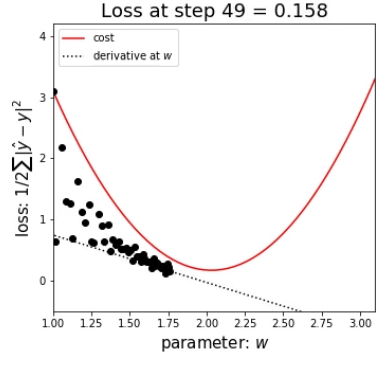
- Batch Gradient Descent: We use the entire dataset to compute gradients.
- Stochastic Gradient Descent: We use a random instance to evaluate the gradient.
- Mini-Batch Gradient Descent: We use a small random set of instances to evaluate gradient (default choice).

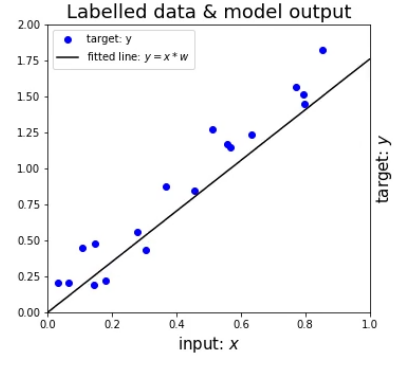

Loss at step 50 = 0.183
cost
derivative at w
loss: 1/2∑|ŷ − y|²
parameter: w


Labelled data & model output
target: y
fitted line: y = x * w
target: y
input: x

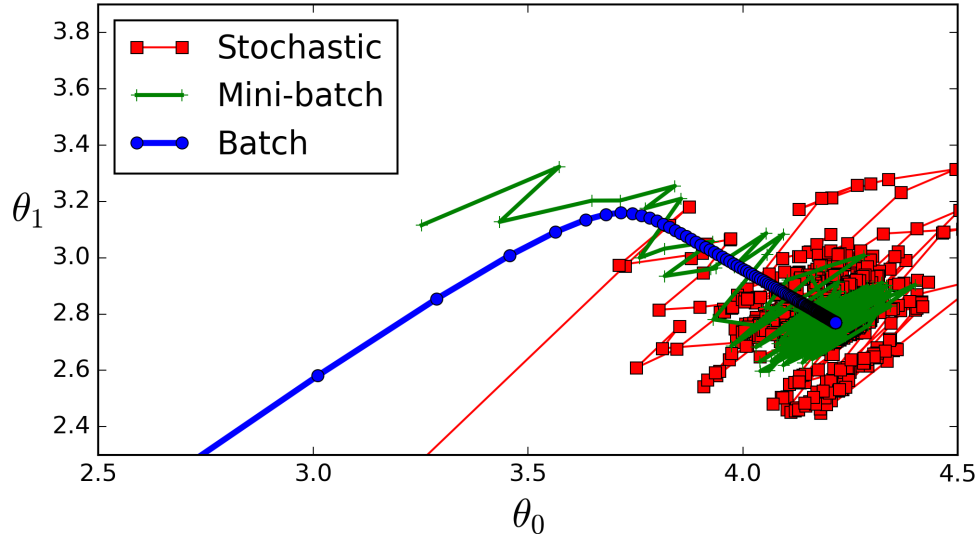

Loss at step 60 = 0.003
cost
derivative at w
loss: 1/2∑|ŷ − y|²
parameter: w


Labelled data & model output
target: y
fitted line: y = x * w
target: y
input: x


Loss at step 49 = 0.158
cost
derivative at w
loss: 1/2∑|ŷ − y|²
parameter: w


Labelled data & model output
target: y
fitted line: y = x * w
target: y
input: x

- Batch gradient descent can converge nicely to the minimum, but since it requires the entire dataset it is very slow (and sometimes impracticable).
- The alternatives converge more erratically, but faster.
- To address this issue we normally use learning rate schedulers (decrease-increase learning rate during training).

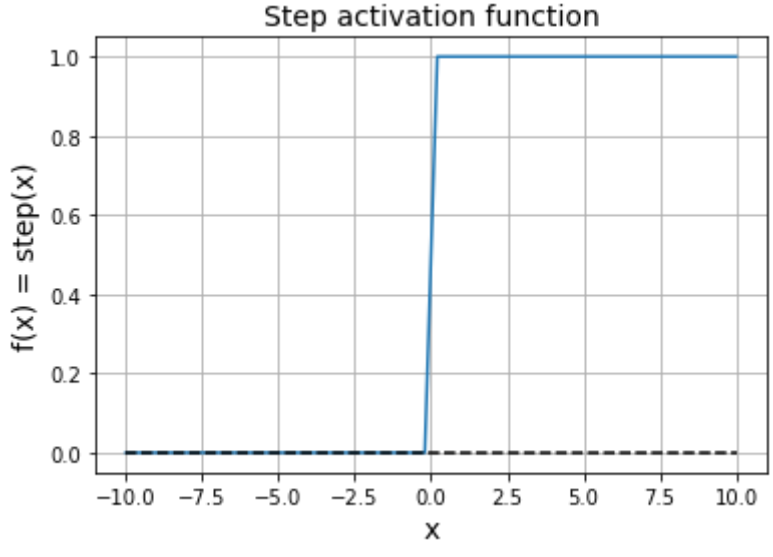

Stochastic
Mini-batch
Batch
3.8
3.6
3.4
3.2
3.0
2.8
2.6
2.4
θ_1
2.5
3.0
3.5
4.0
4.5
θ_0

BINARY CLASSIFICATION

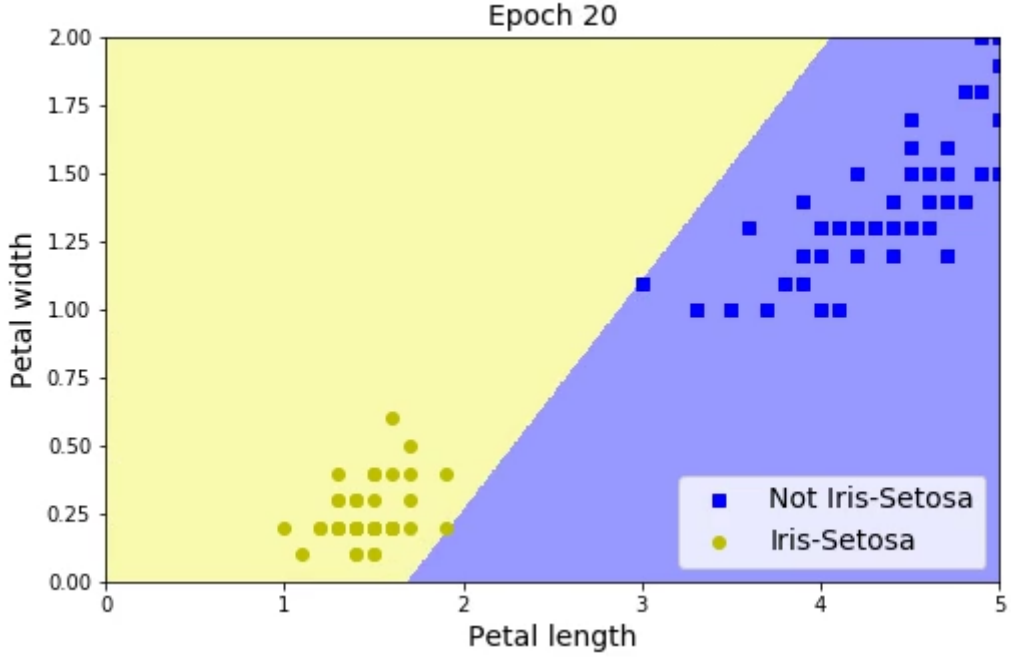
We can use a perceptron to perform binary classification, say if one particular instance belongs to a particular class or not. For that we use a step activation function.

$$\hat{y} = \text{step}(\mathbf{x}^T \mathbf{w}) = \text{step}(w_0 + w_1 x_1 + \ldots + w_m x_m)$$

where the step function sets the output to 0 if $\mathbf{x}^T \mathbf{w} \leq 0$ or 1 if $\mathbf{x}^T \mathbf{w} > 0$.

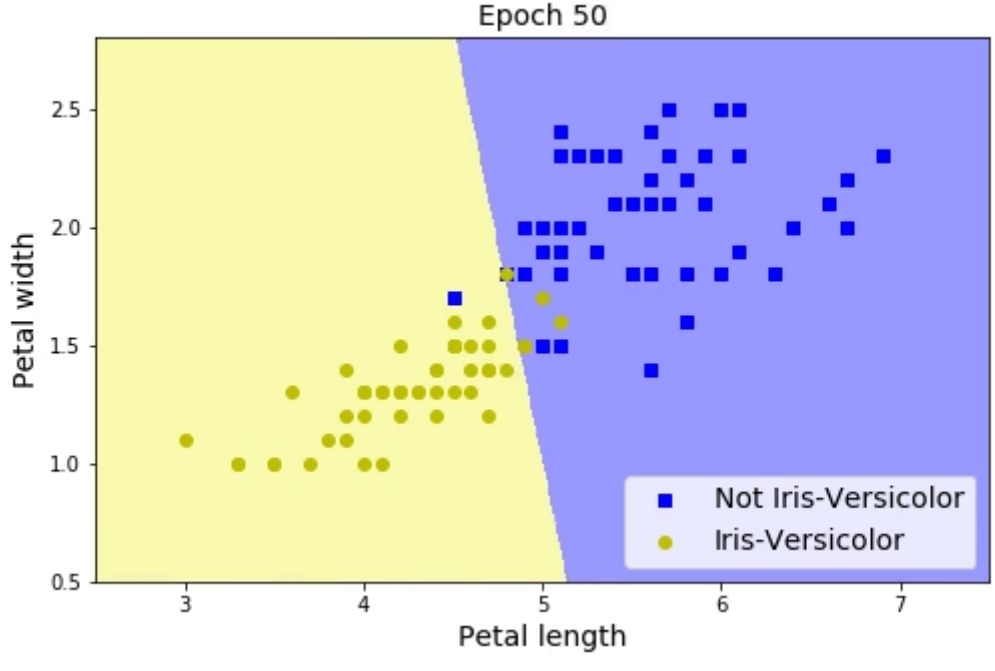

Step activation function
f(x) = step(x)
1.0
0.8
0.6
0.4
0.2
0.0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x

```python
class Perceptron():
    def __init__(self, size):
        self.w = np.random.randn(size)

    def __call__(self, w, x):
        return w.dot(x.T) > 0

    def fit(self, X, Y, epochs, lr):
        X = np.c_[np.ones(len(X)), X]
        for epoch in range(epochs):
            for x, y in zip(X, Y):
                y_hat = self(self.w, x)
                # loss function (MSE)
                l = 0.5*(y_hat - y)**2
                # derivatives
                dldh = (y_hat - y)
                dhdw = x
                dldw = dldh*dhdw
                # update
                self.w = self.w - lr*dldw
```


Epoch 20
Petal width
Petal length
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0
1
2
3
4
5
Not Iris-Setosa
Iris-Setosa

LIMITATIONS

The perceptron is a linear classifier, therefore it will never get to the state with all the input vectors classified correctly if the training set is not linearly separable.

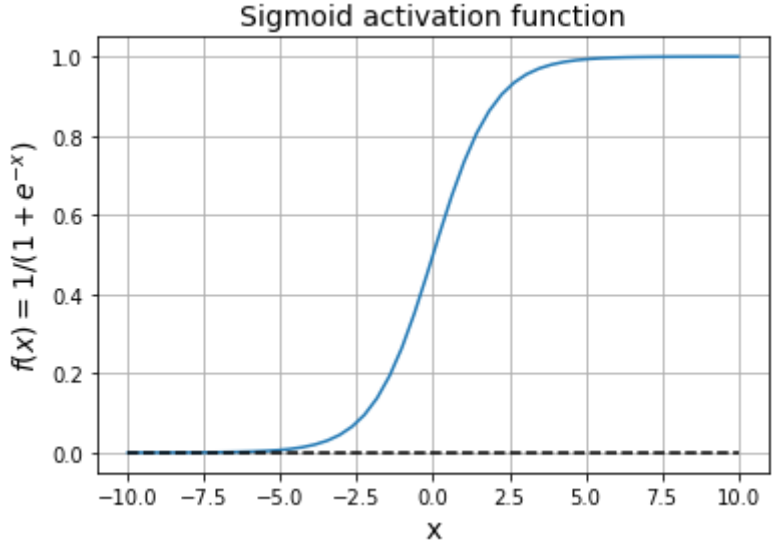


LOGISTIC REGRESSION

We can use a perceptron to predict the probability of an instance to belong to a particular class. For that we use a sigmoid activation function:

$$\hat{y} = \sigma(\mathbf{x}^T \mathbf{w})$$

Where $\sigma(x) = \frac{1}{1+e^{-x}}$ is the sigmoid function, that will limit the value of $\hat{y}$ between 0 and 1.

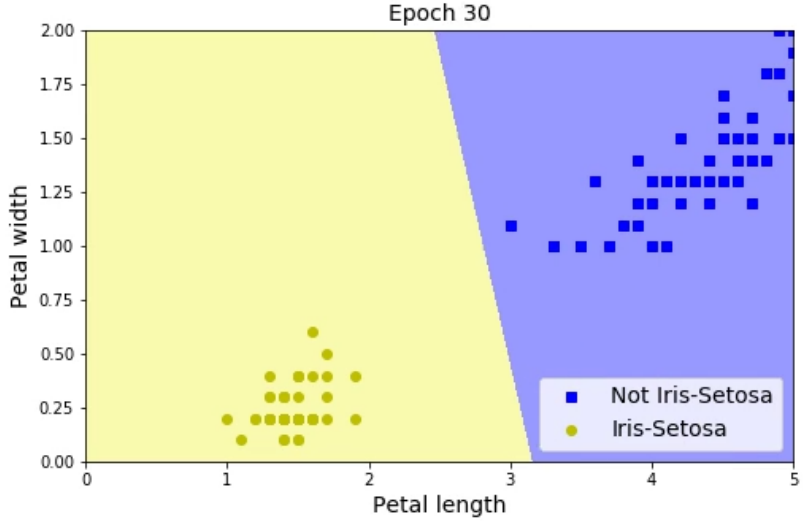

Sigmoid activation function
1.0
0.8
0.6
0.4
0.2
0.0
$f(x) = 1/(1 + e^{-x})$
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x

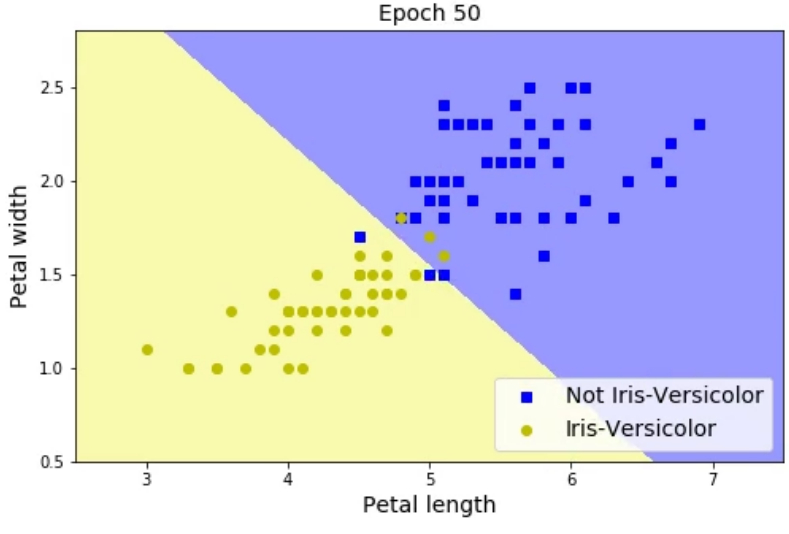
To train a logistic regression model we need a different loss function, called the log loss function.

$$J(\mathbf{w}) = -\frac{1}{N}\sum_{j=1}^{N}\left[y^{(j)}log(\hat{y}^{(j)}) + (1-y^{(j)})log(1-\hat{y}^{(j)})\right]$$

There is no known close-form to minimize this function, but gradient descent is guaranteed to find the global minimum if the learning rate is not too large. For that we need the derivative of the cost function with respect to the model parameters

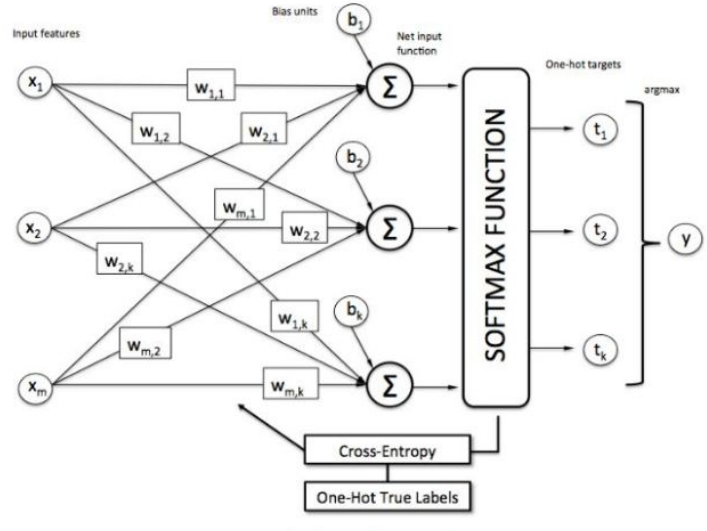
```
class LogisticRegression():
    def __init__(self, size):
        self.w = np.random.randn(size)

    def __call__(self, w, x):
        return self.sigmoid(w.dot(x.T))

    def sigmoid(self, x):
        return 1. / (1. + np.exp(-x))

    def fit(self, X, Y, epochs, lr):
        X = np.c_[np.ones(len(X)), X]
        for epoch in range(epochs):
            for x, y in zip(X, Y):
                y_hat = self(self.w, x)
                self.w = self.w - lr*(y_hat - y)*x
```


Epoch 30
Petal width
Petal length
Not Iris-Setosa
Iris-Setosa


Epoch 50
Petal width
Petal length
Not Iris-Versicolor
Iris-Versicolor

MULTI-CLASS CLASSIFICATION

The logistic regression model can be generalized to support multiple classes. For that we use a Perceptron with multiple outputs (one for each class) and then apply a softmax activation function.



Softmax Regression

The output can be computed as

$$\hat{y} = \arg\max_k \ \sigma(\mathbf{w} \cdot \mathbf{x})_k$$

where now $\sigma(t)_k = \frac{e^{t_k}}{\sum_{k=1}^{K} e^{t_k}}$ is the softmax function, K is the number of classes and t_k is the score of each class for the instance $\mathbf{x}$.

The loss function used for training softmax regression model is called Cross Entropy

$$J(\mathbf{w}) = -\frac{1}{N}\sum_{j=1}^{N}\sum_{k=1}^{K} y_k^{(j)} log(\hat{y}_k^{(j)})$$

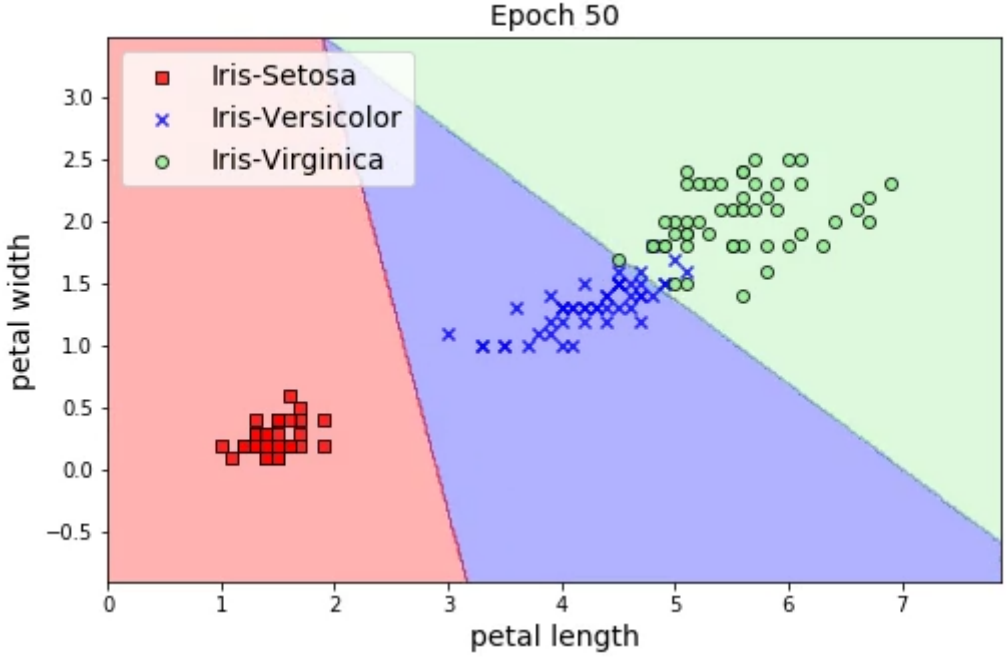
where $y_k^{(j)}$ is the probability that the j^{th} instance belongs to class k (usually 1 or 0). If $K = 2$ the cost function is equivalent to the log loss function. The gradient of the cross entropy cost function is

```python
class SoftmaxRegression():
    def __init__(self, inputs, outputs):
        self.w = np.random.randn(inputs, outputs)

    def fit(self, X, Y, epochs, lr):
        X = np.c_[np.ones(len(X)), X]
        for epoch in range(epochs):
            for x, y in zip(X, Y):
                x = x[None, :]
                y_hat = np.dot(x, self.w)
                loss = crossentropy(y_hat, y).mean()
                dldo = grad_crossentropy(y_hat, y)
                grad_w = np.dot(x.T, dldo)
                self.w = self.w - lr * grad_w

    def predict(self, x):
        x = np.c_[np.ones(len(x)), x]
        y_hat = np.dot(x, self.w)
        return np.argmax(softmax(y_hat), axis=1)
```

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

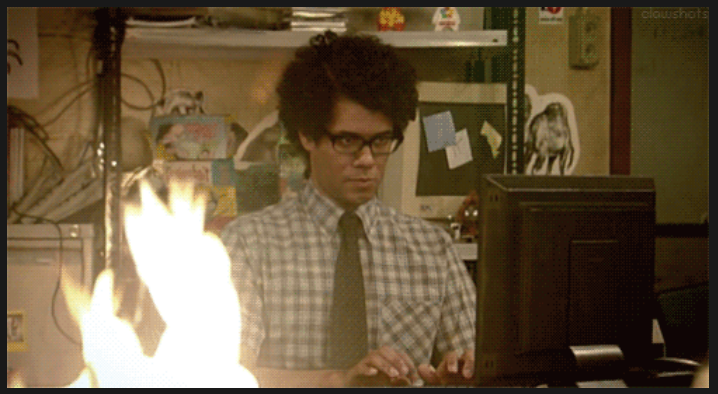

Epoch 50
Iris-Setosa
Iris-Versicolor
Iris-Virginica
petal width
petal length
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
0
1
2
3
4
5
6
7

SUMMARY

- ANNs are inspired in biological NNs
- The perceptron computes a weighted sum of its inputs and applies an activation function
- We can make different tasks depending on the activation and loss functions used
- We use Gradient Descent to find the set of parameters that minimize the loss function
- Linear models have limitations with data that is not linearly separable

Task	Activation	Loss
Regression	Linear	MSE
Binary Classification	Step Sigmoid	MSE LogLoss
Multi-class Classification	Softmax	CE
Multi-label Classification	Sigmoid	LogLoss

See the code



Practice with this exercise