Gradient-Descent Optimization Methods

Rodrigo Castro

MSA 8100, Optimization Methods for Analytics
Master of Science in Analytics Program
Robinson College of Business
Georgia State University

Fall 2018

Outline

Gradient Descent Philosophy

Consider a function $J : \mathbb{R}^n \to \mathbb{R}$, and let $\boldsymbol{\theta} \in \mathbb{R}^n$.

Recall that a *level set* of J is a set $C = \{\boldsymbol{\theta} \in \mathbb{R}^n \mid J(\boldsymbol{\theta}) = c\}$ for some constant c.

Let $\boldsymbol{\theta}_0 \in C$, that is, $J(\boldsymbol{\theta}_0) = c$.

The vector $\nabla J(\boldsymbol{\theta}_0)$ is normal to the level set C, and it determines the direction of maximum rate of increase of J at $\boldsymbol{\theta}_0$.

The direction in which $-\nabla J(\boldsymbol{\theta}_0)$ points is the direction of maximum rate of decrease of J at $\boldsymbol{\theta}_0$.

Hence, the direction of negative gradient is a good direction to search if we want to (eventually) find a function minimizer.

Let $\boldsymbol{\theta}^{(0)}$ be a starting point and consider the point, and consider the point $\boldsymbol{\theta}^{(0)} - \alpha \nabla J(\boldsymbol{\theta}^{(0)})$.

Using Taylor's theorem, we can express $J(\boldsymbol{\theta})$ as

$$J(\boldsymbol{\theta}) = J(\boldsymbol{a}) + \nabla J(\boldsymbol{a}) \cdot (\boldsymbol{\theta} - \boldsymbol{a}) + \dots \quad .$$

By letting $\boldsymbol{a} = \boldsymbol{\theta}^{(0)}$, $\boldsymbol{\theta} = \boldsymbol{\theta}^{(0)} - \alpha \nabla J(\boldsymbol{\theta}^{(0)})$, we get

$$J(\boldsymbol{\theta}^{(0)} - \alpha \nabla J(\boldsymbol{\theta}^{(0)})) = J(\boldsymbol{\theta}^{(0)}) - \alpha ||\nabla J(\boldsymbol{\theta}^{(0)})||^2 + o(\alpha)\,.$$

Thus, if $\nabla J(\boldsymbol{\theta}^{(0)}) \neq 0$, then for sufficiently small $\alpha > 0$, we have

$$J(\boldsymbol{\theta}^{(0)} - \alpha \nabla J(\boldsymbol{\theta}^{(0)})) < J(\boldsymbol{\theta}^{(0)})\,.$$

$J(\boldsymbol{\theta}^{(0)} - \alpha \nabla J(\boldsymbol{\theta}^{(0)})) < J(\boldsymbol{\theta}^{(0)})$ implies that the point $\boldsymbol{\theta}^{(0)} - \alpha \nabla J(\boldsymbol{\theta}^{(0)})$ is an improvement over the point $\boldsymbol{\theta}^{(0)}$ if we are searching for a minimizer.

This idea leads to the implementation of the iterative algorithm:

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \alpha \nabla J(\boldsymbol{\theta}^{(k)}) \,.$$

We refer to this as a *gradient descent algorithm*.

The gradient changes as the search proceeds, tending to zero as we approach the minimizer.

If we are in a convex neighborhood of a local minimum, the steps in the algorithm become smaller and smaller (for sufficiently small α).

Gradient descent converges when every element of the gradient is zero.

In some cases, we may be able to find the critical points by solving the equation $\nabla J(\boldsymbol{\theta}) = \mathbf{0}$ analytically, and avoid a gradient descent search.

However, in practice, in most machine-learning applications solving $\nabla J(\theta) = \mathbf{0}$ analytically is not feasible.

Additionally, in practice, the gradient gets very close to zero, but not exactly zero, especially in high-dimensional spaces.

However, this is not necessarily a major problem for many reasons. One reason is that we usually look for small values of a cost function J in hope of achieving a good enough performance measure P of a model.

Objective Functions of Predictive Models

*For simplicity of exposure, let us set our notation in the context of supervised learning.

A training data set of m training examples consists of a pair of matrices X and Y of the form

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{bmatrix}$$

$$Y = \begin{bmatrix} y^{(1)} & y^{(2)} & \dots & y^{(m)} \end{bmatrix}$$

where each $x^{(i)} = \begin{bmatrix} x_1^{(i)} & x_2^{(i)} & \cdots & x_l^{(i)} \end{bmatrix}^T$ is called *feature vector*. We will refer to $y^{(i)}$ as the *label* of the i-th example.

A pair $(x^{(i)}, y^{(i)})$ constitutes a training example, although sometimes we will refer to just $x^{(i)}$ as the i-th training example.

The numbers $x_1^{(i)}, x_2^{(i)}, \cdots, x_l^{(i)}$ are called the *features* of the i-th traning example.

Generally speaking, what a predictive model does is to combine these features together with a set of parameters $\theta = \{\theta_j\}_{j=1,\ldots,n}$ and apply a mapping f (or a composition of mappings) to that combination in order to make a prediction as close as possible to the associated label.

We will denote a prediction as $\hat{y}^{(i)} = f(x^{(i)}, \theta)$.

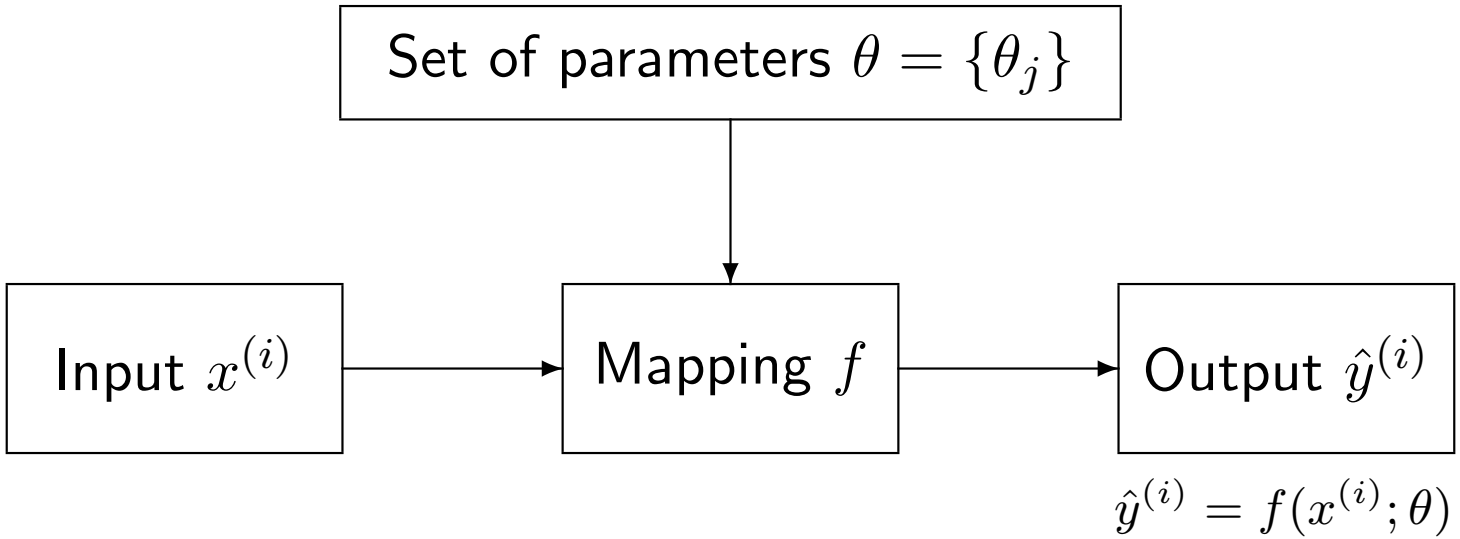


Figure: General Structure of a (Supervised) Predictive Model

In order to train our model to make predictions on the data set that are similar to the label of each example a cost function is constructed, for which we will search a minimizer.

First we start by defining a loss function $\mathcal{L}(\hat{y}^{(i)}, y^{(i)})$ that quantifies the "distance" or "error" between the prediction $\hat{y}^{(i)}$ and the (actual) label $y^{(i)}$ for each training example.

Next, a cost function can be defined as the expectation of the loss function over all examples, that is,

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) \,.$$

Then, our optimization objective will be to minimize $J(\theta)$ with respect to θ. (In practice, the cost function can also have penalty terms such as regularization terms.)

Overview

Batch Gradient Descent

Batch gradient descent is the "straightforward" implementation of gradient descent on a function that depends on a set of training examples (besides the parameters).

The word batch refers to the whole set of training examples (later on we will see mini-batches).

Using the whole batch each time before doing a single update on the parameters ensures that the updates on the parameters are always in the direction of maximum rate of decrease of the cost function (we will see that this is not the case with stochastic gradient descent).

Algorithm:

Set learning rate α;
Initialize parameters $\theta = \{\theta_j\}_{j=1,...,n}$ randomly;
while *stopping criterion not met* **do**

- Compute predictions $\left\{\hat{y}^{(i)} = f(x^{(i)};\theta)\right\}_{i=1,..,m}$;
- Compute $\left\{\mathcal{L}(\hat{y}^{(i)}, y^{(i)})\right\}_{i=1,..,m}$ and $J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)})$;
- Compute $\left\{\frac{\partial J}{\partial \theta_j}\right\}_{j=1,...,n}$;
- Update $\theta_j := \theta_j - \alpha\frac{\partial J}{\partial \theta_j}$, $j = 1, ..., n$;

end
return $\{\theta_j\}_{j=1,...,n}$

Overview

Mini-Batch Gradient Descent

Let us split our set of m training examples into smaller sets (mini-batches) of some fixed size s, $1 \leq s \leq m$.

Let $N = \lceil m/s \rceil$, that is, the number of mini-batches (the last mini-batch might have size smaller than s). Let $X^{\{t\}}$ denote the t-th mini-batch:

$$X^{\{1\}} = \left[x^{(1)} \, x^{(2)} \, \dots \, x^{(s)} \right],$$

$$X^{\{2\}} = \left[x^{(s+1)} \, x^{(s+2)} \, \dots \, x^{(2s)} \right],$$

$$\vdots$$

$$X^{\{N\}} = \left[\dots \dots \, x^{(m)} \right].$$

*For simplicity, let us abuse the notation a little and write $x^{(i)} \in X^{\{t\}}$ if the i-th example is part of the t-th mini-batch.

Algorithm:

Set mini-batch size s, learning rate α;
Permute the training examples randomly and split into mini-batches;
Initialize parameteres $\theta = \{\theta_j\}_{j=1,...,n}$ randomly;
while *stopping criterion not met* **do**
  for *t=1,...,N* **do**
    Compute predictions $\left\{\hat{y}^{(i)} = f(x^{(i)};\theta)\right\}_{x^{(i)} \in X^{\{t\}}}$;
    Compute $\left\{\mathcal{L}(\hat{y}^{(i)}, y^{(i)})\right\}_{x^{(i)} \in X^{\{t\}}}$;
    Compute $J(\theta) = \frac{1}{|X^{\{t\}}|} \sum_{x^{(i)} \in X^{\{t\}}} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$;
    Compute $\left\{\frac{\partial J}{\partial \theta_j}\right\}_{j=1,...,n}$;
    Update $\theta_j := \theta_j - \alpha \frac{\partial J}{\partial \theta_j}$, $j = 1, ..., n$;
  end
end
return $\{\theta_j\}_{j=1,...,n}$

Overview

Stochastic Gradient Descent

The name stochastic gradient descent has been usually reserved for the particular case of mini-bath gradient descent when the mini-batch size is $s = 1$.

Strictly speaking, any mini-batch gradient descent with $1 \leq s < m$ is a stochastic descent.

Whenever the mini-batch size is smaller than m we have no guarantee that in each step the global cost function will decrease.

In some steps the cost function actually increases, but in the long run it tends to converge to a neighbohood of a minimizer.

Algorithm:

Set learning rate α;
Initialize parameters $\theta = \{\theta_j\}_{j=1,...,n}$ randomly;
while *stopping criterion not met* **do**
  for *i=1,...,m* **do**
    Compute prediction $\hat{y}^{(i)} = f(x^{(i)}; \theta)$;
    Compute $\mathcal{L}(\hat{y}^{(i)}, y^{(i)})$ and let $J(\theta) = \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$;
    Compute $\left\{ \frac{\partial J}{\partial \theta_j} \right\}_{j=1,...,n}$;
    Update $\theta_j := \theta_j - \alpha \frac{\partial J}{\partial \theta_j}$, $j = 1, ..., n$;
  end
end
return $\{\theta_j\}_{j=1,...,n}$

Although in batch gradient descent the steps we take are always in the direction of maximum, it can be a slow algorithm when the training set is large.

Stochastic gradient descent is much faster, since we do an update after every single example, but the steps can be very "stochastic" and the neighborhood of convergence around a minimizer might be too large.

Mini-batch gradient descent is the most common implementation of gradient descent for large training sets (and it is usually combined with other techniques, as we will see).

Mini-batch gradient descent is a good balance between speed and good convergence. Usually, the mini-batch size is set up to be a few hundred examples (frequently a power of 2).

Overview

Gradient Descent with Momentum

The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to "move" in their direction.

This is accomplished by introducing a variable $\mathbf{v} = [v_1, \ldots, v_j]$ that plays the role of a velocity vector, it is the direction and speed at which the parameters move through the parameter space. The velocity is set to an exponentially decaying average of the negative gradient, that is, in the iteration of the mini-batch t,

$$v_j^{(t)} = \beta_1 v_j^{(t-1)} + (1 - \beta_1) \frac{\partial J}{\partial \theta_j}^{(t)}, \; j = 1, ..., n,$$

where the hyperparameter $\beta \in [0, 1]$ determines how quickly the contributions from previous gradients exponentially decay. The t-th update in the parameters is then $\theta_j^{(t)} = \theta_j^{(t-1)} - \alpha v_j^{(t)}$, $j = 1, ..., n$.

Set mini-batch size s, learning rate α, momentum parameter β_1;
Permute the training examples randomly and split into mini-batches;
Initialize $\theta = \{\theta_j\}_{j=1,...,n}$ randomly; Initialize velocities $v_j = 0$, $j = 1, ...n$;
while *stopping criterion not met* **do**
  for *t=1,...,N* **do**
    Compute predictions $\left\{\hat{y}^{(i)} = f(x^{(i)}; \theta)\right\}_{x^{(i)} \in X^{\{t\}}}$;
    Compute $\left\{\mathcal{L}(\hat{y}^{(i)}, y^{(i)})\right\}_{x^{(i)} \in X^{\{t\}}}$;
    Compute $J(\theta) = \frac{1}{|X^{\{t\}}|} \sum_{x^{(i)} \in X^{\{t\}}} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$;
    Compute $\{\partial J / \partial \theta_j\}_{j=1,...,n}$;
    Compute $v_j := \beta_1 v_j + (1 - \beta_1) \frac{\partial J}{\partial \theta_j}$, $j = 1, ..., n$;
    Update $\theta_j := \theta_j - \alpha v_j$, $j = 1, ..., n$;
  end
end
return $\{\theta_j\}_{j=1,...,n}$

Overview

RMSprop

The learning rate is one of the most difficult hyperparameters to set. RMSprop is part of a family of algorithms that have adaptive learning rates.

They use a separate learning rate for each parameter and automatically adapt these learning rates throughout the course of learning.

RMSprop is a mini batch-based method that individually adapts the learning rate for a parameter θ_j by dividing the global learning rate α by the square root of a exponentially weighted average s_j of the square of the gradient,

$$s_j^{(t)} = \beta_2 s_j^{(t-1)} + (1-\beta_2)\left(\frac{\partial J}{\partial \theta_j}^{(t)}\right)^2, \quad \theta_j^{(t)} = \theta_j^{(t-1)} - \frac{\alpha}{\sqrt{s_j}}\frac{\partial J}{\partial \theta_j}^{(t)}.$$

We will refer to β_2 as the decay rate, which controls the length scale of the moving average.

By adapting the learning rate in this way, the parameters with the largest partial derivative of the cost function get a correspondingly rapid decrease in their learning rate, while parameters with small partial derivatives get a relatively small decrease in their learning rate.

RMS stands for root mean squared, since we take the square root of a moving average of the square of the gradient.

Set mini-batch size s, learning rate α, decay rate β_2, small constant ε;
Permute the training examples randomly and split into mini-batches;
Initialize $\theta = \{\theta_j\}_{j=1,...,n}$ randomly; Initialize $s_j = 0$, $j = 1, ...n$;
while *stopping criterion not met* **do**
  for *t=1,...,N* **do**
    Compute predictions $\left\{\hat{y}^{(i)} = f(x^{(i)}; \theta)\right\}_{x^{(i)} \in X^{\{t\}}}$;
    Compute $\left\{\mathcal{L}(\hat{y}^{(i)}, y^{(i)})\right\}_{x^{(i)} \in X^{\{t\}}}$;
    Compute $J(\theta) = \dfrac{1}{|X^{\{t\}}|} \displaystyle\sum_{x^{(i)} \in X^{\{t\}}} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$;
    Compute $\{\partial J / \partial \theta_j\}_{j=1,...,n}$;
    Compute $s_j := \beta_2 s_j + (1 - \beta_2) \left(\dfrac{\partial J}{\partial \theta_j}\right)^2$, $j = 1, ..., n$;
    Update $\theta_j := \theta_j - \dfrac{\alpha}{\sqrt{s_j + \varepsilon}} \dfrac{\partial J}{\partial \theta_j}$, $j = 1, ..., n$;
  end
end
return $\{\theta_j\}_{j=1,...,n}$

Overview

Adam

Adam (adaptive moments) is another adaptive learning rate optimization algorithm. It can be seen as a combination of RMSprop and momentum, with one more adaptation: corrected biases.

Adam includes corrections to the estimates of the momentum terms v_j and the terms s_j:

$$\hat{v}_j = \frac{v_j}{1-\beta_1^t}, \quad \hat{s}_j = \frac{s_j}{1-\beta_2^t}.$$

These corrected biases substitute v_j and s_j in the parameter updates.

Having these corrections allows us to have moving averages that are more "fair" representations of the accumulated gradients during the early iterations. (Observe that we usually initialize $v_j = 0$ and $s_j = 0$, so that brings downs the averages, especially during the first iterations.)

Set mini-batch size s, learning rate α, momentum parameter β_1, decay rate β_2, small constant ε;

Initialize $\theta = \{\theta_j\}_{j=1,...,n}$ randomly; Initialize $v_j = 0$, $s_j = 0$, $j = 1, ...n$;

while *stopping criterion not met* **do**

  for *t=1,...,N* **do**

    Proceed first steps as mini-batch gradient descent;

    Compute $\{\partial J/\partial \theta_j\}_{j=1,...,n}$;

    Compute $v_j := \beta_1 v_j + (1-\beta_1)\dfrac{\partial J}{\partial \theta_j}$, $j = 1, ..., n$;

    Compute $s_j := \beta_2 v_j + (1-\beta_2)\left(\dfrac{\partial J}{\partial \theta_j}\right)^2$, $j = 1, ..., n$;

    Correct biases $\hat{v}_j = \dfrac{v_j}{1-\beta_1^t}$, $\hat{s}_j = \dfrac{s_j}{1-\beta_2^t}$;

    Update $\theta_j := \theta_j - \dfrac{\alpha}{\sqrt{\hat{s}_j + \varepsilon}}\hat{v}_j$, $j = 1, ..., n$;

  end

end

return $\{\theta_j\}_{j=1,...,n}$

Learning Rate Decay Methods

Decreasing the learning rate α over time might help to improve the convergence of stochastic gradient methods, since it will "shrink" the size of the neighborhood of convergence over time.

The decay of α is usually a function of the training iteration, or epoch number T. An *epoch* is a single pass through the whole training set.

Let α_0 be a initial learning rate, and γ a decay rate. Some common rate decay methods for α are:

$$\alpha = \alpha_0/(1+\gamma T), \qquad \alpha = \gamma^T \alpha_0,$$

$$\alpha = \gamma \alpha_0/\sqrt{T}, \qquad \alpha = \gamma \alpha_0/\sqrt{t}.$$

Penalty Methods

Let $\theta \in \mathbb{R}^n$, and consider a general constrained optimization problem

$$\text{minimize } f(\theta), \text{ subject to } \theta \in \Omega.$$

We can approximate this constrained optimization problem by the unconstrained optimization problem

$$\text{minimize } f(\theta) + \lambda P(\theta),$$

where $\lambda \in \mathbb{R}$ is a positive constant called the *penalty parameter*, and $P : \mathbb{R}^n \to \mathbb{R}$ is called the *penalty function*.

Definition

A function $P : \mathbb{R}^n \to \mathbb{R}$ is called *penalty function* for the constrained optimization problem above if it satisfies the following conditions:

i) P is continuous.

ii) $P(\theta) \geq 0$ for all $\theta \in \mathbb{R}$.

iii) $P(\theta) = 0$ if and only $\theta \in \Omega$.

From the definition we can see that the role of the penalty function is to "penalize" points that are outside the feasible set.

For the unconstrained problem above to be a good approximation to the original problem, the penalty function P must be chosen appropriately.

Two common penalty functions are L_1 and L_2 regularization, defined as

$$L_1 \text{ regularization} : P(\theta) = ||\theta||_1 = \sum_{j=1}^{n} |\theta_j|,$$

$$L_2 \text{ regularization} : P(\theta) = ||\theta||_2^2 = \frac{1}{2} \sum_{j=1}^{n} \theta_j^2$$

These regularization strategies drive θ closer to the origin, that is, the penalty P is greater as the θ_j are farther away from 0.

One difference between these two regularizations is that in L_1 the parameters tend to be more sparse (due to the shape of the level sets of P), which helps "compressing" the model. However, L_2 regularization is used more often.

The cost function of a predictive model with L_1 regularization would be

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \lambda\sum_{j=1}^{n}|\theta_j|,$$

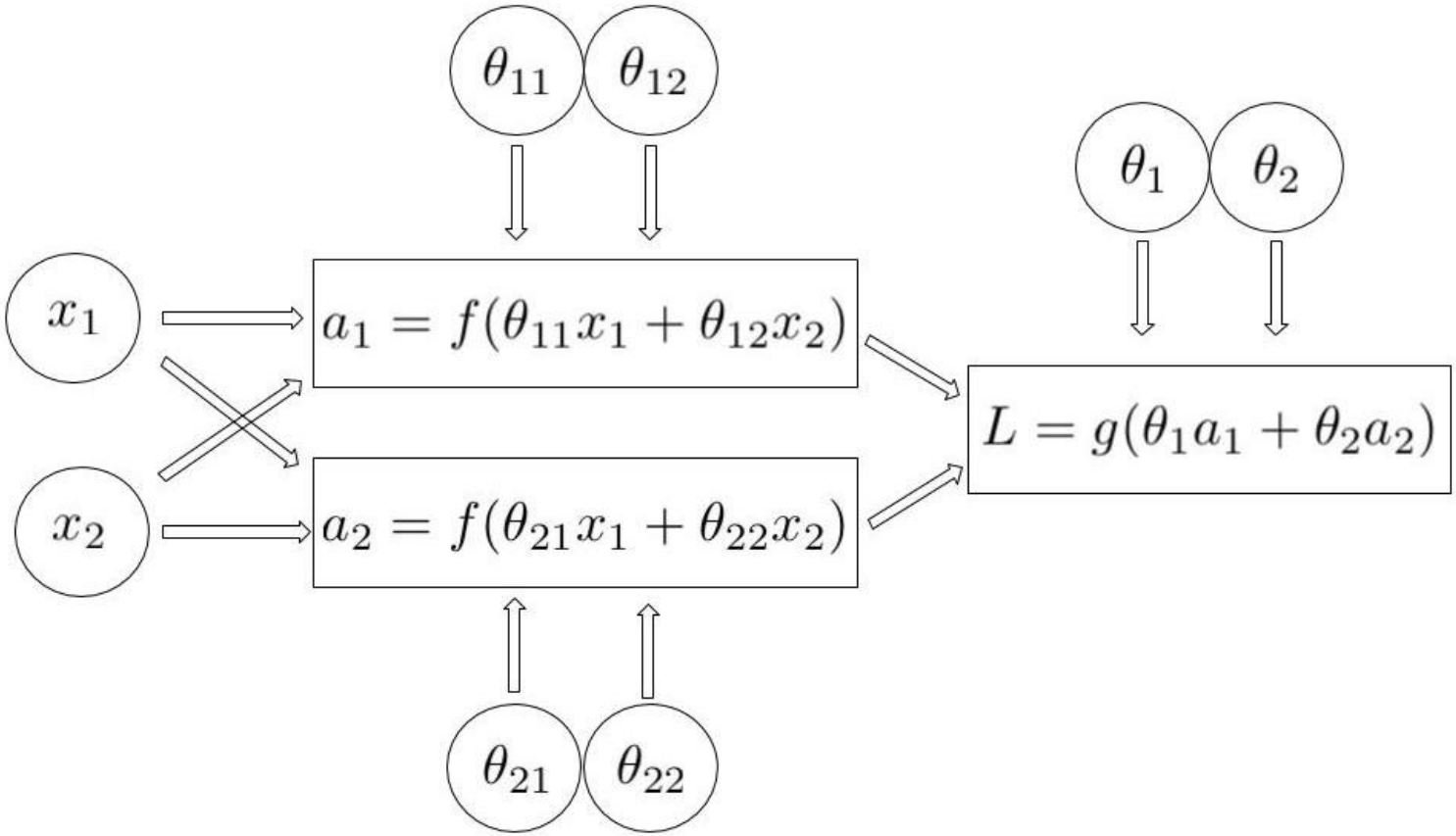
while the cost function of a predictive model with L_2 regularization would be

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2}\sum_{j=1}^{n}\theta_j^2.$$

These strategies are designed mainly to reduce over-fitting and improve generalization, that is, to reduce the test error of a model, possibly at the expense of increased training error. In other words, we want the model to perform well not just on the training data, but also on new inputs.

Computation Graphs

A computation graph is a directed graph where the nodes correspond to variables or operations, and the arrows indicate dependence. This is an example:



Finding the gradient of a "terminal" function of a computation graph with respect to a set of the variables (usually the parameters θ) is known as *backpropagation*. Since the function has a parametric form, the appearance of the chain rule is natural.

For example, say that we want to compute $\partial L / \partial \theta_{21}$ in the example above,

$$\frac{\partial L}{\theta_{21}} = \cancelto{0}{\frac{\partial L}{\partial a_1}\frac{\partial a_1}{\partial \theta_{21}}} + \frac{\partial L}{\partial a_2}\frac{\partial a_2}{\partial \theta_{21}} = \frac{\partial g}{\partial a_2}\frac{\partial f}{\partial \theta_{21}}$$

$$= \theta_2 x_1 g'(\theta_1 a_1 + \theta_2 a_2) f'(\theta_{21} x_1 + \theta_{22} x_2)\,.$$

Artificial neural networks are computation networks, and they are usually trained using gradient descent, so *backpropagation* is used to find the gradient of the cost function in order to update the parameters.

Choosing the Right Optimization Algorithm

Unfortunately, the is currently no consensus on which algorithm should one choose.

The algorithms presented here are the most popular techniques actively in use today.

The algorithms with adaptive learning rates perform fairly robustly, but no single best algorithm has emerged.

The choice of which algorithm to use, at this point, seems to depend largely on the user's familiarity with the algorithm.

Bibliography

The ideas introduced in this presentation are a combination of my own conceptual understanding of the topics together with statements (some copied literally) presented in the sources below.

Chong, E., Zak, S., **An Introduction to Optimization**, *Wiley*, 2013.

Ng, A., **Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization**, *deeplearning.ai on Coursera*, 2017.

Goodfellow, I., Bengio, Y., Courville, A., **Deep Learning**, *MIT Press*, 2016.