Neural Networks

Jerry Peng

1 Dropout

In a machine learning model, if the model has too many parameters and too few training samples, the trained model will easily suffer from overfitting. When training neural networks, we often encounter the problem of over-fitting. In order to solve the over-fitting problem, the method of model integration is generally adopted, that is, multiple models are trained and combined. At this time, the time-consuming training model becomes a big problem. Not only is it time-consuming to train multiple models, but it is also time-consuming to test multiple models.

Dropout can be used as an option for training deep neural networks. In each training batch, overfitting can be significantly reduced by ignoring half of the feature detectors (letting half of the hidden layer nodes have values 0). This approach can reduce the interaction between feature detectors (hidden layer nodes), which means that some detectors depend on other detectors to function. Dropout simply means: during forward propagation, we let the activation value of a certain neuron stop working with a certain probability p. This can make the model more generalizable because it will not rely too much on certain local characteristics.

1.1 How does it work?

Suppose we want to train such a neural network, where the input is x and the output is y. The normal process is: we first propagate x forward through the network, and then propagate the error back to determine how to update the parameters for the network to learn. After using Dropout, the process becomes as follows:

1. First, half of the hidden neurons in the network are randomly (temporarily) deleted, and the input and output neurons remain unchanged.
2. Then the input x is forward propagated through the modified network, and then the obtained loss result is back propagated through the modified network. After a small batch of training samples complete this process, the corresponding parameters (w, b) are updated according to the stochastic gradient descent method on the neurons that have not been deleted.
3. Then continue repeating the process:
   (a) Restore the deleted neurons (the deleted neurons remain intact at this time, while the non-deleted neurons have been updated)
   (b) Randomly select a half-sized subset of hidden layer neurons and temporarily delete them (back up the parameters of the deleted neurons)
   (c) For a small batch of training samples, the loss is propagated forward and then backpropagated and the parameters (w, b) are updated according to the stochastic gradient descent method (the parameters that have not been deleted are updated, and the parameters of the deleted neurons retain the results before they were deleted))

1.2 Mathematical Perspective

Assuming we have a normal simple neural network, and the math expression to calculate the results are:

$$z_i^{(l+1)} = w_i^{(l+1)} + y^l + b$$
$$y_i^{(l+1)} = f(z_i^{(l+1)})$$

After we employing dropout, the math functions become:

$$r_j^{(l)} \sim Bernoulli(p)$$

$$\tilde{y}^{(l)} = r^l \times y^l$$

$$z_i^{(l+1)} = w_i^{(l+1)} + \tilde{y}^l + b$$

$$y_i^{(l+1)} = f(z_i^{(l+1)})$$

The Bernoulli function in the above formula is to generate a probability r vector, that is, to randomly generate a vector of 0 and 1. At the code level, making a neuron stop working with probability p is actually to make its activation function value change to 0 with probability p. For example, the number of neurons in a certain layer of our network is 1000, and the output values of its activation function are $y_1, y_2, \ldots, y_{1000}$. If we choose the dropout ratio of 0.4, then the neurons in this layer will go through dropout. Finally, about 400 of the 1,000 neurons will have their values set to 0.

2 Batch Normalization

In deep learning, internal covariate shift refers to the change in the distribution of network activations due to the change in network parameters during training. This shift can slow down the training process because each layer has to adapt to a new distribution in every training step. Batch normalization aims to reduce this internal covariate shift.

2.1 Mathematical Perceptive

For a given mini-batch, let's consider a particular layer in the network and a specific neuron within that layer. Let $X = [x_1, x_2, \ldots, x_m]$ be the vector of input activations of this neuron for the mini-batch (of size m).

- Calculate Batch Mean and Variance

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_b)^2$$

- Normalize the Batch

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

- Scale and Shift: The normalized values are then scaled and shifted using two trainable parameters, γ (scale) and β (shift), which are learned during training. This step ensures that the normalization process does not restrict the representational capacity of the layer.

$$y_i = \gamma \hat{x}_i + \beta$$

3 Activation Function

An activation function in neural networks is a mathematical function applied to the output of a neuron (or node) in the network. It introduces non-linearity into the network, enabling it to learn and perform more complex tasks. Without activation functions, neural networks would essentially be linear regression models, unable to model the complex relationships in real-world data.

3.1 Sigmoid

The sigmoid function is also called the Logistic function, because the Sigmoid function can be inferred from Logistic Regression (LR) and is also the activation function specified by the LR model. The value range of the sigmod function is between $(0, 1)$, and the output of the network can be mapped to this range to facilitate analysis. The Fig 1 shows the figure of sigmoid function and its derivative.

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

$$\sigma'(z) = \frac{0 - 1(-e^{-z})}{(1 + e^{-z})^2} = \frac{e^{-z}}{(1 + e^{-z})^2} = \frac{1 + e^{-z}}{(1 + e^{-z})} \times \sigma(z) = (1 - \sigma(z)) \times \sigma(z)$$

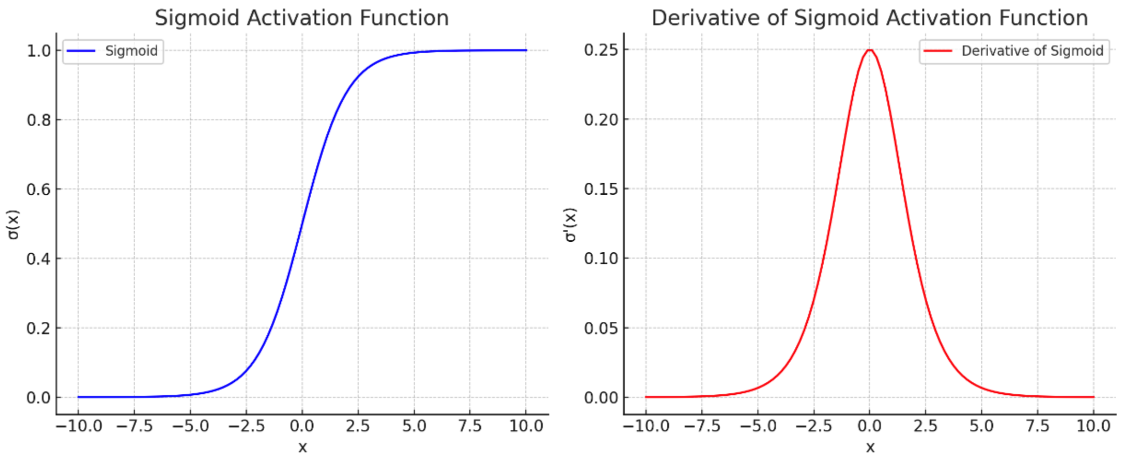

Figure 1: Sigmoid Function

Pros

- Smooth and easy to derive.

Cons

- The activation function is computationally intensive (including exponentiation and division in both forward propagation and back propagation)
- The value range of the Sigmoid derivative is [0, 0.25]. Due to the "chain reaction" during back propagation of the neural network, it is easy for the gradient to disappear.
- The output of Sigmoid is not zero-mean (i.e., zero-centered); this will cause the neurons of the subsequent layer to receive the non-zero mean signal output by the previous layer as input. As the network deepens, the original distribution of the data will be changed.

3.2 tanh

tanh is the hyperbolic tangent function, which is pronounced Hyperbolic Tangent in English. Tanh and sigmoid are similar. They are both saturated activation functions. The difference is that the output value range changes from (0,1) to (-1,1). The tanh function can be regarded as the result of downward translation and stretching of sigmoid. The Fig 2 shows the figure of sigmoid function and its derivative.

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{2}{1 + e^{-2x}} - 1$$

$$tanh'(x) = 1 - tanh(x)^2$$

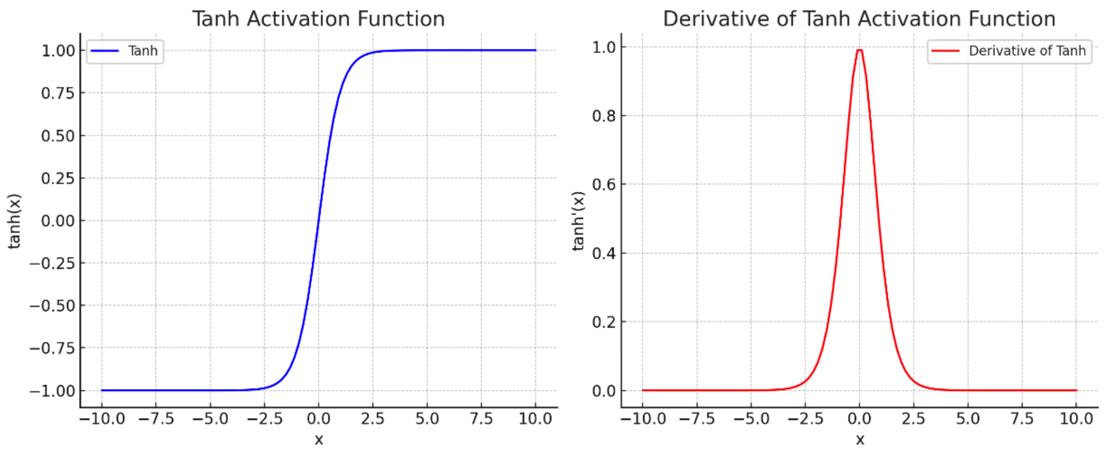

Figure 2: tanh Function

Compare to the sigmoid function, the output range of tanh is (-1, 1), which solves the problem that the Sigmoid function is not zero-centered output. However, the problem of exponentiation still remains. Also, the tanh derivative range is between (0, 1). Compared with sigmoid's (0, 0.25), the gradient vanishing problem will be alleviated, but it will still exist.

3.3 Rectified Linear Unit (ReLU)

ReLU stands for Rectified Linear Unit, and it's a type of activation function widely used in neural networks, especially in deep learning models.

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

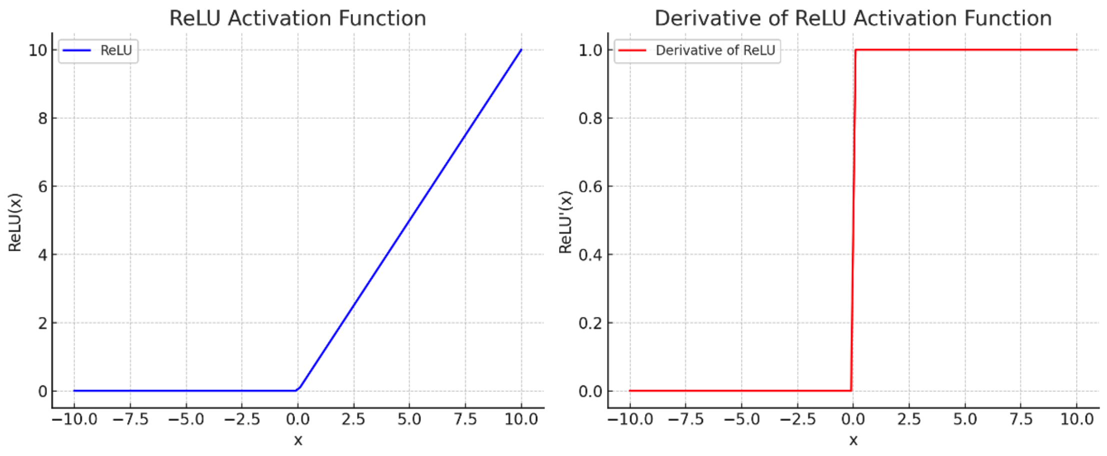

Figure 3: ReLU Function

Based on the Fig 3, the derivative of the ReLU function is either 0 or 1. It's 0 for all negative inputs and 1 for all positive inputs. At $x = 0$, the derivative is undefined; however, in practice, this is often handled in specific ways depending on the context or framework.

ReLU can solve the vanishing gradient problem in deep networks and makes deep networks trainable. At the same time, ReLU is a nonlinear function. The so-called nonlinearity means that the first-order derivative is not a constant; when derivation of ReLU, the derivatives are different when the input values are positive and negative respectively, that is, the derivative of ReLU is not a constant, so ReLU is nonlinear (but unlike Sigmoid and tanh, the nonlinearity of relu is not smooth).

3.3.1 The characteristics of ReLU when $x > 0$, with a constant derivative of 1

The benefit of having a constant derivative of 1 is that it prevents the vanishing gradient problem in ”chain reactions”. However, the intensity of gradient descent relies entirely on the product of the weights, which could lead to the issue of exploding gradients. To solve this, one approach is to control the weights to keep them within the range of $(0, 1)$; another is to clip gradients to control the intensity of the gradient descent, such as in $ReLU(x) = min(6, max(0, x))$.

3.3.2 The characteristic of ReLU setting the output to zero when $x < 0$

Before describing this characteristic, it's important to understand the goal of deep learning: Deep learning aims to extract key information (key features) from complex data relationships based on large batches of sample data. In other words, it's about transforming dense matrices into sparse matrices, retaining key data information while eliminating noise, which enhances the robustness of the model. ReLU sets the output to zero for $x < 0$, which is a process of noise reduction and matrix sparsification. Furthermore, during training, this sparsity is dynamically adjusted, and the network automatically tunes the sparsity ratio to ensure the matrix retains optimal effective features.

However, ReLU forcing the output to zero for x <0 (zeroing out is equivalent to blocking that feature) might prevent the model from learning effective features. Therefore, if the learning rate is set too high, it could result in most neurons in the network being in a 'dead' state. Hence, when using a ReLU network, the learning rate should not be set too high.

4 Gradient Descent

Gradient Descent is an optimization algorithm used to solve machine learning and deep learning models, especially in situations where direct solutions are not feasible. Its goal is to minimize the loss function by iteratively updating the model parameters. Mathematically, the gradient represents the directional derivative of a function at a certain point, which is the direction in which the function changes most rapidly. In the gradient descent algorithm, we update the parameters in the direction of the negative gradient because this direction results in the fastest decrease in the function value. The basic steps of the gradient descent algorithm are as follows:

1. Initialize the model parameters (can be randomly initialize).
2. Calculate the gradient of the loss function. The gradient is the partial derivative of the loss function with respect to the model parameters.
3. Update the model parameters in the direction of the negative gradient. The magnitude of the update is determined by the learning rate (a preset positive number). The formula to update the model parameters θ is:
$$\theta_j := \theta_j - \alpha \times \nabla J(\theta)$$
where α is the learning rate set up by the user, $\nabla J(\theta)$ is the gradient
4. Repeat steps 2 and 3 until the termination criteria are met, such as when the gradient is close to 0 (i.e., a local minimum is found), or when a preset maximum number of iterations is reached.

5 Backpropagation

Backpropagation, short for ”backward propagation of errors,” is a fundamental algorithm in neural networks and deep learning. It's used for training the network, specifically for adjusting the weights of the neurons in a way that minimizes the error between the predicted output and the actual output. Backpropagation is used to calculate the gradient of the loss function with respect to each weight in the network. This gradient is then used to update the weights to minimize the loss.

5.1 Chain Rule

The chain rule is a fundamental principle in calculus used for computing the derivative of composite functions. When functions are composed together, the chain rule allows us to differentiate the entire composition by differentiating each function individually and then multiplying the results. It's particularly important in the context of neural networks for implementing backpropagation, as it helps in calculating gradients efficiently.

Imagin you have two functions f and g, and you want to differentiate their composition $h(x) = f(g(x))$. The chain rule states that the derivative of h with respect to x is the product of the derivative of f with respect to $g(x)$ and the derivative of g with respect to x:

$$\frac{dh}{dx} = \frac{df}{dg} \times \frac{dg}{dx}$$

For more complex compositions involving multiple functions, the chain rule can be extended. For example, if you have three functions f, g, and h, and a composite function $y(x) = f(g(h(x)))$, the derivative is:

$$\frac{dy}{dx} = \frac{df}{dg} \times \frac{dg}{dh} \times \frac{dh}{dx}$$

5.2 Backward Pass

First of all, we know that the purpose of the BP algorithm is to find the derivative of the loss function with respect to the weight/bias parameters, that is, to find:

$$\frac{\partial L(\theta)}{\partial w} = \sum_{n=1}^{N} \frac{\partial C^n(\theta)}{\partial w}$$

where $C(\theta)$ is the cost function, w is the weight, N is the number of neural. Hence, for this pass, we only need to calculate $\frac{\partial C(\theta)}{\partial w}$ and $\frac{\partial L(\theta)}{\partial b}$

Let's take $\frac{\partial C(\theta)}{\partial w}$ for a neuron as an example. Based on the Chain Rule, we have:

$$\frac{\partial C(\theta)}{\partial w} = \frac{\partial z}{\partial w} \times \frac{\partial C}{\partial z}$$

$$z = \sum_{i=1}^{m} x_i w_i + b$$

After the neural network model is defined, the activation function of the neuron $\sigma(z)$(activation function) has been determined, we define a as the output of a neuron, then we follow the chain derivation rule and get:

$$\frac{\partial C(\theta)}{\partial z} = \frac{\partial a}{\partial z} \times \frac{\partial C}{\partial a}$$

Because $a = \sigma(z)$, hence $\frac{\partial a}{\partial z} = \sigma'(z) = \textit{constant}$, and we only need to calculate $\frac{\partial C}{\partial a}$. Based on the chain rule, we get

$$\frac{\partial C}{\partial a} = \sum_{i=1}^{k} \frac{\partial z_i}{\partial a} \frac{\partial C}{\partial z_i}$$

Hence, backward pass is to calculate $\frac{\partial C}{\partial z}$, then go through the above formula step by step in reverse to find the partial derivative of each layer.

6 Gradient Vanishing & Gradient Explosion

In backpropagation, gradients of the loss function are calculated with respect to each weight in the network. These gradients are used to update the weights. The process involves the chain rule from calculus, which for a neural network with multiple layers, means that the gradient is the product of gradients of all layers from the output back to the current weight.

6.1 Gradient Vanishing

Gradient vanishing occurs when the gradients become very small, exponentially decreasing towards zero as the backpropagation algorithm progresses through the layers. This is especially problematic in deep networks with many layers.

Consider a deep neural network with L layers, and the activation function is sigmoid (or any function with a gradient less than 1). The sigmoid function's derivative is at maximum 0.25. During backpropagation, to compute the gradient at the first layer, the derivatives of all L layers are multiplied together. If L is large and the derivative of each layer is less than 1, this product becomes exceedingly small. Mathematically, if each layer's gradient is g where $0 < g < 1$, the gradient at the first layer would be approximately g^L, which approaches zero as L increases.

6.2 Gradient Explosion

Gradient explosion is the opposite issue, where gradients become excessively large. This can lead to huge updates to the weights, causing the network to diverge and fail to converge to a solution.

Using the same scenario as above, but instead, if the derivative of the activation function is large, or the weights are large, the product of these gradients can become very large. If each layer's gradient is g, where $g > 1$, the gradient at the first layer would be approximately g^L. If g is significantly larger than 1 and L is large, this product becomes extremely large, leading to a gradient explosion.

6.3 Solutions

For Gradient Vanishing

- Use of ReLU Activation Function: ReLU (Rectified Linear Unit) has a derivative of either 0 or 1, so it doesn't diminish the gradient during backpropagation.
- Weight Initialization: Proper initialization (like He or Xavier initialization) can help in maintaining the gradients at reasonable levels.
- Batch Normalization: Normalizing the input of each layer to have zero mean and unit variance helps maintain stable gradients.

For Gradient Explosion

- Gradient Clipping: This involves clipping the gradients during backpropagation to a maximum value to prevent them from becoming too large.
- Weight Regularization: Techniques like L1 or L2 regularization can help in keeping the weights (and thus the gradients) from growing too large.
- Adjusting the Learning Rate: Using a smaller learning rate can help mitigate the risk of gradient explosion.

References