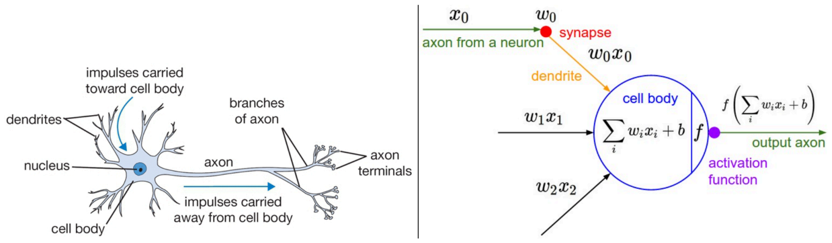
Introduction to the Perceptron

1 What is a Perceptron?

A perceptron is a type of artificial neuron or the simplest form of a neural network, serving as the foundational building block for more complex neural networks. Introduced by Frank Rosenblatt in 1957, it mimics the way a neuron in the brain works.



2 How does a Perceptron Work?

A perceptron takes multiple input values, each multiplied by a weight, sums them up, and produces a single binary output based on whether the sum is above a certain threshold.

2.1 Mathematical Model

The perceptron decision is based on these formulas:

$$f(x) = sign(w \cdot x + b)$$

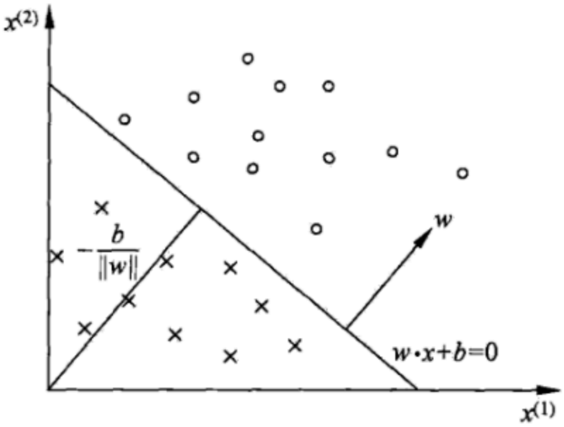
$$sign(x) = \begin{cases} +1 \text{ if } x \geq 0 \\ -1 \text{ if } x < 0 \end{cases}$$

Components of a Perceptron :

1. **Inputs (x)**: The features or data points provided to the perceptron.
2. **Weights (w)**: Coefficients determining the importance of each input.
3. **Bias (b)**: An extra input (always 1) with its own weight, allowing the activation function to shift, fitting the data better.
4. **Activation Function(sign function)**: Decides whether the neuron should activate, typically a step function for basic perceptrons.

2.1.1 Geometric interpretation

The perceptron divides the input space into two parts with a decision boundary, where for the inputs x that satisfy $w \cdot x + b > 0$, the predicted label is $+1$, and for those that satisfy $w \cdot x + b < 0$, the predicted label is -1. The decision boundary of the perceptron is a **hyperplane** defined by $w \cdot x + b = 0$, which separates the **space** into two parts. On one side of the hyperplane, all the points are classified into one class, and on the other side, they are classified into another class. The direction of the vector w is perpendicular to the decision boundary, and the distance from the origin to the boundary is determined by the bias b.



3 How to train a Perceptron

3.1 Loss function

Since we want to classify all points correctly, a natural idea is to directly use the total number of misclassified points as a loss function :

$$\begin{aligned} L_1(w,b) &= \sum_{i=1}^{N} -y_i * f(x_i)\ (when\ \ y_i * f(x_i) < 0) \\ &= \sum_{i=1}^{N} -y_i * sign(w \cdot x_i + b)\ (when\ \ y_i * sign(w \cdot x_i + b) < 0) \end{aligned}$$

The sign function cannot be differentiated, so the perceptron algorithm chooses the **total distance from the misclassification point to the hyperplane S** as the loss function:

$$\begin{aligned} L_2(w,b) &= \sum_{i=1}^{N} \frac{1}{||w||} |w \cdot x_i + b|\ (when\ \ y_i * (w \cdot x_i + b) < 0) \\ &= -\frac{1}{||w||} \sum_{i=1}^{N} y_i * (w \cdot x_i + b)\ (when\ \ y_i * (w \cdot x_i + b) < 0) \end{aligned}$$

Regardless of coefficient $\frac{1}{\|w\|}$, we can get the final loss function of perceptron :

$$L_3(w,b) = -\sum_{i=1}^{N} y_i * (w \cdot x_i + b)\ \ (when\ \ y_i * (w \cdot x_i + b) < 0)$$

The derivation of Loss function

$$\nabla_w L(w,b) = -\sum_{x_i \in M} y_i x_i$$

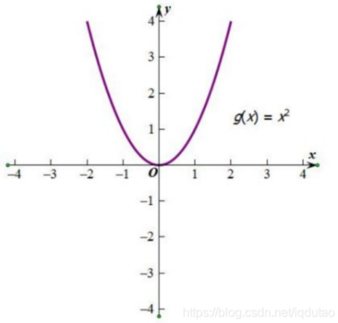
$$\nabla_b L(w,b) = -\sum_{x_i \in M} y_i$$

3.2 Gradient descent

3.2.1 Introdcution

How to **update the parameters** by minimizing the Loss function?

if we want to minimize the function $g(x) = x^2$, we can just use its derivation $g'(x) = 0$



how about a more complex function?

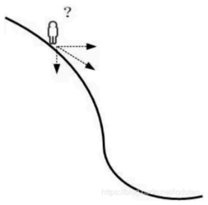
$$f(x,y) = x^3 - 2x^2 + e^{xy} - y^3 + 10y^2 + 100\sin(xy)$$

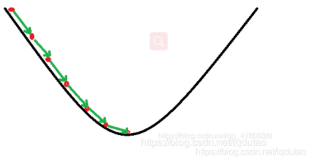
We take the partial derivatives of x and y respectively and set them to 0 to obtain the following system of equations:

$$\begin{cases} 3x^2 - 4x + ye^{xy} + 100y\cos(xy) = 0 \\ xe^{xy} - 3y^2 + 20y + x\cos(xy) = 0 \end{cases}$$

Hard to solve!

When implemented in engineering, the **iterative method** is usually adopted, which starts from an initial point, repeatedly uses some rules to move from the next point, and constructs such a series until it converges to the point where the gradient is 0, that is, the gradient descent algorithm.





Imagine a scenario where a person is stuck on a mountain (where the red circle is in the picture) and needs to get down from the mountain (to find the lowest point of the mountain, which is the valley), but the fog on the mountain is heavy at this time, resulting in poor visibility. Therefore, the path down the mountain cannot be determined, and he must use the information around him to find the path down the mountain. At this point, he can use the **gradient descent algorithm to help him down the mountain**. To be specific, take his current position as a benchmark, **find the steepest**

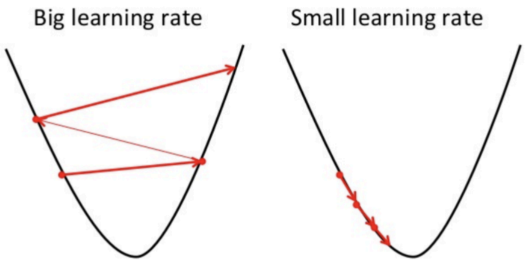
place in this position (gradient), and then walk in the direction of the height of the mountain, and then every distance, and repeat the same method, and finally successfully reach the valley.

The equation below describes what the gradient descent algorithm does: b is the next position of our climber, while a represents his current position. The minus sign refers to the minimization part of the gradient descent algorithm. The gamma in the middle is a waiting factor and the gradient term $\bigtriangledown f(a)$ is simply the direction of the steepest descent.

$$b = a - \gamma \bigtriangledown f(a)$$

3.2.2 Select appropriate Learning Rate

For the gradient descent algorithm to reach the local minimum we must set the learning rate **to an appropriate value**, which is neither too low nor too high. This is important because if the steps it takes are too big, it may not reach the local minimum because it bounces back and forth between the convex function of gradient descent (see left image below). If we set the learning rate to a very small value, gradient descent will eventually reach the local minimum but that may take a while (see the right image)



3.2.3 Types of Gradient Descent

- **Batch gradient descent**, also called vanilla gradient descent, calculates the error for each example **within the training dataset**, but only after all training examples have been evaluated does the model get updated. This whole process is like a cycle and it's called a training epoch.

  Some advantages of batch gradient descent are its computational efficiency: it produces a stable error gradient and a stable convergence. Some disadvantages are that the stable error gradient can sometimes result in a state of convergence that isn't the best the model can achieve. It also **requires the entire training dataset to be in memory** and available to the algorithm.

  ```
  for i in range(nb_epochs):
    params_grad = evaluate_gradient(loss_function,data,params)
    params = params - learning_rate * params_grad
  ```

- **Stochastic gradient descent** (SGD), by contrast, does this for **each training example within the dataset**, meaning it updates the parameters for each training example one by one. Depending on the problem, this can make SGD faster than batch gradient descent. One advantage is the frequent updates allow us to have a pretty detailed rate of improvement.

  The frequent updates, however, are more computationally expensive than the batch gradient descent approach. Additionally, the frequency of those updates can result in noisy gradients, which may cause the error rate to jump around instead of slowly decreasing.

  ```
  for i in range(nb_epochs):
     np.random.shuffle(data)
     for example in data:
        params_grad = evaluate_gradient(loss_functon,example,params)
        params = params - learning_rate * params_grad
  ```

- **Mini-batch gradient descent**, is the go-to method since it's a combination of the concepts of SGD and batch gradient descent. It simply **splits the training dataset into small batches and performs an update for each of those batches**. This creates a balance between the robustness of stochastic gradient descent and the efficiency of batch gradient descent.

  ```
  for i in range(nb_epochs):
     np.random.shuffle(data)
     for batch in get_batches(data,batch_size=50):
         params_grad = evaluate_gradient(loss_function,batch,params)
         params = params - learning_rate * params_grad
  ```

3.3 The "standard" algorithm

Given a training set $D = \{(x_i, y_i)\}, x_i \in \mathbb{R}^N, y_i \in \{-1, 1\}$

1. Initialize $\boldsymbol{w} = 0 \in \mathbb{R}^n$
2. For epoch = 1 … T:

   a. Compute the predictions of Perceptron of the whole training set.

   b. Compute the gradient of the loss function with respect to $\boldsymbol{w}$:

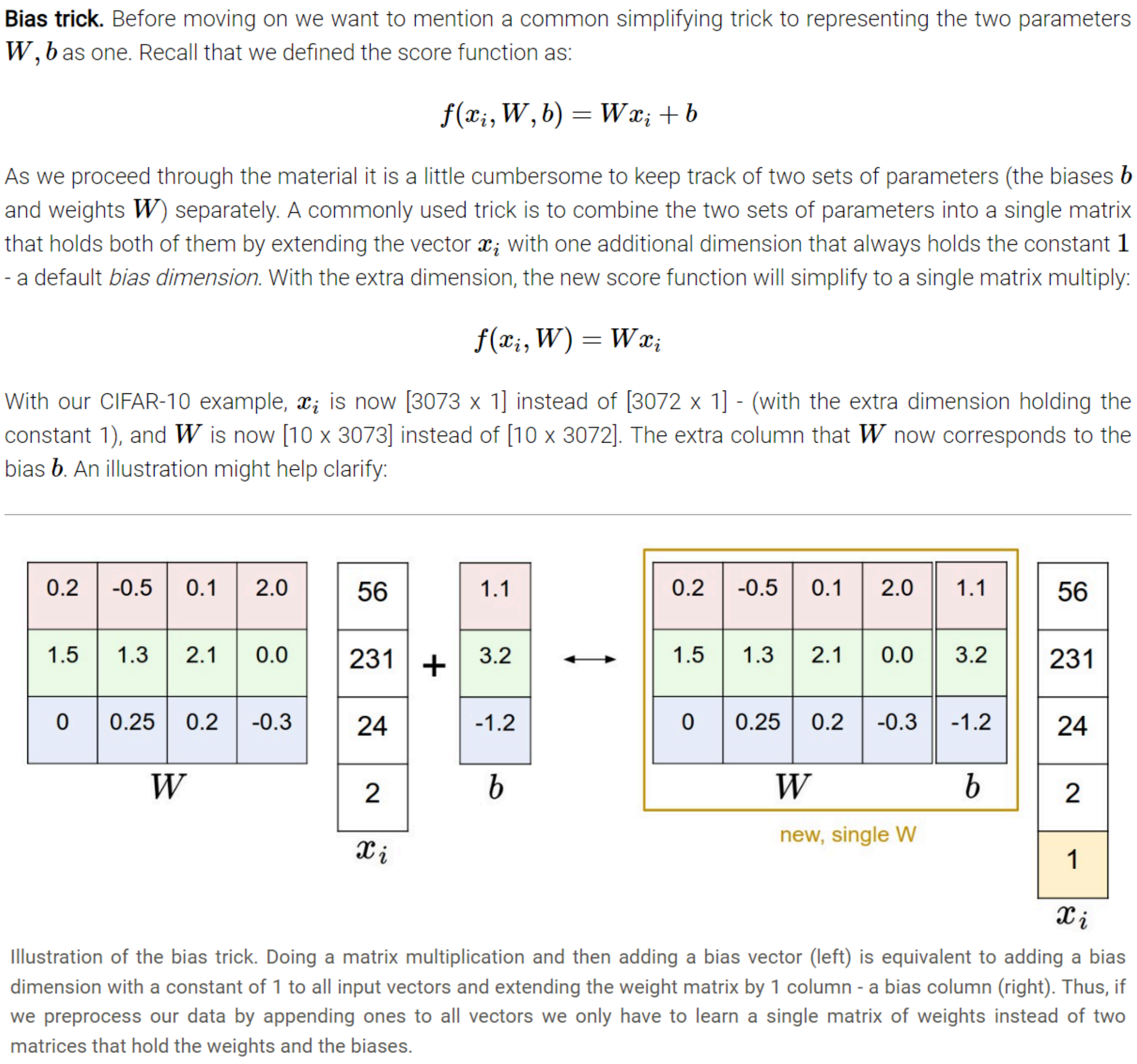
   $$gradient = -\frac{1}{N}\sum(x_i \cdot y_i),\ \ for\ sample_i : p_i * y_i < 0$$

   where p_i is i th prediction, y_i is the related ground truth of sample i, N is the number of misclassification points.

   c. Update $w \leftarrow w - lr * gradient$
3. Return $\boldsymbol{w}$

How to turn W&b into W

https://cs231n.github.io/linear-classify



Bias trick. Before moving on we want to mention a common simplifying trick to representing the two parameters W, b as one. Recall that we defined the score function as:

$$f(x_i, W, b) = W x_i + b$$

As we proceed through the material it is a little cumbersome to keep track of two sets of parameters (the biases b and weights W) separately. A commonly used trick is to combine the two sets of parameters into a single matrix that holds both of them by extending the vector x_i with one additional dimension that always holds the constant 1 - a default *bias dimension*. With the extra dimension, the new score function will simplify to a single matrix multiply:

$$f(x_i, W) = W x_i$$

With our CIFAR-10 example, x_i is now [3073 x 1] instead of [3072 x 1] - (with the extra dimension holding the constant 1), and W is now [10 x 3073] instead of [10 x 3072]. The extra column that W now corresponds to the bias b. An illustration might help clarify:

$$\underset{W}{\begin{bmatrix} 0.2 & -0.5 & 0.1 & 2.0 \\ 1.5 & 1.3 & 2.1 & 0.0 \\ 0 & 0.25 & 0.2 & -0.3 \end{bmatrix}} \underset{x_i}{\begin{bmatrix} 56 \\ 231 \\ 24 \\ 2 \end{bmatrix}} + \underset{b}{\begin{bmatrix} 1.1 \\ 3.2 \\ -1.2 \end{bmatrix}} \longleftrightarrow \underset{\text{new, single W}}{\boxed{\underset{W \qquad\qquad b}{\begin{bmatrix} 0.2 & -0.5 & 0.1 & 2.0 & 1.1 \\ 1.5 & 1.3 & 2.1 & 0.0 & 3.2 \\ 0 & 0.25 & 0.2 & -0.3 & -1.2 \end{bmatrix}}}} \underset{x_i}{\begin{bmatrix} 56 \\ 231 \\ 24 \\ 2 \\ 1 \end{bmatrix}}$$

Illustration of the bias trick. Doing a matrix multiplication and then adding a bias vector (left) is equivalent to adding a bias dimension with a constant of 1 to all input vectors and extending the weight matrix by 1 column - a bias column (right). Thus, if we preprocess our data by appending ones to all vectors we only have to learn a single matrix of weights instead of two matrices that hold the weights and the biases.