

# **Natural Language Processing with Pytorch**

## Build Intelligent Language Applications Using Deep Learning

Stephen CUI

April 20, 2023

# Contents

<b>1</b>	<b>Foundational Components of Neural Networks</b>	<b>3</b>
1.1	The Perceptron: The Simplest Neural Network . . . . .	3
1.2	Activation Functions . . . . .	3
1.2.1	Sigmoid . . . . .	3
1.2.2	Tanh . . . . .	4
1.2.3	ReLU . . . . .	4
1.2.4	Softmax . . . . .	4
1.3	Loss Functions . . . . .	5
1.3.1	Mean Squared Error Loss . . . . .	5
1.3.2	Categorical Cross-Entropy Loss . . . . .	5
1.3.3	Binary Cross-Entropy Loss . . . . .	5
1.4	Diving Deep into Supervised Training . . . . .	5
1.4.1	Constructing Toy Data . . . . .	5
1.4.2	Putting It Together: Gradient-Based Supervised Learning . . . . .	6

# Chapter 1

## Foundational Components of Neural Networks

### 1.1 The Perceptron: The Simplest Neural Network

Each perceptron unit has an input ( $x$ ), an output ( $y$ ), and three “knobs” : a set of weights ( $w$ ), a bias ( $b$ ), and an activation function ( $f$ ). The weights and the bias are learned from the data, and the activation function is handpicked depending on the network designer’s intuition of the network and its target outputs. Mathematically, we can express this as follows:

$$y = f(\mathbf{w}x + \mathbf{b})$$

It is usually the case that there is more than one input to the perceptron. We can represent this general case using vectors. That is,  $\mathbf{x}$ , and  $\mathbf{w}$  are vectors, and the product of  $\mathbf{w}$  and  $\mathbf{x}$  is replaced with a dot product:

$$y = f(\mathbf{w}\mathbf{x} + \mathbf{b})$$

essentially, a perceptron is a composition of a linear and a nonlinear function. The linear expression  $\mathbf{w}\mathbf{x} + \mathbf{b}$  is also known as an *affine transform*.

### 1.2 Activation Functions

Activation functions are nonlinearities introduced in a neural network to capture complex relationships in data.

#### 1.2.1 Sigmoid

The sigmoid function saturates (i.e., produces extreme valued outputs) very quickly and for a majority of the inputs. This can become a problem because it can lead to the gradients becoming either zero or diverging to an overflowing floating-point value. These phenomena are also known as *vanishing gradient problem* and *exploding gradient problem*, respectively. As a consequence, it is rare to see sigmoid units used in neural networks other than at the output, where the squashing property allows one to interpret outputs as probabilities.

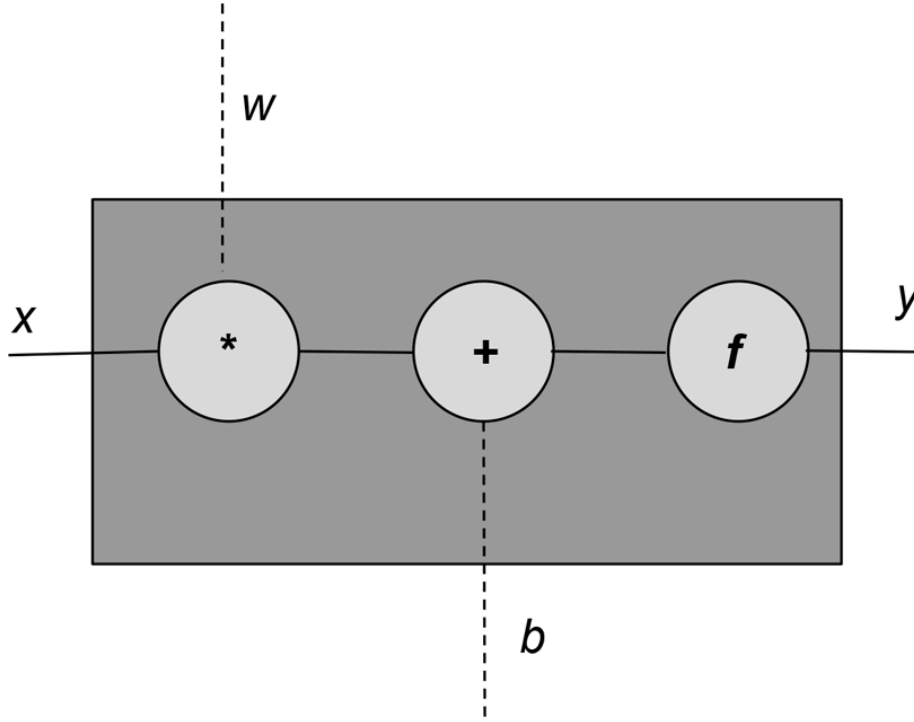


Figure 1.1: The computational graph for a perceptron with an input ( $x$ ) and an output ( $y$ ). The weights ( $w$ ) and bias ( $b$ ) constitute the parameters of the model.

### 1.2.2 Tanh

The tanh activation function is a cosmetically different variant of the sigmoid.

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

### 1.2.3 ReLU

ReLU (pronounced ray-luh) stands for *rectified linear unit*. This is arguably the most important of the activation functions.

$$f(x) = \max(0, x)$$

The clipping effect of ReLU that helps with the vanishing gradient problem can also become an issue, where over time certain outputs in the network can simply become zero and never revive again. This is called the “dying ReLU” problem. To mitigate that effect, variants such as the Leaky ReLU and Parametric ReLU (PReLU) activation functions have proposed, where the leak coefficient  $a$  is a learned parameter.

$$f(x) = \max(x, ax)$$

### 1.2.4 Softmax

Another choice for the activation function is the softmax. Like the sigmoid function, the softmax function squashes the output of each unit to be between 0 and 1.

$$\text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^k \exp(x_j)}$$

This transformation is usually paired with a probabilistic training objective, such as categorical cross entropy.

## 1.3 Loss Functions

### 1.3.1 Mean Squared Error Loss

For regression problems for which the network's output ( $\hat{y}$ ) and the target ( $y$ ) are continuous values, one common loss function is the mean squared error (MSE):

$$L_{MSE}(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^n (y - \hat{y})^2$$

The MSE is simply the average of the squares of the difference between the predicted and target values. There are several other loss functions that you can use for regression problems, such as mean absolute error (MAE) and root mean squared error (RMSE), but they all involve computing a real-valued distance between the output and target.

### 1.3.2 Categorical Cross-Entropy Loss

The categorical cross-entropy loss is typically used in a multiclass classification setting in which the outputs are interpreted as predictions of class membership probabilities. The target ( $y$ ) is a vector of  $n$  elements that represents the true multinomial distribution over all the classes. If only one class is correct, this vector is a one-hot vector. The network's output ( $\hat{y}$ ) is also a vector of  $n$  elements but represents the network's prediction of the multinomial distribution. Categorical cross entropy will compare these two vectors ( $y, \hat{y}$ ) to measure the loss:

$$L_{cross\_entropy}(y, \hat{y}) = - \sum_i y_i \log(\hat{y}_i)$$

如果类索引是 $[0, C)$ 的范围，这里 $C$ 是类别的数量，如果`ignore_index`指定的情况下，交叉熵损失函数接收这个类索引（这个索引未必在类范围内）。这种情况下未衰减的损失可以用下式描述：

$$l(x, y) = L = \{l_1, \dots, l_N\}^T, l_n = -w_{y_n} \log \frac{\exp(x_{n, y_n})}{\sum_{c=1}^C \exp(x_{n, c})} I(y_n \neq \text{ignore\_index}) \quad (1.1)$$

式中 $x$ 是输入， $y$ 是目标， $w$ 是权重， $C$ 是类别数量， $N$ 是minibatch的跨度，更多查看PyTorch的[CrossEntropyLoss](#)。

### 1.3.3 Binary Cross-Entropy Loss

Sometimes, our task involves discriminating between two classes—also known as binary classification. For such situations, it is efficient to use the binary cross-entropy (BCE) loss.

$$l(x, y) = L = \{l_1, \dots, l_N\}^T, l_n = -w_{y_n} [y_n \log x_n + (1 - y_n) * \log(1 - x_n)] \quad (1.2)$$

## 1.4 Diving Deep into Supervised Training

### 1.4.1 Constructing Toy Data

#### Choosing an optimizer

The PyTorch library offers several choices for an optimizer. Stochastic gradient descent (SGD) is a classic algorithm of choice, but for difficult optimization problems, SGD has convergence issues, often leading to poorer models. The current preferred alternative are adaptive optimizers, such as Adagrad or Adam, which use information about updates over time.

### 1.4.2 Putting It Together: Gradient-Based Supervised Learning

Let's take a look at how this gradient-stepping algorithm looks. First, any bookkeeping information, such as gradients, currently stored inside the model object is cleared with a function named `zero_grad()`. Then, the model computes outputs (`y_pred`) given the input data (`x_data`). Next, the loss is computed by comparing model outputs (`y_pred`) to intended targets (`y_target`). This is the supervised part of the supervised training signal. The PyTorch loss object (criterion) has a function named `backward()` that iteratively propagates the loss backward through the computational graph and notifies each parameter of its gradient. Finally, the optimizer (opt) instructs the parameters how to update their values knowing the gradient with a function named `step()`.

In the literature, and also in this notes, the term minibatch is used interchangeably with batch to highlight that each of the batches is significantly smaller than the size of the training data; for example, the training data size could be in the millions, whereas the minibatch could be just a few hundred in size.