Lecture 4: Simple Machine Learning Algorithms for Classification

Dr. Huiping Cao

Outline

- Classification problem
- Artificial neurons
  - Intuition
  - Formal definitions
- Perceptron learning
- Perceptron implementation in Python

Classifications

- Binary classification problem: $y \in \{0, 1\}$
  - 0 (or -1): Negative class (e.g., benign tumors, normal emails, normal transactions)
  - 1: Positive class (e.g., malignant tumors, spam emails, fraudulent transactions)
- Multiclass classification problem: $y \in \{0, 1, 2, 3, 4\}$
  - Power system disturbances (fault, generation loss, load switch on/off)
- Examples and datasets: UCI machine learning repository (https://archive.ics.uci.edu/ml/index.php)

Applications

- Well defined vs. Not well defined
- Example 1: a climatologist needs to monitor the air quality and give warnings if he foresees that there will be dust storms on a road segment. The data that we can have access to include the temperatures, wind speed, wind velocity, images taken every 15 or 20 minutes.
- Example 2: a rancher needs to monitor the health of his cattle. He sets some sensors on the cattle and record the animal's movement.
- Questions:
  - Can this problem be modeled as a classification problem?
  - If so, what are the class labels and data instances?
  - What issues that regular classification does not consider?

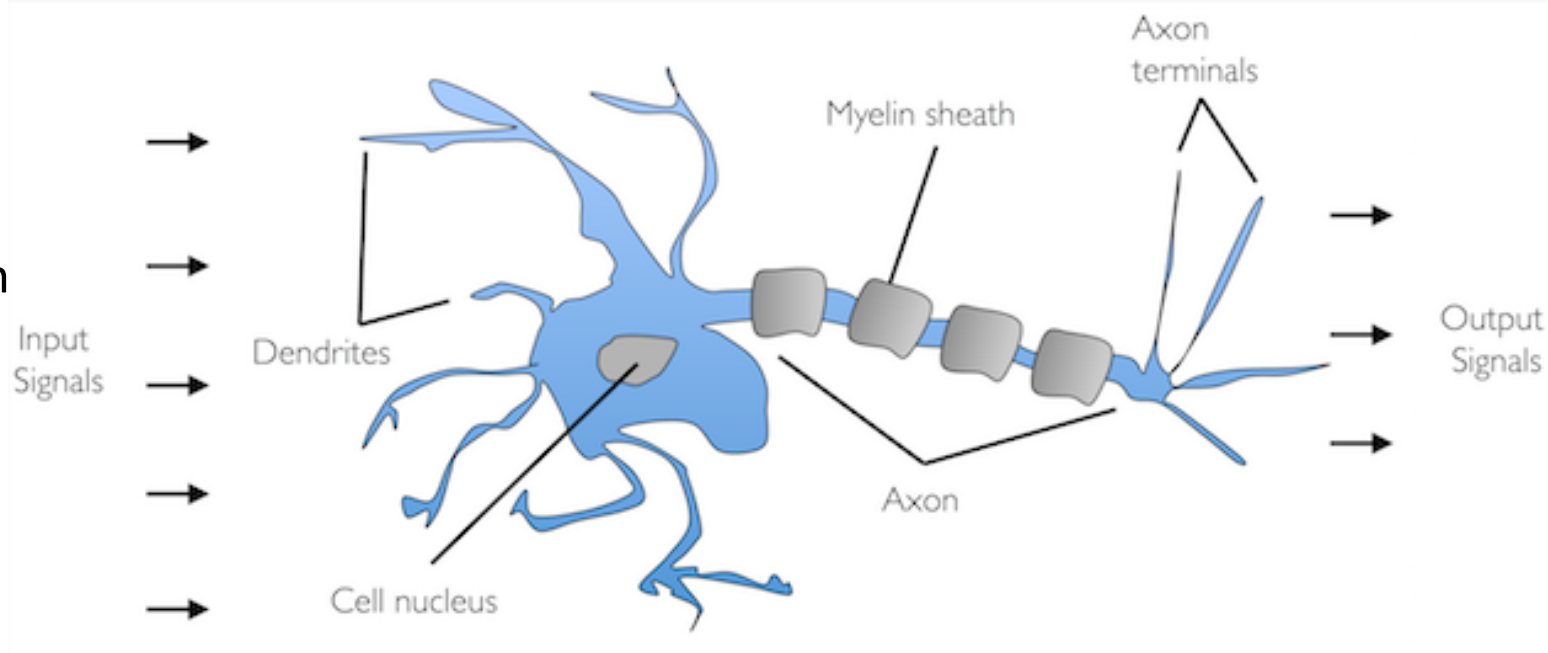
Artificial neurons – intuition and history

- The first concept of a simplified brain cell, McCullock-Pitts **(MCP) neuron**, in 1943*.
- Neurons are interconnected **nerve cells** in the brain. They are involved in the processing and transmitting of chemical and electrical signals.
- We are interested in replicating the **biological function**. The first step is to replicate the **biological structure**.

** A logical calculus of the ideas immanent in nervous activity, W. S. McCullock and W. Pitts, Bulletin of Mathematical Biophysics, 5(4): 115-133, 1943).*

Biological Structures

- **Neurons:** nerve cells
- **Axons**: strands of fibers, for linking neurons
- **Dendrites**: connects neurons and axons
- **Synapses**: the contact point between a dendrite and an axon
- **Logic gate** with binary outputs
  - multiple signals arrive at the dendrites
  - are integrated into the cell body
  - if the accumulated signal exceeds a certain threshold, an output signal is generated that will be passed on by the axon



Perceptron

- In 1957, Frank Rosenblatt published the first concept of the **perceptron learning** rule based on the MCP neuron model.
- In the context of classification, this algorithm can be used to predict if a sample belongs to one class or the other.
- An **algorithm to** automatically learn the optimal weight coefficients.
  - The coefficients are multiplied with the input features to decide whether a neuron fires or not.

Artificial neurons – formal definition

- Context of binary classification: 1 (positive class) and -1 (negative class)
- Task: given input **x,** predict its class label.
- Given **x** and **w**

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_m \end{bmatrix} \qquad \boldsymbol{w} = \begin{bmatrix} w_1 \\ w_2 \\ \dots \\ w_m \end{bmatrix}$$

- The **net input** is $z = \boldsymbol{w}^T \boldsymbol{x} = w_1 x_1 + w_2 x_2 + \cdots + w_m x_m$
- We define a decision function (a variant of a unit step function)

$$\varphi(z) = \begin{cases} 1 \; if \; z \geq \theta \\ -1 \; otherwise \end{cases}$$

Equation rewriting

- The **net input** is $z = \boldsymbol{w}^T \boldsymbol{x} = w_0 x_0 + w_1 x_1 + w_2 x_2 + \cdots + w_m x_m$
  - Where x_0=1 and w_0=$-\theta$
- We define a decision function $\varphi(z) = \begin{cases} 1 \; if \; z \geq 0 \\ -1 \; otherwise \end{cases}$
- **Bias unit**: $-\theta$

Perceptron learning rule

- Initialize the weights small random numbers
- For given number of iterations
  - For each training sample $\mathbf{x}^{(i)}$
    - Compute the output value $\hat{y}^{(i)}$
    - Update the weights $w_j := w_j + \Delta w_j$ simultaneously for all $j = 0, \cdots, m$
- **Epoch**: one pass over the training dataset is one epoch.

Weight update

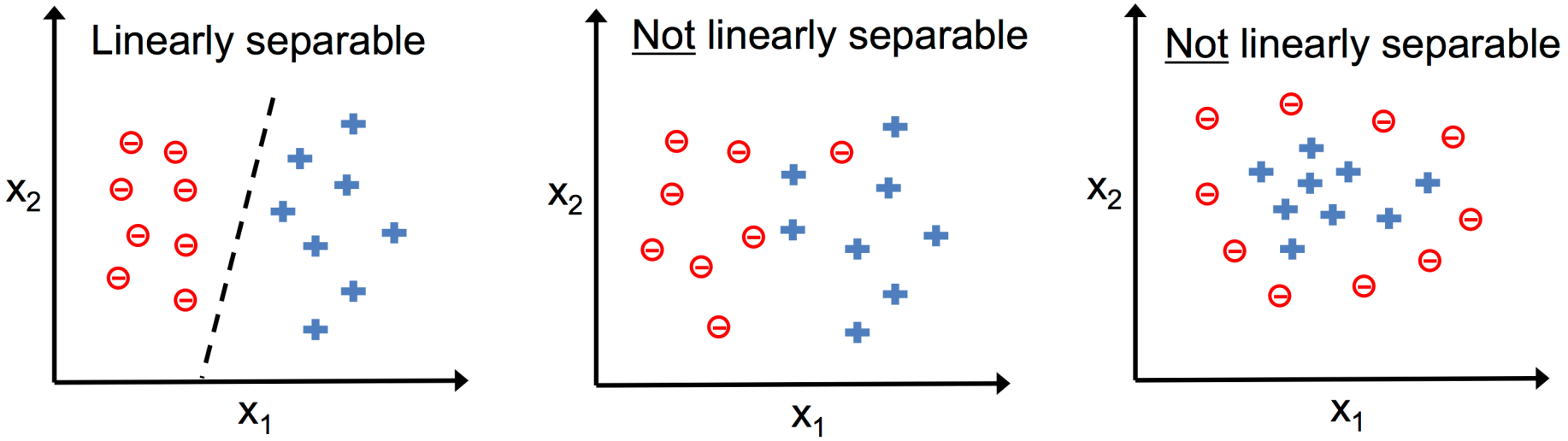
- The calculation of Δw_j: $\Delta w_j = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)x_j^{(i)}$
  - $\eta \in [0,1]$ is learning rate. It is typically a constant number
  - $y^{(i)}$ is the true class label of the i-th sample
  - $\hat{y}^{(i)}$ is the predicted class label of the i-th sample
- All m+1 weights are being updated simultaneously. For a m-dimensional dataset, the update can be written as
  - $\Delta w_0 = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)$ because $x_0^{(i)}$=1
  - $\Delta w_1 = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)x_1^{(i)}$
  - $\Delta w_2 = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)x_2^{(i)}$
  - …
  - $\Delta w_m = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)x_m^{(i)}$

More discussions about the perceptron learning rule

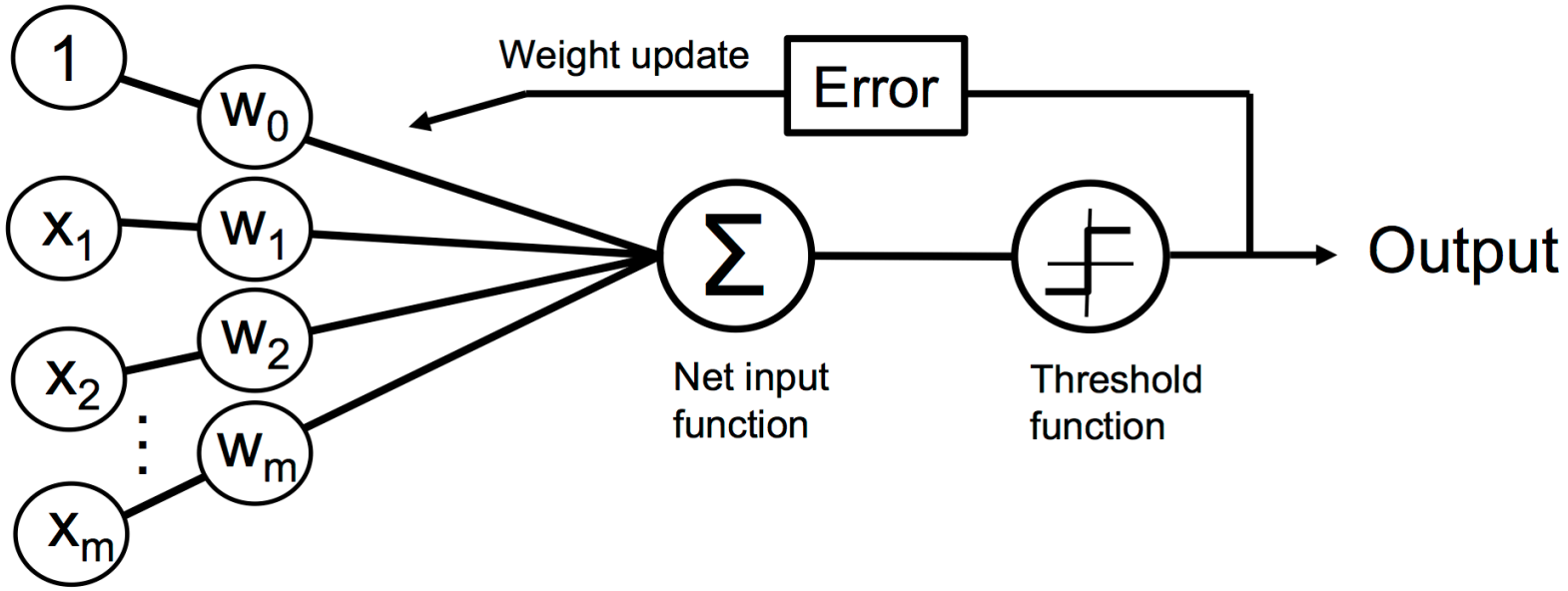
- Consider any example **x** = (x_1 , x_2 , … , x_m) (i.e., removing the instance index)
- If the perceptron predicts the class label **correctly**, the weights remain unchanged.
  - $\Delta w_j = \eta(1-1)x_j \ = 0$
  - $\Delta w_j = \eta((-1) \ -(-1))x_j \ = 0$
- If the perceptron predicts the class label **wrong**, the weights are being pushed **towards the direction of the target** class.
  - If class label 1 is predicted to be class label -1
    - $\Delta w_j = \eta(1-(-1))x_j \ = 2\eta x_j$
    - $w_j x_j = (w_j + \Delta w_j)\, x_j = w_j x_j + \Delta w_j \, x_j = w_j x_j + 2\eta x_j^2$ (next iteration)
  - Similarly, if class label -1 is predicted to be class label 1
    - $\Delta w_j = \eta((-1)-1)x_j \ = -2\eta x_j$, $w_j x_j = w_j x_j - 2\eta x_j^2$

Convergence & decision boundaries

- The **convergence** of the perceptron is only guaranteed if the two classes are linearly separable and the learning rate is sufficiently small.
- If the two classes can't be separated by a linear decision boundary, we can set a maximum number of epoch and/or a threshold for the number of tolerated misclassifications.
- **Decision boundary**: Three example of separable.



Perceptron model diagram



Implementation

- import numpy as np
- import pandas as pd

- **class** Perceptron (object) :

Parameters and attributes

Parameters

---

eta : float
  Learning rate (between 0.0 and 1.0)
n_iter : int
  Passes over the training dataset.
random_state: int
  Random number generator seed for random weight initialization.

Attributes

---

w_ : 1d-array
  Weights after fitting.
errors_ : list
  Number of misclassifications (updates) in each epoch.

Initialization

```
def _init_(self , eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter=n_iter
        self.random_state = random_state
```

Training

```
def fit(self, X, y):
        rgen = np.random.RandomSta
        self.w_ = rgen.normal(loc=0.0,
        self.errors_ = []

        for _ in range(self.n_iter):
            errors = 0
            for xi, target in zip(X, y):
                update = self.eta * (target - self.predict(xi))  # η(y^(i) − ŷ^(i))
                self.w_[1:] += update * xi
                self.w_[0] += update
                errors += int(update != 0.0)
            self.errors.append(errors)
        return self
```

The zip() function returns a zip object, which is an iterator of tuples where the first item in each passed iterator is paired together, and then the second item in each passed iterator are paired together etc. If the passed iterators have different lengths, the iterator with the least items decides the length of the new iterator.

Training

```
def fit(self, X, y):
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
        self.errors_ = []

        for _ in range(self.n_iter):
            errors = 0
            for xi, target in zip(X, y):
                update = self.eta * (target - self.predict(xi))
                self.w_[1:] += update * xi
                self.w_[0] += update
                errors += int(update != 0.0)
            self.errors.append(errors)
        return self
```

Training

```
def fit(self, X, y):
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
        self.errors_ = []

        for _ in range(self.n_iter):
            errors = 0
            for xi, target in zip(X, y):
                update = self.eta * (target - self.predict(xi))
                self.w_[1:] += update * xi
                self.w_[0] += update
                errors += int(update != 0.0)
            self.errors.append(errors)
        return self
```

numpy.random.RandomState: Random number generator.
numpy.random.normal: Draw random samples from a normal (Gaussian) distribution.

Prediction

```
def net_input(self, X):
        return np.dot(X, self.w_[1:]) + self.w_[0]

def predict(self, X):
        return np.where(self.net_input(X) >= 0.0, 1, -1)
```

numpy.dot(a, b, out=None): Dot product of two arrays.

numpy.where(condition[, x, y]) Return elements, either from x or y, depending on condition.

Train a perceptron model for Isis data

- **Binary class**: only use Setosa and Versicolor from the Iris dataset.
- Steps
  - Read the file to a Pandas *DataFrame* object
  - Extract the first 100 class labels that corresponds to 50 *Iris−setosa* and 50 *Iris−versicolor*.
  - Convert the class labels to two integer class labels 1 (versicolor) and -1 (Setosa) to a vector *y*.
  - (optional) plot the data
  - **Train our perceptron algorithm**
  - **(for verification) plot the misclassification error for each epoch to check whether the algorithm con- verges and finds a decision boundary**.

Train the model

- DataFrame **iloc** function : selection based on location
- DataFrame has the **values method**, which yields the corresponding NumPy representation.

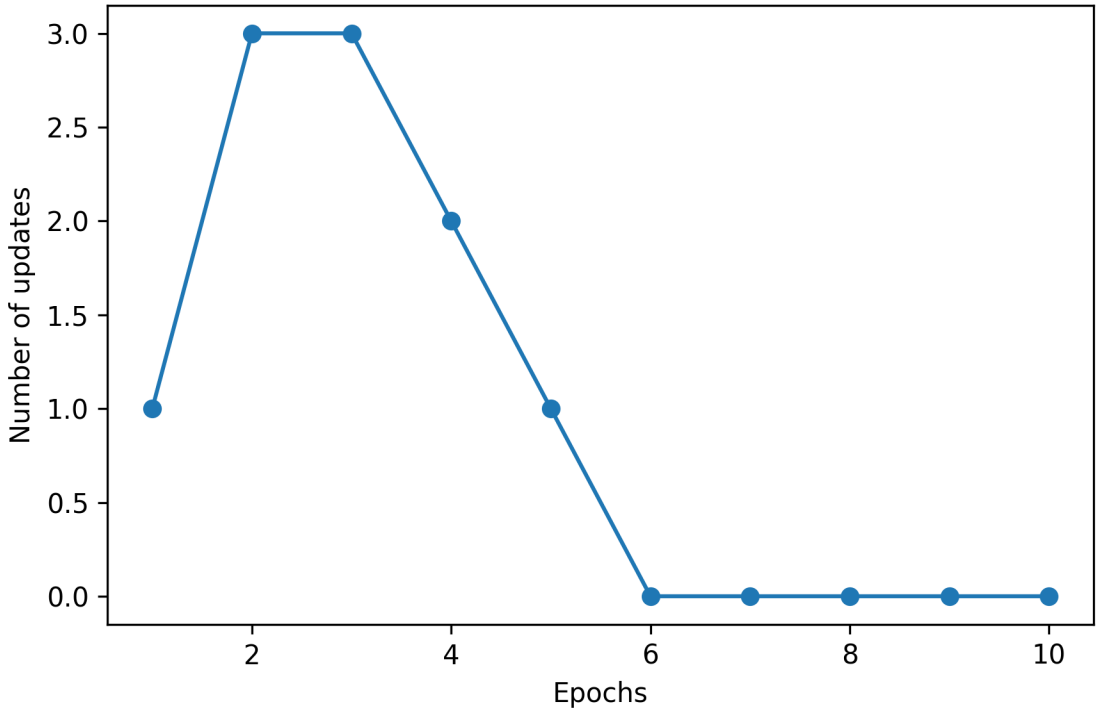
```
df = pd.read_csv('iris.data', header=None)

y = df.iloc[0:100, 4].values                # select setosa and versicolor
y = np.where(y == 'Iris-setosa', -1, 1)     # Convert the class labels to two integer
X = df.iloc[0:100, [0, 2]].values           # extract sepal length and petal length

ppn = Perceptron(eta=0.1, n_iter=10)
ppn.fit(X, y)

plt.plot(range(1, len(ppn.errors_) + 1), ppn.errors_, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Number of updates')
```

Errors in different iterations



Initialize the weights to be non-zero small values (S.S.)

- Zero $\boldsymbol{w}^{(0)}$ = 0 will leads to that "η affects only the scale of the weight vector, but not the direction"
- Let superscript (i) represents iteration
- Let $\boldsymbol{w}^{(0)}$ = 0
- Iteration 1:
  - $\hat{y}^{(1)} = \varphi\left(\boldsymbol{w}^{(0)^T}\boldsymbol{x}^{(1)}\right)$
  - $\Delta\boldsymbol{w}^{(1)} = \eta\left(y^{(1)} - \hat{y}^{(1)}\right)\boldsymbol{x}^{(1)}$
  - $\boldsymbol{w}^{(1)} = \boldsymbol{w}^{(0)} + \Delta\boldsymbol{w}^{(1)}$
- Iteration 2
  - $\hat{y}^{(2)} = \varphi\left(\boldsymbol{w}^{(1)^T}\boldsymbol{x}^{(2)}\right)$
  - $\Delta\boldsymbol{w}^{(2)} = \eta\left(y^{(2)} - \hat{y}^{(2)}\right)\boldsymbol{x}^{(2)}$
  - $\boldsymbol{w}^{(2)} = \boldsymbol{w}^{(1)} + \Delta\boldsymbol{w}^{(2)}$

(S.S.)

$$\Delta \boldsymbol{w}^{(2)} = \eta\left(y^{(2)} - \hat{y}^{(2)}\right)\boldsymbol{x}^{(2)}$$

$$= \eta\left(y^{(2)} - \varphi\left(\boldsymbol{w}^{(1)^T}\boldsymbol{x}^{(2)}\right)\right)\boldsymbol{x}^{(2)}$$

$$= \eta\left(y^{(2)} - \varphi\left(\left(\boldsymbol{w}^{(0)} + \Delta\boldsymbol{w}^{(1)}\right)^T\boldsymbol{x}^{(2)}\right)\right)\boldsymbol{x}^{(2)}$$

$$= \eta\left(y^{(2)} - \varphi\left(\left(\boldsymbol{w}^{(0)} + \eta\left(y^{(1)} - \hat{y}^{(1)}\right)\boldsymbol{x}^{(1)}\right)^T\boldsymbol{x}^{(2)}\right)\right)\boldsymbol{x}^{(2)}$$

$$= \eta\left(y^{(2)} - \varphi\left(\left(\boldsymbol{w}^{(0)} + \eta\left(y^{(1)} - \varphi\left(\boldsymbol{w}^{(0)^T}\boldsymbol{x}^{(1)}\right)\right)\boldsymbol{x}^{(1)}\right)^T\boldsymbol{x}^{(2)}\right)\right)\boldsymbol{x}^{(2)}$$

$$\hat{y}^{(2)} = \varphi\left(\boldsymbol{w}^{(1)^T}\boldsymbol{x}^{(2)}\right)$$

$$\boldsymbol{w}^{(1)} = \boldsymbol{w}^{(0)} + \Delta\boldsymbol{w}^{(1)}$$

$$\Delta \boldsymbol{w}^{(1)} = \eta\left(y^{(1)} - \hat{y}^{(1)}\right)\boldsymbol{x}^{(1)}$$

$$\hat{y}^{(1)} = \varphi\left(\boldsymbol{w}^{(0)^T}\boldsymbol{x}^{(1)}\right)$$

When $\boldsymbol{w}^{(0)^T} = \boldsymbol{0}$, $\boldsymbol{w}^{(0)^T}\boldsymbol{x}^{(1)} = \boldsymbol{0}$ and $\varphi(0)=\boldsymbol{1}$

$$\Delta \boldsymbol{w}^{(2)} = \eta\left(y^{(2)} - \varphi\left(\eta\left(y^{(1)} - 1\right)\boldsymbol{x}^{(1)^T}\boldsymbol{x}^{(2)}\right)\right)\boldsymbol{x}^{(2)}$$

Since $\eta>0$, $\varphi\left(\eta\left(y^{(1)} - 1\right)\boldsymbol{x}^{(1)^T}\boldsymbol{x}^{(2)}\right) = \varphi\left(\left(y^{(1)} - 1\right)\boldsymbol{x}^{(1)^T}\boldsymbol{x}^{(2)}\right)$

$$\Delta \boldsymbol{w}^{(2)} = \eta\left(y^{(2)} - \varphi\left(\left(y^{(1)} - 1\right)\boldsymbol{x}^{(1)^T}\boldsymbol{x}^{(2)}\right)\right)\boldsymbol{x}^{(2)}$$

Thus, $\Delta \boldsymbol{w}^{(i)} = \eta(a\ factor\ not\ affected\ by\ \eta)$

Summary

- Perceptron learning rule
- Implementation of perceptron model
- Utilize perceptron model