Improving the way neural networks learn

부산대학교 김호원

부산대 컴퓨터공학과 교수
부산대 사물인터넷센터 센터장
howonkim@pusan.ac.kr

2017.12.4

Improving the way neural networks learn

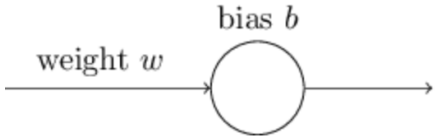
http://neuralnetworksanddeeplearning.com/chap3.html

Neural network 학습

- Neural network 학습을 위한 기법
  - Cost function
    - Cross entropy function
  - 주요 네가지 regularization 기법
    - L1 regularization
    - L2 regularization
    - Dropout
    - Artificial expansion of training data

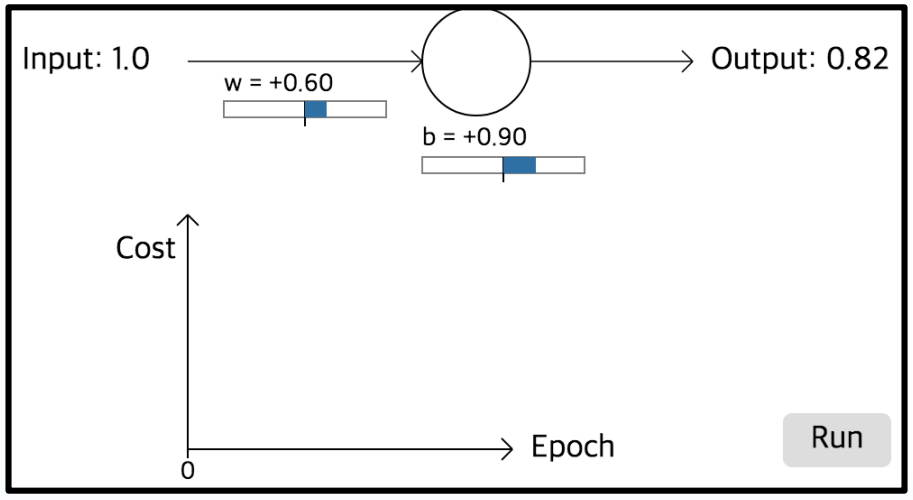
Cross-entropy cost function

- Inversion을 수행 neural network의 초기값 설정에 따른 학습 특성 비교(1/2)
  - 입력이 '1'일때 '0'을 출력하는 neural network을 학습하는 경우
    - 즉, inversion을 수행하도록 weight w와 b를 학습하는 경우임
    - 또한, cost function은 quadratic cost를 사용하는 경우임 → $C = \frac{(y-a)^2}{2}$,

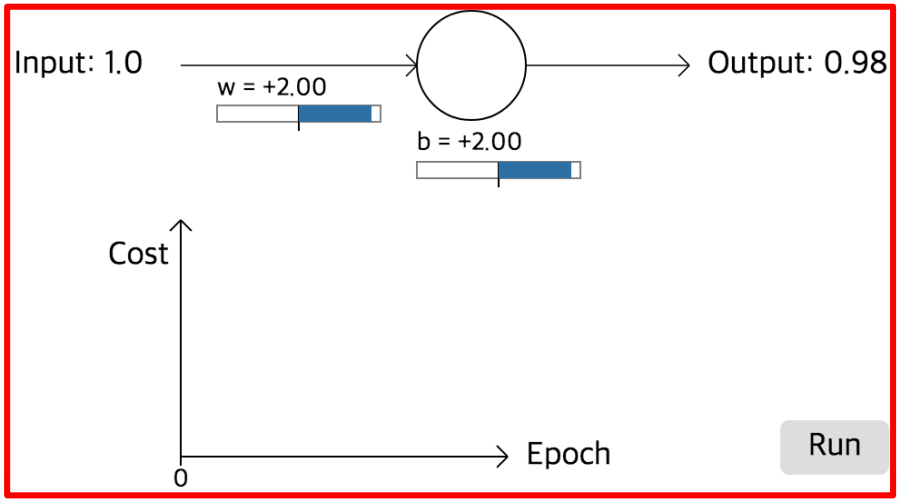


  - Weight w와 bias b 초기값에 따른 학습 속도 비교(학습율 $\eta = 0.15$로 동일하게 설정)

(1) 초기 w=0.6, 0.9일 경우의 학습 특성?



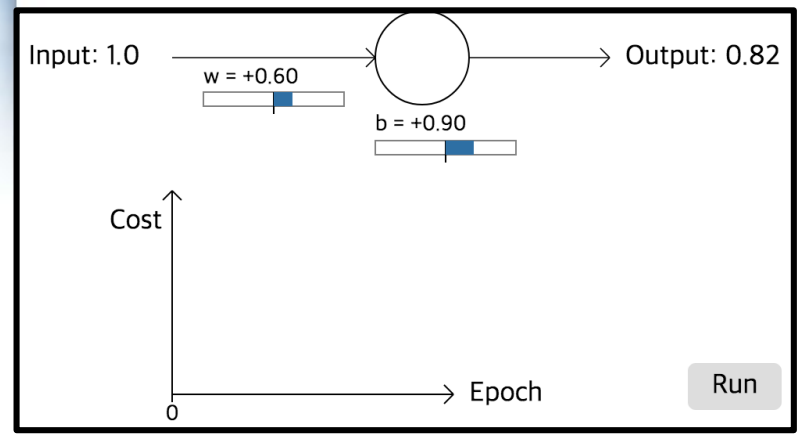
(2) 초기 w=2.0, 2.0일 경우의 학습 특성?

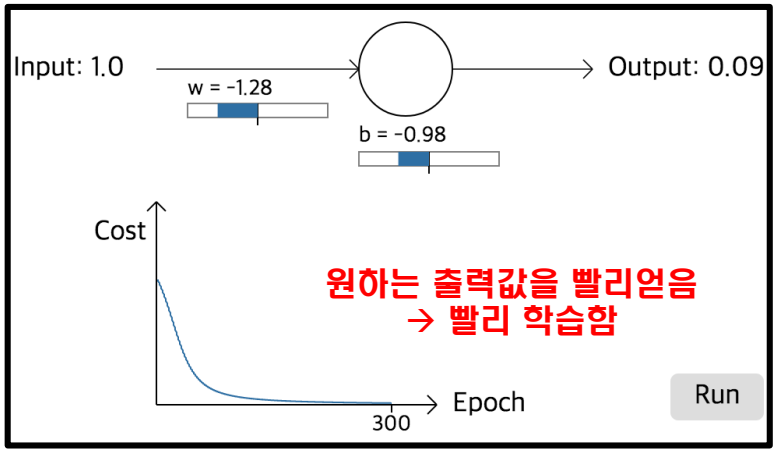


Cross-entropy cost function

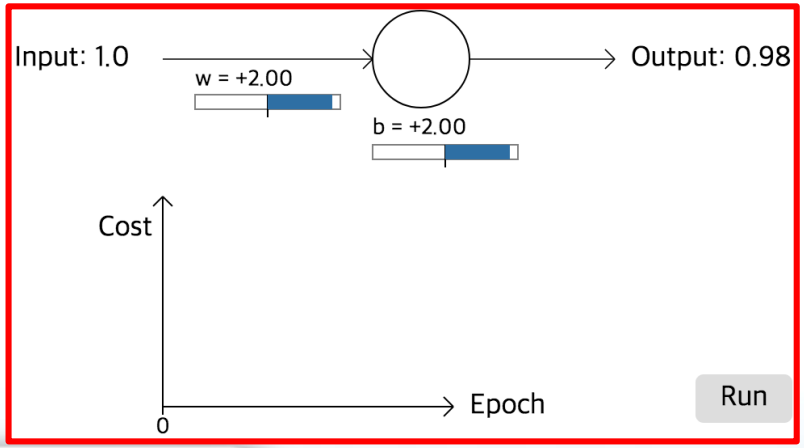
Inversion을 수행 neural network의 초기값 설정에 따른 학습 특성 비교(2/2)

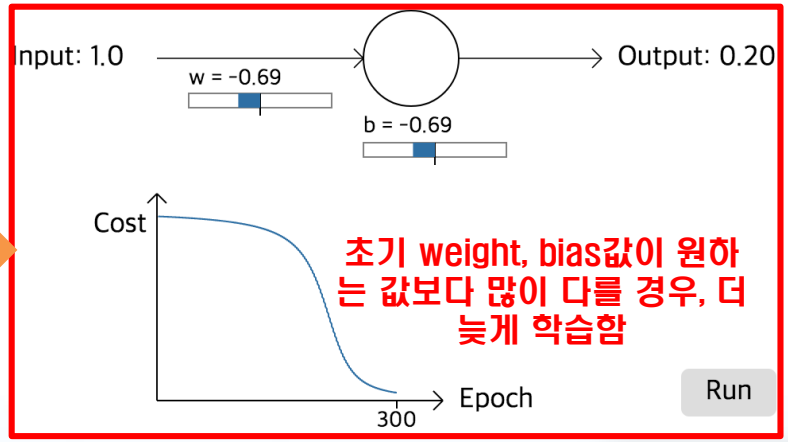
(1) 초기 w=0.6, 0.9일 경우의 학습 특성?





(2) 초기 w=2.0, 2.0일 경우의 학습 특성?





Cross-entropy cost function

- 학습 속도가 느리다는 것은 $\partial C/\partial w$, $\partial C/\partial b$ 미분값이 작다는 것을 의미함
  - 왜 미분값들이 작은건가?

To understand the origin of the problem, consider that our neuron learns by changing the weight and bias at a rate determined by the partial derivatives of the cost function, $\partial C/\partial w$ and $\partial C/\partial b$. So saying "learning is slow" is really the same as saying that those partial derivatives are small. The challenge is to understand why they are small. To understand that, let's compute the partial derivatives. Recall that we're using the quadratic cost function, which, from Equation (6), is given by

$$C = \frac{(y-a)^2}{2}, \tag{54}$$

Cross-entropy cost function

- **학습 속도가 느리다는 것은 $\partial C/\partial w$, $\partial C/\partial b$ 미분값이 작다는 것을 의미함**
  - **왜 미분값들이 작은건가?**

$$C = \frac{(y-a)^2}{2}, \tag{54}$$

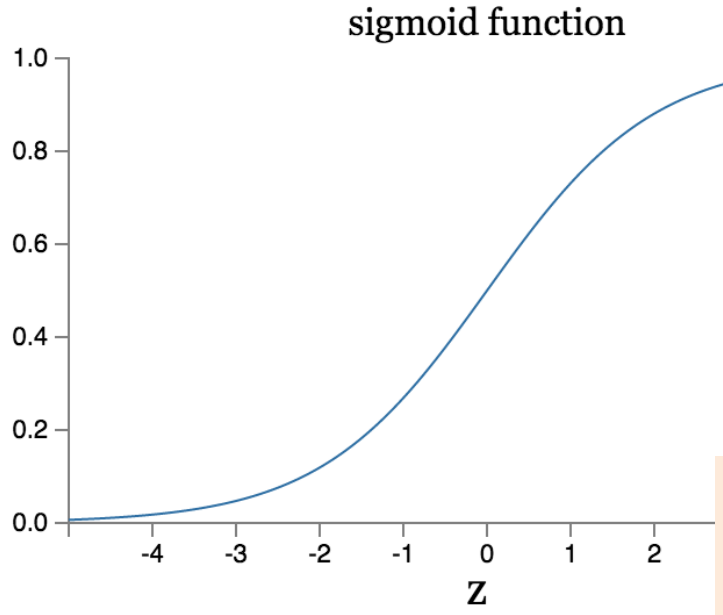
where a is the neuron's output when the training input $x = 1$ is used, and $y = 0$ is the corresponding desired output. To write this more explicitly in terms of the weight and bias, recall that $a = \sigma(z)$, where $z = wx + b$. Using the chain rule to differentiate with respect to the weight and bias we get

$$\frac{\partial C}{\partial w} = (a-y)\sigma'(z)x = a\sigma'(z) \tag{55}$$

$$\frac{\partial C}{\partial b} = (a-y)\sigma'(z) = a\sigma'(z), \tag{56}$$

where I have substituted $x = 1$ and $y = 0$. To understand the behaviour of these expressions, let's look more closely at the $\sigma'(z)$ term on the right-hand side. Recall the shape of the σ function:

term on the right-hand side. Recall the shape of the σ function:



$$\frac{\partial C}{\partial w} = (a-y)\sigma'(z)x = a\sigma'(z) \tag{55}$$

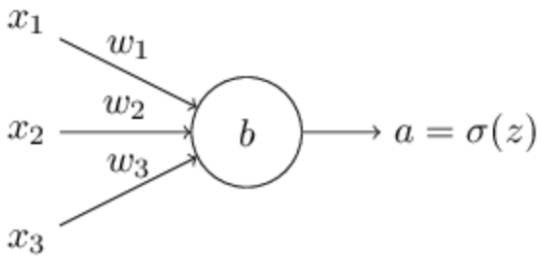
$$\frac{\partial C}{\partial b} = (a-y)\sigma'(z) = a\sigma'(z), \tag{56}$$

$\partial C/\partial w$, $\partial C/\partial b$는 $\sigma'(z)$에 **의존함.**
Neuron의 출력이 1에 가까울 수록 $\sigma'(z)$ 값은 매우 작아짐 → 이때문에 학습 속도가 낮아지는 것임

We can see from this graph that when the neuron's output is close to 1, the curve gets very flat, and so $\sigma'(z)$ gets very small. Equations (55) and (56) then tell us that $\partial C/\partial w$ and $\partial C/\partial b$ get very small. This is the origin of the learning slowdown. What's more, as we shall see a little later, the learning slowdown occurs for essentially the same reason in more general neural networks, not just the toy example we've been playing with.

Cross-entropy cost function

- Cost 함수로 quadratic cost 대신, cross-entropy 사용하여 slow down 해결 가능함
  - different cost function, known as the cross-entropy. To understand the cross-entropy, let's move a little away from our super-simple toy model. We'll suppose instead that we're trying to train a neuron with several input variables, $x_1, x_2, \ldots$, corresponding weights $w_1, w_2, \ldots$, and a bias, b:



The output from the neuron is, of course, $a = \sigma(z)$, where $z = \sum_j w_j x_j + b$ is the weighted sum of the inputs. We define the cross-entropy cost function for this neuron by

$$C = -\frac{1}{n} \sum_x [y \ln a + (1-y) \ln(1-a)], \tag{57}$$

where n is the total number of items of training data, the sum is over all training inputs, x, and y is the corresponding desired output.

Cross-entropy cost function

see this, let's compute the partial derivative of the cross-entropy cost with respect to the weights. We substitute $a = \sigma(z)$ into (57), and apply the chain rule twice, obtaining:

$$\frac{\partial C}{\partial w_j} = -\frac{1}{n}\sum_x \left(\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)}\right)\frac{\partial \sigma}{\partial w_j} \tag{58}$$

$$= -\frac{1}{n}\sum_x \left(\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)}\right)\sigma'(z)x_j. \tag{59}$$

Putting everything over a common denominator and simplifying this becomes:

$$\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_x \frac{\sigma'(z)x_j}{\sigma(z)(1-\sigma(z))}(\sigma(z)-y). \tag{60}$$

Using the definition of the sigmoid function, $\sigma(z) = 1/(1+e^{-z})$, and a little algebra we can show that $\sigma'(z) = \sigma(z)(1-\sigma(z))$. I'll ask you to verify this in an exercise below, but for now let's accept it as given.

Cross-entropy cost function

We see that the $\sigma'(z)$ and $\sigma(z)(1-\sigma(z))$ terms cancel in the equation just above, and it simplifies to become:

$$\frac{\partial C}{\partial w_j} = \frac{1}{n} \sum_x x_j (\sigma(z) - y). \tag{61}$$

This is a beautiful expression. It tells us that the rate at which the weight learns is controlled by $\sigma(z) - y$, i.e., by the error in the output. The larger the error, the faster the neuron will learn. This is just what we'd intuitively expect. In particular, it avoids the learning slowdown caused by the $\sigma'(z)$ term in the analogous equation for the quadratic cost, Equation (55). When we use the cross-entropy, the $\sigma'(z)$ term gets canceled out, and we no longer need worry about it being small. This cancellation is the special miracle ensured by the cross-entropy cost function. Actually, it's not really a miracle. As

Cross-entropy cost function

the cross-entropy cost function. Actually, it's not really a miracle. As we'll see later, the cross-entropy was specially chosen to have just this property.

In a similar way, we can compute the partial derivative for the bias. I won't go through all the details again, but you can easily verify that

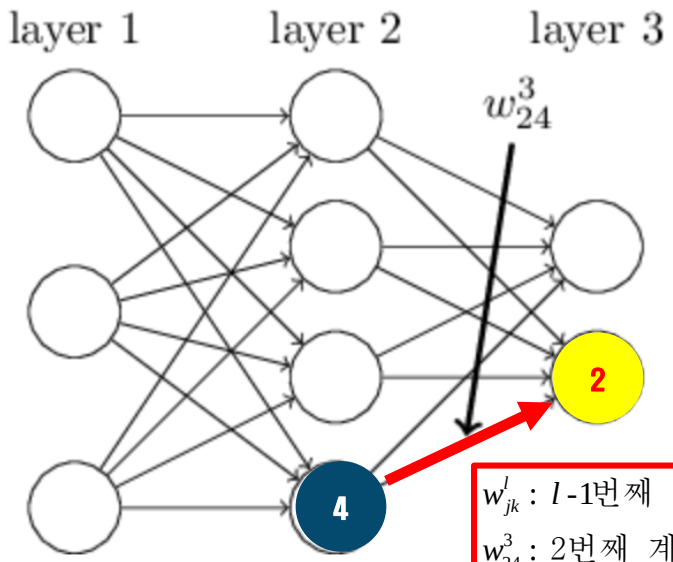
$$\frac{\partial C}{\partial b} = \frac{1}{n} \sum_x (\sigma(z) - y). \tag{62}$$

Again, this avoids the learning slowdown caused by the $\sigma'(z)$ term in the analogous equation for the quadratic cost, Equation (56).

Notation 정의

본 자료에서 사용하는 neural network 관련 notation : weight 정의

Let's begin with a notation which lets us refer to weights in the network in an unambiguous way. We'll use w^l_{jk} to denote the weight for the connection from the k^{th} neuron in the $(l-1)^{\text{th}}$ layer to the j^{th} neuron in the l^{th} layer. So, for example, the diagram below shows the weight on a connection from the fourth neuron in the second layer to the second neuron in the third layer of a network:



w^l_{jk} is the weight from the k^{th} neuron in the $(l-1)^{\text{th}}$ layer to the j^{th} neuron in the l^{th} layer

w^l_{jk} : l-1번째 계층의 k번째 neuron에서 l번째 계층 j번째 neuron간의 weight값

w^3_{24} : 2번째 계층의 4번째 neuron에서 3번째 계층 2번 neuron간의 weight값

Notation 정의

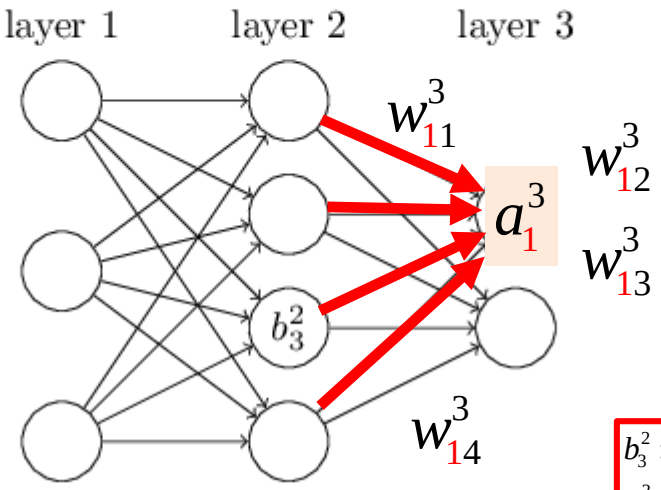
본 자료에서 사용하는 neural network 관련 notation: Bias와 activation

We use a similar notation for the network's biases and activations.

Explicitly, we use b^l_j for the bias of the j^{th} neuron in the l^{th} layer.

And we use a^l_j for the activation of the j^{th} neuron in the l^{th} layer.

The following diagram shows examples of these notations in use:



b^2_3 : 계층2의 3번째 neuron의 bais 값

a^3_1 : 계층3의 첫번째 neuron의 activation 값

– Activation은 다음과 같이 정의됨

$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right), \qquad (23)$$

z^l_j : Weighted input으로 정의함

Softmax

0에서 1의 값을 가지는 Softmax

- Softmax의 출력을 확률 분포로 생각할 수 있음
  - 출력에 Sigmoid 함수를 사용할 경우는 확률 분포를 만들어내지 못함

The idea of softmax is to define a new type of output layer for our neural networks. It begins in the same way as with a sigmoid layer, by forming the weighted inputs* $z^L_j = \sum_k w^L_{jk} a^{L-1}_k + b^L_j$. However, we don't apply the sigmoid function to get the output. Instead, in a softmax layer we apply the so-called *softmax function* to the z^L_j. According to this function, the activation a^L_j of the jth output neuron is

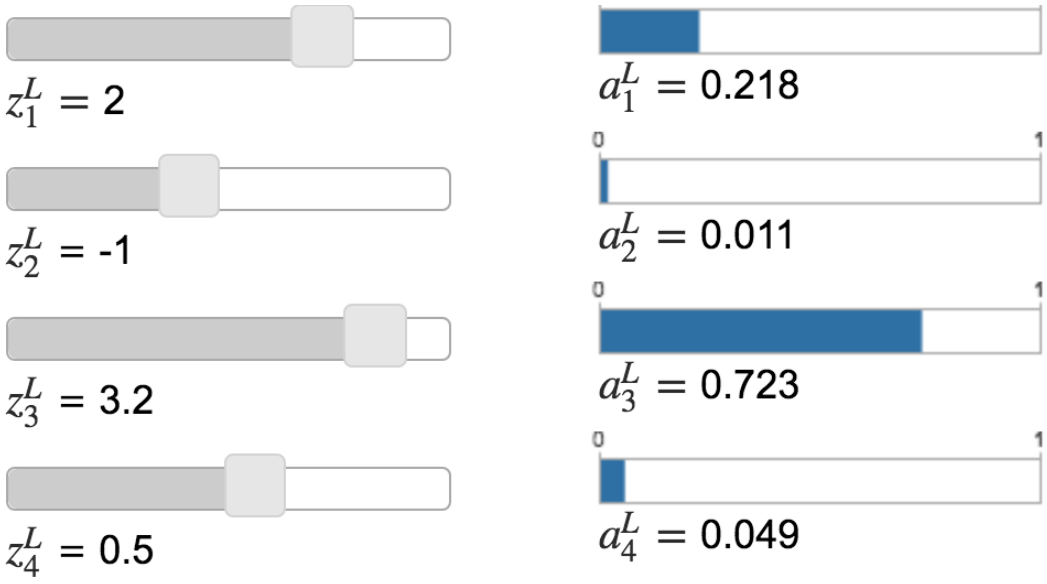
$$a^L_j = \frac{e^{z^L_j}}{\sum_k e^{z^L_k}}, \tag{78}$$

where in the denominator we sum over all the output neurons.

Softmax

0에서 1의 값을 가지는 Softmax

– 아래의 weighted input 값 조정에 따른 output activation 값 확인 가능함



As you increase z_4^L, you'll see an increase in the corresponding output activation, a_4^L, and a decrease in the other output activations.

$$\sum_j a_j^L = \frac{\sum_j e^{z_j^L}}{\sum_k e^{z_k^L}} = 1. \qquad (79)$$

Overfitting과 Regularization

Overfitting 이슈

The Nobel prizewinning physicist Enrico Fermi was once asked his opinion of a mathematical model some colleagues had proposed as the solution to an important unsolved physics problem. The model gave excellent agreement with experiment, but Fermi was skeptical. He asked how many free parameters could be set in the model. "Four" was the answer. Fermi replied*: "I remember my friend Johnny von Neumann used to say, with four parameters I can fit an elephant, and with five I can make him wiggle his trunk.".

The point, of course, is that models with a large number of free parameters can describe an amazingly wide range of phenomena. Even if such a model agrees well with the available data, that doesn't make it a good model. It may just mean there's enough freedom in the model that it can describe almost any data set of the given size, without capturing any genuine insights into the underlying phenomenon. When that happens the model will work well for the existing data, but will fail to generalize to new situations. The true test of a model is its ability to make predictions in situations it hasn't been exposed to before.

많은 parameter를 사용하면, 주어진 data에 대해서 매우 잘 적용됨. 하지만, 새로운 데이터에 대한 일반화 특성은 좋지 않은 경우 발생 → overfitting 이슈임

Overfitting과 Regularization

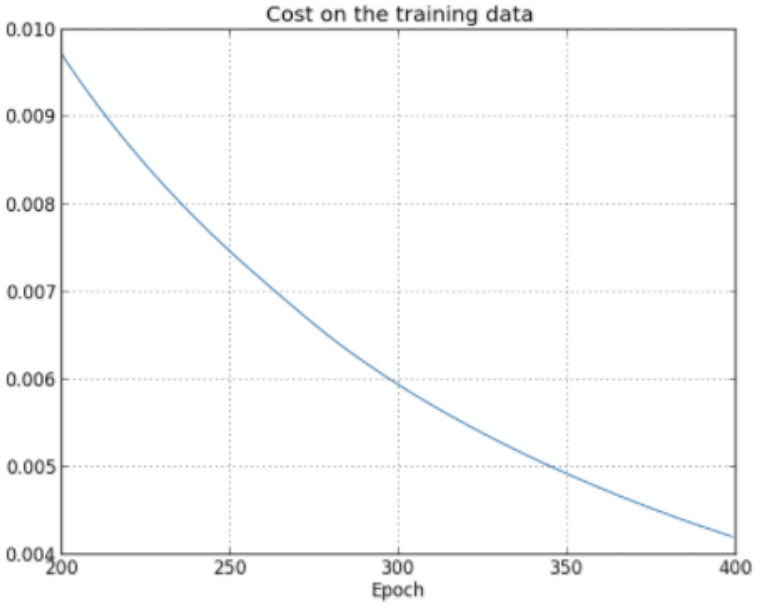
Overfitting 이슈

- 앞 페이지의 예에서, Von Neumann은 4개의 free parameter만 있으면 많은 다양한 현상을 모델링할 수 있다고 했음(일반화하지는 못하면서..)
- 그런데, Neural network의 경우에는
  - 30개의 hidden neural network을 가지는 경우, 24,000 개의 parameter가 있을 수 있으며,
  - 100개의 hidden neural network을 가지는 경우, 80,000개의 parameter를 가짐 (MNIST 0…9 분류 문제에서..)
  - 심지어, deep neural network일 경우, 수 백만~ 수 십억 개의 parameter를 가질 수 있음

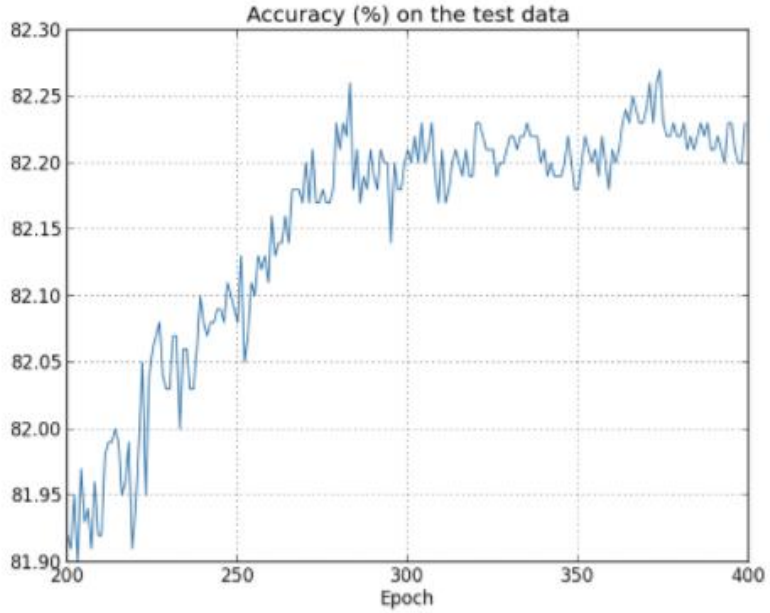
Overfitting과 Regularization

Overfitting 이슈

Using the results we can plot the way the cost changes as the network learns* :



Let's now look at how the classification accuracy on the test data changes over time:



Cost는 학습이 반복됨에 따라 계속 줄어듬. 즉 error가 계속 줄어듦

하지만, 실제 test data에 대한 정확도를 보면, epoch 280부터는 더 이상 정확도가 높아지지 않음, 즉 이때 overfitting 상황이 시작된 것임

Overfitting과 Regularization

- Overfitting 줄이는 방법
  - 더 많은 training data 사용
  - L1 regularization, dropout 사용
  - Network size를 줄이는 방법 → 큰 network이 더 많은 능력을 가짐 (상충됨)
  - Training data와 fixed network size인 경우:
    - Regularization 기법 사용 (L2 regularization)

I describe one of the most commonly used regularization techniques, a technique sometimes known as *weight decay* or *L2 regularization*. The idea of L2 regularization is to add an extra term to the cost function, a term called the *regularization term*. Here's the regularized cross-entropy:

$$C = -\frac{1}{n}\sum_{xj}\left[y_j \ln a_j^L + (1-y_j)\ln(1-a_j^L)\right] + \frac{\lambda}{2n}\sum_w w^2. \qquad (85)$$

The first term is just the usual expression for the cross-entropy. But we've added a second term, namely the sum of the squares of all the weights in the network. This is scaled by a factor $\lambda/2n$, where $\lambda > 0$ is known as the *regularization parameter*, and n is, as usual, the size of our training set. I'll discuss later how λ is chosen. It's also worth

Overfitting과 Regularization

Overfitting 줄이는 방법 : L2 Regularization 사용

Intuitively, the effect of regularization is to make it so the network prefers to learn small weights, all other things being equal. Large weights will only be allowed if they considerably improve the first part of the cost function. Put another way, regularization can be viewed as a way of compromising between finding small weights and minimizing the original cost function. The relative importance of the two elements of the compromise depends on the value of λ: when λ is small we prefer to minimize the original cost function, but when λ is large we prefer small weights.

$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2, \tag{87}$$

where C_0 is the original, unregularized cost function.

Overfitting과 Regularization

Overfitting 줄이는 방법 : L1 Regularization 사용

L1 regularization: In this approach we modify the unregularized cost function by adding the sum of the absolute values of the weights:

$$C = C_0 + \frac{\lambda}{n} \sum_w |w|. \tag{95}$$

Intuitively, this is similar to L2 regularization, penalizing large weights, and tending to make the network prefer small weights. Of course, the L1 regularization term isn't the same as the L2 regularization term, and so we shouldn't expect to get exactly the same behaviour. Let's try to understand how the behaviour of a network trained using L1 regularization differs from a network trained using L2 regularization.

Overfitting과 Regularization

L1 vs. L2 Regularization

L1 $$C = C_0 + \frac{\lambda}{n} \sum_w |w|. \tag{95}$$

L2 $$C = C_0 + \frac{\lambda}{2n} \sum_w w^2, \tag{87}$$

where C_0 is the original, unregularized cost function.

– L1 regularization:
  • Constant 양만큼 줄임

– L2 regularization:
  • Weight w의 크기에 비례하여 줄임

L2 regularization	L1 regularization
Computational efficient due to having analytical solutions	Computational inefficient on non-sparse cases
Non-sparse outputs	Sparse outputs

– W값이 크다면 L2 regularization이 더 크게 줄임. 하지만 w 값이 작은 값이라면, L1 regularization이 weight를 더 많이 줄이는 효과를 가짐

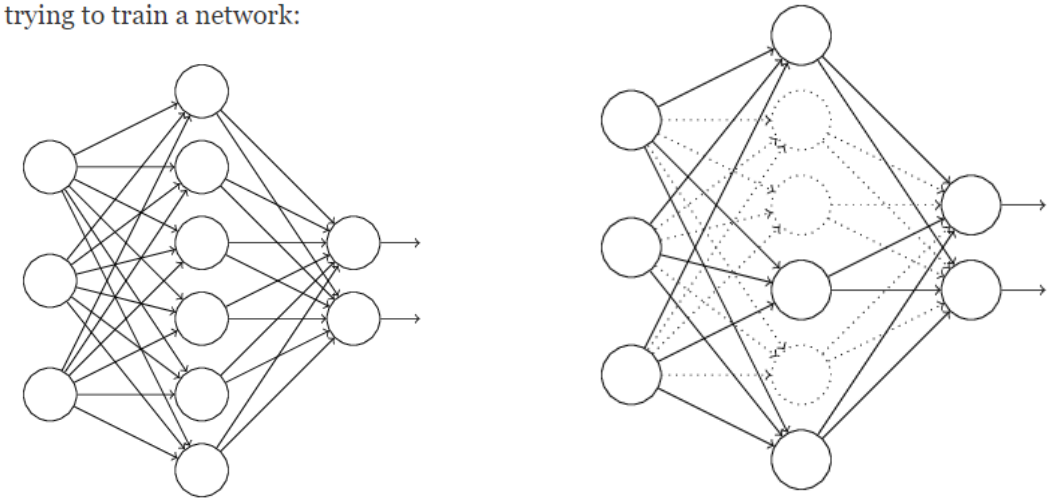
– 이 때문에. L1은 연결이 많지 않은, 하지만, 몇 개의 중요한 연결이 있는 sparse한 network에서 사용하는 것이 좋음

Dropout 기법

Dropout 기법

– L1, L2 regularization 기법은 cost를 건드렸지만, Dropout 기법은 cost function과는 관계없이 network 그 자체를 다룸

Suppose we're trying to train a network:



In particular, suppose we have a training input x and corresponding desired output y. Ordinarily, we'd train by forward-propagating x through the network, and then backpropagating to determine the contribution to the gradient. With dropout, this process is modified. We start by randomly (and temporarily) deleting half the hidden neurons in the network, while leaving the input and output neurons untouched. After doing this, we'll end up with a network along the following lines. Note that the dropout neurons, i.e., the neurons which have been temporarily deleted, are still ghosted in:

Dropout 기법

What's this got to do with dropout? Heuristically, when we dropout different sets of neurons, it's rather like we're training different neural networks. And so the dropout procedure is like averaging the effects of a very large number of different networks. The different networks will overfit in different ways, and so, hopefully, the net effect of dropout will be to reduce overfitting.

A related heuristic explanation for dropout is given in one of the earliest papers to use the technique*: "This technique reduces complex co-adaptations of neurons, since a neuron cannot rely on the presence of particular other neurons. It is, therefore, forced to learn more robust features that are useful in conjunction with many different random subsets of the other neurons." In other words, if we think of our network as a model which is making predictions, then we can think of dropout as a way of making sure that the model is robust to the loss of any individual piece of evidence. In this, it's somewhat similar to L1 and L2 regularization, which tend to reduce weights, and thus make the network more robust to losing any individual connection in the network.

Thanks~

Q & A