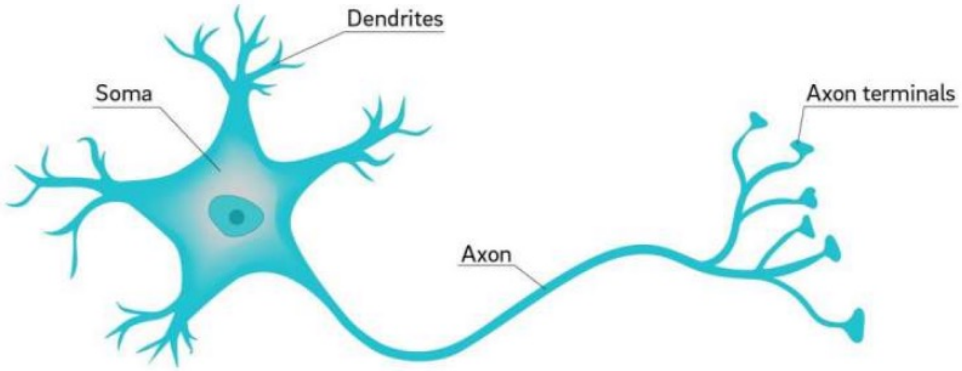
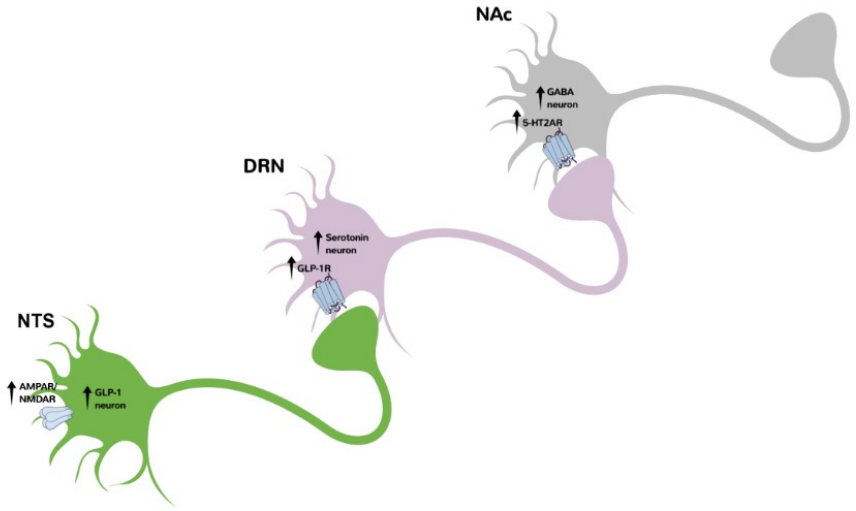
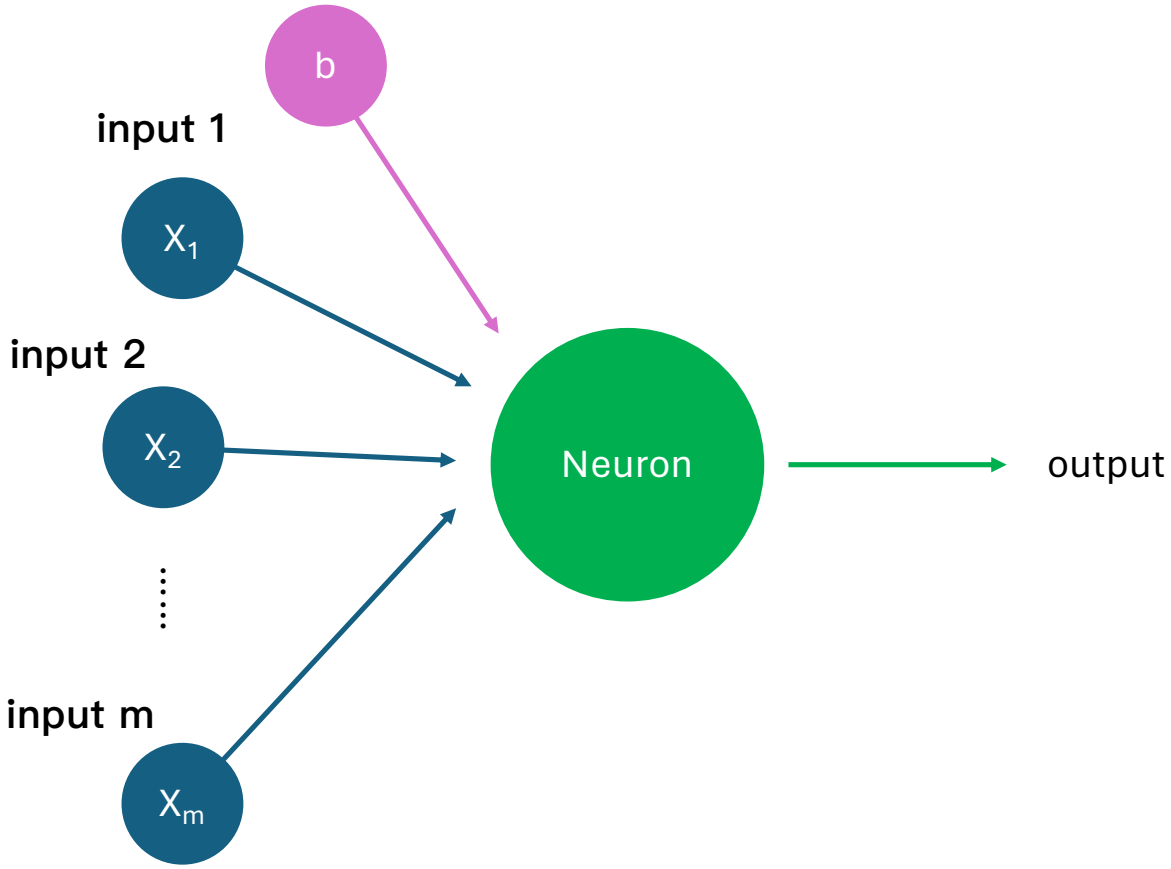
Neuron Network

Neuron

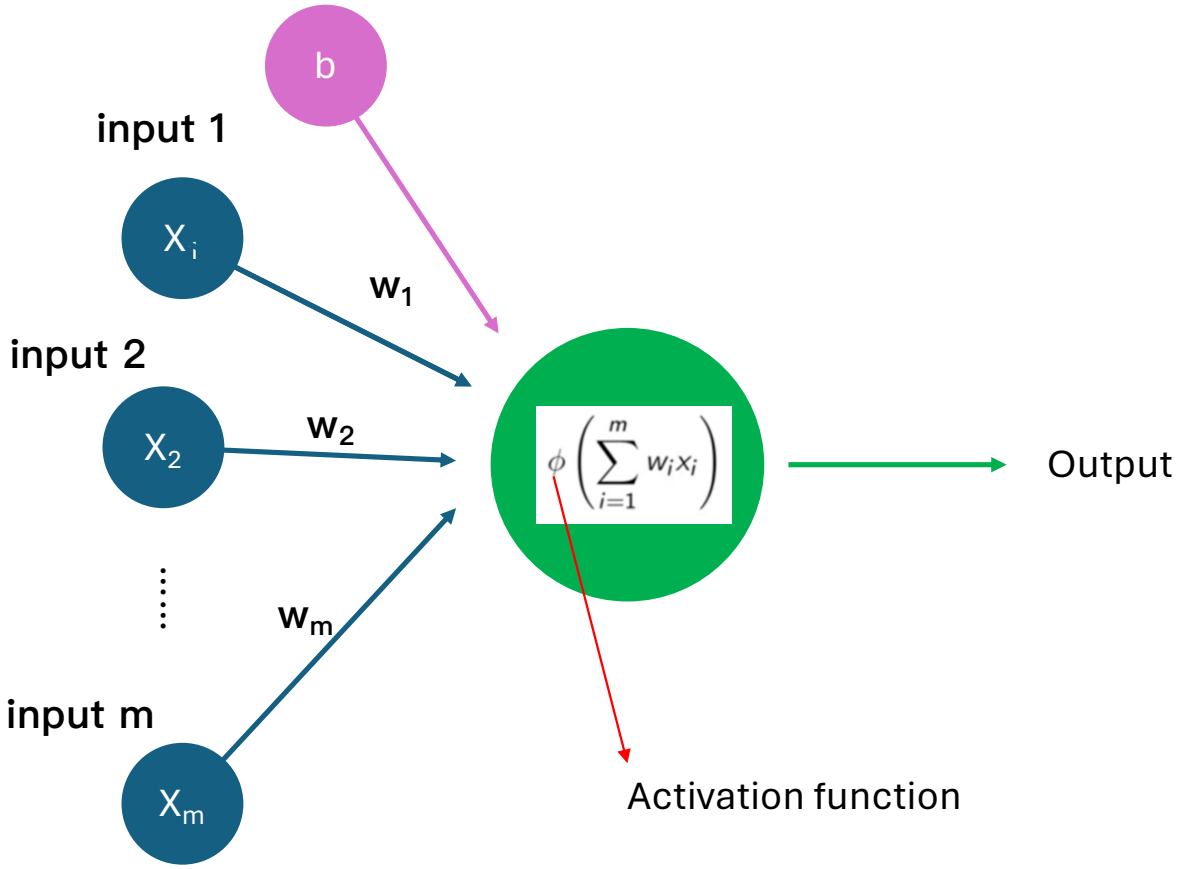

Dendrites
Soma
Axon terminals
Axon


NAc
GABA neuron
5-HT2AR
DRN
Serotonin neuron
GLP-1R
NTS
AMPAR
NMDAR
GLP-1 neuron

ANN (Artificial Neuron)



ANN (Artificial Neuron)



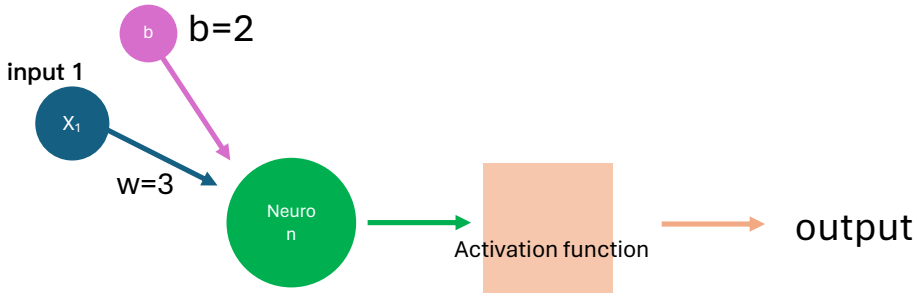
The parameters *w* and *b* are constant, obtained through model training.

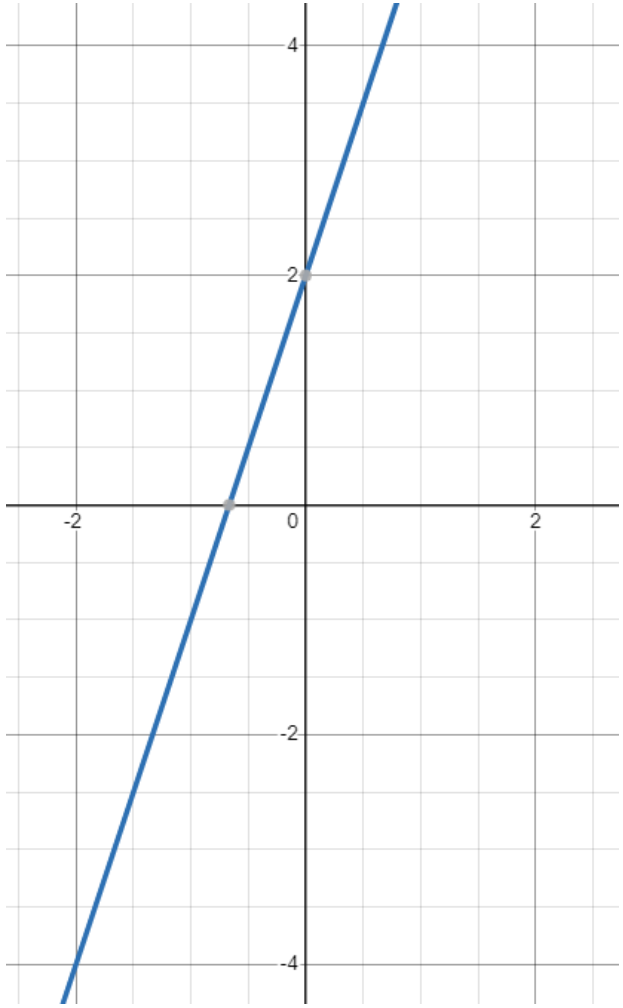
1 node neuron: 1 input

$$z = activate\left[\sum_{i=1}^{n} w_i x_i + b\right] = activate(wx + b)$$

let w=3, b=2

z = 3x+2



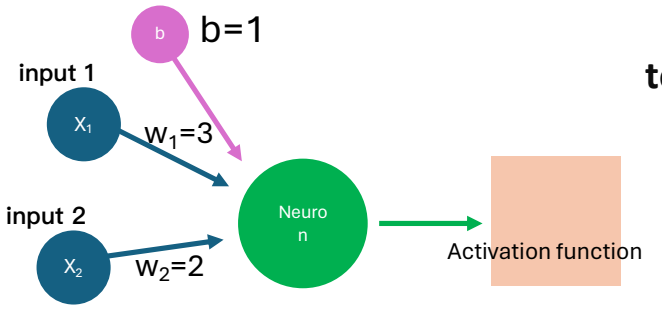


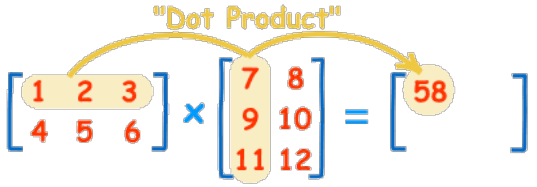
1 node neuron: 2 input

$$z = activate\left[\sum_{i=1}^{n} w_i x_i + b\right] = activate(wx + b)$$

tensor_w tensor_x

torch.matmul(*w*, *x*)





$$\begin{pmatrix} 2 & 1 \\ 4 & 3 \end{pmatrix} \times \begin{pmatrix} 1 & 2 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 3 & 4 \\ 7 & 8 \end{pmatrix}$$

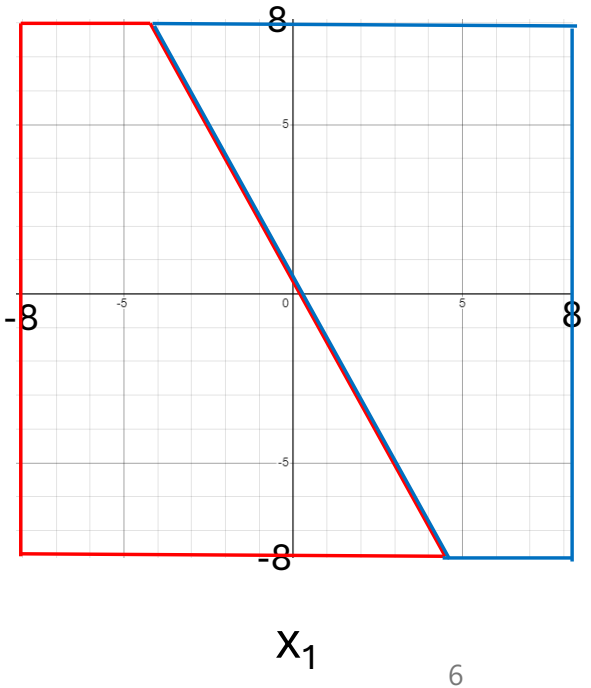
let $w_1=3$, $w_2=2$, $b=1$

$z = 3x_1+2x_2 + 1$

$$z = activate\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} [3 \quad 2] + 1\right)$$

$$f(x) = \begin{cases} 1, if\ x < 0 \\ 0, if\ x \geq 0 \end{cases}$$

$3x_1+2x_2$



Perceptron

感知器

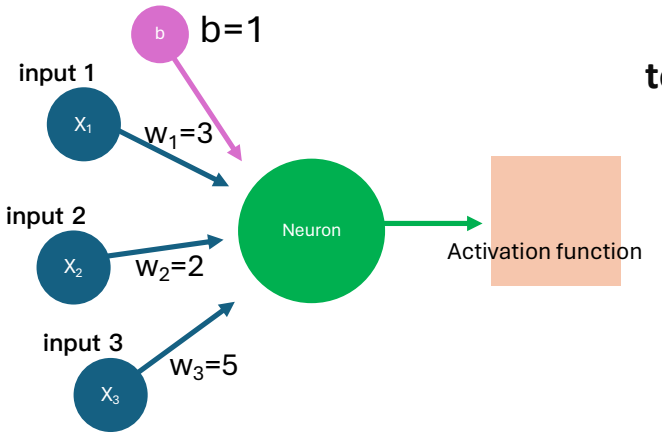
A perceptron is a single-layer neural network that serves as a fundamental building block of more complex neural network architectures.

1 node neuron: 3 input

$$z = activate\left[\sum_{i=1}^{n} w_i x_i + b\right] = activate(wx + b)$$

tensor_w tensor_x

torch.matmul(*w, x*)



let $w_1=3$, $w_2=2$, $w_3=5$, b=1

$z = 3x_1+2x_2 + 5x_3+ 1$

$$z = activate\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} [3 \quad 2 \quad 5] + 1\right)$$

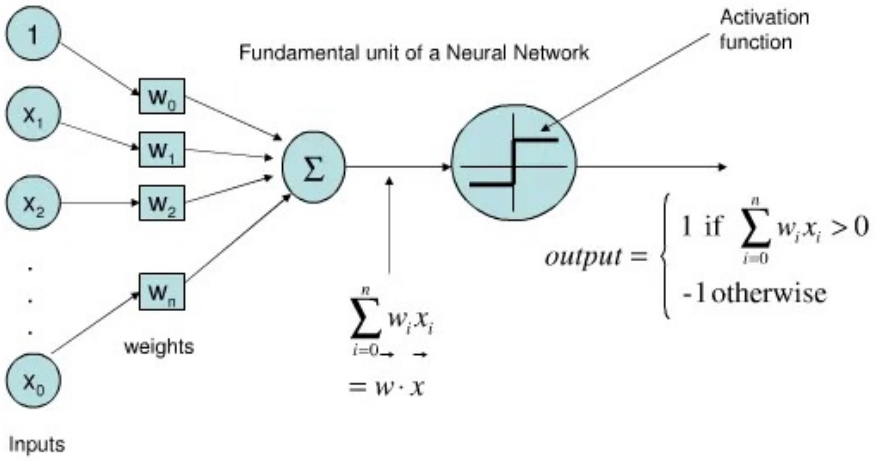
$$f(x) = \begin{cases} 1, if\ x < 0 \\ 0, if\ x \geq 0 \end{cases}$$

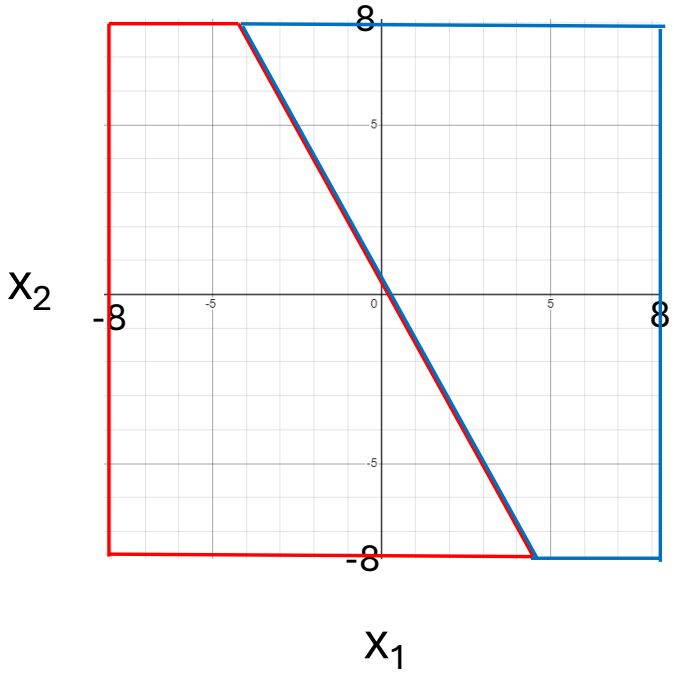
$3x_1+2x_2$

Perceptron

感知器

A perceptron is a single-layer neural network that serves as a fundamental building block of more complex neural network architectures.


1
X1
X2
Xn
X0
W0
W1
W2
Wn
Σ
Fundamental unit of a Neural Network
Activation function
weights
Inputs
$\sum_{i=0}^{n} w_i x_i = w \cdot x$
$output = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$


8
5
-8
-5
0
5
8
-5
-8
X2
X1

Activate functions

- Sigmoid Function (Logistic Function):

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

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

- It compresses the input to the range between 0 and 1, commonly used in binary classification tasks, although less prevalent in deep neural networks due to the vanishing gradient problem.

Activate functions

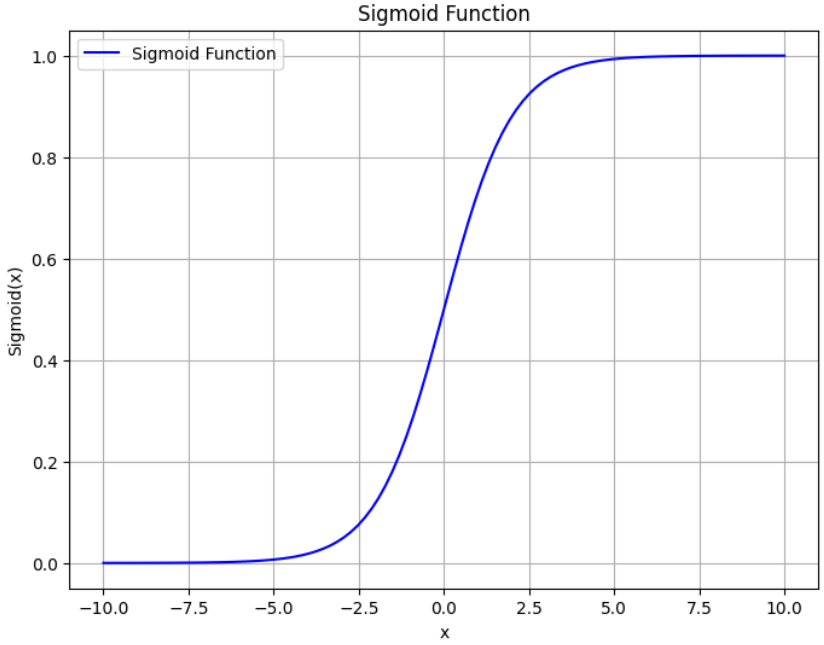
```python
import numpy as np
import matplotlib.pyplot as plt

# Define the sigmoid function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Generate x values (start, stop, and num)
x_values = np.linspace(-10, 10, 100)

# Calculate corresponding y values using the sigmoid function
y_values = sigmoid(x_values)

# Plot the sigmoid function
plt.figure(figsize=(8, 6))
plt.plot(x_values, y_values, label='Sigmoid Function', color='b')
plt.title('Sigmoid Function')
plt.xlabel('x')
plt.ylabel('Sigmoid(x)')
plt.grid(True)
plt.legend()
plt.show()
```



Activate functions

- Tanh Function (Hyperbolic Tangent Function):

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

```
def tanh(x):
    return np.tanh(x)
```

- It compresses the input to the range between -1 and 1, similar to the sigmoid but with a wider output range, also facing the vanishing gradient problem.

Activate functions

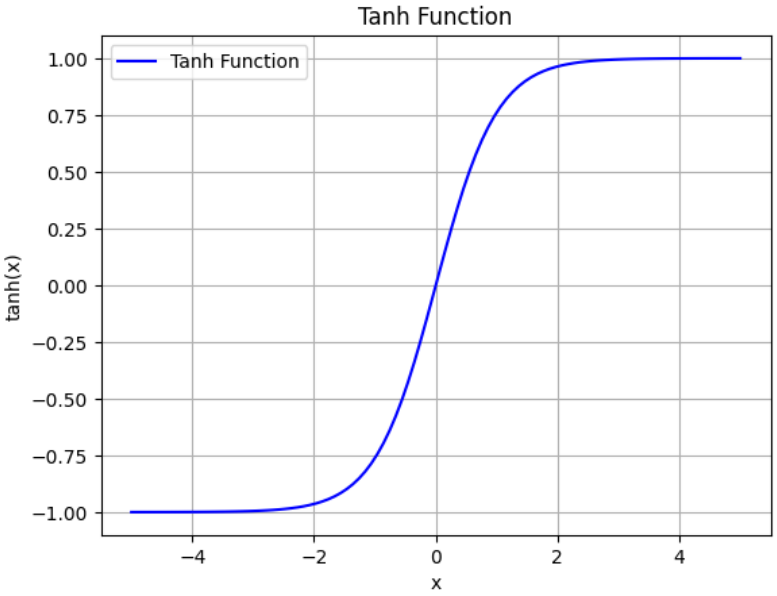
- ReLU Function (Rectified Linear Unit):

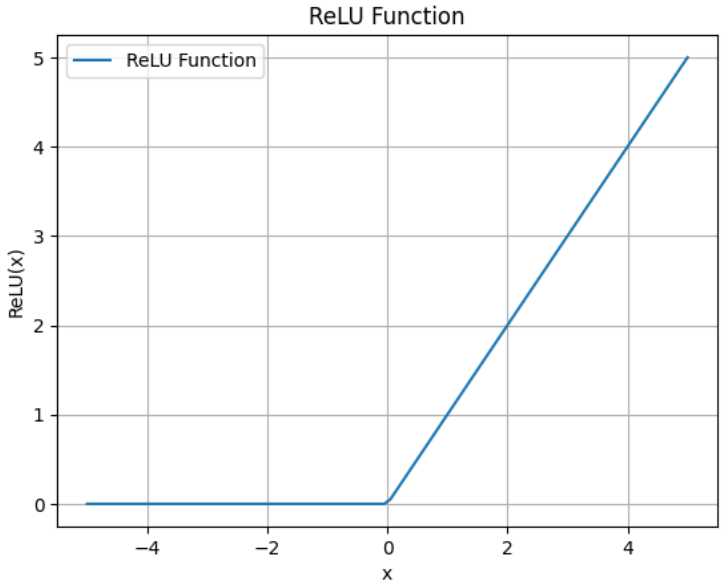
$$\mathrm{ReLU}(x) = \max(0, x)$$

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

- It outputs the input if it's greater than 0; otherwise, it outputs 0.
- ReLU is simple and efficient but suffers from the "dying ReLU" problem (neurons stuck at 0 output) and the issue of zero gradient for negative inputs.

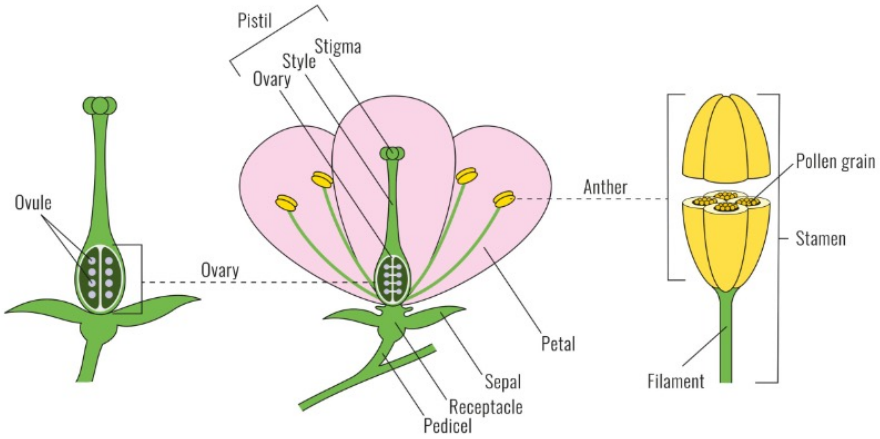
Exercise


Tanh Function
Tanh Function
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
tanh(x)
−4
−2
0
2
4
x


ReLU Function
ReLU Function
5
4
3
2
1
0
ReLU(x)
−4
−2
0
2
4
x

Perceptron-iris-classification

- The features used in this training example are the petal(花瓣) and sepal(花萼) length of the seratosa and verisolor iris species. This is the format of the chart with the given data. Each flower has 50 data samples and thus 50 data points for the perceptron to train and classify from.



Type	Petal Length	Sepal Length
Versicolor	6.7	3.0
Seratosa	6.3	2.5
...	...	...

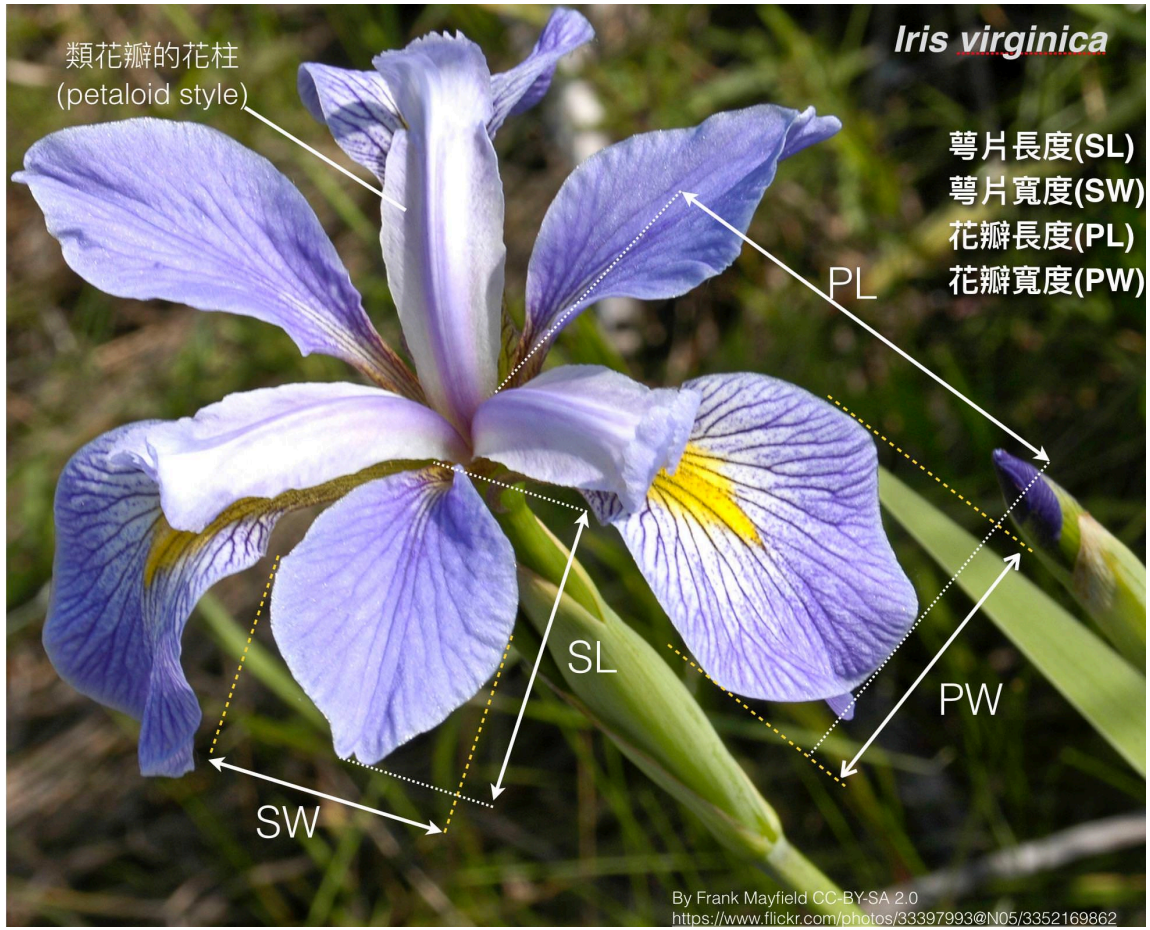
Iris setosa - Wikipedia

Data and code from:
https://github.com/Gregory-Eales/perceptron-iris-classification

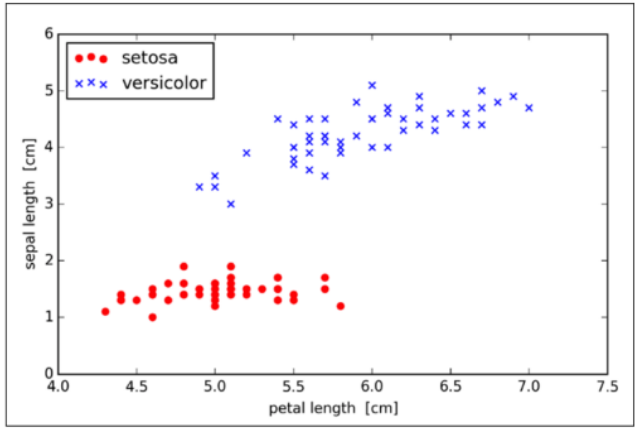
Dataset

id	Sepal Length	Sepal Width	Petal Length	Petal Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
51	7.0	3.2	4.7	1.4	versicolor
52	6.4	3.2	4.5	1.5	versicolor
101	6.3	3.3	6.0	2.5	virginica
102	5.8	2.7	5.1	1.9	virginica

perceptron_iris.py

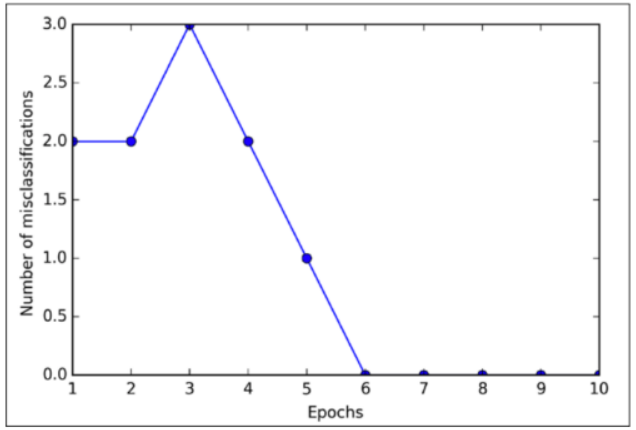


Data and code from:
https://github.com/Gregory-Eales/perceptron-iris-classification



This is a graph showing plots of petal length vs sepal length. The setosa species is show as red dots and the versicolor species is shown as blue 'X's. There is a clear gap between the two species on the graph which should be easy for a perceptron model to identify.

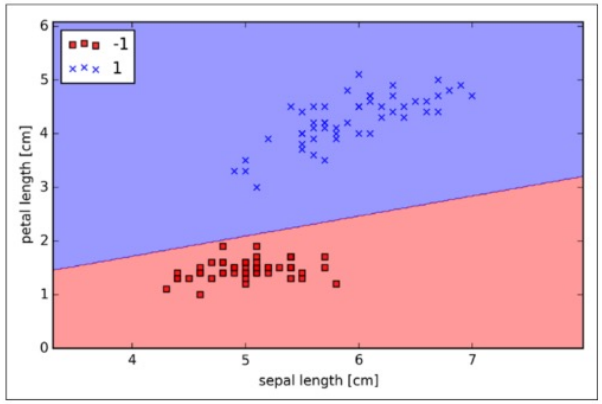
Training:



This graph shows the number of misclassifications the perceptron model makes for each epoch. The number of misclassifications begins to drop after the third epoch with an error rate of 0% only after five epochs.

Data and code from:
https://github.com/Gregory-Eales/perceptron-iris-classification

Results:



This graph shows the decision regions for each species in their corresponding color labeling. The linear distinction makes a clear separation between the species, showing that the perceptron has successfully learned how to predict the species type given the sepal and petal length.

Data and code from:
https://github.com/Gregory-Eales/perceptron-iris-classification

perceptron_predict_iris.py

```
# Creation of the main perceptron object.
# 建立主要感知器物件。
class Perceptron(object):
    # Initiating the learning rate and number of iterations.
    # 初始化學習速率和迭代次數。
    def __init__(self, Learn_Rate=0.5, Iterations=10):
        self.learn_rate = Learn_Rate
        self.Iterations = Iterations
        self.errors = []
        self.weights = np.zeros(1 + x.shape[1])

    # Defining fit method for model training.
    # 定義適合模型訓練的方法。
    def fit(self, x, y):
        self.weights = np.zeros(1 + x.shape[1])
        for i in range(self.Iterations):
            error = 0
            for xi, target in zip(x, y):
                update = self.learn_rate * (target - self.predict(xi))
                self.weights[1:] += update*xi
                self.weights[0] += update
                error += int(update != 0)
            self.errors.append(error)
        return self

    # Net Input method for summing the given matrix inputs and their corresponding weights.
    # 淨輸入方法用於加總給定矩陣輸入及其相應的權重。
    def net_input(self, x):
        return np.dot(x, self.weights[1:]) + self.weights[0]

    # Predict method for predicting the classification of data inputs.
    # 預測方法用於預測數據輸入的分類。
    def predict(self, x):
        return np.where(self.net_input(x) >= 0.0, 1, -1)
```

perceptron_predict_iris.py

```python
# Creation of the main perceptron object.
# 建立主要感知器物件。
class Perceptron(object):
    # Initiating the learning rate and number of iterations.
    # 初始化學習速率和迭代次數。
    def __init__(self, Learn_Rate=0.5, Iterations=10):
        self.learn_rate = Learn_Rate
        self.Iterations = Iterations
        self.errors = []
        self.weights = np.zeros(1 + x.shape[1])

    # Defining fit method for model training.
    # 定義適合模型訓練的方法。
    def fit(self, x, y):
        self.weights = np.zeros(1 + x.shape[1])
        for i in range(self.Iterations):
            error = 0
            for xi, target in zip(x, y):
                update = self.learn_rate * (target - self.predict(xi))
                self.weights[1:] += update*xi
                self.weights[0] += update
                error += int(update != 0)
            self.errors.append(error)
        return self

    # Net Input method for summing the given matrix inputs and their corresponding weights.
    # 淨輸入方法用於加總給定矩陣輸入及其相應的權重。
    def net_input(self, x):
        return np.dot(x, self.weights[1:]) + self.weights[0]

    # Predict method for predicting the classification of data inputs.
    # 預測方法用於預測數據輸入的分類。
    def predict(self, x):
        return np.where(self.net_input(x) >= 0.0, 1, -1)
```

perceptron_predict_iris_sklearnPerceptron.py

```python
# Model training and evaluation.
# 模型訓練與評估。
classifier = Perceptron(alpha=0.01, max_iter=50, random_state=0)
classifier.fit(x, y)
```

sklearn.linear_model.Perceptron

class `sklearn.linear_model.Perceptron`(*, *penalty=None, alpha=0.0001, l1_ratio=0.15, fit_intercept=True, max_iter=1000, tol=0.001, shuffle=True, verbose=0, eta0=1.0, n_jobs=None, random_state=0, early_stopping=False, validation_fraction=0.1, n_iter_no_change=5, class_weight=None, warm_start=False*) [source]

Linear perceptron classifier.

The implementation is a wrapper around `SGDClassifier` by fixing the `loss` and `learning_rate` parameters as:

```
SGDClassifier(loss="perceptron", learning_rate="constant")
```

Other available parameters are described below and are forwarded to `SGDClassifier`.

Read more in the User Guide.

Parameters:

penalty : *{'l2','l1','elasticnet'}, default=None*
The penalty (aka regularization term) to be used.

alpha : *float, default=0.0001*
Constant that multiplies the regularization term if regularization is used.

l1_ratio : *float, default=0.15*
The Elastic Net mixing parameter, with `0 <= l1_ratio <= 1`. `l1_ratio=0` corresponds to L2 penalty, `l1_ratio=1` to L1. Only used if `penalty='elasticnet'`.

New in version 0.24.

fit_intercept : *bool, default=True*
Whether the intercept should be estimated or not. If False, the data is assumed to be already centered.

max_iter : *int, default=1000*
The maximum number of passes over the training data (aka epochs). It only impacts the behavior in the `fit` method, and not the `partial_fit` method.

New in version 0.19.

tol : *float or None, default=1e-3*
The stopping criterion. If it is not None, the iterations will stop when (loss > previous_loss - tol).

New in version 0.19.

shuffle : *bool, default=True*
Whether or not the training data should be shuffled after each epoch.

verbose : *int, default=0*
The verbosity level.

n_jobs : *int, default=None*
The number of CPUs to use to do the OVA (One Versus All, for multi-class problems) computation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See Glossary for more details.

random_state : *int, RandomState instance or None, default=0*
Used to shuffle the training data, when `shuffle` is set to `True`. Pass an int for reproducible output across multiple function calls. See Glossary.

early_stopping : *bool, default=False*
Whether to use early stopping to terminate training when validation score is not improving. If set to True, it will automatically set aside a stratified fraction of training data as validation and terminate training when validation score is not improving by at least `tol` for `n_iter_no_change` consecutive epochs.

New in version 0.20.

validation_fraction : *float, default=0.1*
The proportion of training data to set aside as validation set for early stopping. Must be between 0 and 1. Only used if early_stopping is True.

New in version 0.20.

n_iter_no_change : *int, default=5*
Number of iterations with no improvement to wait before early stopping.

New in version 0.20.

class_weight : *dict, {class_label: weight} or "balanced", default=None*
Preset for the class_weight fit parameter.

Weights associated with classes. If not given, all classes are supposed to have weight one.

The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`.

warm_start : *bool, default=False*
When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. See the Glossary.

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Perceptron.html

scikit-learn digits dataset

- The "Digits" dataset in scikit-learn, often referred to as load_digits, is a classic machine learning dataset used for learning and demonstrating algorithms in the field of pattern recognition and classification.
- It consists of 8x8 pixel images of handwritten digits from 0 to 9. Each image is represented as a flattened array of 64 pixels, and the dataset contains a total of 1,797 samples.

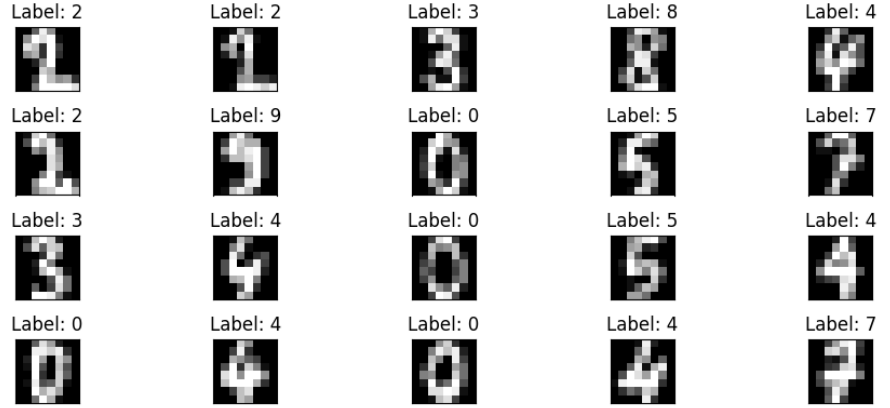
```python
# Importing dependencies.
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_digits

# Load the digits dataset
digits = load_digits()

# Plotting 20 random images
fig, axes = plt.subplots(4, 5, figsize=(10, 4), subplot_kw={'xticks':[], 'yticks':[]})
for i, ax in enumerate(axes.flat):
    index = np.random.randint(0, len(digits.images))
    ax.imshow(digits.images[index], cmap='gray')
    ax.set_title(f"Label: {digits.target[index]}")

plt.tight_layout()
plt.show()

```

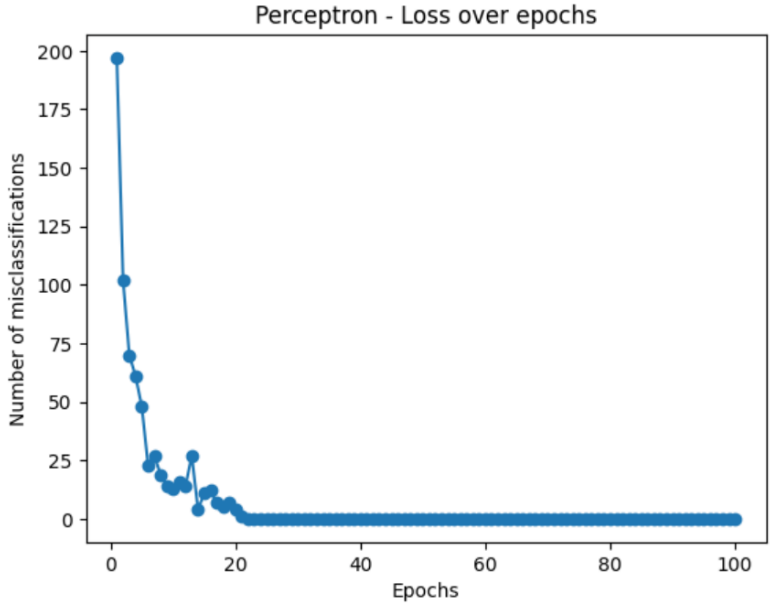


Perceptron-digits-classification

```
7 class MultiClassPerceptron(object):
8     def __init__(self, Learn_Rate=0.01, Iterations=100, n_classes=10):
9         self.learn_rate = Learn_Rate
10        self.Iterations = Iterations
11        self.n_classes = n_classes
12        self.weights = None
13        self.errors_ = []  # Storing the number of misclassifications for each epoch
14
15    def fit(self, X, y):
16        self.weights = np.zeros((self.n_classes, X.shape[1] + 1))
17        for _ in range(self.Iterations):
18            errors = 0
19            for xi, target in zip(X, y):
20                actual_class = self.predict(xi)
21                if actual_class != target:
22                    self.weights[target, 1:] += self.learn_rate * xi
23                    self.weights[target, 0] += self.learn_rate
24                    self.weights[actual_class, 1:] -= self.learn_rate * xi
25                    self.weights[actual_class, 0] -= self.learn_rate
26                    errors += 1
27            self.errors_.append(errors)
28
29    def net_input(self, X, w):
30        return np.dot(X, w[1:]) + w[0]
31
32    def predict(self, X):
33        results = np.dot(self.weights[:, 1:], X) + self.weights[:, 0]
34        return np.argmax(results)
```

perceptron_predict_digits.py

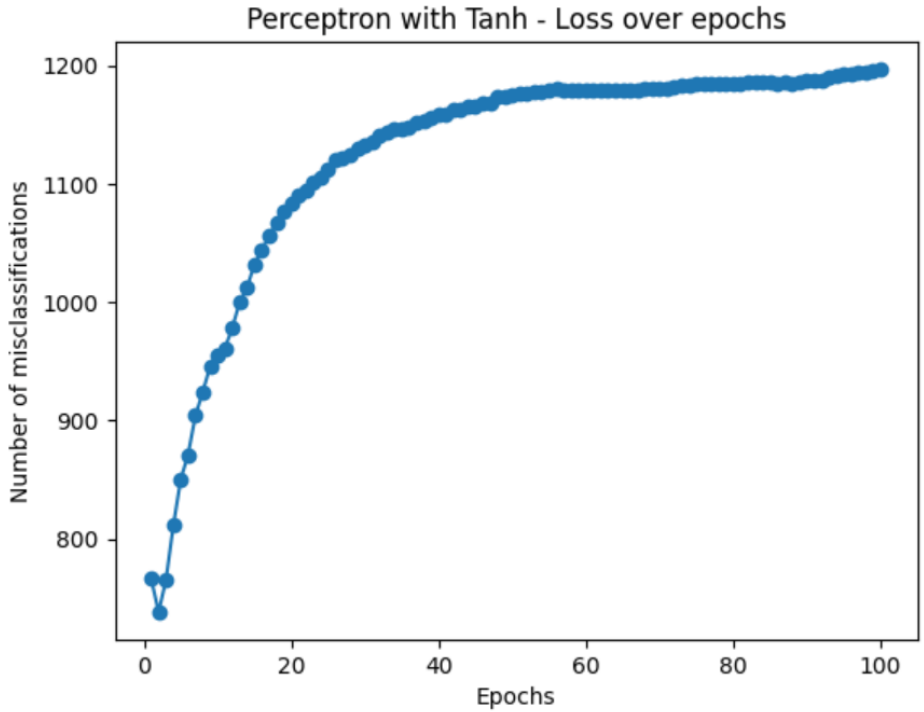
Perceptron-digits-classification




```
Accuracy: 94.81%
```

Perceptron-digits-classification: tanh

```
33      def activation(self, z):
34          return np.tanh(z)
```

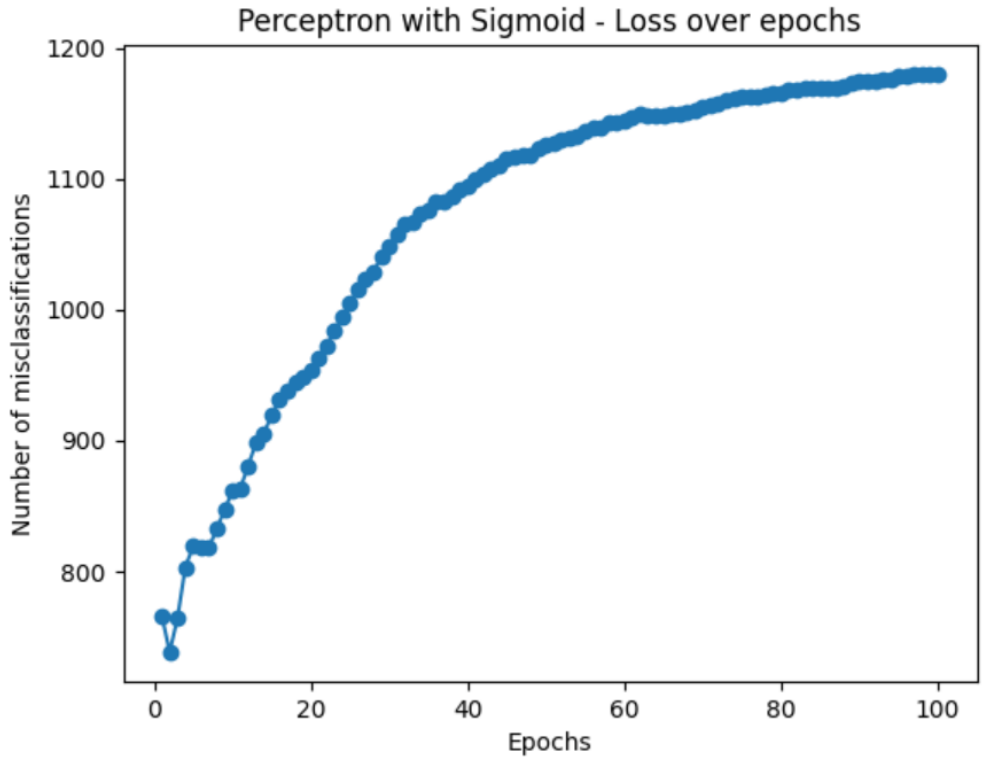



```
Accuracy: 5.93%
```

perceptron_predict_digits_tanh.py

Perceptron-digits-classification: sigmoid

```
33      def activation(self, z):
34          return 1 / (1 + np.exp(-z))
```




```
Accuracy: 6.85%
```

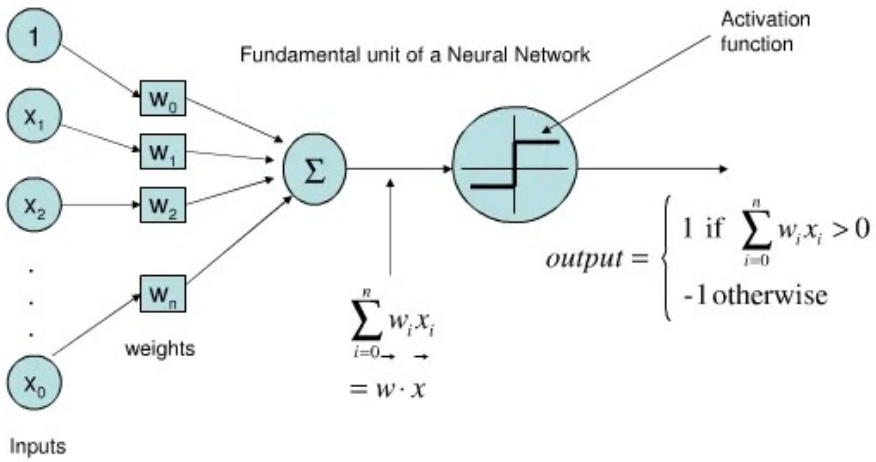
perceptron_predict_digits_ sigmoid.py

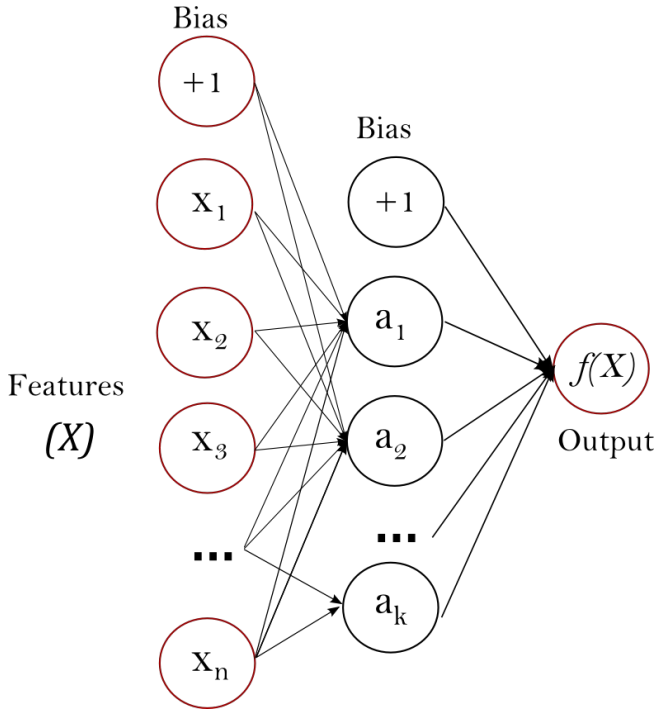
Fully Connected Network (FCN)

- A Fully Connected Network (FCN), also known as a Multilayer Perceptron (MLP,多層感知器), is a fundamental neural network architecture.
- In a fully connected network, each neuron is connected to every neuron in the previous layer, forming a fully connected structure.
- Each neuron receives inputs from the previous layer, computes a weighted sum, and generates an output through an activation function.
- Fully connected networks typically consist of multiple hidden layers and an output layer, and are trained using the backpropagation algorithm.

- 全連接網路（Fully Connected Network）是一種基本的神經網路架構，也被稱為多層感知器（Multilayer Perceptron, MLP）。
- 在全連接網路中，每個神經元都與前一層的每個神經元相連接，形成了完全連接的架構。每個神經元接收來自前一層的輸入，進行加權求和，並通過激活函數產生輸出。
- 全連接網路通常由多個隱藏層和一個輸出層組成，並使用反向傳播算法來訓練模型。

Fully Connected Network (FCN)





Exercise:

Increase accuracy of digits-classification

Submission requirements:

1. source code (predict.py)
2. PDF documents
   Explaining your strategy.
   Show the outputs (before and after)
3. Upload to e-learning before 3/29 14:10