Convolutional Neural Networks

John Chidiac

October 7, 2023

Contents

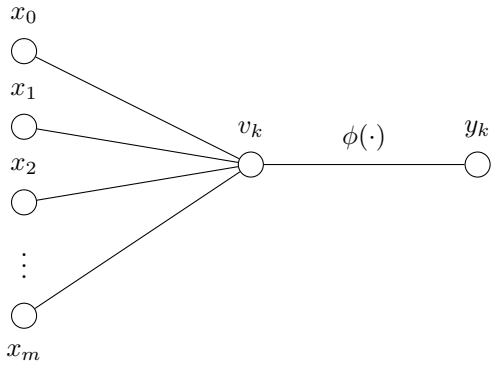
1 Basic Concepts in Machine Learning

1.1 Neurons

A neuron is a mathematical function inspired by biological neurons. It receives one or more inputs and sums them to produce an output. Each input is separately weighted and the sum is passed through an activation function. They are often monotonically increasing, bounded, continuous, and differentiable.

1.1.1 Basic Structure

Given an artificial neuron k, we consider $m + 1$ inputs with signals $x_0 \to x_m$ and weights $w_{k0} \to w_{km}$. Typically we assign "+1" to x_0 which makes it a bias input with $w_{k0} = b_k$. This leaves m actual inputs to the neuron. $\phi(\cdot)$ is the activation function, v_k is our neuron, and y_k is the output.



1.1.2 Implementing a Threshold Logic Unit

A TLU is a type of artificial neuron that takes a set of binary inputs, applies a weight to them, sums them, subtracts a threshold value, then passes them through a step function. The TLU we look at is a very simple and elementary example.

Neuron 1 Threshold Logic Unit

```
1: threshold ← number
2: weights ← numbers[X]
3: procedure FIRE(booleans[X])
4:     T ← 0
5:     for i ← 1 to X do
6:         if booleans[i] = true then
7:             T ← T + weights[i]
8:         end if
9:     end for
10:    return T > threshold
11: end procedure
```

1.2 Activation Functions

The activation function of a node in a neural network is a function that calculates the output of the node based on its inputs and the weights on individual inputs. Nonlinear activation functions are essential to solve more complicated problems.

1.2.1 Different Properties of Activation Functions

Nonlinearity. When the activation function is non-linear, then a two-layer neural network can be proven to be a universal function approximator by the *Universal Approximation Theorem*. The identity function, which is linear, does not satisfy this property. When multiple layers use the identity function, the entire model is equivalent to a single-layer model.

Range. When the range of the activation function is finite, gradient-based training methods tend to be more stable, because pattern presentations significantly affect only limited weights. When the range is infinite, training is generally more efficient because pattern presentations significantly affect most of the weights. In the case of infinite ranges, smaller learning rates are necessary.

Continuously Differentiable. This property is desirable for enabling gradient-based optimization methods.

1.2.2 Mathematical Details and Activation Functions

A function f is *saturating* if $\lim_{|v|\to\infty} |\nabla f(v)| = 0$. Networks with non-saturating activation functions may be better than saturating ones because they are less likely to suffer from the vanishing gradient problem. We have ridge activation functions:

- Linear activation: $\phi(v) = a + v'b$
- ReLU activation: $\max(0, \phi(v) = a + v'b)$
- Heaviside activation: $\phi(v) = 1$ if $a + v'b > 0$
- Logistic activation: $\phi(v) = (1 + \exp(-a - v'b))^{-1}$

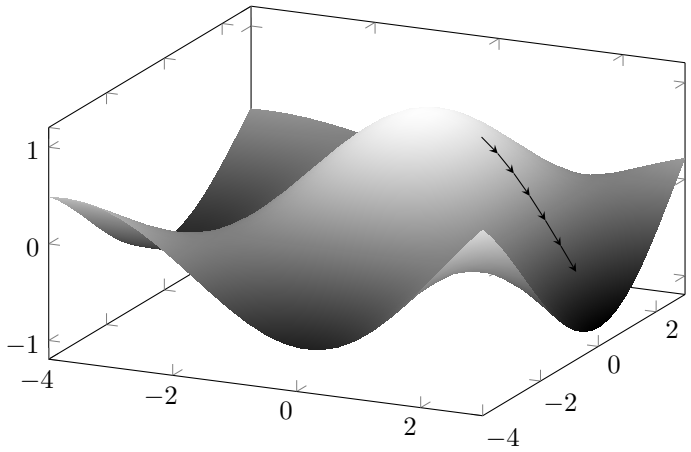
Radial activation functions used in radial basis function networks:

- Gaussian activation: $\phi(v) = \exp\left(-\frac{||v-c||^2}{2\sigma^2}\right)$
- Multiquadratic activation: $\phi(v) = \sqrt{||v - c||^2 + a^2}$
- Inverse multiquadratic activation: $\phi(v) = \left(||v - c||^2 + a^2\right)^{-\frac{1}{2}}$

And finally, folding activation functions.

1.3 Gradient Descent

Gradient descent is a first-order iterative optimization algorithm for finding a local minimum of a differentiable function. The idea is to take repeated steps in the opposite direction of the gradient of the function at the current point. Doing otherwise will lead us to the local maximum. It is useful for minimizing the cost or loss function.



Gradient descent is based on the observation that if a multivariable function $F(x)$ is defined and differentiable in a neighborhood of point a, then $F(x)$ decreases the fastest if one goes from a in the direction of the negative gradient of F at a, $-\nabla F(a)$. Then, if $a_{n+1} = a_n - \gamma \nabla F(a_n)$ where γ is the learning rate, for a small enough learning rate, $\gamma \in \mathbb{R}_+$, we have $F(a_n) \geq F(a_{n+1})$. With that observation, we have

$$F(x_0) \geq F(x_1) \geq \cdots \geq F(x_n) \tag{1}$$

and we hope that the sequence x_n converges to the desired local minimum.

1.4 Learning Rate

The learning rate is a tuning parameter in an optimization algorithm that determines the step size at each iteration while moving toward a minimum of a loss function. When setting a learning rate, there is a trade-off between the rate of convergence and overshooting. While the descent direction is determined by the gradient of the loss function, the learning rate determines how big a step is taken in that direction. Too high might jump above the minimum and too low may never reach it. In order to achieve faster convergence, prevent oscillations and getting stuck in undesirable local minima the learning rate is often varied during training either in accordance to a learning rate schedule or by using an adaptive learning rate.

1.5 Loss Function

A loss function is a function that maps an event or values of one or more variables onto a real number representing some cost associated with the event. Optimization algorithms seek to minimize the loss function.

1.5.1 Quadratic Loss Function

The use of a quadratic loss function is common since it is often more mathematically manageable than other loss functions because of its symmetry and properties of variances. If the target is t, x is the value predicted by the model, and C is some constant scaling factor, then the squared error loss is:

$$\lambda(x) = C(t - x)^2 \tag{2}$$

1.5.2 0-1 Loss Function

This loss function is most commonly used in statistics and uses the Iverson bracket notation (1 when true, 0 when false).

$$L(\hat{y}, y) = [y \neq \hat{y}] \tag{3}$$

2 Building a Neural Network

The essential steps of machine learning are:

1. Feed input $y = \text{network}(x, w)$
2. Calculate the error $E = \frac{1}{2}(\hat{y} - y)^2$
3. Adjust the parameters using gradient descent $w \leftarrow w - \gamma \frac{\partial E}{\partial w}$
4. Repeat

where W is a set of parameters that are updated with gradient descent.

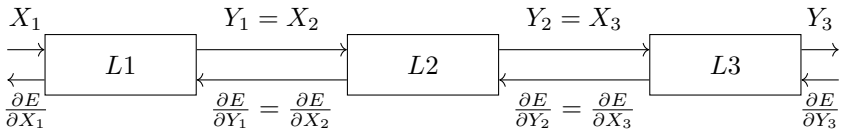
2.1 Implementation Design

We want modular code, so we need to implement every layer separately. We need a model that fits every type of layer. A layer is just a large function. For every input we give a layer, we get an output. This is called *forward propagation*. The next step is called *backward propagation*. During backward propagation, if the layer is given the derivative of its error with respect to its output, $\frac{\partial E}{\partial Y}$, it gives back the derivative of the error with respect to its input $\frac{\partial E}{\partial X}$.

$$\frac{\partial E}{\partial W} = \frac{\partial E}{\partial Y}\frac{\partial Y}{\partial W} \tag{4}$$

where

$$\frac{\partial E}{\partial X} = \frac{\partial E}{\partial Y}\frac{\partial Y}{\partial X} \tag{5}$$



2.2 The Dense Layer

In the dense layer, every input neuron is connected to every output neuron. Each connection represents a weight w_{ji}, or the weight that connects the output neuron j to the output neuron i. The bias b_j is a trainable parameter. The below is the *forward propagation*.

$$y_j = \sum_{k=1}^{i} x_k w_{jk} + b_j \tag{6}$$

We can write $y_1 \to y_j$ using matrices for simplicity and ease of computation:

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_j \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1i} \\ w_{21} & w_{22} & \dots & w2i \\ \vdots & \vdots & \ddots & \vdots \\ w_{j1} & w_{j2} & \dots & w_{ji} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_j \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_j \end{bmatrix} \tag{7}$$

Now, we look at the backward step, *back propagation*. Given the error of the output of the depth layer, we need the derivative of the error with respect to the weights and biases, since these are the trainable parameters. Next, we need the error with respect to the input since it will be passed to the layer before the one computing the dense layer.

$$\frac{\partial E}{\partial Y} = \begin{bmatrix} \frac{\partial E}{\partial y_1} \\ \frac{\partial E}{\partial y_2} \\ \vdots \\ \frac{\partial E}{\partial y_j} \end{bmatrix} \qquad \frac{\partial E}{\partial W} = \begin{bmatrix} \frac{\partial E}{\partial w_{11}} & \frac{\partial E}{\partial w_{12}} & \dots & \frac{\partial E}{\partial w_{1i}} \\ \frac{\partial E}{\partial w_{21}} & \frac{\partial E}{\partial w_{22}} & \dots & \frac{\partial E}{\partial w_{2i}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial w_{j1}} & \frac{\partial E}{\partial w_{j2}} & \dots & \frac{\partial E}{\partial w_{ji}} \end{bmatrix} \tag{8}$$

but, $\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial y_j} x_i$

$$\frac{\partial E}{\partial W} = \begin{bmatrix} \frac{\partial E}{\partial y_1} \\ \frac{\partial E}{\partial y_2} \\ \vdots \\ \frac{\partial E}{\partial y_j} \end{bmatrix} \begin{bmatrix} x_1 & x_2 & \dots & x_n \end{bmatrix} = \frac{\partial E}{\partial Y} X^t \tag{9}$$

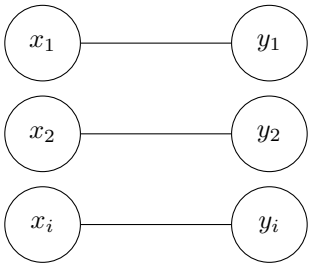
where X^t is X transposed. Moreover,

$$\frac{\partial E}{\partial b_j} = \frac{\partial E}{\partial y_j} \iff \frac{\partial E}{\partial B} = \frac{\partial E}{\partial Y} \tag{10}$$

And finally,

$$\frac{\partial E}{\partial X} = \begin{bmatrix} w_{11} & w_{21} & \dots & w_{j1} \\ w_{12} & w_{22} & \dots & w_{j2} \\ \vdots & \vdots & \ddots & \vdots \\ w_{1i} & w_{2i} & \dots & w_{ji} \end{bmatrix} \begin{bmatrix} \frac{\partial E}{\partial y_1} \\ \frac{\partial E}{\partial y_2} \\ \vdots \\ \frac{\partial E}{\partial y_j} \end{bmatrix} = W^t \frac{\partial E}{\partial Y} \tag{11}$$

2.3 The Activation Layer



The *forward propagation* of the activation layer is described by

$$Y = f(X) \tag{12}$$

From equation (12), we have

$$\frac{\partial E}{\partial X} = \frac{\partial E}{\partial Y} \odot f'(X) \tag{13}$$

3 Building a Convolutional Neural Network

3.1 Cross-Corrolation and Convolution

The equation described below is known as valid cross-corrolation.

$$\underbrace{\begin{bmatrix} 1 & 6 & 2 \\ 5 & 3 & 1 \\ 7 & 0 & 4 \end{bmatrix}}_{\text{input}} \star \underbrace{\begin{bmatrix} 1 & 2 \\ -1 & 0 \end{bmatrix}}_{\text{kernel}} = \underbrace{\begin{bmatrix} 8 & 7 \\ 4 & 5 \end{bmatrix}}_{\text{output}} \tag{14}$$

We note that the $\star$ operator denotes cross-corrolation. From the above, by rotating the kernel by 180 degrees, we can get the convolution:

$$\text{conv}(I, K) = I \star rot180(K) \tag{15}$$

Now, we introduce full cross-corrolation. This starts whenever the kernel intersects with the input, in the case before, it only worked with full intersection.

$$\underbrace{\begin{bmatrix} 1 & 6 & 2 \\ 5 & 3 & 1 \\ 7 & 0 & 4 \end{bmatrix}}_{\text{input}} \underset{\text{full}}{\star} \underbrace{\begin{bmatrix} 1 & 2 \\ -1 & 0 \end{bmatrix}}_{\text{kernel}} = \underbrace{\begin{bmatrix} 0 & -1 & -6 & -2 \\ 2 & 8 & 7 & 1 \\ 10 & 4 & 5 & -3 \\ 14 & 7 & 8 & 4 \end{bmatrix}}_{\text{output}} \tag{16}$$

3.2 The Convolutional Layer

The convolutional layer takes in a 3 dimensional array as input which may have depth. The layer has trainable parameters such as the kernels and biases. This produces an output with the same depth as the kernel. Below, we describe the *forward propagation* of a convolutional neural network.

$$Y_i = B_i + \sum_{j=1}^{n} X_j \star K_{ij} \tag{17}$$

Now, we find the necessary derivatives for *backward propagation*.

$$\frac{\partial E}{\partial K_{ij}} = X_j \star \frac{\partial E}{\partial Y_i} \tag{18}$$

$$\frac{\partial E}{\partial B_i} = \frac{\partial E}{\partial Y_i} \tag{19}$$

$$\frac{\partial E}{\partial X_j} = \sum_{i=1}^{n} \frac{\partial E}{\partial Y_i} \underset{\text{full}}{*} K_{ij} \tag{20}$$

3.3 Cross-Entropy

3.3.1 Binary Cross-Entropy Loss

Binary cross-entropy loss is a loss function for binary datasets defined as

$$E = -\frac{1}{n}\sum_{i=1}^{n} \hat{y}_i \log(y_i) + (1 - \hat{y}_i) \log(1 - y_i) \tag{21}$$

Its derivative with respect to y is described as

$$\frac{\partial E}{\partial y_i} = \frac{1}{n}\left(\frac{1 - \hat{y}_i}{1 - y_i} - \frac{\hat{y}_i}{y_i}\right) \tag{22}$$

For this type of problem, we typically use the sigmoid activation function.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{23}$$

$$\sigma'(x) = \sigma(x) \times (1 - \sigma(x)) \tag{24}$$