

**Government College of Engineering and
Technology, Jammu**



**UNIVERSITY OF JAMMU INDUSTRIAL INTERNSHIP PROJECT
REPORT ON**

“Self-Driving car Simulator using CNN”

*Submitted in partial fulfillment of the requirements for the Award of
Bachelor’s Degree in Computer Engineering from University of Jammu By*

Faizan Hamid Lone

191103024

Under the Guidance of

Dr. Simmi Dutta

Dr. Shabir Ahmad Sofi

Er. Probjot Singh

Project Guides

Project Supervisor

GCET Jammu

NIT Srinagar

Certificate of Approval of Project Report

Faizan Hamid Lone 191103024

“Self-Driving car Simulator using CNN”

NIT srinagar

2

ACKNOWLEDGEMENT

We are deeply grateful to our Project Supervisor, **Dr. Shabir Ahmad Sofi**, and Project Guide, Head of the Department, **Dr. Simmi Dutta** for their exemplary guidance, monitoring and constant encouragement throughout the course of the project.

We also take this opportunity to express a deep sense of gratitude to the entire teaching and non-teaching faculty of the Computer Engineering Department for their support and appreciation.

Last but not the least we are thankful to our family members and friends for their much-needed moral support and encouragement.

Submitted by:

Ubaid Ullah Bhat 191103023

Sheikh Mohammad Hamiz 191103022

Faizan Hamid Lone 191103024

CERTIFICATE

DEPARTMENT OF COMPUTER ENGINEERING GCET JAMMU

CERTIFICATE



राष्ट्रीय प्रौद्योगिकी संस्थान श्रीनगर
NATIONAL INSTITUTE OF TECHNOLOGY SRINAGAR
(An autonomous Institute of National Importance under the aegis of Ministry of Education, Govt. of India)

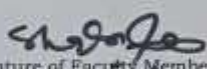
प्रशिक्षण और प्लेसमेंट विभाग
DEPARTMENT OF TRAINING & PLACEMENT

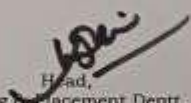
Tel/Fax: +91-9419226538, 9419226574 Extn: 2130/31 Email: placements@nitae.ac.in
Hazratbal, Srinagar Jammu and Kashmir, 190006, INDIA

No: NIT/T & P/2023-24/CERT/043
Dated: 02/08/2023

CERTIFICATE

This is to certify that **Mr. /Ms. FAIZAN HAMID LONE**, Enrolment No. **191103024**, student of Department of Computer Science Engineering **Government College of Engineering & Technology, Jammu** has completed his/her Internship cum Project on the topic "**MACHINE LEARNING APPLICATIONS**" in the Department of Information Technology at National Institute of Technology Srinagar from **01st April 2023 to 31st July 2023**. The performance of the Trainee during the above training period was **Satisfactory/Good/Best/Excellent**.


Signature of Faculty Member/
Supervisor/Coordinator


Head,
Training & Placement Deptt.

प्रशिक्षण और प्लेसमेंट विभाग
राष्ट्रीय प्रौद्योगिकी संस्थान श्रीनगर
सनातन, श्रीनगर, जम्मू और कश्मीर, भारत
NIT Training & Placement
National Institute of Technology Srinagar
Hazratbal Srinagar, J&K, India

Table of Contents

List of Figures.....	7
Abstract.....	8
Perceptron	9
Sigmoid Neuron	11
The architecture of neural networks.....	14
A simple network to classify handwritten digits	16
Learning with gradient descent	20
The four fundamental equations behind backpropagation	25
The backpropagation algorithm.....	26
Introducing the cross-entropy cost function.....	28
Softmax.....	29
Regularization	31
Introducing convolutional networks	32
Recent progress in image recognition	40
Other approaches to deep neural nets.....	45
On the future of neural networks	48
Project.....	50

List of figures

Figure1(a)	10
Figure1(b)	11
Figure2(a)	13
Figure2(b)	14
Figure3(a)	15
Figure3(b)	15
Figure4(a)	17
Figure4(b)	18
Figure4(c)	19
Figure5(a)	22
Figure5(b)	24
Figure6(a)	26
Figure8(a)	29
Figure11(a)	32
Figure11(b)	33
Figure11(c)	33
Figure11(d)	34
Figure11(e)	35
Figure11(f)	36
Figure11(g)	37
Figure11(h)	38
Figure11(i)	39
Figure12(a)	43
Figure12(b)	45
Figure15(a)	57
Figure15(b)	58
Figure15(c)	59
Figure15(d)	60

Abstract

Simulators play a crucial role in the development and testing of self-driving car technologies, offering a safe and controlled environment for training and validation. This abstract presents an overview of a self-driving car simulator that leverages Convolutional Neural Networks (CNNs) to simulate and enhance various aspects of autonomous driving tasks.

Convolutional Neural Networks have demonstrated exceptional performance in computer vision tasks, making them a valuable tool in simulating real-world driving scenarios within a controlled virtual environment. The self-driving car simulator employs CNNs for a range of functions, including perception, control, and decisionmaking.

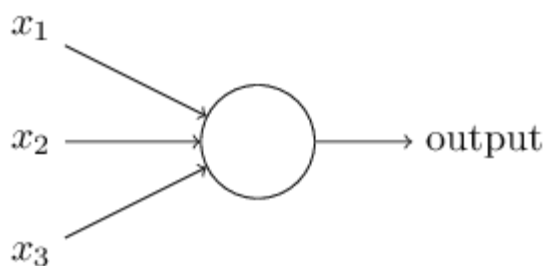
The perception module of the simulator employs CNNs to process virtual sensor data, replicating the inputs that a real self-driving car would receive from cameras, LiDAR, radar, and other sensors. By using CNNs for object detection, lane detection, and scene segmentation, the simulator can realistically emulate the challenges of recognizing and understanding the surroundings, such as identifying pedestrians, other vehicles, and road obstacles.

Perceptrons

What is a neural network? To get started, I'll explain a type of artificial neuron called a perceptron. Perceptrons were developed in the 1950s and 1960s by the scientist Frank Rosenblatt, inspired by earlier work by Warren McCulloch and Walter Pitts. Today, it's more common to use other models of artificial neurons - in this book, and in much modern work on neural networks, the main neuron model used is one called the sigmoid neuron. We'll get to sigmoid neurons shortly. But to understand why sigmoid neurons are defined the way they are, it's worth taking the time to first understand

perceptrons.

So how do perceptrons work? A perceptron takes several binary inputs, x_1, x_2, \dots , and produces a single binary output:



In the example shown the perceptron has three inputs, x_1, x_2, x_3 . In general it could have more or fewer inputs. Rosenblatt proposed a simple rule to compute the output. He introduced *weights*, w_1, w_2, \dots , real numbers expressing the importance of the respective inputs to the output. The neuron's output, 00 or 11, is determined by whether the weighted sum $\sum_j w_j x_j$ is less than or greater than some *threshold value*. Just like the weights, the threshold is a real number which is a parameter of the neuron. To put it in more precise algebraic terms:

$$\text{Output} = \left\{ \begin{array}{ll} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{array} \right\}$$

That's the basic mathematical model. A way you can think about the perceptron is that it's a device that makes decisions by weighing up evidence. Let me give an example. It's not a very realistic example, but it's easy to understand, and we'll soon get to more realistic examples. Suppose the weekend is coming up, and you've heard that there's going to be a cheese festival in your city. You like cheese, and are trying to decide whether or not to go to the festival. You might make your decision by weighing up three factors:

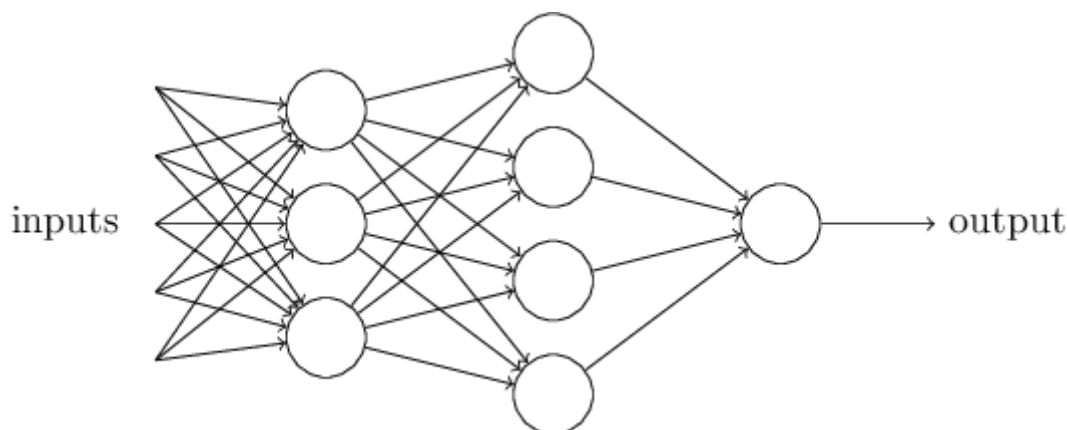
1. Is the weather good?
2. Does your boyfriend or girlfriend want to accompany you?
3. Is the festival near public transit? (You don't own a car).

We can represent these three factors by corresponding binary variables x_1 , x_2 , and x_3 . For instance, we'd have $x_1=1$ if the weather is good, and $x_1=0$ if the weather is bad. Similarly, $x_2=1$ if your boyfriend or girlfriend wants to go, and $x_2=0$ if not. And similarly, again for x_3 and public transit.

Now, suppose you absolutely adore cheese, so much so that you're happy to go to the festival even if your boyfriend or girlfriend is uninterested and the festival is hard to get to. But perhaps you really loathe bad weather, and there's no way you'd go to the festival if the weather is bad. You can use perceptrons to model this kind of decisionmaking. One way to do this is to choose a weight $w_1=6$ for the weather, and $w_2=2$ and $w_3=2$ for the other conditions. The larger value of w_1 indicates that the weather matters a lot to you, much more than whether your boyfriend or girlfriend joins you, or the nearness of public transit. Finally, suppose you choose a threshold of 55 for the perceptron. With these choices, the perceptron implements the desired decisionmaking model, outputting 1 whenever the weather is good, and 0 whenever the weather is bad. It makes no difference to the output whether your boyfriend or girlfriend wants to go, or whether public transit is nearby.

By varying the weights and the threshold, we can get different models of decisionmaking. For example, suppose we instead chose a threshold of 3. Then the perceptron would decide that you should go to the festival whenever the weather was good *or* when both the festival was near public transit *and* your boyfriend or girlfriend was willing to join you. In other words, it'd be a different model of decision-making. Dropping the threshold means you're more willing to go to the festival.

Obviously, the perceptron isn't a complete model of human decision-making! But what the example illustrates is how a perceptron can weigh up different kinds of evidence in order to make decisions. And it should seem plausible that a complex network of perceptrons could make quite subtle decisions:



In this network, the first column of perceptrons - what we'll call the first *layer* of perceptrons - is making three very simple decisions, by weighing the input evidence. What about the perceptrons in the second layer? Each of those perceptrons is making a decision by weighing up the results from the first layer of decision-making. In this way a perceptron in the second layer can make a decision at a more complex and more abstract level than perceptrons in the first layer. And even more complex decisions can be made by the perceptron in the third layer. In this way, a many-layer network of perceptrons can engage in sophisticated decision making.

In the network above the perceptrons look like they have multiple outputs. In fact, they're still single output. The multiple output arrows are merely a useful way of indicating that the output from a perceptron is being used as the input to several other perceptrons. It's less unwieldy than drawing a single output line which then splits.

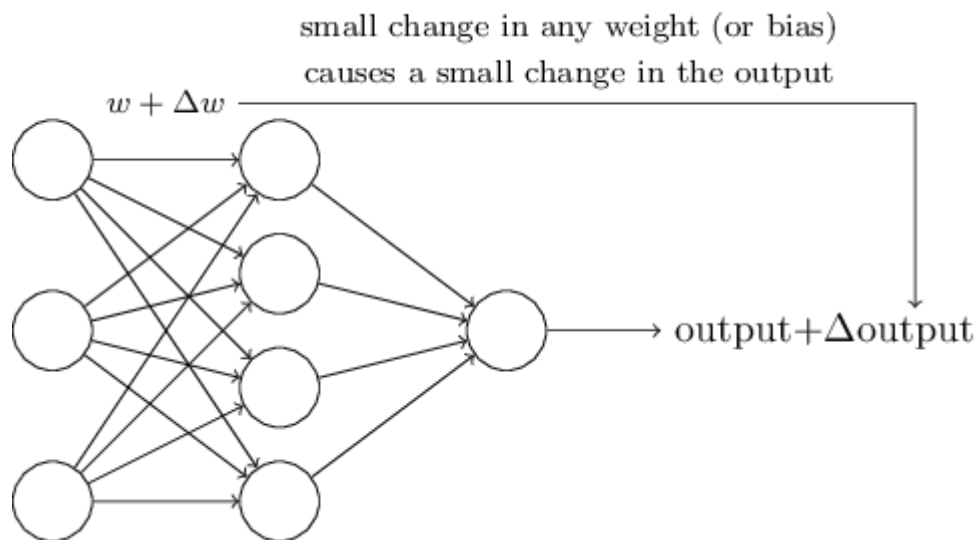
The condition $\sum_j w_j x_j > \text{threshold}$ is cumbersome, and we can make two notational changes to simplify it. The first change is to write $\sum_j w_j x_j$ as a dot product, $w \cdot x \equiv \sum_j w_j x_j$, where w and x are vectors, whose components are the weights and inputs, respectively. The second change is to move the threshold to the other side of the inequality, and to replace it by what's known as the perceptron's *bias*, $b \equiv -\text{threshold}$. Using the bias instead of the threshold, the perceptron rule can be rewritten:

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

Sigmoid neurons

Suppose we have a network of perceptrons that we'd like to use to learn to solve some problem. For example, the inputs to the network might be the raw pixel data from a scanned, handwritten image of a digit. And we'd like the network to learn weights and biases so that the output from the network correctly classifies the digit. To see how learning might work, suppose we make a small change in some weight (or bias) in the network. What we'd like is for this small change in weight to cause only a small corresponding change in the output from

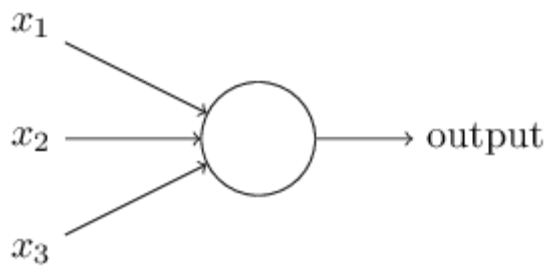
the network. As we'll see in a moment, this property will make learning possible. Schematically, here's what we want (obviously this network is too simple to do handwriting recognition!):



If it were true that a small change in a weight (or bias) causes only a small change in output, then we could use this fact to modify the weights and biases to get our network to behave more in the manner we want. For example, suppose the network was mistakenly classifying an image as an "8" when it should be a "9". We could figure out how to make a small change in the weights and biases so the network gets a little closer to classifying the image as a "9". And then we'd repeat this, changing the weights and biases over and over to produce better and better output. The network would be learning.

The problem is that this isn't what happens when our network contains perceptrons. In fact, a small change in the weights or bias of any single perceptron in the network can sometimes cause the output of that perceptron to completely flip, say from 0 to 1. That flip may then cause the behaviour of the rest of the network to completely change in some very complicated way. So while your "9" might now be classified correctly, the behaviour of the network on all the other images is likely to have completely changed in some hard-to-control way. That makes it difficult to see how to gradually modify the weights and biases so that the network gets closer to the desired behaviour. Perhaps there's some clever way of getting around this problem. But it's not immediately obvious how we can get a network of perceptrons to learn.

We can overcome this problem by introducing a new type of artificial neuron called a *sigmoid* neuron. Sigmoid neurons are similar to perceptrons, but modified so that small changes in their weights and bias cause only a small change in their output. That's the crucial fact which will allow a network of sigmoid neurons to learn.



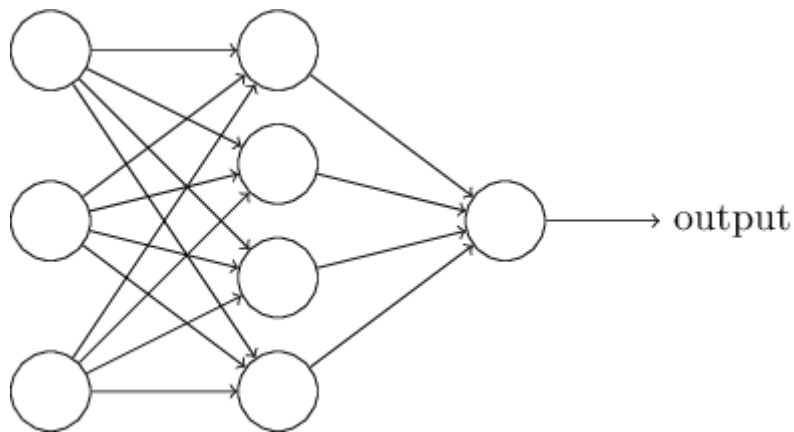
Just like a perceptron, the sigmoid neuron has inputs, x_1, x_2, \dots . But instead of being just 0 or 1, these inputs can also take on any values *between* 0 and 1. So, for instance, 0.638...0.638... is a valid input for a sigmoid neuron. Also just like a perceptron, the sigmoid neuron has weights for each input, w_1, w_2, \dots , and an overall bias, b . But the output is not 0 or 1. Instead, it's $\sigma(w \cdot x + b)$, where σ is called the *sigmoid function*.^{**}Incidentally, σ is sometimes called the *logistic function*, and this new class of neurons called *logistic neurons*. It's useful to remember this terminology, since these terms are used by many people working with neural nets. However, we'll stick with the sigmoid terminology., and is defined by:

$$\sigma(z) \equiv 1 / 1 + e^{-z}.$$

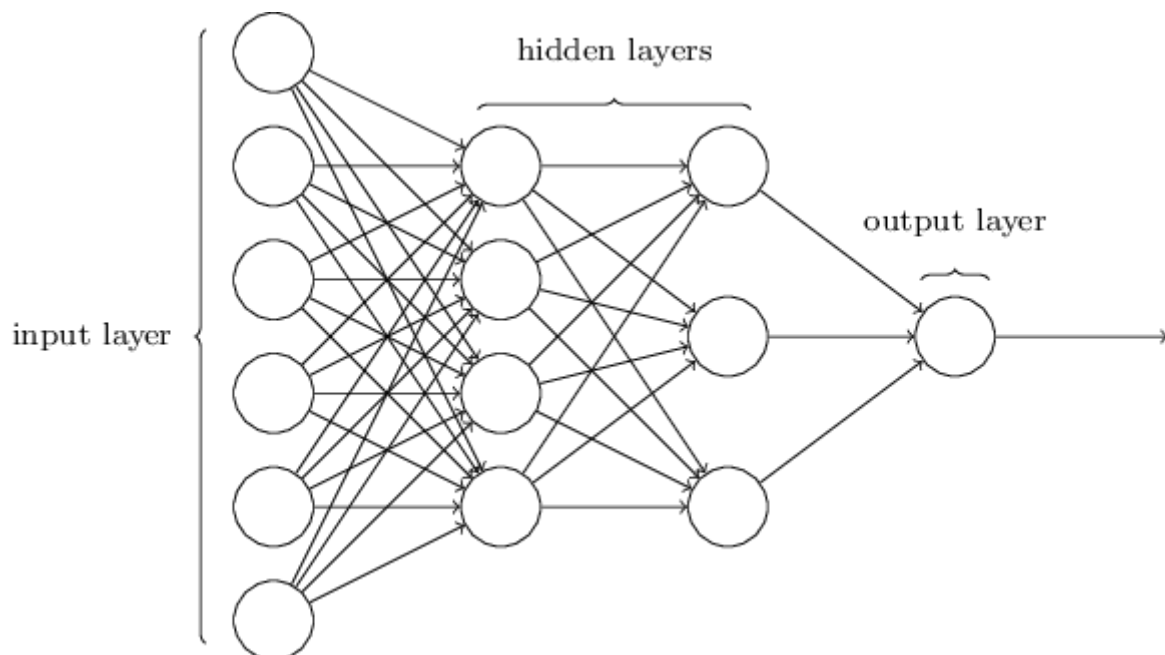
At first sight, sigmoid neurons appear very different to perceptrons. The algebraic form of the sigmoid function may seem opaque and forbidding if you're not already familiar with it. In fact, there are many similarities between perceptrons and sigmoid neurons, and the algebraic form of the sigmoid function turns out to be more of a technical detail than a true barrier to understanding.

The architecture of neural networks

Suppose we have the network:



As mentioned earlier, the leftmost layer in this network is called the input layer, and the neurons within the layer are called *input neurons*. The rightmost or *output* layer contains the *output neurons*, or, as in this case, a single output neuron. The middle layer is called a *hidden layer*, since the neurons in this layer are neither inputs nor outputs. The term "hidden" perhaps sounds a little mysterious - the first time I heard the term I thought it must have some deep philosophical or mathematical significance - but it really means nothing more than "not an input or an output". The network above has just a single hidden layer, but some networks have multiple hidden layers. For example, the following four-layer network has two hidden layers:



Somewhat confusingly, and for historical reasons, such multiple layer networks are sometimes called *multilayer perceptrons* or *MLPs*, despite being made up of sigmoid

neurons, not perceptrons. I'm not going to use the MLP terminology in this book, since I think it's confusing, but wanted to warn you of its existence.

The design of the input and output layers in a network is often straightforward. For example, suppose we're trying to determine whether a handwritten image depicts a "9" or not. A natural way to design the network is to encode the intensities of the image pixels into the input neurons. If the image is a 64 by 64 greyscale image, then we'd have $4,096=64 \times 64$ input neurons, with the intensities scaled appropriately between 0 and 1. The output layer will contain just a single neuron, with output values of less than 0.50 indicating "input image is not a 9", and values greater than 0.50 indicating "input image is a 9".

While the design of the input and output layers of a neural network is often straightforward, there can be quite an art to the design of the hidden layers. In particular, it's not possible to sum up the design process for the hidden layers with a few simple rules of thumb. Instead, neural networks researchers have developed many design heuristics for the hidden layers, which help people get the behaviour they want out of their nets. For example, such heuristics can be used to help determine how to trade off the number of hidden layers against the time required to train the network.

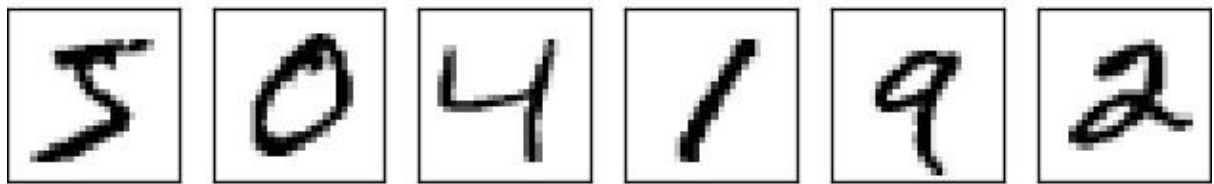
Up to now, we've been discussing neural networks where the output from one layer is used as input to the next layer. Such networks are called *feedforward* neural networks. This means there are no loops in the network - information is always fed forward, never fed back. If we did have loops, we'd end up with situations where the input to the σ function depended on the output. That'd be hard to make sense of, and so we don't allow such loops.

A simple network to classify handwritten digits

Having defined neural networks, let's return to handwriting recognition. We can split the problem of recognizing handwritten digits into two sub-problems. First, we'd like a way of breaking an image containing many digits into a sequence of separate images, each containing a single digit. For example, we'd like to break the image



into six separate images,

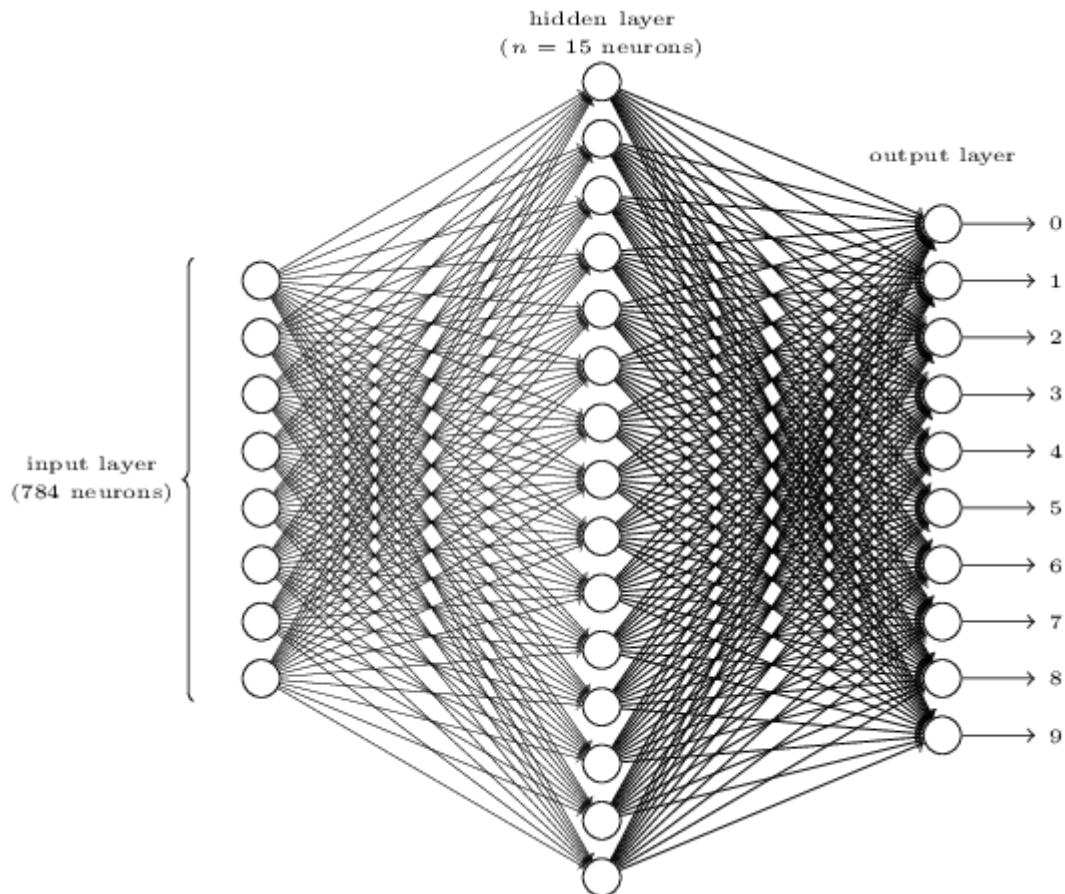


We humans solve this *segmentation problem* with ease, but it's challenging for a computer program to correctly break up the image. Once the image has been segmented, the program then needs to classify each individual digit. So, for instance, we'd like our program to recognize that the first digit above,



is a 5. There are many approaches to solving the segmentation problem. One approach is to trial many different ways of segmenting the image, using the individual digit classifier to score each trial segmentation. A trial segmentation gets a high score if the individual digit classifier is confident of its classification in all segments, and a low score if the classifier is having a lot of trouble in one or more segments. The idea is that if the classifier is having trouble somewhere, then it's probably having trouble because the segmentation has been chosen incorrectly. This idea and other variations can be used to solve the segmentation problem quite well. So instead of worrying about segmentation we'll concentrate on developing a neural network which can solve the more interesting and difficult problem, namely, recognizing individual handwritten digits. To recognize individual digits we will use a three-layer neural network:

The input layer of the network contains neurons encoding the values of the input pixels. As discussed in the next section, our training data for the network will consist of many 28 by 28 pixel images of scanned handwritten digits, and so the input layer contains $784=28 \times 28$ neurons. For simplicity I've omitted most of the 784 input neurons in the diagram above. The input pixels are greyscale, with a value of 0.0 representing white, a value of 1.0 representing black, and in between values representing gradually darkening shades of grey.

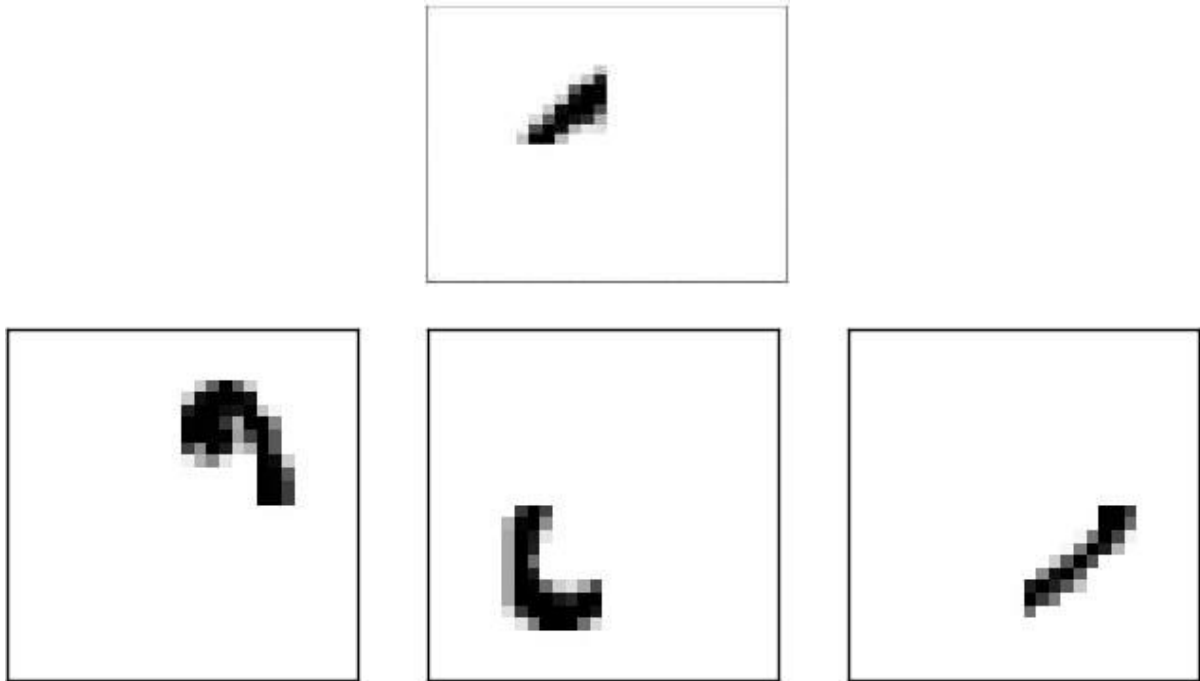


The second layer of the network is a hidden layer. We denote the number of neurons in this hidden layer by n , and we'll experiment with different values for n . The example shown illustrates a small hidden layer, containing just $n=15$ neurons.

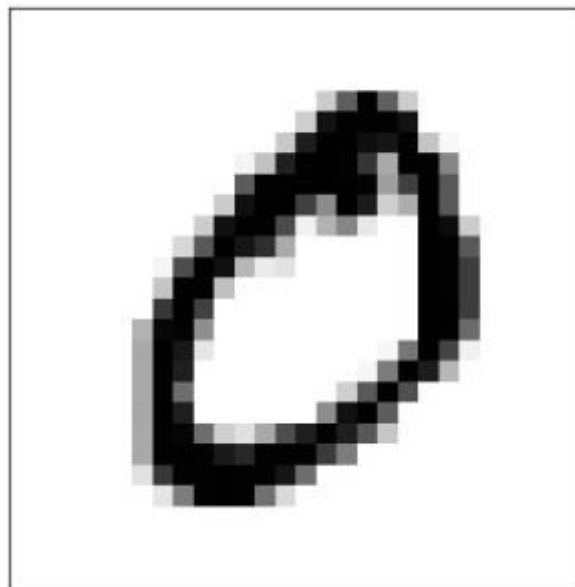
The output layer of the network contains 10 neurons. If the first neuron fires, i.e., has an output ≈ 1 , then that will indicate that the network thinks the digit is a 0. If the second neuron fires then that will indicate that the network thinks the digit is a 1. And so on. A little more precisely, we number the output neurons from 0 through 9, and figure out which neuron has the highest activation value. If that neuron is, say, neuron number 6, then our network will guess that the input digit was a 6. And so on for the other output neurons.

Consider first the case where we use 1010 output neurons. Let's concentrate on the first output neuron, the one that's trying to decide whether or not the digit is a 00. It does this by weighing up evidence from the hidden layer of neurons. What are those hidden neurons doing? Well, just suppose for the sake of argument that the first neuron in the hidden layer detects whether or not an image like the following is present:

It can do this by heavily weighting input pixels which overlap with the image, and only lightly weighting the other inputs. In a similar way, let's suppose for the sake of argument that the second, third, and fourth neurons in the hidden layer detect whether or not the following images are present:



As you may have guessed, these four images together make up the 00 image that we saw in the line of digits shown earlier:



So if all four of these hidden neurons are firing then we can conclude that the digit is a 0. Of course, that's not the *only* sort of evidence we can use to conclude that the image was a 0 - we could legitimately get a 0 in many other ways (say, through translations of the above

images, or slight distortions). But it seems safe to say that at least in this case we'd conclude that the input was a 0.

Supposing the neural network functions in this way, we can give a plausible explanation for why it's better to have 10 outputs from the network, rather than 4. If we had 4 outputs, then the first output neuron would be trying to decide what the most significant bit of the digit was. And there's no easy way to relate that most significant bit to simple shapes like those shown above. It's hard to imagine that there's any good historical reason the component shapes of the digit will be closely related to (say) the most significant bit in the output.

Learning with gradient descent

What we'd like is an algorithm which lets us find weights and biases so that the output from the network approximates $y(x)$ for all training inputs x . To quantify how well we're achieving this goal we define a *cost function***Sometimes referred to as a *loss* or *objective* function.

We use the term cost function throughout this book, but you should note the other terminology, since it's often used in research papers and other discussions of neural networks.:

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

Here, w denotes the collection of all weights in the network, b all the biases, n is the total number of training inputs, a is the vector of outputs from the network when x is input, and the sum is over all training inputs, x . Of course, the output a depends on x , w and b , but to keep the notation simple I haven't explicitly indicated this dependence. The notation $\|v\|$ just denotes the usual length function for a vector v . We'll call C the *quadratic* cost function; it's also sometimes known as the *mean squared error* or just *MSE*. Inspecting the form of the quadratic cost function, we see that $C(w,b)$ is non-negative, since every term in the sum is non-negative. Furthermore, the cost $C(w,b)$ becomes small, i.e., $C(w,b) \approx 0$, precisely when $y(x)$ is approximately equal to the output, a , for all training inputs, x . So our training algorithm has done a good job if it can find weights and biases so that $C(w,b) \approx 0$. By contrast, it's not doing so well when $C(w,b)$ is large - that would mean that $y(x)$ is not close to the output a for a large number of inputs. So the aim of our training algorithm will be to minimize the cost $C(w,b)$ as a function of the weights and biases. In other words, we want to find a set of weights and biases which make the cost as small as possible. We'll do that using an algorithm known as *gradient descent*.

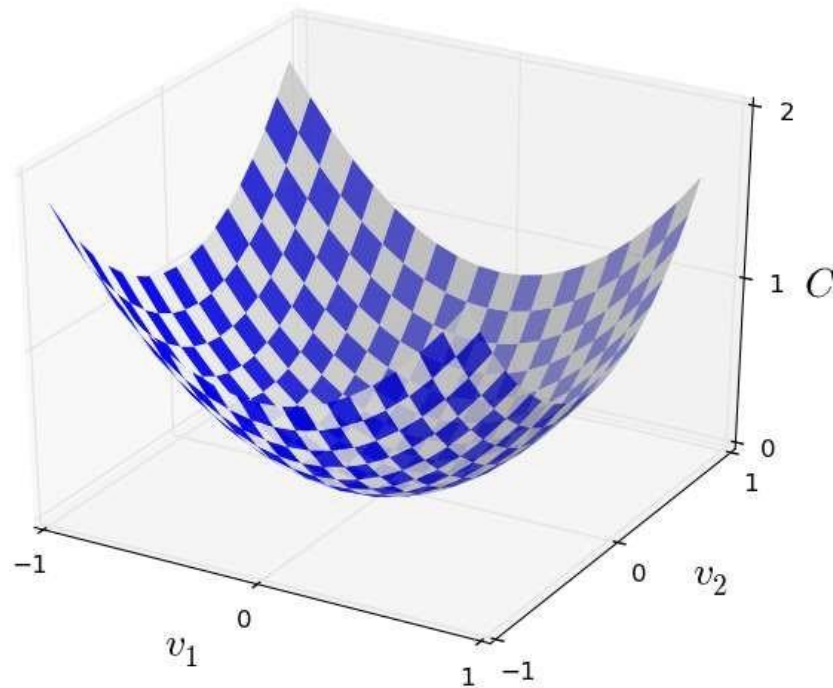
Why introduce the quadratic cost? After all, aren't we primarily interested in the number of images correctly classified by the network? Why not try to maximize that number directly, rather than minimizing a proxy measure like the quadratic cost? The problem with that is that the number of images correctly classified is not a smooth function of the weights and biases in the network. For the most part, making small changes to the weights and biases won't cause any change at all in the number of training images classified correctly. That makes it difficult to figure out how to change the weights and biases to get improved performance. If we instead use a smooth cost function like the quadratic cost it turns out to be easy to figure out how to make small changes in the weights and biases so as to get an improvement in the cost. That's why we focus first on minimizing the quadratic cost, and only after that will we examine the classification accuracy.

Even given that we want to use a smooth cost function, you may still wonder why we choose the quadratic function used in Equation . Isn't this a rather *ad hoc* choice?

Perhaps if we chose a different cost function we'd get a totally different set of minimizing weights and biases? This is a valid concern, and later we'll revisit the cost function, and make some modifications. However, the quadratic cost function of Equation works perfectly well for understanding the basics of learning in neural networks, so we'll stick with it for now.

Recapping, our goal in training a neural network is to find weights and biases which minimize the quadratic cost function $C(w,b)$. This is a well-posed problem, but it's got a lot of distracting structure as currently posed - the interpretation of w and b as weights and biases, the σ function lurking in the background, the choice of network architecture, MNIST, and so on. It turns out that we can understand a tremendous amount by ignoring most of that structure, and just concentrating on the minimization aspect. So for now we're going to forget all about the specific form of the cost function, the connection to neural networks, and so on. Instead, we're going to imagine that we've simply been given a function of many variables and we want to minimize that function. We're going to develop a technique called *gradient descent* which can be used to solve such minimization problems. Then we'll come back to the specific function we want to minimize for neural networks.

Okay, let's suppose we're trying to minimize some function, $C(v)$. This could be any realvalued function of many variables, $v=v_1,v_2,\dots$. Note that I've replaced the w and b notation by v to emphasize that this could be any function - we're not specifically thinking in the neural networks context any more. To minimize $C(v)$ it helps to imagine C as a function of just two variables, which we'll call v_1 and v_2 :



What we'd like is to find where C achieves its global minimum. Now, of course, for the function plotted above, we can eyeball the graph and find the minimum. In that sense, I've perhaps shown slightly *too* simple a function! A general function, C , may be a complicated function of many variables, and it won't usually be possible to just eyeball the graph to find the minimum.

Fortunately, there is a beautiful analogy which suggests an algorithm which works pretty well. We start by thinking of our function as a kind of a valley. If you squint just a little at the plot above, that shouldn't be too hard. And we imagine a ball rolling down the slope of the valley. Our everyday experience tells us that the ball will eventually roll to the bottom of the valley. Perhaps we can use this idea as a way to find a minimum for the function? We'd randomly choose a starting point for an (imaginary) ball, and then simulate the motion of the ball as it rolled down to the bottom of the valley. We could do this simulation simply by computing derivatives (and perhaps some second derivatives) of C - those derivatives would tell us everything we need to know about the local "shape" of the valley, and therefore how our ball should roll.

To make this question more precise, let's think about what happens when we move the ball a small amount Δv_1 in the v_1 direction, and a small amount Δv_2 in the v_2 direction. Calculus tells us that C changes as follows:

$$\Delta C \approx \frac{\partial C}{\partial v_1} \Delta v_1 + \frac{\partial C}{\partial v_2} \Delta v_2$$

In a moment we'll rewrite the change ΔC in terms of Δv and the gradient, ∇C . Before getting to that, though, I want to clarify something that sometimes gets people hung up on the gradient. When meeting the ∇C notation for the first time, people sometimes wonder how they should think about the ∇ symbol. What, exactly, does ∇ mean? In fact, it's perfectly fine to think of ∇C as a single mathematical object - the vector defined above - which happens to be written using two symbols. In this point of view, ∇ is just a piece of notational flag-waving, telling you "hey, ∇C is a gradient vector". There are more advanced points of view where ∇ can be viewed as an independent mathematical entity in its own right (for example, as a differential operator), but we won't need such points of view.

With these definitions, the expression for ΔC can be rewritten as

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

This equation helps explain why ∇C is called the gradient vector: ∇C relates changes in v to changes in C , just as we'd expect something called a gradient to do. But what's really exciting about the equation is that it lets us see how to choose Δv so as to make ΔC negative. In particular, suppose we choose

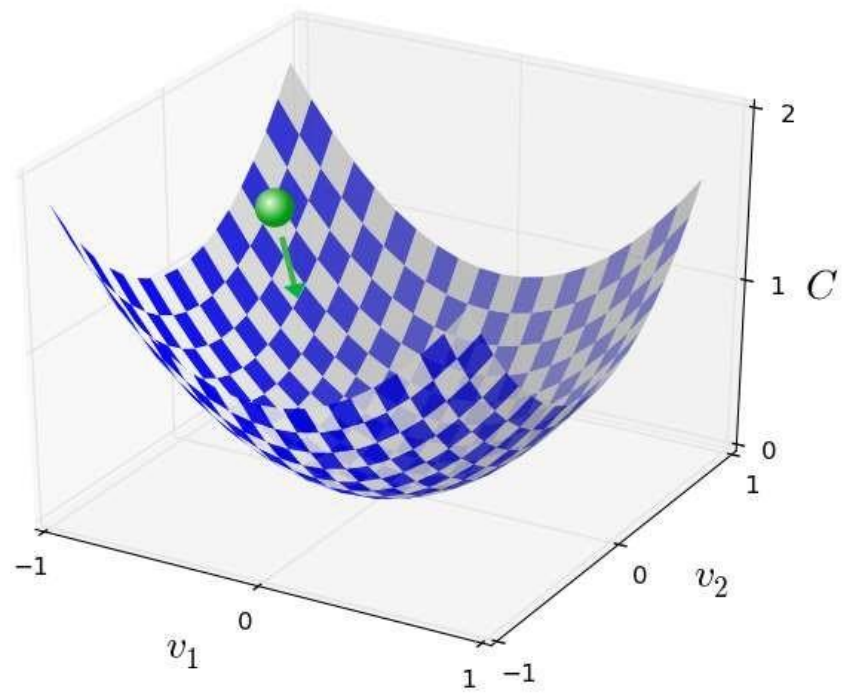
$$\Delta v = -\eta \nabla C$$

where η is a small, positive parameter (known as the *learning rate*). Then Equation (9) tells us that $\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2$. Because $\|\nabla C\|^2 \geq 0$, this guarantees that $\Delta C \leq 0$, i.e., C will always decrease, never increase, if we change v according to the prescription in. (Within, of course, the limits of the approximation in Equation. This is exactly the property we wanted! And so we'll take Equation to define the "law of motion" for the ball in our gradient descent algorithm. That is, we'll use Equation to compute a value for Δv , then move the ball's position v by that amount: $v \rightarrow v' = v - \eta \nabla C$.

Then we'll use this update rule again, to make another move. If we keep doing this, over and over, we'll keep decreasing C until - we hope - we reach a global minimum.

Summing up, the way the gradient descent algorithm works is to repeatedly compute the gradient ∇C , and then to move in the *opposite* direction, "falling down" the slope of the valley.

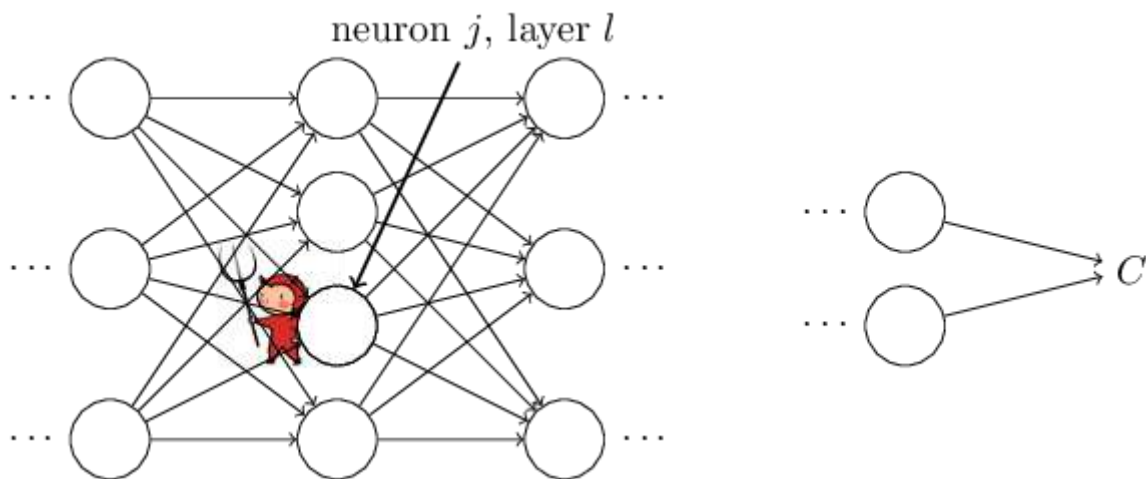
We can visualize it like this:



The four fundamental equations behind backpropagation

Backpropagation is about understanding how changing the weights and biases in a network changes the cost function. Ultimately, this means computing the partial derivatives $\partial C / \partial w_{ljk}$ and $\partial C / \partial b_{lj}$. But to compute those, we first introduce an intermediate quantity, δ_{lj} , which we call the *error* in the j th neuron in the l th layer. Backpropagation will give us a procedure to compute the error δ_{lj} , and then will relate δ_{lj} to $\partial C / \partial w_{ljk}$ and $\partial C / \partial b_{lj}$.

To understand how the error is defined, imagine there is a demon in our neural network:



The demon sits at the j th neuron in layer l . As the input to the neuron comes in, the demon messes with the neuron's operation. It adds a little change Δz_{lj} to the neuron's weighted input, so that instead of outputting $\sigma(z_{lj})$, the neuron instead outputs $\sigma(z_{lj} + \Delta z_{lj})$. This change propagates through later layers in the network, finally causing the overall cost to change by an amount $\partial C / \partial z_{lj} \Delta z_{lj}$.

Now, this demon is a good demon, and is trying to help you improve the cost, i.e., they're trying to find a Δz_{lj} which makes the cost smaller. Suppose $\partial C / \partial z_{lj}$ has a large value (either positive or negative). Then the demon can lower the cost quite a bit by choosing Δz_{lj} to have the opposite sign to $\partial C / \partial z_{lj}$. By contrast, if $\partial C / \partial z_{lj}$ is close to zero, then the demon can't improve the cost much at all by perturbing the weighted input z_{lj} . So far as the demon can tell, the neuron is already pretty near optimal. This is only the case for small changes Δz_{lj} of course. We'll assume that the demon is constrained to make such small changes. And so there's a heuristic sense in which $\partial C / \partial z_{lj}$ is a measure of the error in the neuron.

Motivated by this story, we define the error δ_{lj} of neuron j in layer l by $\delta_{lj} \equiv \partial C / \partial z_{lj}$.

As per our usual conventions, we use δ^l to denote the vector of errors associated with layer l . Backpropagation will give us a way of computing δ^l for every layer, and then relating those errors to the quantities of real interest, $\partial C / \partial w_{ljk}$ and $\partial C / \partial b_l$.

Backpropagation is based around four fundamental equations. Together, those equations give us a way of computing both the error δ^l and the gradient of the cost function.

An equation for the error in the output layer, δ^L : The components of δ^L are given by $\delta^L_j = \partial C / \partial a^L_j \cdot \sigma'(z^L_j)$.

An equation for the error δ^l in terms of the error in the next layer, δ^{l+1} : In particular $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$

An equation for the rate of change of the cost with respect to any bias in the network: In particular: $\partial C / \partial b_l = \delta^l_j$

An equation for the rate of change of the cost with respect to any weight in the network: In particular:

$$\partial C / \partial w_{ljk} = a^{l-1}_k \delta^l_j.$$

The backpropagation algorithm

The backpropagation equations provide us with a way of computing the gradient of the cost function. Let's explicitly write this out in the form of an algorithm:

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

2. **Feedforward:** For each $l=2,3,\dots$, 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 $l=L-1, L-2, \dots, 2$ 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 $\partial C / \partial w_{ljk} = a_{l-1k} \delta_{lj}$ and $\partial C / \partial b_{lj} = \delta_{lj}$.

Examining the algorithm you can see why it's called *backpropagation*. We compute the error vectors δ_l backward, starting from the final layer. It may seem peculiar that we're going through the network backward. But if you think about the proof of backpropagation, the backward movement is a consequence of the fact that the cost is a function of outputs from the network. To understand how the cost varies with earlier weights and biases we need to repeatedly apply the chain rule, working backward through the layers to obtain usable expressions.

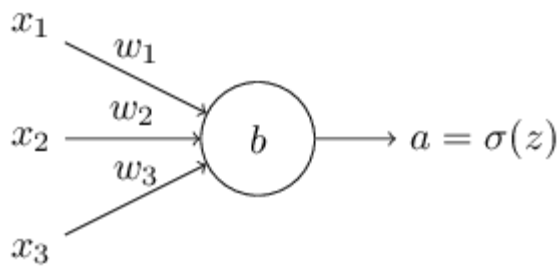
The backpropagation algorithm computes the gradient of the cost function for a single training example, $C=C_x$. In practice, it's common to combine backpropagation with a learning algorithm such as stochastic gradient descent, in which we compute the gradient for many training examples. In particular, given a mini-batch of m training examples, the following algorithm applies a gradient descent learning step based on that mini-batch:

1. **Input a set of training examples**
2. **For each training example x :** Set the corresponding input activation $a_{x,1}$, and perform the following steps:
 - **Feedforward:** For each $l=2,3,\dots, L$ compute $z_{x,l} = w_{l,x} a_{x,l-1} + b_l$ and $a_{x,l} = \sigma(z_{x,l})$.
 - **Output error $\delta_{x,L}$:** Compute the vector $\delta_{x,L} = \nabla_a C_x \odot \sigma'(z_{x,L})$.
 - **Backpropagate the error:** For each $l=L-1, L-2, \dots, 2$ compute $\delta_{x,l} = ((w_{l+1})^T \delta_{x,l+1}) \odot \sigma'(z_{x,l})$.
3. **Gradient descent:** For each $l=L, L-1, \dots$, update the weights according to the rule $w_l \rightarrow w_l - \eta m \sum x \delta_{x,l} (a_{x,l-1})^T$, and the biases according to the rule $b_l \rightarrow b_l - \eta m \sum x \delta_{x,l}$.

Of course, to implement stochastic gradient descent in practice you also need an outer loop generating mini-batches of training examples, and an outer loop stepping through multiple epochs of training

Introducing the cross-entropy cost function

How can we address the learning slowdown? It turns out that we can solve the problem by replacing the quadratic cost with a different cost function, known as the crossentropy. 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, \dots , corresponding weights w_1, w_2, \dots , 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 $C = -1/n \sum_x [y \ln a + (1-y) \ln(1-a)]$

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.

Two properties in particular make it reasonable to interpret the cross-entropy as a cost function. First, it's non-negative, that is, $C \geq 0$. To see this, notice that: (a) all the individual terms in the sum in (57) are negative, since both logarithms are of numbers in the range 00 to 11; and (b) there is a minus sign out the front of the sum.

Second, if the neuron's actual output is close to the desired output for all training inputs, x , then the cross-entropy will be close to zero. To prove this I will need to assume that the desired outputs y are all either 0 or 1. This is usually the case when solving classification problems, for example, or when computing Boolean functions. To understand what happens when we don't make this assumption, see the exercises at the end of this section. To see this, suppose for example that $y=0$ and $a \approx 0$ for some input x . This is a case when the neuron is doing a good job on that input. We see that the first term in the expression for the cost vanishes, since $y=0$, while the second term is just $-\ln(1-a) \approx 0$. A similar analysis holds when $y=1$ and $a \approx 1$. And so the contribution to the cost will be low provided the actual output is close to the desired output.

Summing up, the cross-entropy is positive, and tends toward zero as the neuron gets better at computing the desired output, y , for all training inputs, x . These are both properties we'd intuitively expect for a cost function. Indeed, both properties are also satisfied by the quadratic cost. So that's good news for the cross-entropy. But the crossentropy cost function has the benefit that, unlike the quadratic cost, it avoids the problem of learning slowing down.

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. You may wish to revisit that chapter if you need to refresh your memory about the meaning of the notation. $z_{lj} = \sum_k w_{ljk} a_{lk} + b_{lj}$. 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_{Lj} . According to this function, the activation a_{Lj} of the j th output neuron is $a_{Lj} = e^{z_{Lj}} / \sum_k e^{z_{Lk}}$

The fact that a softmax layer outputs a probability distribution is rather pleasing. In many problems it's convenient to be able to interpret the output activation a_{Lj} as the network's estimate of the probability that the correct output is j . So, for instance, in the MNIST classification problem, we can interpret a_{Lj} as the network's estimated probability that the correct digit classification is j .

By contrast, if the output layer was a sigmoid layer, then we certainly couldn't assume that the activations formed a probability distribution. I won't explicitly prove it, but it should be plausible that the activations from a sigmoid layer won't in general form a probability distribution. And so with a sigmoid output layer we don't have such a simple interpretation of the output activations.

Regularization

Increasing the amount of training data is one way of reducing overfitting. Are there other ways we can reduce the extent to which overfitting occurs? One possible approach is to reduce the size of our network. However, large networks have the potential to be more powerful than small networks, and so this is an option we'd only adopt reluctantly.

Fortunately, there are other techniques which can reduce overfitting, even when we have a fixed network and fixed training data. These are known as *regularization* techniques. In this section 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: $C = -\frac{1}{n} \sum_j [y_j * \ln a^{L_j} + (1 - y_j) \ln(1 - a^{L_j})] + \frac{\lambda}{2n} \sum w^2$.

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 noting that the regularization term *doesn't* include the biases.

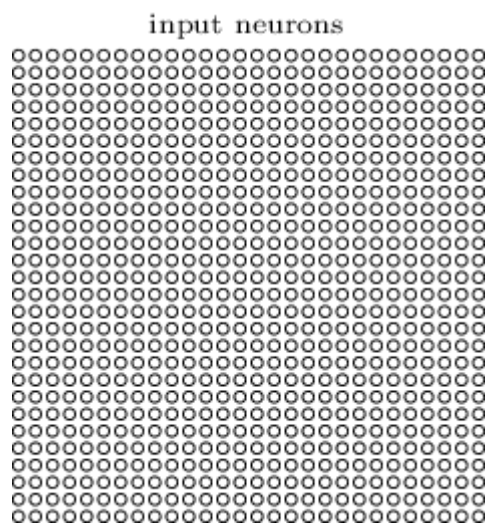
Intuitively, the effect of regularization is to make it so the network prefers to learn small weights, all other things being equal. Large weights will only be allowed if they considerably improve the first part of the cost function. Put another way, regularization can be viewed as a way of compromising 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.

Introducing convolutional networks

Convolutional neural networks are that networks which use a special architecture which is particularly well-adapted to classify images. Using this architecture makes convolutional networks fast to train. This, in turn, helps us train deep, many-layer networks, which are very good at classifying images. Today, deep convolutional networks or some close variant are used in most neural networks for image recognition.

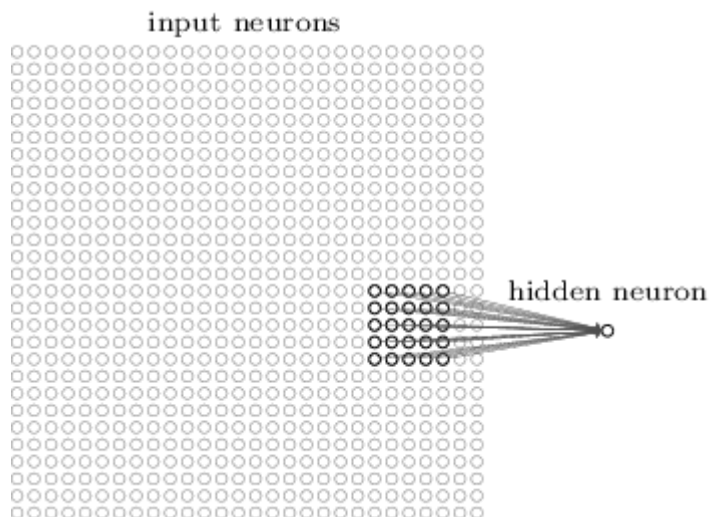
Convolutional neural networks use three basic ideas: *local receptive fields*, *shared weights*, and *pooling*. Let's look at each of these ideas in turn.

Local receptive fields: In the fully-connected layers shown earlier, the inputs were depicted as a vertical line of neurons. In a convolutional net, it'll help to think instead of the inputs as a 28×28 square of neurons, whose values correspond to the 28×28 pixel intensities we're using as inputs:



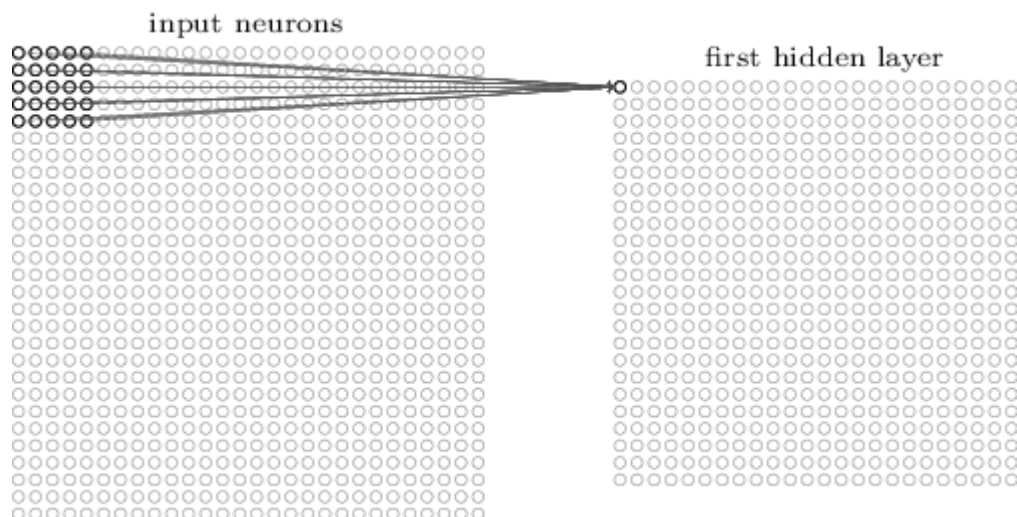
As per usual, we'll connect the input pixels to a layer of hidden neurons. But we won't connect every input pixel to every hidden neuron. Instead, we only make connections in small, localized regions of the input image.

To be more precise, each neuron in the first hidden layer will be connected to a small region of the input neurons, say, for example, a 5×5 region, corresponding to 25 input pixels. So, for a particular hidden neuron, we might have connections that look like this:

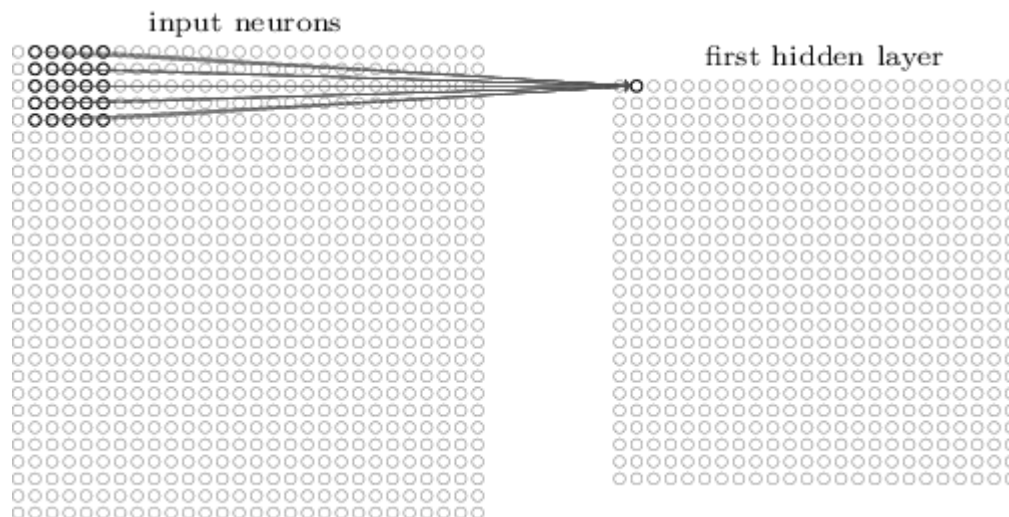


That region in the input image is called the *local receptive field* for the hidden neuron. It's a little window on the input pixels. Each connection learns a weight. And the hidden neuron learns an overall bias as well. You can think of that particular hidden neuron as learning to analyze its particular local receptive field.

We then slide the local receptive field across the entire input image. For each local receptive field, there is a different hidden neuron in the first hidden layer. To illustrate this concretely, let's start with a local receptive field in the top-left corner:



Then we slide the local receptive field over by one pixel to the right (i.e., by one neuron), to connect to a second hidden neuron:



And so on, building up the first hidden layer. Note that 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. This is because we can only move the local receptive field 23 neurons across (or 23 neurons down), before colliding with the right-hand side (or bottom) of the input image.

Shared weights and biases: I've said that each hidden neuron has a bias and 5×5 weights connected to its local receptive field. What I did not yet mention is that we're going to use the *same* weights and bias for each of the 24×24 hidden neurons. In other words, for the j , k th hidden neuron, the output is:

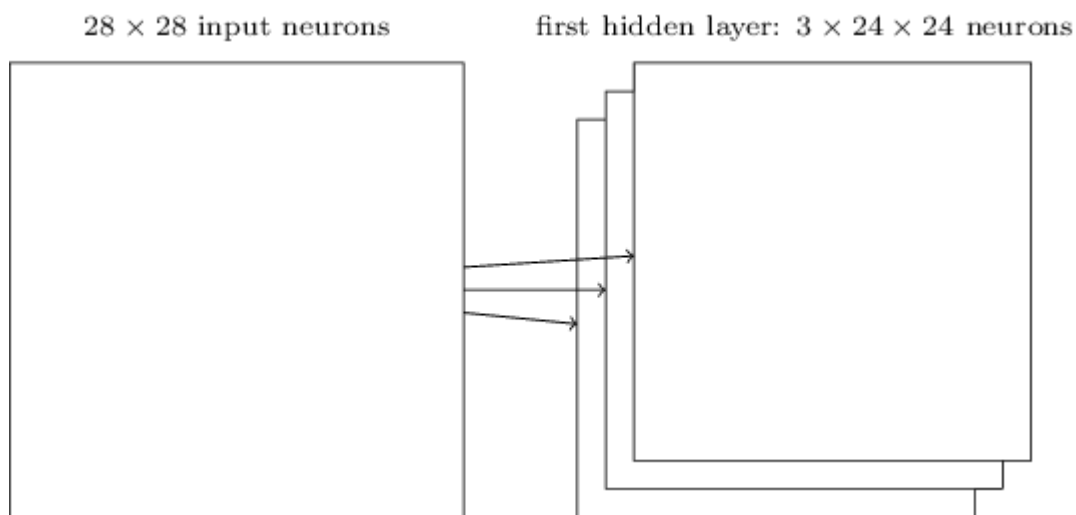
$$\sigma(b + \sum_l \sum_m w_{l,m} a_{j+l, k+m}) \quad \text{where } l = 0 \text{ to } 4 \text{ and } m = 0 \text{ to } 4$$

Here, σ is the neural activation function - perhaps the sigmoid function we used in earlier chapters. b is the shared value for the bias. $w_{l,m}$ is a 5×5 array of shared weights. And, finally, we use $a_{x,y}$ to denote the input activation at position x,y .

This means that all the neurons in the first hidden layer detect exactly the same feature**I haven't precisely defined the notion of a feature. Informally, think of the feature detected by a hidden neuron as the kind of input pattern that will cause the neuron to activate: it might be an edge in the image, for instance, or maybe some other type of shape., just at different locations in the input image. To see why this makes sense, suppose the weights and bias are such that the hidden neuron can pick out, say, a vertical edge in a particular local receptive field. That ability is also likely to be useful at other places in the image. And so it is useful to apply the same feature detector everywhere in the image. To put it in slightly more abstract terms, convolutional networks are well adapted to the translation invariance of images: move a picture of a cat (say) a little ways, and it's still an image of a cat**In fact, for the MNIST digit classification problem we've been studying, the images are centered and size normalized. So MNIST has less translation invariance than images found "in the wild", so to speak. Still, features like edges and corners are likely to be useful across much of the input space..

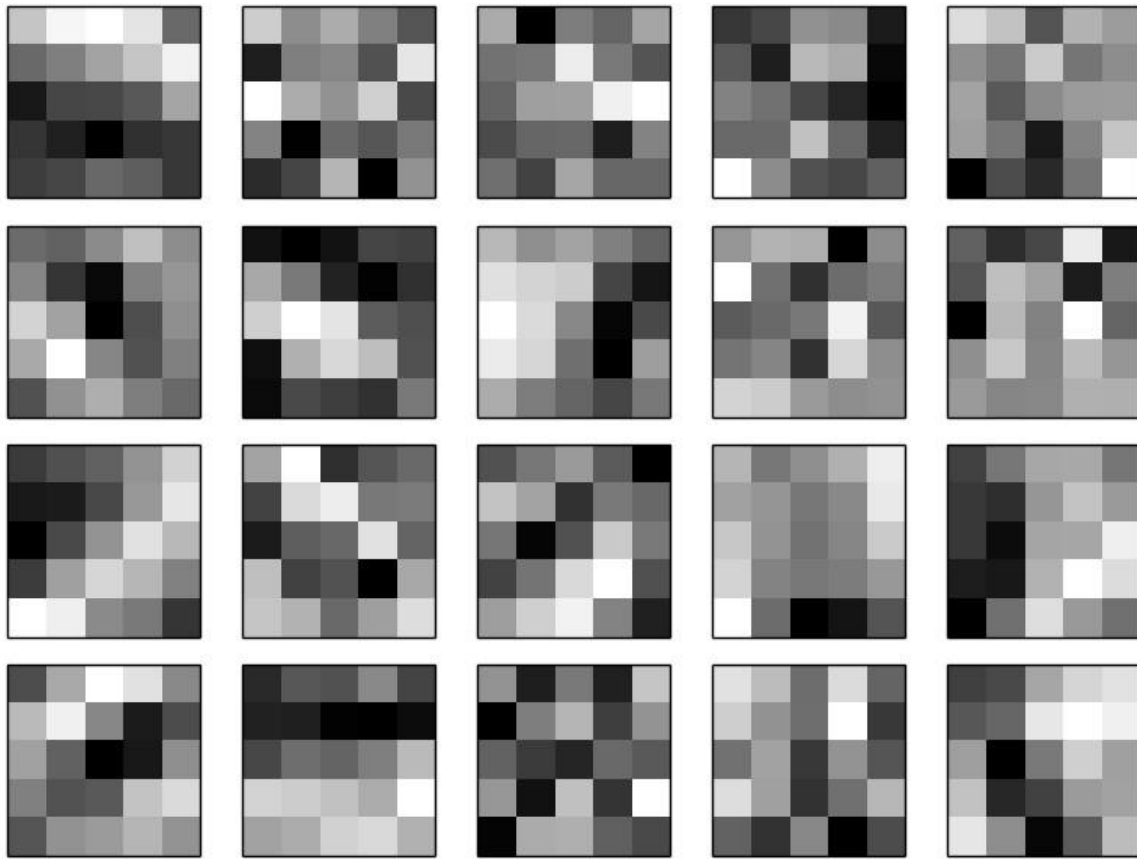
For this reason, we sometimes call the map from the input layer to the hidden layer a *feature map*. We call the weights defining the feature map the *shared weights*. And we call the bias defining the feature map in this way the *shared bias*. The shared weights and bias are often said to define a *kernel* or *filter*. In the literature, people sometimes use these terms in slightly different ways, and for that reason I'm not going to be more precise; rather, in a moment, we'll look at some concrete examples.

The network structure I've described so far can detect just a single kind of localized feature. To do image recognition we'll need more than one feature map. And so a complete convolutional layer consists of several different feature maps:



In the example shown, there are 33 feature maps. Each feature map is defined by a set of $5 \times 5 \times 5$ shared weights, and a single shared bias. The result is that the network can detect 33 different kinds of features, with each feature being detectable across the entire image.

I've shown just 33 feature maps, to keep the diagram above simple. However, in practice convolutional networks may use more (and perhaps many more) feature maps. One of the early convolutional networks, LeNet-5, used 66 feature maps, each associated to a $5 \times 5 \times 5$ local receptive field, to recognize MNIST digits. So the example illustrated above is actually pretty close to LeNet-5. In the examples we develop later in the chapter we'll use convolutional layers with 2020 and 4040 feature maps. Let's take a quick peek at some of the features which are learned*:



The 20 images correspond to 20 different feature maps (or filters, or kernels). Each map is represented as a 5×5 block image, corresponding to the 5×5 weights in the local receptive field. Whiter blocks mean a smaller (typically, more negative) weight, so the feature map responds less to corresponding input pixels. Darker blocks mean a larger weight, so the feature map responds more to the corresponding input pixels. Very roughly speaking, the images above show the type of features the convolutional layer responds to.

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. So each feature map requires 26 parameters. If we have 20 feature maps that's a total of $20 \times 26 = 520$ parameters defining the convolutional layer. By comparison, suppose we had a fully connected first layer, with $784 = 28 \times 28$ input neurons, and a relatively modest 30 hidden neurons, as we used in many of the examples earlier in the book. That's a total of 784×30 weights, plus an extra 30 biases, for a total of 23,550 parameters. In other words, the fully-connected layer would have more than 40 times as many parameters as the convolutional layer.

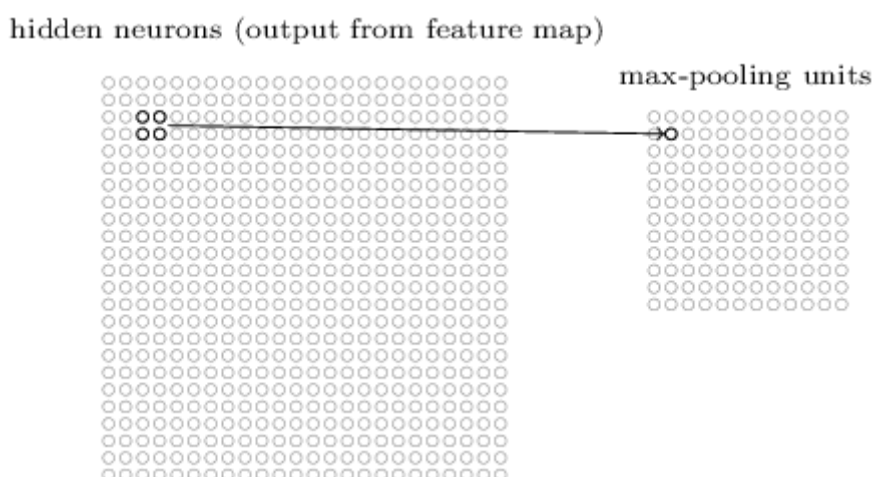
Of course, we can't really do a direct comparison between the number of parameters, since the two models are different in essential ways. But, intuitively, it seems likely that the use of translation invariance by the convolutional layer will reduce the number of parameters it

needs to get the same performance as the fully-connected model. 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 is sometimes known as a *convolution*. A little more precisely, people sometimes write that equation as $a_1 = \sigma(b + w * a_0)$ where a_1 denotes the set of output activations from one feature map, a_0 is the set of input activations, and $*$ is called a convolution operation. We're not going to make any deep use of the mathematics of convolutions, so you don't need to worry too much about this connection. But it's worth at least knowing where the name comes from.

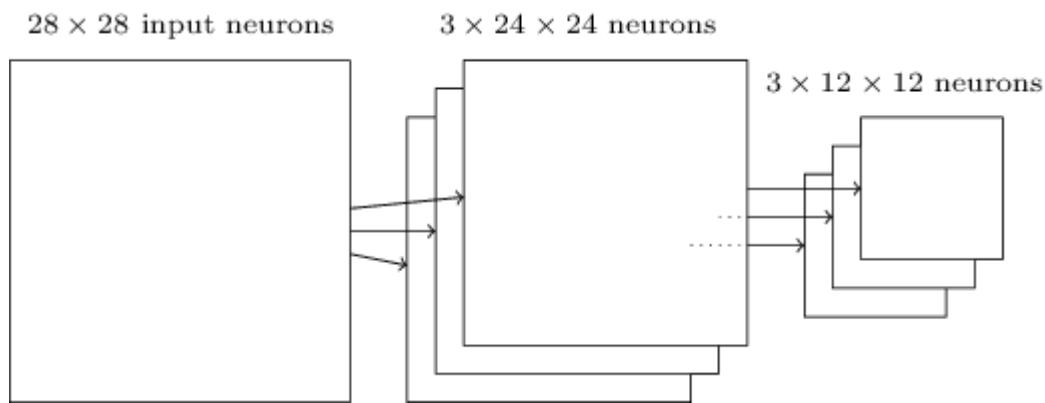
Pooling layers: In addition to the convolutional layers just described, convolutional neural networks also contain *pooling layers*. Pooling layers are usually used immediately after convolutional layers. What the pooling layers do is simplify the information in the output from the convolutional layer.

In detail, a pooling layer takes each feature map *output from the convolutional layer and prepares a condensed feature map. For instance, each unit in the pooling layer may summarize a region of (say) 2×2 neurons in the previous layer. As a concrete example, one common procedure for pooling is known as *max-pooling*. In max-pooling, a pooling unit simply outputs the maximum activation in the 2×2 input region, as illustrated in the following diagram:



Note that since we have 24×24 neurons output from the convolutional layer, after pooling we have 12×12 neurons.

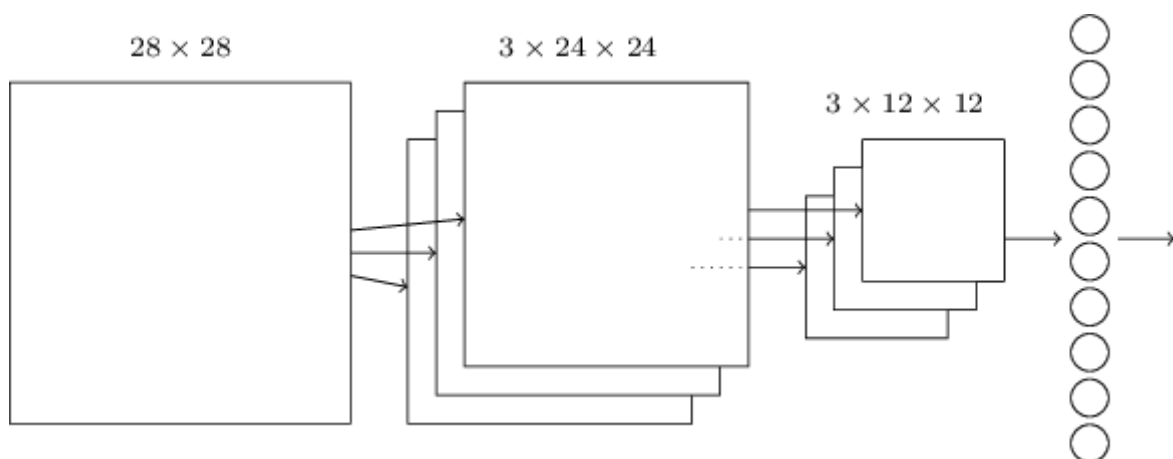
As mentioned above, the convolutional layer usually involves more than a single feature map. We apply max-pooling to each feature map separately. So if there were three feature maps, the combined convolutional and max-pooling layers would look like:



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. A big benefit is that there are many fewer pooled features, and so this helps reduce the number of parameters needed in later layers.

Max-pooling isn't the only technique used for pooling. Another common approach is known as *L2 pooling*. Here, instead of taking the maximum activation of a 2×2 region of neurons, we take the square root of the sum of the squares of the activations in the 2×2 region. While the details are different, the intuition is similar to max-pooling: L2 pooling is a way of condensing information from the convolutional layer. In practice, both techniques have been widely used. And sometimes people use other types of pooling operation. If you're really trying to optimize performance, you may use validation data to compare several different approaches to pooling, and choose the approach which works best. But we're not going to worry about that kind of detailed optimization.

Putting it all together: We can now put all these ideas together to form a complete convolutional neural network. It's similar to the architecture we were just looking at, but has the addition of a layer of 10 output neurons, corresponding to the 10 possible values for MNIST digits ('0', '1', '2', etc):



The network begins with 28×28 input neurons, which are used to encode the pixel intensities for the MNIST image. This is then followed by a convolutional layer using a 5×5 local receptive field and 33 feature maps. The result is a layer of $3 \times 24 \times 24$ hidden feature neurons. The next step is a max-pooling layer, applied to 2×2 regions, across each of the 33 feature maps. The result is a layer of $3 \times 12 \times 12$ hidden feature neurons.

The final layer of connections in the network is a fully-connected layer. That is, this layer connects *every* neuron from the max-pooled layer to every one of the 1010 output neurons. This fully-connected architecture is the same as we used in earlier chapters. Note, however, that in the diagram above, I've used a single arrow, for simplicity, rather than showing all the connections. Of course, you can easily imagine the connections.

This convolutional architecture is quite different to the architectures used in earlier chapters. But the overall picture is similar: a network made of many simple units, whose behaviours are determined by their weights and biases. And the overall goal is still the same: to use training data to train the network's weights and biases so that the network does a good job classifying input digits.

In particular, just as earlier in the book, we will train our network using stochastic gradient descent and backpropagation.

Recent progress in image recognition

In 1998, the year MNIST was introduced, it took weeks to train a state-of-the-art workstation to achieve accuracies substantially worse than those we can achieve using a GPU and less than an hour of training. Thus, MNIST is no longer a problem that pushes the limits of available technique; rather, the speed of training means that it is a problem good for teaching and learning purposes. Meanwhile, the focus of research has moved on, and modern work involves much more challenging image recognition problems. In this section, I briefly describe some recent work on image recognition using neural networks.

The section is different to most of the book. Through the book I've focused on ideas likely to be of lasting interest - ideas such as backpropagation, regularization, and convolutional networks. I've tried to avoid results which are fashionable as I write, but whose long-term value is unknown. In science, such results are more often than not ephemera which fade and have little lasting impact. Given this, a skeptic might say: "well, surely the recent progress in image recognition is an example of such ephemera? In another two or three years, things will have moved on. So surely these results are only of interest to a few specialists who want to compete at the absolute frontier? Why bother discussing it?"

Such a skeptic is right that some of the finer details of recent papers will gradually diminish in perceived importance. With that said, the past few years have seen extraordinary improvements using deep nets to attack extremely difficult image recognition tasks. Imagine a historian of science writing about computer vision in the year 2100. They will identify the years 2011 to 2015 (and probably a few years beyond) as a time of huge breakthroughs, driven by deep convolutional nets. That doesn't mean deep convolutional nets will still be used in 2100, much less detailed ideas such as dropout, rectified linear units, and so on. But it does mean that an important transition is taking place, right now, in the history of ideas. It's a bit like watching the discovery of the atom, or the invention of antibiotics: invention and discovery on a historic scale. And so while we won't dig down deep into details, it's worth getting some idea of the exciting discoveries currently being made.

The 2012 LRMD paper: Let me start with a 2012 paper. These are from the 2014 dataset, which is somewhat changed from 2011. Qualitatively, however, the dataset is extremely similar. Details about ImageNet are available in the original ImageNet paper, ImageNet: a large-scale hierarchical image database, by Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei (2009).:

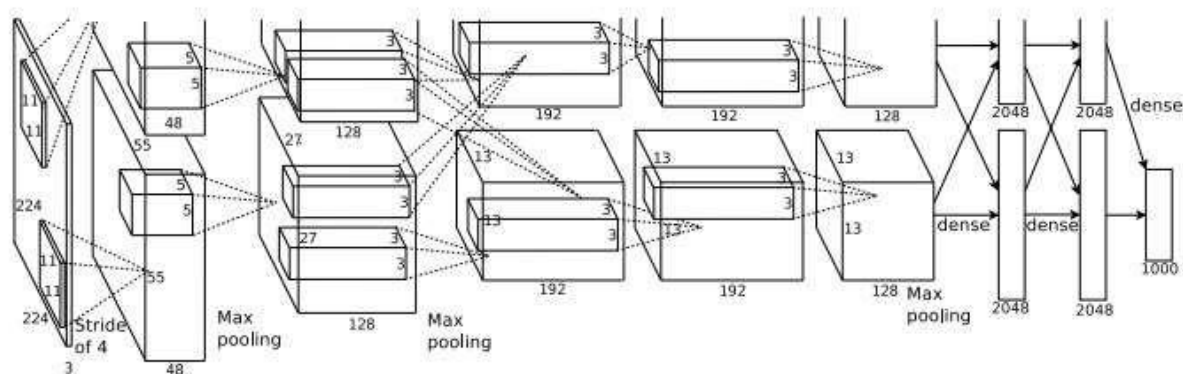
The 2012 KSH paper: The work of LRMD was followed by a 2012 paper of Krizhevsky, Sutskever and Hinton (KSH). KSH trained and tested a deep convolutional neural network using a restricted subset of the ImageNet data. The subset they used came from a popular machine learning competition - the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC). Using a competition dataset gave them a good way of comparing their

approach to other leading techniques. The ILSVRC-2012 training set contained about 1.2 million ImageNet images, drawn from 1,000 categories. The validation and test sets contained 50,000 and 150,000 images, respectively, drawn from the same 1,000 categories.

One difficulty in running the ILSVRC competition is that many ImageNet images contain multiple objects. Suppose an image shows a labrador retriever chasing a soccer ball. The so-called "correct" ImageNet classification of the image might be as a labrador retriever. Should an algorithm be penalized if it labels the image as a soccer ball? Because of this ambiguity, an algorithm was considered correct if the actual ImageNet classification was among the 55 classifications the algorithm considered most likely. By this top-55 criterion, KSH's deep convolutional network achieved an accuracy of 84.784.7 percent, vastly better than the next-best contest entry, which achieved an accuracy of 73.873.8 percent. Using the more restrictive metric of getting the label exactly right, KSH's network achieved an accuracy of 63.363.3 percent.

It's worth briefly describing KSH's network, since it has inspired much subsequent work. It's also, as we shall see, closely related to the networks we trained earlier in this chapter, albeit more elaborate. KSH used a deep convolutional neural network, trained on two GPUs. They used two GPUs because the particular type of GPU they were using (an NVIDIA GeForce GTX 580) didn't have enough on-chip memory to store their entire network. So they split the network into two parts, partitioned across the two GPUs.

The KSH network has 7 layers of hidden neurons. The first 5 hidden layers are convolutional layers (some with max-pooling), while the next 2 layers are fullyconnected layers. The output layer is a 1,000-unit softmax layer, corresponding to the 1,000 image classes. Here's a sketch of the network, taken from the KSH paper**Thanks to Ilya Sutskever.. The details are explained below. Note that many layers are split into 22 parts, corresponding to the 2 GPUs.



The input layer contains $3 \times 224 \times 224$ neurons, representing the RGB values for a $224 \times 224 \times 224 \times 224$ image. Recall that, as mentioned earlier, ImageNet contains images of varying resolution. This poses a problem, since a neural network's input layer is usually of a

fixed size. KSH dealt with this by rescaling each image so the shorter side had length 256. They then cropped out a 256×256 area in the center of the rescaled image. Finally, KSH extracted random 224×224 subimages (and horizontal reflections) from the 256×256 images. They did this random cropping as a way of expanding the training data, and thus reducing overfitting. This is particularly helpful in a large network such as KSH's. It was these 224×224 images which were used as inputs to the network. In most cases the cropped image still contains the main object from the uncropped image.

Moving on to the hidden layers in KSH's network, the first hidden layer is a convolutional layer, with a max-pooling step. It uses local receptive fields of size 11×11, and a stride length of 44 pixels. There are a total of 96 feature maps. The feature maps are split into two groups of 48 each, with the first 48 feature maps residing on one GPU, and the second 48 feature maps residing on the other GPU. The max-pooling in this and later layers is done in 3×3 regions, but the pooling regions are allowed to overlap, and are just 22 pixels apart.

The second hidden layer is also a convolutional layer, with a max-pooling step. It uses 5×5 local receptive fields, and there's a total of 256 feature maps, split into 128 on each GPU. Note that the feature maps only use 48 input channels, not the full 96 output from the previous layer (as would usually be the case). This is because any single feature map only uses inputs from the same GPU. In this sense the network departs from the convolutional architecture we described earlier in the chapter, though obviously the basic idea is still the same.

The third, fourth and fifth hidden layers are convolutional layers, but unlike the previous layers, they do not involve max-pooling. Their respective parameters are: (3) 384384 feature maps, with 3×3 local receptive fields, and 256 input channels; (4) 384384 feature maps, with 3×3 local receptive fields, and 192 input channels; and (5) 256256 feature maps, with 3×3 local receptive fields, and 192 input channels.

Note that the third layer involves some inter-GPU communication (as depicted in the figure) in order that the feature maps use all 256 input channels.

The sixth and seventh hidden layers are fully-connected layers, with 4,096 neurons in each layer.

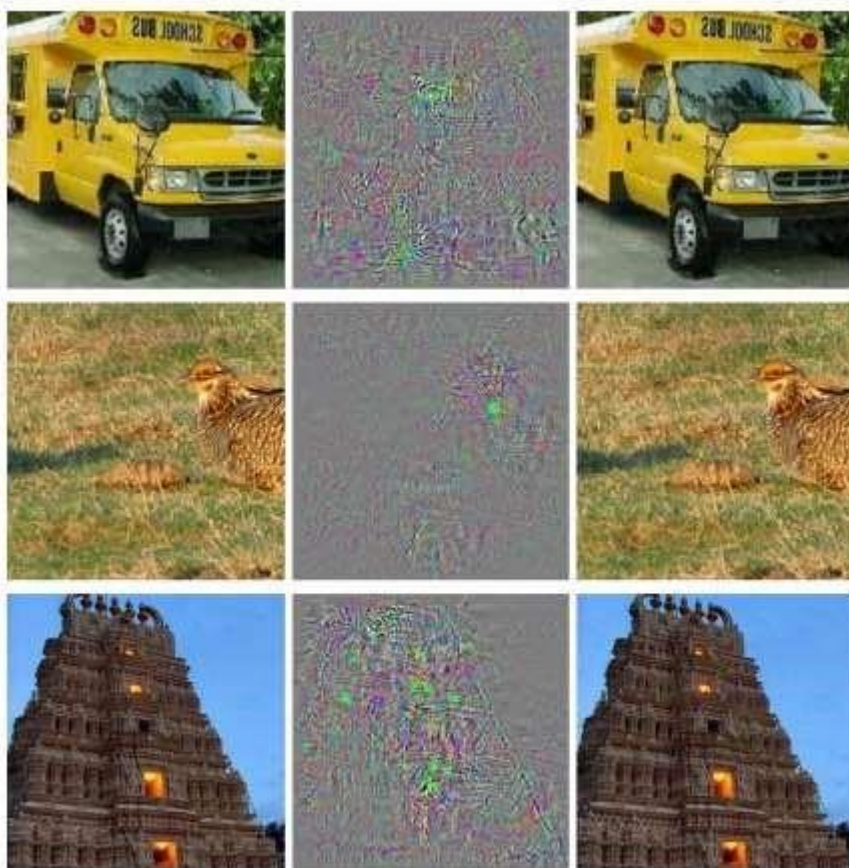
The output layer is a 1,000-unit softmax layer.

The KSH network takes advantage of many techniques. Instead of using the sigmoid or tanh activation functions, KSH use rectified linear units, which sped up training significantly. KSH's network had roughly 60 million learned parameters, and was thus, even with the large training set, susceptible to overfitting. To overcome this, they expanded the training set using the random cropping strategy we discussed above. They also further addressed overfitting by using a variant of l2 regularization, and dropout. The network itself was trained using momentum-based mini-batch stochastic gradient descent.

Other activity: I've focused on ImageNet, but there's a considerable amount of other activity using neural nets to do image recognition. Let me briefly describe a few interesting recent results, just to give the flavour of some current work.

One encouraging practical set of results comes from a team at Google, who applied deep convolutional networks to the problem of recognizing street numbers in Google's Street View imagery, by Ian J. Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, and Vinay Shet (2013).. In their paper, they report detecting and automatically transcribing nearly 100 million street numbers at an accuracy similar to that of a human operator. The system is fast: their system transcribed all of Street View's images of street numbers in France in less than an hour! They say: "Having this new dataset significantly increased the geocoding quality of Google Maps in several countries especially the ones that did not already have other sources of good geocoding." And they go on to make the broader claim: "We believe with this model we have solved [optical character recognition] for short sequences [of characters] for many applications."

I've perhaps given the impression that it's all a parade of encouraging results. Of course, some of the most interesting work reports on fundamental things we don't yet understand. For instance, a 2013 paper, by Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus (2013) showed that deep networks may suffer from what are effectively blind spots. Consider the lines of images below. On the left is an ImageNet image classified correctly by their network. On the right is a slightly perturbed image (the perturbation is in the middle) which is classified *incorrectly* by the network. The authors found that there are such "adversarial" images for every sample image, not just a few special ones.



This is a disturbing result. The paper used a network based on the same code as KSH's network - that is, just the type of network that is being increasingly widely used. While such neural networks compute functions which are, in principle, continuous, results like this suggest that in practice they're likely to compute functions which are very nearly discontinuous. Worse, they'll be discontinuous in ways that violate our intuition about what is reasonable behavior. That's concerning. Furthermore, it's not yet well understood what's causing the discontinuity: is it something about the loss function? The activation functions used? The architecture of the network? Something else? We don't yet know.

Now, these results are not quite as bad as they sound. Although such adversarial images are common, they're also unlikely in practice. As the paper notes:

The existence of the adversarial negatives appears to be in contradiction with the network's ability to achieve high generalization performance. Indeed, if the network can generalize well, how can it be confused by these adversarial negatives, which are indistinguishable from the regular examples? The explanation is that the set of adversarial negatives is of extremely low probability, and thus is never (or rarely) observed in the test set, yet it is dense (much like the rational numbers), and so it is found near virtually every test case.

Other approaches to deep neural nets

Recurrent neural networks (RNNs): In 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, with a frozen, crystalline quality to it. But suppose we allow the elements in the network to keep changing in a dynamic way. For instance, 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. Indeed, a neuron's activation might be determined in part by its own activation at an earlier time. That's certainly not what happens in a feedforward network. Or perhaps the activations of hidden and output neurons won't be determined just by the current input to the network, but also by earlier inputs.

Neural networks with this kind of time-varying behaviour are known as *recurrent neural networks* or *RNNs*. There are many different ways of mathematically formalizing the informal description of recurrent nets given in the last paragraph. You can get the flavour of some of these mathematical models by glancing at [the Wikipedia article on RNNs](#). As I write, that page lists no fewer than 13 different models. But mathematical details aside, the broad idea is that RNNs are neural networks in which there is some notion of dynamic change over time. And, not surprisingly, they're particularly useful in analysing data or processes that change over time. Such data and processes arise naturally in problems such as speech or natural language, for example.

One way RNNs are currently being used is to connect neural networks more closely to traditional ways of thinking about algorithms, ways of thinking based on concepts such as Turing machines and (conventional) programming languages. [A 2014 paper](#) developed an RNN which could take as input a character-by-character description of a (very, very simple!) Python program, and use that description to predict the output. Informally, the network is learning to "understand" certain Python programs. [A second paper, also from 2014](#), used RNNs as a starting point to develop what they called a neural Turing machine (NTM). This is a universal computer whose entire structure can be trained using gradient descent. They trained their NTM to infer algorithms for several simple problems, such as sorting and copying.

Long short-term memory units (LSTMs): One challenge affecting RNNs is that early models turned out to be very difficult to train, harder even than deep feedforward networks. Recall that the usual manifestation of this problem is that the gradient gets smaller and smaller as it is propagated back through layers. This makes learning in early layers extremely slow. The problem actually gets worse in RNNs, since gradients aren't just propagated backward through layers, they're propagated backward through time. If the network runs for a long time that can make the gradient extremely unstable and hard to learn from. Fortunately, it's possible to incorporate an idea known as long shortterm memory units (LSTMs) into RNNs. The 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, and many recent papers (including many that I linked above) make use of LSTMs or related ideas.

Deep belief nets, generative models, and Boltzmann machines: Modern interest in deep learning began in 2006, with papers explaining how to train a type of neural network known as a *deep belief network* (DBNs were influential for several years, but have since lessened in popularity, while models such as feedforward networks and recurrent neural nets have become fashionable. Despite this, DBNs have several properties that make them interesting.

One reason DBNs are interesting is that they're an example of what's called a *generative model*. In a feedforward network, we specify the input activations, and they determine the activations of the feature neurons later in the network. A generative model like a DBN can be used in a similar way, but it's also possible to specify the values of some of the feature neurons and then "run the network backward", generating values for the input activations. More concretely, a DBN trained on images of handwritten digits can (potentially, and with some care) also be used to generate images that look like handwritten digits. In other words, the DBN would in some sense be learning to write. In this, a generative model is much like the human brain: not only can it read digits, it can also write them. In Geoffrey Hinton's memorable phrase, to recognize shapes, first learn to generate images.

A second reason DBNs are interesting is that they can do unsupervised and semisupervised learning. For instance, when trained with image data, DBNs can learn useful features for understanding other images, even if the training images are unlabelled. And the ability to do unsupervised learning is extremely interesting both for fundamental scientific reasons, and - if it can be made to work well enough - for practical applications.

Given these attractive features, why have DBNs lessened in popularity as models for deep learning? Part of the reason is that models such as feedforward and recurrent nets have achieved many spectacular results, such as their breakthroughs on image and speech recognition benchmarks. It's not surprising and quite right that there's now lots of attention being paid to these models. There's an unfortunate corollary, however. The marketplace of ideas often functions in a winner-take-all fashion, with nearly all attention going to the current fashion-of-the-moment in any given area. It can become extremely difficult for people to work on momentarily unfashionable ideas, even when those ideas are obviously of real long-term interest. My personal opinion is that DBNs and other generative models likely deserve more attention than they are currently receiving. And I won't be surprised if DBNs or a related model one day surpass the currently fashionable models. For an introduction to DBNs, see this overview. I've also found this article helpful. It isn't primarily about deep belief nets, *per se*, but does contain much useful information about restricted Boltzmann machines, which are a key component of DBNs.

Other ideas: What else is going on in neural networks and deep learning? Well, there's a huge amount of other fascinating work. Active areas of research include using neural networks to do natural language processing (see also this informative review paper), machine translation,

as well as perhaps more surprising applications such as music informatics. There are, of course, many other areas too. In many cases, having read this book you should be able to begin following recent work, although (of course) you'll need to fill in gaps in presumed background knowledge.

On the future of neural networks

Machine learning, data science, and the virtuous circle of innovation: Of course, machine learning isn't just being used to build intention-driven interfaces. Another notable application is in data science, where machine learning is used to find the "known unknowns" hidden in data. This is already a fashionable area, and much has been written about it, so I won't say much. But I do want to mention one consequence of this fashion that is not so often remarked: over the long run it's possible the biggest breakthrough in machine learning won't be any single conceptual breakthrough. Rather, the biggest breakthrough will be that machine learning research becomes profitable, through applications to data science and other areas. If a company can invest 1 dollar in machine learning research and get 1 dollar and 10 cents back reasonably rapidly, then a lot of money will end up in machine learning research. Put another way, machine learning is an engine driving the creation of several major new markets and areas of growth in technology. The result will be large teams of people with deep subject expertise, and with access to extraordinary resources. That will propel machine learning further forward, creating more markets and opportunities, a virtuous circle of innovation.

The role of neural networks and deep learning: I've been talking broadly about machine learning as a creator of new opportunities for technology. What will be the specific role of neural networks and deep learning in all this?

To answer the question, it helps to look at history. Back in the 1980s there was a great deal of excitement and optimism about neural networks, especially after backpropagation became widely known. That excitement faded, and in the 1990s the machine learning baton passed to other techniques, such as support vector machines. Today, neural networks are again riding high, setting all sorts of records, defeating all comers on many problems. But who is to say that tomorrow some new approach won't be developed that sweeps neural networks away again? Or perhaps progress with neural networks will stagnate, and nothing will immediately arise to take their place?

For this reason, it's much easier to think broadly about the future of machine learning than about neural networks specifically. Part of the problem is that we understand neural networks so poorly. Why is it that neural networks can generalize so well? How is it that they avoid overfitting as well as they do, given the very large number of parameters they learn? Why is it that stochastic gradient descent works as well as it does? How well will neural networks perform as data sets are scaled? For instance, if ImageNet was expanded by a factor of 1010, would neural networks' performance improve more or less than other machine learning techniques? These are all simple, fundamental questions. And, at present, we understand the answers to these questions very poorly. While that's the case, it's difficult to say what role neural networks will play in the future of machine learning.

I will make one prediction: I believe deep learning is here to stay. The ability to learn hierarchies of concepts, building up multiple layers of abstraction, seems to be fundamental to making sense of the world. This doesn't mean tomorrow's deep learners won't be radically

different than today's. We could see major changes in the constituent units used, in the architectures, or in the learning algorithms. Those changes may be dramatic enough that we no longer think of the resulting systems as neural networks. But they'd still be doing deep learning.

Project

Utils.py:

```
import pandas as pd import numpy as np
import os import matplotlib.image as mpimg from imgaug import augmenters as iaa
import cv2 import random from sklearn.utils import shuffle
from tensorflow.keras.models import Sequential from tensorflow.keras.layers import Convolution2D, Flatten, Dense
from tensorflow.keras.optimizers import Adam
def getName(fpath):
    return fpath.split('\\')[-1]
def importDataInfo(path):
    columns = ["Center", "Left", "Right", "Steering", "Throttle", "Break", "Speed"]
    data = pd.read_csv(os.path.join(path, "driving_log.csv"), names=columns)
    # print(data.head())
    # print(getName(data["Center"][0]))
    data["Center"] = data["Center"].apply(getName)
    # print(data["Center"][0:4]) print("Total number of images: ", data.shape[0]) return data
    def balanceData(data, display = True):
        nBins = 31
        samplesPerBin = 1000
        hist, bins = np.histogram(data['Steering'], nBins)
        print(bins)
        center = (bins[:-1]+bins[1:]) * 0.5
        removedIndexList = [] for j in range(nBins):
            binDataList = []
            for i in range(len(data['Steering'])):
                if data['Steering'][i] >= bins[j] and data['Steering'][i] <= bins[j+1]:
                    binDataList.append(i)
            binDataList = shuffle(binDataList)
            binDataList = binDataList[samplesPerBin:]
            removedIndexList.extend(binDataList)
        print('removed Images', len(removedIndexList))
    def loadData(path, data):
        imagesPath = [] steering = [] for i in range(len(data)):
            indexedData = data.iloc[i] #
            print(indexedData)
            imagesPath.append(os.path.join(path, 'IMG', indexedData[0]))
```

```

steering.append(float(indexedData[3]))    imagesPath =
np.asarray(imagesPath)    steering = np.asarray(steering)
return imagesPath, steering
def
augmentImage(imgPath,steering):
img = mpimg.imread(imgPath)
    ## Pan    if
np.random.rand() < 0.5:
    pan = iaa.Affine(translate_percent = {'x' : (-0.1, 0.1), 'y' : (-0.1,
0.1)})    img =
pan.augment_image(img)
    ## Zoom    if
np.random.rand() < 0.5:
    zoom = iaa.Affine(scale = (1, 1.2))
img = zoom.augment_image(img)
    ### Brightness    if
np.random.rand() < 0.5:
    Brightness = iaa.Multiply((0.2, 1.2))
img = Brightness.augment_image(img)
    ### Flip    if
np.random.rand() < 0.5:
img = cv2.flip(img, 1)
steering = -steering
return img, steering
def preProcessing(img):
img = img[60:135,:,:]
    img = cv2.cvtColor(img, cv2.COLOR_RGB2YUV)
img = cv2.GaussianBlur(img, (3, 3), 0)    img
= cv2.resize(img, (200, 66))    img = img /
255    return img
def batchGen(imagesPath, steeringList, batchSize,
trainFlag):    while True:
    imgBatch = []
    steeringBatch = []
for i in range(batchSize):
    index = random.randint(0, len(imagesPath) - 1)
if trainFlag:
    img, steering = augmentImage(imagesPath[index],
steeringList[index])    else:
    img = mpimg.imread(imagesPath[index])
steering = steeringList[index]    img =
preProcessing(img)    imgBatch.append(img)
steeringBatch.append(steering)    yield
(np.asarray(imgBatch), np.asarray(steeringBatch)) def
createModel():
    model = Sequential()
    model.add(Convolution2D(24, (5, 5), (2, 2), input_shape = (66, 200,
3), activation = 'elu'))    model.add(Convolution2D(36, (5, 5), (2, 2),

```

```

activation = 'elu'))      model.add(Convolution2D(48, (5, 5), (2, 2),
activation = 'elu'))      model.add(Convolution2D(64, (3, 3), activation =
'elu'))      model.add(Convolution2D(64, (3, 3), activation = 'elu'))
        model.add(Flatten())
model.add(Dense(100, activation = 'elu'))
model.add(Dense(50, activation = 'elu'))
model.add(Dense(10, activation = 'elu'))
model.add(Dense(1))
        model.compile(Adam(learning_rate = 0.00001), loss =
'mse')
        return
model

```

TrainingSimulation.py

```

from utils import * from sklearn.model_selection
import train_test_split import matplotlib.pyplot as
plt
### Step 1 path
= "DataSet"
data = importDataInfo(path)
### Step 2 balanceData(data)

### Step 3 imagesPath, steering =
loadData(path,data)
# print(imagesPath)
# print(steering) ### step 4 x_train, x_val, y_train, y_val =
train_test_split(imagesPath,steering,test_size = 0.2, random_state = 5)
print('Total Training images: ',len(x_train)) print('Total validation
images: ', len(x_val))
### step 5 model = createModel() model.summary() ### step 6 history =
model.fit(batchGen(x_train, y_train, 10, 1),steps_per_epoch = 20, epochs =
2,
        validation_data =
batchGen(x_val,y_val,20,0),validation_steps = 20)

```

```

### step 6 model.save('model.h5')
print('model saved')
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.legend(['Training', 'Validation'])
plt.ylim([0, 1]) plt.title('Loss')
plt.xlabel('Epoch') plt.show()

```

Drive.py

```

print('Setting up') import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3' import
socketio import eventlet import numpy as np
from flask import Flask from
tensorflow.keras.models import load_model
import base64 from io import BytesIO from PIL
import Image import cv2 sio =
socketio.Server()
app = Flask(__name__) #
'__main__' maxspeed = 10
def
preProcess(img):
    img = img[60:135,:,:] img =
    cv2.cvtColor(img, cv2.COLOR_RGB2YUV) img =
    cv2.GaussianBlur(img, (3, 3), 0) img =
    cv2.resize(img, (200, 66)) img = img / 255
    return img

@sio.on('telemetry') def
telemetry(sid,data):
    speed = float(data['speed']) image =
    Image.open(BytesIO(base64.b64decode(data['image']))) image =
    np.asarray(image) image = preProcess(image) image =
    np.array([image]) steering = float(model.predict(image))
    throttle = 1.0 - speed / maxspeed
    print('{} {} {}'.format(steering, throttle, speed))
    sendControl(steering, throttle)

@sio.on('connect') def
connect(sid, environ):

```

```

print('connected')
sendControl(0, 0)
def sendControl(steering,
throttle):    sio.emit('steer',
data = {
    'steering_angle':steering.__str__(),
    'throttle':throttle.__str__()
}) if __name__ ==
'__main__':
    model = load_model('model.h5')
app = socketio.Middleware(sio, app)
eventlet.wsgi.server(eventlet.listen(('',4567))), app)

```

Output:

