Notes for Neural Networks and Deep Learning

Moonsu Kang

July 2016

1 Introduction

This note is written to cover contents from *Neural Networks and Deep Learning* by Michael Nielson. (http://neuralnetworksanddeeplearning.com/index.html)

2 Chapter 1: using neural nets to recognize handwritten digits

Neural networks - Learning Algorithm to tune weights and biases of a network of neurons

2.1 Sigmoid Neurons

Neuron that make small change in outputs from small input changes. Smoothness is the key differentiation from perceptrons. Sigmoid function:

$$\sigma(z) = \frac{1}{1 + exp(-\sum_j w_j x_j - b)}$$

2.2 Architecture of neural networks

1) Feedforward neural networks
2) recurrent neural networks
3) hidden layer

2.3 Gradient Descent in neural network

We are interested in quantifying the goal and measure the impacts of adjustments on weights and biases. For this purpose, cost function is served. While there may be a number of functions that can serve the purspose, one of them is MSE:

$$C(w,b) = \frac{1}{2n} \sum_x \|(y(x) - a)\|^2$$

where x is input and a is the vector of activations output from x input. y(x) is the corresponding desired output.

our goal in training a neural network is to find weights and biases which minimize the quadratic cost function C(w,b).

$$\Delta C \approx \frac{dC}{dv_1}\Delta v_1 + \frac{dC}{dv_1}\Delta v_1$$

Above equation equals to:

$$\Delta C \approx \nabla C \cdot \Delta v$$

where $\nabla C \equiv (\frac{dC}{dv_1}, \frac{dC}{dv_2})^T$. ∇C relates changes in v to changes in C. Then as long as

$$\Delta v = -\eta \nabla C$$

, where η is small, positive, then

$$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2$$

In other words, ΔC is always decreasing. So when we compute for v, we look for v':

$$v \rightarrow v' = v - \eta \nabla C$$

In Neural network, we would apply gradient descent to weights and bias. Stochastic gradient descent - to estimate the gradient ΔC by computing ΔC_x for a small sample of randomly chosen x.

$$w_k \rightarrow w_k' = w_k - \frac{\eta}{m}\sum_j \frac{dC_{X_j}}{dw_k}$$

$$b_l \rightarrow b_l' = b_l - \frac{\eta}{m}\sum_j \frac{dC_{X_j}}{db_l}$$

Hyperparameters to choose from: epochs (number of training), eta (learning rate), mini-batch size, size of hidden layer

3 Chapter 2: How the backpropagation algorithm works

Use backpropagation for gradient of cost function.

3.1 Computation of output from neural network

$$a_j^l = \sigma(\sum_k w_{jk}^l a_k^{l-1} + b_j^l)$$

where w_{jk}^l is weight from the k^{th} neuron of $(l-1)^{th}$ layer to j^{th} neuron of l^{th} and a_k^{l-1} is activation of k^{th} neuron from $(l-1)^{th}$ layer.

3.2 Assumptions on cost fuction

1) Assumption 1:

$$C = C(w, b) = \frac{1}{2n}\sum_x \|(y(x) - a)\|^2$$

$$= \frac{1}{n}\sum_x C_x$$

where $C_x = \frac{1}{2}\|y - a^L\|^2$.
Backpropagation lets us compute partial derivatives $\frac{dC_x}{dw}$ and $\frac{dC_x}{db}$ for a single training example. So assumption that we can find $\frac{dC}{dw}$ and $\frac{dC}{db}$ from averaging over training examples is necessary. 2) Assumption 2: cost fuction can be written as a function of the outputs from the neural network.

$$C = C(a^L)$$

3.3 Hadamard product

$$s \odot t$$

for elementwise product of two vectors

3.4 Fundamental Equations

$$\delta_j^l \equiv \frac{dC}{dz_j^l}$$

where δ_j^l is the vector of errors associated with layer l.

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$$

$$a^l = \sigma(z^l)$$

Backpropagation will compute δ^l for every layer and relating those errors with $\frac{dC}{dw_{jk}^l}$ and $\frac{dC}{db_j^l}$.

1) Equation for the error in the output layer

$$\delta_j^L = \frac{dC}{da_j^L}\sigma^{'}(z_j^L) \tag{1}$$

The first term measures how fast the cost is changing as a function of the j^{th} output activation. Ther second term measures how fast the activation function σ is changing at z_j^L. To rewrite, it is:

$$\delta^L = \nabla_a C \odot \sigma^{'}(z^L)$$

2) Equation for the error in terms of the error in the next layer

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma^{'}(z^l) \tag{2}$$

where $(w^{l+1})^T$ is the transpose of the weight matrix w^{l+1}.

3) Equation for the rate of change of the cost with respect to any bias in the network

$$\frac{dC}{db_j^l} = \delta_j^l \tag{3}$$

4) Equation for the rate of change of the cost with respect to any weight in the network:

$$\frac{dC}{dw_{jk}^l} = a_k^{l-1}\delta_j^l = a_{in}\delta_{out} \tag{4}$$

3.5 Interpretation

When $\sigma(z_j^l)$ is approximately 0 or 1, $\sigma^{'}(z_j^l)$ is close to 0, i.e. the output neuron has saturated and the weight has stopped learning. This can be observed from both (1) and (2) equations. Any weights input to a saturated neuron will learn slowly. From equation (4), when activation input is low, the learning rate is slow as well. In summary, weight will learn slowly if either the input neuron is low activation or if the output neuron has saturated (either high or low activation).

3.6 Backpropagation Algorithm

1. **Input x**:
Set the corresponding activation a^1 for the input layer.

2. **Feedforward**:
For $l = 2, 3, ..L$, compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$.

3. **Output error δ^L**:
compute the vector $\delta^L = \nabla_a C \odot \sigma^{'}(z^l)$.

4. **Backpropagate the error**:
For each $l = L-1, L-2, ...,$ compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma^{'}(z^l)$

5. **Output**:
The gradient of the cost function is given by $\frac{dC}{dw_{jk}^l} = a_k^{l-1}\delta_j^l$ and $\frac{dC}{db_j^l} = \delta_j^l$. 6.
Gradient descent:
For each $l = L, L-1, ...,$ update the weights and bias based on results from 5.

4 Chapter 3: Improving the way neural networks learn

This chapter covers a choice of cost function, known as cross-entropy cost function; four regularization methods, better initializing weights in the network, and a set of heuristics to help choosing good hyper-parameters for the network.

4.1 Cross-entropy cost function

Learning rate is slow at far off because sigmoid function is quite flat at ends. Thus resulting $\frac{dC}{dw}$ and $\frac{dC}{db}$ to be small. Cross-entropy cost function can be useful for this reason.

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

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 function is non-negative and if the neuron's actual output is close to the desired output then the cross-entropy will be close to zero.

$$\frac{dC}{dw_j} = -\frac{1}{n}\sum_x (\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)})\frac{d\sigma}{dw_j}$$

$$= -\frac{1}{n}\sum_x (\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)})\sigma^{'}(z)x_j$$

$$= \frac{1}{n}\sum_x (\frac{\sigma^{'} x_j}{\sigma(z)(1-\sigma(z))})(\sigma(z) - y)$$

Using the definition of sigmoid function, $\sigma(z) = 1/(1+e^{-z})$, and that $\sigma^{'} = \sigma(z)(1-\sigma(z))$,

$$\frac{dC}{dw_j} = \frac{1}{n}\sum_x x_j(\sigma(z) - y) \tag{5}$$

Note that we are free from $\sigma^{'}$! Similarly,

$$\frac{dC}{db} = \frac{1}{n}\sum_x (\sigma(z) - y) \tag{6}$$

It's about how the speed of learning changes. In particular, when we use the quadratic cost learning is slower when the neuron is unambiguously wrong than it is later on, as the neuron gets closer to the correct output; while with the cross-entropy learning is faster when the neuron is unambiguously wrong.

4.2 Softmax

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* *In describing the softmax we'll make frequent use of notation introduced in the last chapter. You may wish to revisit that chapter if you need to refresh your memory about the meaning of the notation. $z_j^L = \sum_k w_{jk}^L a_k^{L-1} + b_j^L$. 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_j^L. According to this function, the activation a_j^L of the jth output neuron is

$$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}}$$

where $\sum_j a_j^L = \frac{\sum_j e^{z_j^L}}{\sum_k e^{z_k^L}} = 1$.
Softmax is a way of rescaling z_j^L and forming a probability distribution.

With log likelihood cost function, $C \equiv -\ln a_y^L$, the derivatives come out to be:

$$\frac{dC}{dw_{jk}^L} = a_k^{L-1}(a_j^L - y_j) \tag{7}$$

$$\frac{dC}{db_j^L} = a_j^L - y_j \tag{8}$$

These equations are the same as ones from cross-entropy. Both (sigmoid layer + cross-entropy cost function) and (softmax layer + log likelihood cost function) approaches work well. Softmax allows one to interpret the output activations as probabilities.

4.3 Overfit

Overfitting is a major problem in neural networks. There are number of approaches to reduce the impacts of overfitting. As a part, we use validation data set to determine when to stop training. This strategy is called 'early stopping'. When Validation set is used to evaluate trial choices of hyperparameters such as the number of epochs to train for, the learning rate, the best network architecture,..., it's called as 'hold out' method.

4.4 L2 Regularization

L2 regularization (weight decay): add the weights to cost function

$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2 \tag{9}$$

where C_0 is the original, unregularized cost function. Regularization is a way of trading off between small weights and minimizing original cost function. λ is significant.

$$\frac{dC}{dw} = \frac{dC_0}{dw} + \frac{\lambda}{n}w$$
$$\frac{dC}{db} = \frac{dC_0}{dw}$$

Then, gradient descent learning for weights would be:

$$w \to w - \eta\frac{dC_0}{dw} - \frac{\eta\lambda}{n}w \tag{10}$$
$$= (1 - \frac{\eta\lambda}{n})w - \eta\frac{dC_0}{dw} \tag{11}$$

This means that weight decay faster than before, or is considered as weight rescaling before learning. Bias is not affected. Regularized network learns to respond to types of evidence which are seen often across the training set. Regularized networks are constrained to build relatively simple models based on patterns seen often in the training data, and are resistant to learning peculiarities of the noise in the training data.

4.5 L1 Regularization

The cost function for L1 regularization is:

$$C = C_0 + \frac{\lambda}{n}\sum_w |w| \tag{12}$$

This means that derivatives would be

$$\frac{dC}{dw} = \frac{dC_0}{dw} + \frac{\lambda}{n}sgn(w) \tag{13}$$

and gradient descent for weights will be:

$$w \to w' = w - \frac{\eta\lambda}{n}sgn(w) - \eta\frac{dC_0}{dw} \tag{14}$$

In L1 regularization, the weights shrink by a constant amount toward 0. In L2 regularization, the weights shrink by an amount which is proportional to w. So when a particular weight has a large magnitude, $|w|$, L1 regularization shrinks the weight much less than L2 regularization does.By contrast, when $|w|$ is small, L1 regularization shrinks the weight much more than L2 regularization.

4.6 Dropout

Unlike L1 and L2 regularization, dropout doesn't rely on modifying the cost function. Instead, in dropout we modify the network itself. We start by randomly (and temporarily) deleting half the hidden neurons in the network, while leaving the input and output neurons untouched.We forward-propagate the input xx through the modified network, and then backpropagate the result, also through the modified network. After doing this over a mini-batch of examples, we update the appropriate weights and biases. We then repeat the process, first restoring the dropout neurons, then choosing a new random subset of hidden neurons to delete, estimating the gradient for a different mini-batch, and updating the weights and biases in the network. 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.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.

4.7 Artificially expanding the training data

The general principle is to expand the training data by applying operations that reflect real-world variation. For example, we can expand our training data by making many small rotations of all the MNIST training images, and then using the expanded training data to improve our network's performance.

4.8 Weight Initialization

Choosing both the weights and biases using independent Gaussian random variables, normalized to have mean 0 and standard deviation 1, is not the best as it leaves z, a hidden neuron, with broad Gaussian distribution. This means that when $|z| >= 1$ the output $\sigma(z)$ from the hidden neuron will be very close to 1 or 0, meaning that it will have saturated. Once it happens, making small changes in the weights will make only absolutely miniscule changes in the activation of our hidden neuron. That miniscule change in the activation of the hidden neuron will, in turn, barely affect the rest of the neurons in the network at all, and we'll see a correspondingly miniscule change in the cost function.As a result, those weights will only learn very slowly when we use the gradient descent algorithm.

Suppose we have a neuron with n_{in} input weights. Then we shall initialize those weights as Gaussian random variables with mean 0 and standard deviation $1/\sqrt{(}n_{in})$. That is, we'll squash the Gaussians down, making it less likely that our neuron will saturate. $z = \sum_j w_j x_j + b$ will again be a Gaussian random variable with mean 0, but it'll be much more sharply peaked than before. We'll continue to initialize the biases as before, as Gaussian random variables with a mean of 0 and a standard deviation of 1. This is okay, because it doesn't make it too much more likely that our neurons will saturate. Better initialization is not only the speed of learning which is improved, it's sometimes also the final performance.

4.9 Hyperparameters: λ, η, minibatch size

Broad Strategy:
1. Simplify by reducing to smaller goal and experiment Get rid of all the training and validation images except images which are 0s or 1s. Then try to train a network to distinguish 0s from 1s. Not only is that an inherently easier problem than distinguishing all ten digits, it also reduces the amount of training data by 80 percent, speeding up training by a factor of 5. That enables much more rapid experimentation, and so gives you more rapid insight into how to build a good network.

2. Simple network You can speed up experimentation by stripping your network down to the simplest network likely to do meaningful learning.

3. Increase frequency of monitoring; one monitoring for smaller training and validation set size We can get feedback more quickly by monitoring the validation accuracy more often, say, after every 1,000 training images. Furthermore, instead of using the full 10,000 image validation set to monitor performance, we can get a much faster estimate using just 100 validation images.

We continue, individually adjusting each hyper-parameter, gradually improving performance and bump up the network repeatedly 1. experiment with η, λ 2. Increase network 3. Decrease frequency of monitoring As we do so, it typically takes longer to witness the impact due to modifications of the hyper-parameters, and so we can gradually decrease the frequency of monitoring. In essence, break it down to simplest form and build it up.

4.9.1 Early stopping

Use early stopping to determine **the number of training epochs**: As we discussed earlier in the chapter, early stopping means that at the end of each epoch we should compute the classification accuracy on the validation data. When that stops improving, terminate. This makes setting the number of epochs very simple. In particular, it means that we don't need to worry about explicitly figuring out how the number of epochs depends on the other hyper-parameters. Instead, that's taken care of automatically. Furthermore, early stopping also automatically prevents us from overfitting. A good rule is to terminate if the best classification accuracy doesn't improve for quite some time. I suggest using the no-improvement-in-ten rule for initial experimentation, and gradually adopting more lenient rules, as you better understand the way your network trains: no-improvement-in-twenty, no-improvement-in-fifty, and so on.

4.9.2 Learning rate schedule

Instead of constant learning rate, it's best to use a large learning rate that causes the weights to change quickly and gradually decrease for fine tuning. One natural approach is to use the same basic idea as early stopping. The idea is to hold the learning rate constant until the validation accuracy starts to get worse. Then decrease the learning rate by some amount, say a factor of two or ten.

4.9.3 regularization parameter, λ

I suggest starting initially with no regularization ($\lambda = 0.0$), and determining a value for η, as above. Using that choice of η, we can then use the validation data to select a good value for λ. Start by trialling $\lambda = 1.0$, and then increase or decrease by factors of 10, as needed to improve performance on the validation data. Once you've found a good order of magnitude, you can fine tune your value of λ. That done, you should return and re-optimize η again.

4.9.4 minibatch size

Choosing the best mini-batch size is a compromise. Too small, and you don't get to take full advantage of the benefits of good matrix libraries optimized for fast hardware. Too large and you're simply not updating your weights often enough. What you need is to choose a compromise value which maximizes the speed of learning. Fortunately, the choice of mini-batch size at which the speed is maximized is relatively independent of the other hyper-parameters (apart from the overall architecture), so you don't need to have optimized those hyper-parameters in order to find a good mini-batch size. The way to go is therefore to use some acceptable (but not necessarily optimal) values for the other hyper-parameters, and then trial a number of different mini-batch sizes, scaling η as above. Plot the validation accuracy versus time (as in, real elapsed time, not epoch!), and choose whichever mini-batch size gives you the most rapid improvement in performance. With the mini-batch size chosen you can then proceed to optimize the other hyper-parameters.

4.9.5 Automated techniques

A common technique is grid search, which systematically searches through a grid in hyper-parameter space. (Random search for hyper-parameter optimization, by James Bergstra and Yoshua Bengio (2012)) Many more sophisticated approaches have also been proposed. It is worthwhile to read 2012 paper which used a Bayesian approach to automatically optimize hyper-parameters. (Practical Bayesian optimization of machine learning algorithms, by Jasper Snoek, Hugo Larochelle, and Ryan Adams.)

There are a few particularly useful papers that synthesize and distill out much of this lore. Yoshua Bengio has a 2012 paper (Practical recommendations for gradient-based training of deep architectures, by Yoshua Bengio (2012)) that gives some practical recommendations for using backpropagation and gradient descent to train neural networks, including deep neural nets. Bengio discusses many issues in much more detail than I have, including how to do more systematic hyper-parameter searches.

Another good paper is a 1998 paper, Efficient BackProp, by Yann LeCun, Léon Bottou, Genevieve Orr and Klaus-Robert Müller (1998), by Yann LeCun, Léon Bottou, Genevieve Orr and Klaus-Robert Müller. Both these papers appear in an extremely useful 2012 book, Neural Networks: Tricks of the Trade, edited by Grégoire Montavon, Geneviève Orr, and Klaus-Robert Müller., that collects many tricks commonly used in neural nets.

In a sense, we've been learning how to think about neural nets. Over the remainder of this chapter I briefly sketch a handful of other techniques. These sketches are less in-depth than the earlier discussions, but should convey some feeling for the diversity of techniques available for use in neural networks.

4.10 Other techniques

4.10.1 Variations on stochastic gradient descent

Hessian Technique:
By taylor's theorem,

$$C(w+\delta w) = C(w) + \sum_j \frac{dC}{dw_j}\Delta w_j + \frac{1}{2}\sum_{jk}\Delta w_j \frac{d^2C}{dw_j dw_k}\Delta w_k + \ldots \tag{15}$$

$$= C(w) + \nabla C \cdot \Delta w + \frac{1}{2}\Delta w^T H \Delta w + \ldots \tag{16}$$

where H is known as Hassian matrix, whose jkth entry is $d^C/dw_j dw_k$. Using calculus we can show that the expression on the right-hand side can be minimized by choosing

$$\Delta w = -H^{-1}\nabla C$$

This means that we could minimize the cost by
1. Choose w
2. Update w to $w' = w - \eta H^{-1}\nabla C$, where H and ∇C are computed at w
3. Update w to $w'' = w' - \eta H^{-1}\nabla C$, where H and ∇C are computed at w'
4. ...

There are theoretical and empirical results showing that Hessian methods converge on a minimum in fewer steps than standard gradient descent. But it's very difficult to apply in practice.

Momentum-based gradient descent:
The momentum technique modifies gradient descent in two ways that make it more similar to the physical picture. First, it introduces a notion of "velocity" for the parameters we're trying to optimize. The gradient acts to change the velocity, not (directly) the "position", in much the same way as physical forces change the velocity, and only indirectly affect position. Second, the momentum method introduces a kind of friction term, which tends to gradually reduce the velocity.

$$v \rightarrow v' = \mu v - \eta \nabla C \tag{17}$$

$$w \rightarrow w' = w + v' \tag{18}$$

where μ,momentum co-efficient, is a hyper-parameter which controls the amount of damping or friction in the system. Intuitively, we build up the velocity by repeatedly adding gradient terms to it. That means that if the gradient is in (roughly) the same direction through several rounds of learning, we can build up quite a bit of steam moving in that direction. With each step the velocity gets larger down the slope, so we move more and more quickly to the bottom of the valley. This can enable the momentum technique to work much faster than standard gradient descent. When μ=1, there is no friction; however $\mu = 0$ means that friction is so strong that velocity does not get build up. In this way, we can get some of the advantages of the Hessian technique, using information about how the gradient is changing. In practice, the momentum technique is commonly used, and for easy use without disadvantages.

4.11 Other models of artificial neuron

Networks built using other model neurons sometimes outperform sigmoid networks. Depending on the application, networks based on such alternate models may learn faster, generalize better to test data, or perhaps do both.
Tanh neuron

$$tanh(w \cdot x + b)$$

Difference between tanh neurons and sigmoid neurons is that the output from tanh neurons ranges from -1 to 1, not 0 to 1. This means that if you're going to build a network based on tanh neurons you may need to normalize your outputs.
Rectified linear neuron
The output of a rectified linear unit with input x, weight vector w, and bias b is given by:

$$\max(0, w \cdot x + b)$$

Increasing the weighted input to a rectified linear unit will never cause neurons to saturate, and so there is no corresponding learning slowdown. On the other hand, when the weighted input to a rectified linear unit is negative, the gradient vanishes, and so the neuron stops learning entirely. These are just two of the many issues that make it non-trivial to understand when and why rectified linear units perform better than sigmoid or tanh neurons.

5 Universality Theorem

Neural networks can compute any function with good approximation. The class of functions which can be approximated in the way described are the continuous functions.

Suppose f(x) is the function that we want to approximate to. What's output from the network is $\sigma(\sum_j w_j a_j + b)$ where b is the bias on the output neuron. Design a neural network whose hidden layer has a weighted output given by $\sigma^{-1} \cdot f(x)$, where σ^{-1} is just the inverse of the σ function. If we can make the weighted output before output layers to be like this, then the output from the network as a whole will be a good approximation to f(x). Your challenge, then, is to design a neural network to approximate the goal function. How well you're doing is measured by the average deviation between the goal function and the function the network is actually computing. Your challenge is to drive the average deviation as low as possible. Increasing the number of pairs of hidden neurons allows more bumps - higher approximation. In summary, weights and bias of neurons within hidden layers determine the affected areas by x (by filtering to 0 for ones that do not meet criteria; one neuron for each area) and the weights and bias of output layer determine the heights of regions. Taking multiple inputs in hidden layer meets criteria for areas involve multiple inputs. So the right question to ask is not whether any particular function is computable, but rather what's a good way to compute the function. To sum up: universality tells us that neural networks can compute any function; and empirical evidence suggests that deep networks are the networks best adapted to learn the functions useful in solving many real-world problems.

6 Deep Network

Training deep networks with learning algorithm, stochastic gradient descent by backpropagation, often get stuck as different layers in deep network are learning at vastly different speeds. In fact, we'll find that there's an intrinsic instability associated to learning by gradient descent in deep, many-layer neural networks. This instability tends to result in either the early or the later layers getting stuck during training.

6.1 Unstable gradient: Vanishing/exploding gradient problem

In vanishing gradient problem, neurons in the earlier layers learn much more slowly than neurons in later layers. In exploding gradient problem, it's the opposite case.

$$\frac{dC}{db_1} = \sigma^{'}(z_1) w_2 \sigma^{'}(z_2) \sigma^{'}(z_3) \frac{dC}{da_3}$$

Since $|w_j \sigma^{'}(z_j)| < \frac{1}{4}$ usually, $\frac{dC}{db_1}$ is a factor of 4 smaller than $\frac{dC}{db_2}$, 16 than $\frac{dC}{db_3}$.

This is partly from initializing w_j to be in Gaussian distribution with mean 0 and sd 1. Also $\sigma^{'}$ for sigmoid is at max .25. As weights get trained and is bigger, vanishing gradient problem is solved. If it's too big, then we will see exploding gradient problem. The essential problem is that the gradient in early layers is the product of terms from all the later layers; and that neural networks is exposed to unstable gradient problem.

Among difficulties of training deep networks, there is also a role played by the choice of activation function, the way weights are initialized, and even details of how learning by gradient descent is implemented. And, of course, choice of network architecture and other hyper-parameters is also important. Thus, many factors can play a role in making deep networks hard to train.

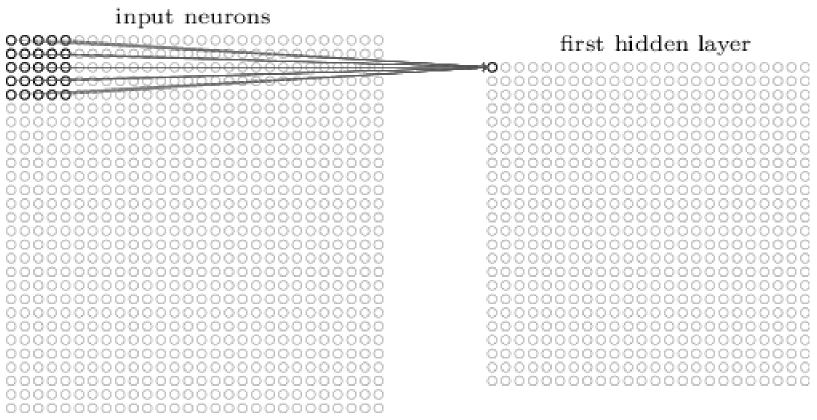
7 Deep Learning

7.1 Convolutional Network

Convolutional neural networks use three basic ideas: local receptive fields, shared weights, and pooling. Let's look at each of these ideas in turn.

7.1.1 Local Receptive Field

Local receptive field for the hidden neuron is a little window on the input pixels. Each connection learns a weight. And the hidden neuron learns an overall bias as well.



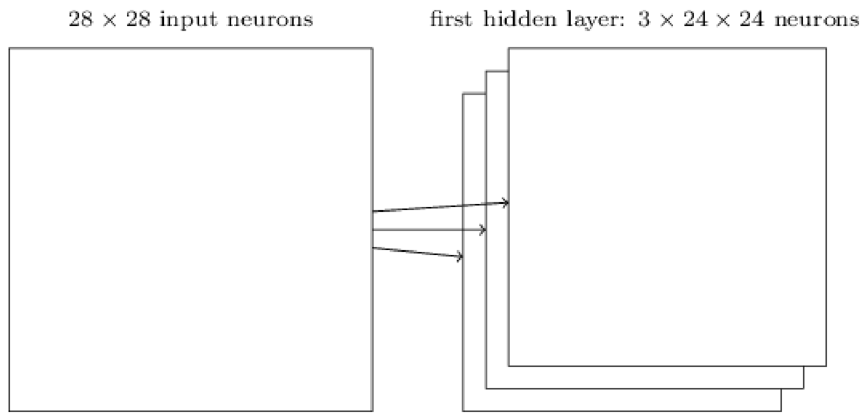
Then slide the local receptive field across the entire input image. If we have a 28×28 input image, and 5×5 local receptive fields, then there will be 24×24 neurons in the hidden layer. In fact, sometimes a different stride length is used. For instance, we might move the local receptive field 22 pixels to the right (or down), in which case we'd say a stride length of 22 is used.

7.1.2 Shared weights

Each hidden neuron has a bias and 5×5 weights connected to its local receptive field where same weights and bias is used for each of $24x24$ hidden neurons. For instance, for j, k^{th} hidden neuron, the output is:

$$\sigma(b + \sum_{l=0}^{4} \sum_{m=0}^{4} w_{l,m} a_{j+l,k+m}) \tag{19}$$

This means that all the neurons in the first hidden layer detect exactly the same feature, just at different locations in the input image. The map from the input layer to the hidden layer is called a **feature map**. The weights defining the feature map as the **shared weights**. The shared weights and bias are often said to define a kernel or filter.

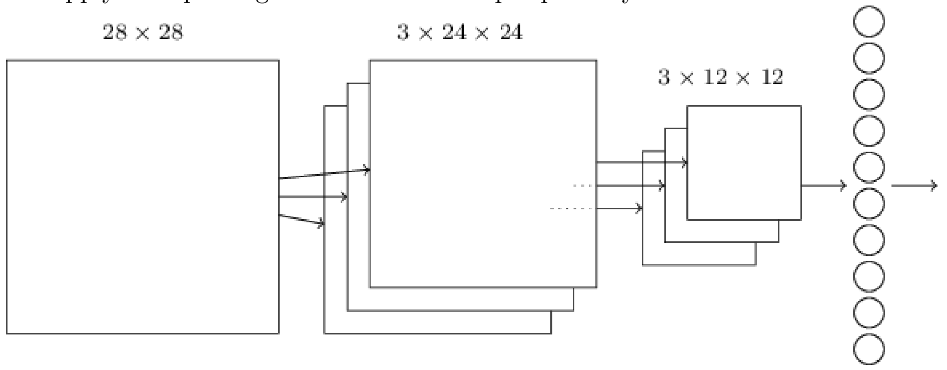


In the example shown, there are 33 feature maps. Each feature map is defined by a set of 5×5 shared weights, and a single shared bias. The result is that the network can detect 3 different kinds of features, with each feature being detectable across the entire image. A big advantage of sharing weights and biases is that it greatly reduces the number of parameters involved in a convolutional network. For each feature map we need $25 = 5 \times 5$ shared weights, plus a single shared bias. By comparison, suppose we had a fully connected first layer, with $784 = 28 \times 28$ input neurons. That, in turn, will result in faster training for the convolutional model, and, ultimately, will help us build deep networks using convolutional layers. Incidentally, the name convolutional comes from the fact that the operation in Equation 19.

7.1.3 Pooling Layers

Pooling layers are usually used immediately after convolutional layers. What the pooling layers do is simplify the information in the output from the convolutional layer. In detail, a pooling layer takes each feature map output from the convolutional layer and prepares a condensed feature map. One common procedure for pooling is known as **max-pooling**. In max-pooling, a pooling unit simply outputs the maximum activation in the region of hidden neurons.

We apply max-pooling to each feature map separately.



We can think of max-pooling as a way for the network to ask whether a given feature is found anywhere in a region of the image. It then throws away the exact positional information. The intuition is that once a feature has been found, its exact location isn't as important as its rough location relative to other features. This drastically reduces number of parameters in later layers.
Another common approach is known as L2 pooling. Instead of taking the maximum activation, take the square root of the sum of the squares of the activations in the region.

Fully-connected layer: Lastly, fully-connected layer connects every neuron from the max-pooled layer to every one of the output neurons.

7.2 Techniques to improve deep learning

7.2.1 Inserting multiple convolutional layer

By inserting additional convolutional layer between the existing convolutional-pooling layer and the fully-connected hidden layer, the accuracy can be improved. In fact, you can think of the second convolutional-pooling layer as having as input 12×12 "images", whose "pixels" represent the presence (or absence) of particular localized features in the original input image. So you can think of this layer as having as input a version of the original input image. That version is abstracted and condensed, but still has a lot of spatial structure, and so it makes sense to use a second convolutional-pooling layer. Each neuron in this second convolutional layer will learn from all $20 \times 5 \times 5$ input neurons in its local receptive field from 20 pooled layers. The feature detectors in the second convolutional-pooling layer have access to all the features from the previous layer, but only within their particular local receptive field.

7.2.2 Rectified Linear Activation

Rectified Linear activation function outperforms sigmoid or tanh functions. A common justification is that max(0,z) doesn't saturate in the limit of large z, unlike sigmoid neurons, and this helps rectified linear units continue learning.

7.2.3 Expand training data

Can expand training data by rotating, translating, and skewing the data.

7.2.4 Fully-connected layer

1. Expand the size of the fully-connected layer 2. Insert extra fully-connected layer with dropout method, it can improve the results. Apply dropout to only the fully-connected section of the network. The convolutional layers have considerable inbuilt resistance to overfitting. The reason is that the shared weights mean that convolutional filters are forced to learn from across the entire image.

7.2.5 Using an ensemble of networks

Create several neural networks, and then get them to vote to determine the best classification.

7.3 Tackling unstability of deep networking

Tying back to the previous chapter on unstability of deep networking, some of techniques discussed in this chapter help us avoiding/proceeding over the problem:
(1) Using convolutional layers greatly reduces the number of parameters in those layers, making the learning problem much easier;
(2) Using more powerful regularization techniques (notably dropout and convolutional layers) to reduce overfitting, which is otherwise more of a problem in more complex networks;
(3) Using rectified linear units instead of sigmoid neurons, to speed up training - empirically, often by a factor of 33-55;
(4) Using GPUs and being willing to train for a long period of time.

7.4 Other approaches to deep neural nets

Other topics include recurrent neural networks, Boltzmann machines, generative models, transfer learning, reinforcement learning, and etcs.

7.4.1 Recurrent neural networks

n the feedforward nets we've been using there is a single input which completely determines the activations of all the neurons through the remaining layers. It's a very static picture: everything in the network is fixed. In RNN, the elements in the network keep changing in a dynamic way; the behaviour of hidden neurons might not just be determined by the activations in previous hidden layers, but also by the activations at earlier times. RNNs are neural networks in which there is some notion of dynamic change over time.

7.4.2 Long short-term memory units

Long short-term memory units were introduced by Hochreiter and Schmidhuber in 1997 with the explicit purpose of helping address the unstable gradient problem. LSTMs make it much easier to get good results when training RNNs

7.4.3 Deep belief nets, generative models, and Boltzmann machines

DBN is a generative model that it can generate values for specific input activations. In other words, the DBN would in some sense be learning to write after learning to read them. A second reason DBNs are interesting is that they can do unsupervised and semi-supervised learning.