Deep Learning Training Program

Sasank Chilamkurthy, Qure.ai

May 10, 2017

Contents

Calculus: Recap

Let's recap what a derivative is.

Derivative

Derivative of function $f(v)$ ($f'(v)$ or $\frac{df}{dv}$) measures sensitivity of change in $f(v)$ with respect of change in v.

Direction (i.e sign) of the derivative at a points gives the direction of (greatest) increment of the function at that point.

Gradient

The gradient is a multi-variable generalization of the derivative. It is a vector valued function. For function $f(v_1, v_2, \ldots, v_n)$, gradient is a vector whose components are n partial derivatives of f:

$$\nabla f = (\frac{\partial f}{\partial v_1}, \frac{\partial f}{\partial v_2}, \ldots, \frac{\partial f}{\partial v_n})$$

Similar to derivative, direction of the gradient at a point is the steepest ascent of the function starting from that point.

Optimization

Given a function $C(v_1, v_2, \ldots, v_n)$, how do we optimize it? i.e, find a point which minimizes this function globally.

This is a very generic problem; lot of practical and theoretical problems can be posed in this form. So, there is no general answer.

Gradient Descent

However, for a class of functions called convex functions, there is a simple algorithm which is guaranteed to converge to minima. Convex functions have only one minima. They look something like a valley:

To motivate our algorithm, imagine a ball is put at a random point on our valley and allowed to roll. Our common sense tells that ball will eventually roll to the bottom of the valley.

Let's roughly simulate the motion of the ball! Key observation is that the ball moves in the steepest direction of descent. This is negative [1] of the gradient's direction.

Great! Let's put together our algorithm:

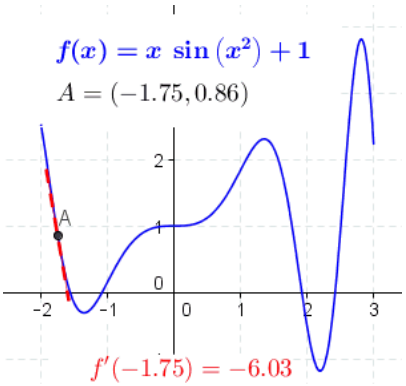


Figure 1: Derivative illustration. Red is for positive v direction and green is for negative v direction. Source.

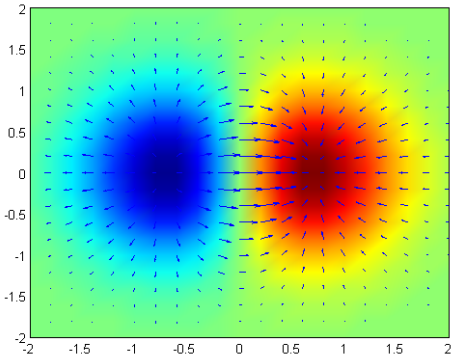


Figure 2: Gradient of the 2D function $f(x, y) = xe^{(x^2+y^2)}$ is plotted as blue arrows over the pseudocolor (red is for high values while blue is for low values) plot of the function. Source.

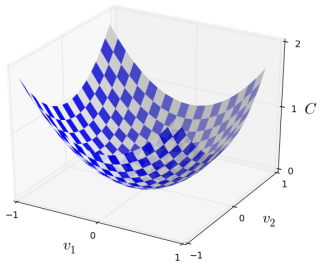


Figure 3: A 2d convex function as a valley. Source.

[1] Gradient gives us steepest direction of *ascent*.

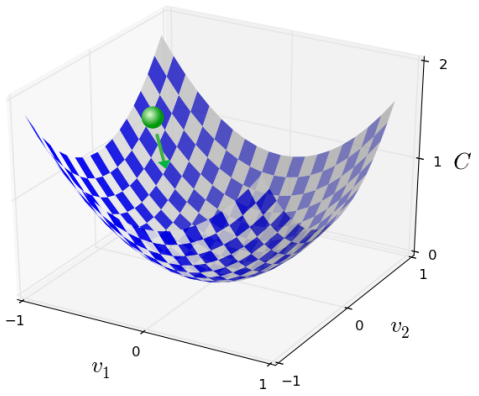


Figure 4: Gradient descent illustration: a ball on a valley. Source.

Algorithm 1: Gradient Descent

1: Start at a random point: v
2: **while** v hasn't converged **do**
3: Update v in the direction of steepest descent:

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

4: **end while**

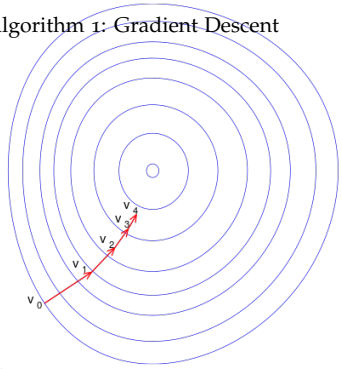


Figure 5: Gradient descent on a series of level sets. Source.

Here η is called *learning rate*. If it is too small, algorithm can be very slow and might take too many iterations to converge. If it is too large, algorithm might not even converge.

Example: Regression

Let's apply the things learnt above on linear regression problem. Here's a recap of linear regression:

Our model is $y(x) = wx + b$. Parameters are $v = (w, b)$ We are given training pairs $(x^1, y^1), (x^2, y^2), \ldots, (x^n, y^n)$ [2].

We want to find w and b which minimize the following cost/loss function:

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

where $C_i(w,b) = \frac{1}{2}\|y(x^i) - y^i\|^2 = \frac{1}{2}\|wx^i + b - y^i\|^2$ is the loss of the model for i th training pair.

Let's calculate gradients,

$$\nabla C = \frac{1}{n}\sum_{i=1}^{n} \nabla C_i$$

where ∇C_i is computed using:

[2] Notation clarifiction: Here, superscripts are indices, not powers

$$\nabla C_i = \left(\frac{\partial C_i}{\partial w}, \frac{\partial C_i}{\partial b}\right) = \Big((wx^i + b - y^i)x^i, (wx^i + b - y^i)\Big)$$

Update rule is

$$v \to v' = v - \eta \nabla C = v - \frac{\eta}{n} \sum_{i=1}^{n} \nabla C_i$$

Stochastic Gradient Descent

In the above example, what if we have very large number of samples i.e $n \gg 0$? At every optimization step, we have to compute ∇C_i for each sample $i = 1, 2, \ldots, n$. This can be very time consuming!

Can we approximate ∇C with a very few samples? Yes!

$$\nabla C = \frac{1}{n} \sum_{i=1}^{n} \nabla C_i \approx \frac{1}{m} \sum_{j \in S_m} \nabla C_j$$

where S_m is random subset of size $m \ll n$ of $1, 2, \ldots, n$. It turns out that this approximation, even though estimated only from a small random subset of samples, is good enough for convergence of gradient descent. This subset of data is called *minibatch* and this technique is called *stochastic gradient descent*.

Then stochastic gradient descent works by picking a randomly chosen subset of data and trains (i.e updates v) with gradient approximation computed from them. Next, another subset is picked up and trained with them. And so on, until we've exhausted all the training data, which is said to complete an *epoch* of training. Concretely, following is the stochastic gradient descent algorithm.

Algorithm 2: Stochastic Gradient Descent

1: Start at a random point: v
2: **for** a fixed number of epochs **do**
3: Randomly partition the data into minibatches each of size m
4: **for** For each minibatch S_m **do**
5: Update the parameters using

$$v \to v' = v - \frac{\eta}{m} \sum_{i \in S_m} \nabla C_j$$

6: **end for**
7: **end for**

Neural Networks

With this background, we are ready to start with neural networks.

Perceptron

Perceptron, a type of artificial neuron was developed in the 1950s and 1960s. Today, it's more common to use Rectified Linear Units

(ReLUs). Nevertheless, it's worth taking time to understand the older models.

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

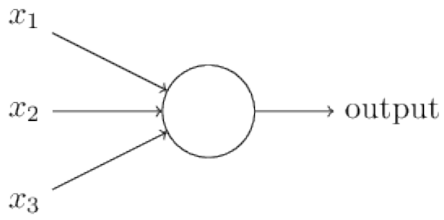


Figure 6: Perceptron Model. Source.

Weights $w_1, w_2, \ldots w_n$ decide the importance of each of the inputs on output $y(x)$. There is also a *threshold* b to decide the output. These are the parameters of the model. The expression of the output is

$$y(x) = \sigma\left(\sum_j w_j x_j - b\right)$$

where $\sigma(z)$ is step function,

$$\sigma(z) \quad = \quad \begin{cases} 0 & \text{if } z \leq 0 \\ 1 & \text{if } z > 0 \end{cases}$$

Therefore

$$y(x) \quad = \quad \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq b \\ 1 & \text{if } \sum_j w_j x_j > b \end{cases}$$

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. An example [3]:

[3] This example is straight from here

Another way perceptrons can be used is to compute the elementary logical functions we usually think of as underlying computation, functions such as `AND`, `OR`, and `NAND`. Check that following perceptron implements `NAND`:

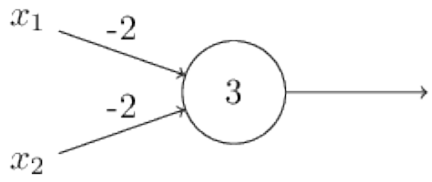


Figure 7: `NAND` implemented by perceptron. Source

If you are familiar with digital logic, you will know that `NAND` gate is a universal gate. That is, any logical computation can be computed using just `NAND` gates. Then, the same property follows for perceptrons.

Multi Layer Perceptrons and Sigmoid

Although perceptron isn't a complete model of human decision-making, the above example illustrates 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

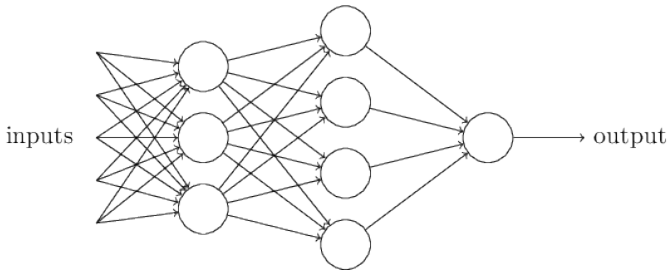


Figure 8: Multi layer perceptron. Source

This network is called *Multi layered perceptron (MLP)*. In this MLP, 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.

How do we learn the parameters of the above model? Gradient Descent! [4] However, the network is very discontinuous. 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. This makes it very difficult for gradient descent to converge.

[4] MLP model is far from convex. Therefore, gradient descent is not guaranteed to converge! But it turns out that it works fine with a few tweaks which we describe below.

How do we overcome this? What is the source of this discontinuity? Remember that output of perceptron is given by $y(x) = \sigma\left(\sum_j w_j x_j - b\right)$ where $\sigma(z)$ is step function

$$\sigma(z) = \begin{cases} 0 & \text{if } z \leq 0 \\ 1 & \text{if } z > 0 \end{cases}$$

This $\sigma(z)$ is the source of discontinuity. Can we replace step function with a smoother version of it?

Check out the following function:

$$\sigma(z) = \frac{1}{1+e^{-z}}.$$

If you graph it, it's quite clear that this function is smoothed out version of a step function. This function is called *sigmoid*.

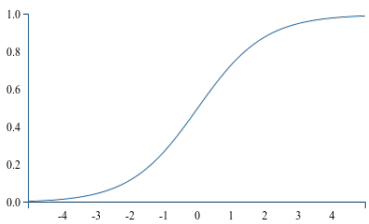


Figure 9: Sigmoid function. When z is large and positive, Then $e^{-z} \approx 0$ and so $\sigma(z) \approx 1$. Suppose on the other hand that z is very negative. Then $e^{-z} \to \infty$, and $\sigma(z) \approx 0$. Source.

With the *sigmoid neuron*, gradient descent can converges. Before computing gradients for gradient descent, we need to discuss loss and activation functions.

ReLU Activation

By now, you have seen that general form of a artificial neuron is $y(x) = \sigma\left(\sum_j w_j x_j - b\right)$. Here the function $\sigma(z)$ is called *activation function*. So far, we have seen two different activations:

1. Step function
2. Sigmoid function

Let me introduce another activation function, *rectifier* or *rectified linear unit* (ReLU):

$$\sigma(z) = \max(0, z)$$

Because of reasons we describe later [5], ReLUs are preferred activation functions these days. You will almost never see sigmoid activation function in modern deep neural networks.

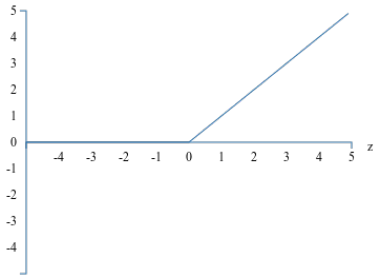


Figure 10: ReLU. Source.

[5] Vanishing gradients problem. Read more here

Loss functions

To train any machine learning model, we need to measure who well the model fits to training set. This is called loss/cost [6] function. In regression problem we discussed before, cost function was $C(w, b) = \frac{1}{2n}\sum_{i=1}^{n} \|y(x^i) - y^i\|^2$. Minimizing the cost function trains the model.

We will make two assumptions about our cost function:

1. The cost function can be written as an average over cost functions C_i for individual training examples, (x^i, y^i). i.e, $C = \frac{1}{n}\sum_{i=1}^{n} C_i$
2. Cost can be written as a function of the outputs from the neural network. i.e, $C_i = L(y(x^i), y^i)$ where $y(x)$ is the output from the network.

[6] I use terms loss function and cost function interchangeably.

In the case of regression, we used L_2 loss, $L_2(o, y) = \frac{1}{2}\|o - y\|^2$. We could have also used L_1 loss, $L_1(o, y) = |o - y|$.

Cross Entropy Loss

What if we have a classification problem? What loss do we use? Consider a MLP for digit classification on the right.

Here, we have output o of size 10 and a target class y. We could have used $L_1(o, e_y)$ or $L_2(o, e_y)$ loss where e_y is yth unit vector. But it turns out that this doesn't work very well.

Instead, we will consider the outputs as a probability distribution over 10 classes and use what is called a cross entropy loss:

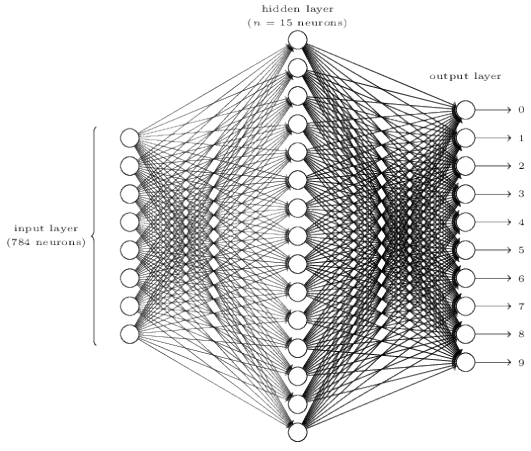


Figure 11: MLP for digit classification. Source.

$$L(o, y) = -\log(o_y)$$

To understand this function, realize that o_y, yth output is the predicted probability of the target class y. Minimizing negative of log of this probability, maximizes the probability. Also, $L \geq 0$ because $\log(x) < 0$ for $x \in [0, 1]$.

Softmax Activation

If we use cross entropy loss, we cannot use sigmoids as activations of output layer because sigmoids do not guarantee a probability distribution. Although each component of the output will be in $[0, 1]$, they need not add up to 1.

We therefore use an activation layer called *softmax*. According to this function, the activation a_j of the jth output neuron is

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

where in the denominator we sum over all the output neurons. This expression may seem opaque if you are not familiar with it. Observe the following:

1. Activations are positive: $a_j \geq 0$
2. Activations sum to 1: $\sum_j a_j = 1$
3. If you increase z_m keeping others constant, a_m increases. Other activations decrease to ensure sum remains 1.
4. If z_m is much larger than the others, $a_m \approx 1$ and $a_k \approx 0$ for $k \neq m$.

Therefore softmax is a probability distribution which behaves like smooth version of `argmax`.

Backpropogation

We have so far discussed the model component of neural networks. We haven't yet discussed how we learn the parameters of the networks i.e, weights and biases of the layers.

As expected, we will use stochastic gradient descent. For this, we need gradients of C_i, loss for the ith training example, with respect to all the parameters of network. Computation of this quantity, ∇C_i is slightly involved. Let's start with writing the expression for C_i. Let's represent all the parameters of the network with θ:

$$C_i(\theta) = L\left(y(x^i, \theta), y^i\right)$$

Let's break the above function into composition of layers (or functions in general)[7]

[7] Dropping subscript for brevity

$$C = y_L \circ y_{L-1} \circ \cdots \circ y_l \circ \cdots \circ y_1$$

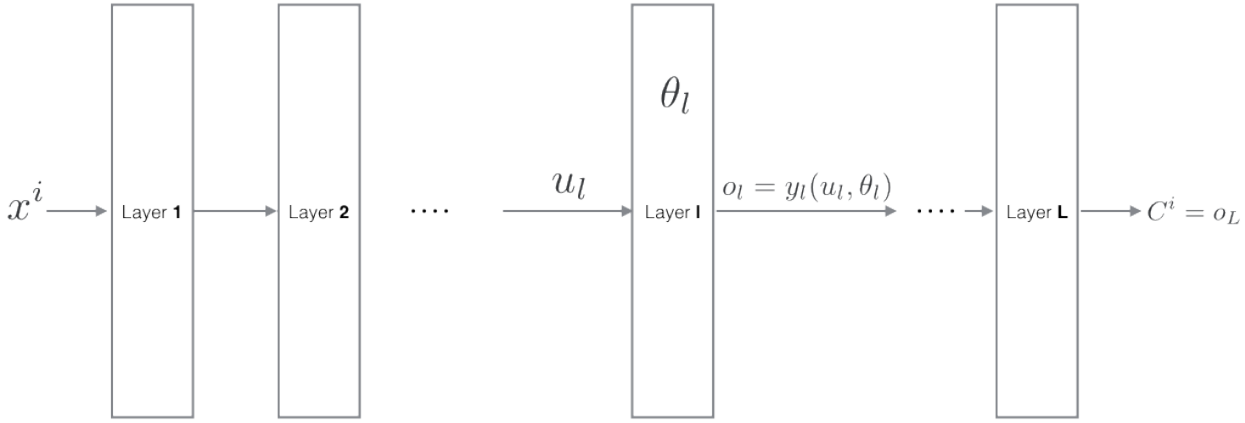


Figure 12: Cost function as composition

Here, lth [8] layer/function takes in input u_l and outputs o_l

$$o_l = y_l(u_l, \theta_l) \tag{1}$$

where θ_l are learnable parameters of this layer. Since output of $l-1$th layer is fed to l th layer as input,

$$u_l = o_{l-1} \tag{2}$$

We require $\nabla C = \left(\frac{\partial C}{\partial \theta_1}, \frac{\partial C}{\partial \theta_2}, \ldots, \frac{\partial C}{\partial \theta_L}\right)$. Therefore, we need to compute, for $l = 1, 2, \ldots L$

$$\frac{\partial C}{\partial \theta_l} = \frac{\partial o_L}{\partial \theta_l}$$

To compute this quantity, we will compute generic [9]:

$$\frac{\partial o_m}{\partial \theta_l}$$

Before getting started, let's write down the chain rule. Chain rule is the underlying operation of our algorithm.

Chain Rule: For function f(x, y),

$$\frac{\partial f}{\partial t} = \frac{\partial f}{\partial x} * \frac{\partial x}{\partial t} + \frac{\partial f}{\partial y} * \frac{\partial y}{\partial t} \tag{3}$$

If $m > l$,

$$\frac{\partial o_m}{\partial \theta_l} = 0 \tag{4}$$

because output of the earlier layers doesn't depend on parameters of the later layer.

[8] In particular, $C = o_L = y_L(u_L) = L(u_L, y^i)$

[9] You will see ahead why this quantity is useful.

If $m = l$, using equation (1) and the fact that u_l and θ_l are independent,

$$\frac{\partial o_l}{\partial \theta_l} = \frac{\partial y_l}{\partial \theta_l} \tag{5}$$

$\frac{\partial y_l}{\partial \theta_l}$ is a computable quantity which depends on the form of layer y_l.

If $m < l$, using equation (1), (2), chain rule (3),

$$\begin{aligned}
\frac{\partial o_m}{\partial \theta_l} &= \frac{\partial y_m}{\partial \theta_l} \\
&= \frac{\partial y_m}{\partial u_m} * \frac{\partial u_m}{\partial \theta_l} + \frac{\partial y_m}{\partial \theta_m} * \frac{\partial \theta_m}{\partial \theta_l} \\
&= \frac{\partial y_m}{\partial u_m} * \frac{\partial u_m}{\partial \theta_l} \\
&= \frac{\partial y_m}{\partial u_m} * \frac{\partial o_{m-1}}{\partial \theta_l}
\end{aligned}$$

Therefore,

$$\frac{\partial o_m}{\partial \theta_l} = \frac{\partial y_m}{\partial u_m} * \frac{\partial o_{m-1}}{\partial \theta_l} \tag{6}$$

Like $\frac{\partial y_l}{\partial \theta_l}$, $\frac{\partial y_l}{\partial u_l}$ is a computable quantity which depends on layer y_l. Let's put everything together and compute the required quantity [10]:

[10] Apply (6) repetitively

$$\begin{aligned}
\frac{\partial C}{\partial \theta_l} &= \frac{\partial o_L}{\partial \theta_l} \\
&= \frac{\partial y_L}{\partial u_L} * \frac{\partial o_{L-1}}{\partial \theta_l} \\
&= \frac{\partial y_L}{\partial u_L} * \frac{\partial y_{L-1}}{\partial u_{L-1}} * \frac{\partial o_{L-2}}{\partial \theta_l} \\
&\vdots \\
&= \frac{\partial y_L}{\partial u_L} * \frac{\partial y_{L-1}}{\partial u_{L-1}} * \cdots * \frac{\partial y_{l-1}}{\partial u_{l-1}} * \frac{\partial o_l}{\partial \theta_l} \\
&= \frac{\partial y_L}{\partial u_L} * \frac{\partial y_{L-1}}{\partial u_{L-1}} * \cdots * \frac{\partial y_{l-1}}{\partial u_{l-1}} * \frac{\partial y_l}{\partial \theta_l}
\end{aligned}$$

Now, algorithm to compute gradients ∇C, i.e. $\frac{\partial C}{\partial \theta_l}$ for all l is fairly clear:

Algorithm 3: Back Propogation

{Forward pass}
Set $u_0 = x$
for $l = 1, \dots L$ **do**
    Store $u_l = y_l(u_{l-1}, \theta_l)$
end for
{Backward pass}
Set `buffer` $= 1$
for $l = L, L-1, \dots 1$ **do**
    Store $\frac{\partial C}{\partial \theta_l} = \frac{\partial y_l}{\partial \theta_l} *$ `buffer`
    Update `buffer` $= \frac{\partial y_l}{\partial u_l} *$ `buffer`
end for
return $\left(\frac{\partial C}{\partial \theta_1}, \frac{\partial C}{\partial \theta_2}, \dots, \frac{\partial C}{\partial \theta_L}\right)$.

Although this derivation is for scalar functions, it will work with vector functions with a few modifications.

Deep Networks and why they are hard to train

Whenever you are asked to do any complex task, you usually break it down to sub tasks and solve the component subtasks. For instance, suppose you're designing a logical circuit to multiply two numbers. Chances are your circuit will look something like this:

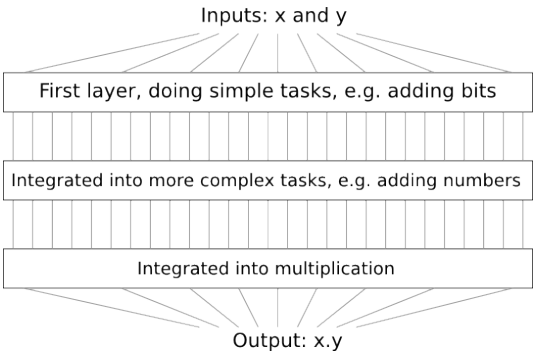


Figure 13: Logical circuit for multiplication. Source.

Similarly deep neural networks (i.e lot of layers) can build up multiple layers of abstraction. Consider the following network:

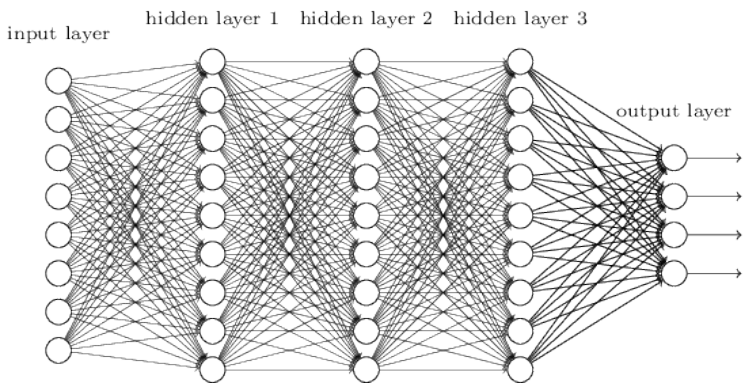


Figure 14: Deep Neural Network. Source.

If we're doing visual pattern recognition, then the neurons in the

first layer might learn to recognize edges, the neurons in the second layer could learn to recognize more complex shapes, say triangle or rectangles, built up from edges. The third layer would then recognize still more complex shapes. And so on. These multiple layers of abstraction seem likely to give deep networks a compelling advantage in learning to solve complex pattern recognition problems.

How do we train such deep networks? Stochastic gradient descent as usual. But we'll run into trouble, with our deep networks not performing much (if at all) better than shallow networks. Let's try to understand why are deep networks hard to train:

1. Consider the number of parameters in the network. They are huge! If we have to connect 1000 unit hidden layer to 224x224 (50,176) image, we have $65,536 * 1000 \approx 50e6$ parameters in that layer alone! There are so many parameters that network can easily overfit on the data without generalization.

2. Gradients are unstable. Recall the expression for the gradients, $\frac{\partial C}{\partial \theta_l} = \frac{\partial y_L}{\partial u_L} * \frac{\partial y_{L-1}}{\partial u_{L-1}} * \cdots * \frac{\partial y_{l-1}}{\partial u_{l-1}} * \frac{\partial y_l}{\partial \theta_l}$. If few of $\frac{\partial y_m}{\partial u_m} \ll 1$, they will multiply up and make $\frac{\partial C}{\partial \theta_l} \approx 0$ [11]. Similarly if few of $\frac{\partial y_m}{\partial u_m} \gg 1$, they make $\frac{\partial C}{\partial \theta_l} \to \infty$.

[11] This is the reason why sigmoids are avoided. For sigmoid, $\frac{\partial y}{\partial u} = \frac{d\sigma}{dz}|_{z=u}$ is close to zero if u is either too large or too small. It's maximum is only $1/4$

Keep these two points in mind. We will see several approaches to deep learning that to some extent manage to overcome or route around these.

Convolutional Neural Networks

Let's go back to the problem of handwritten digit recognition. MLP looks like this:

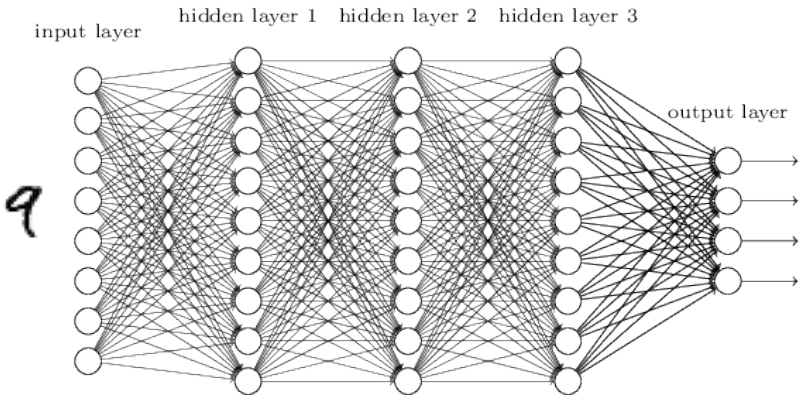


Figure 15: MLP for MNIST. Source.

In particular, we have connected all the pixels in 28x28 images i.e, 784 pixels to each neuron in hidden layer 1.

Upon reflection, it's strange to use networks with fully-connected layers to classify images. The reason is that such a network architecture does not take into account the spatial structure of the

images. For instance, it treats input pixels which are far apart and close together on exactly the same footing. Such concepts of spatial structure must instead be inferred from the training data.

But what if, instead of starting with a network architecture which is *tabula rasa*, we used an architecture which tries to take advantage of the spatial structure? In this section I describe convolutional neural networks. These networks 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:

1. Local receptive fields
2. Shared weights
3. Pooling.

Local receptive fileds

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.

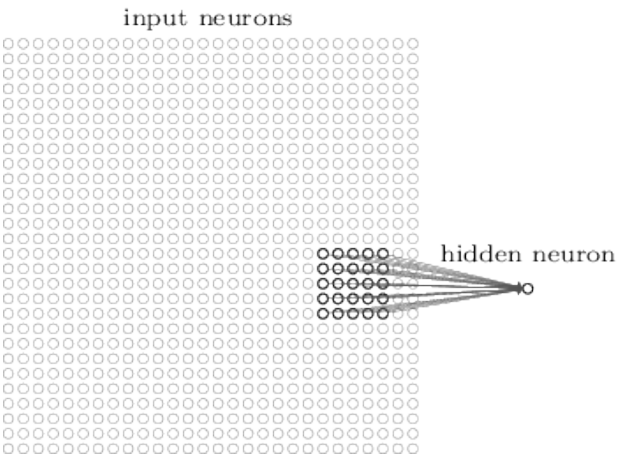


Figure 16: Local receptive fields of convolution. Source.

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:

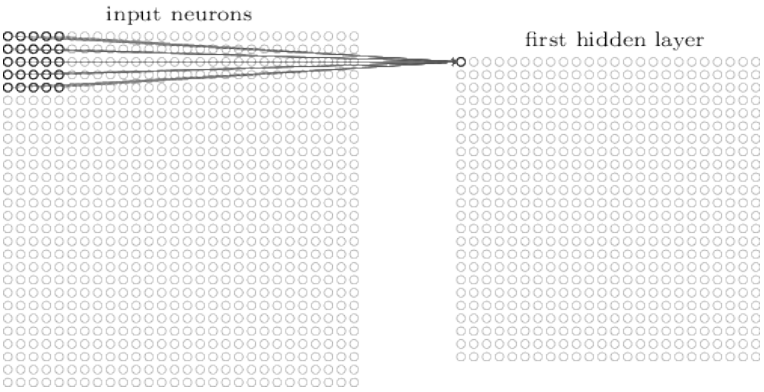


Figure 17: Convolution. Source.

Then we slide the local receptive field over by one pixel [12] to the right (i.e., by one neuron), to connect to a second hidden neuron [13]:

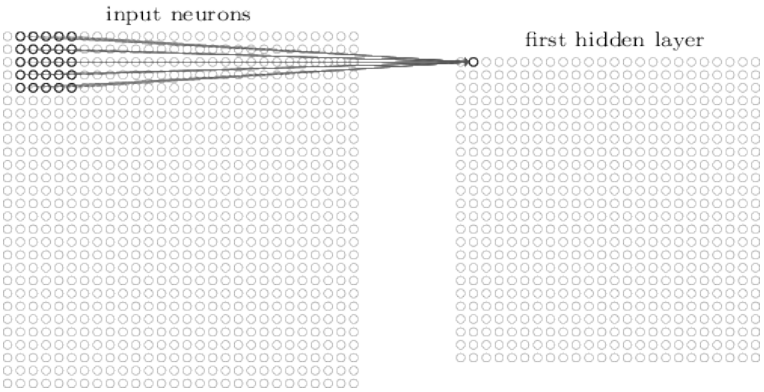


Figure 18: Slide the convolution. Source.

[12] Sometimes a different *stride* length (e.g. 2) is used.

[13] 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.

Shared weights and biases

I've said that each hidden neuron has a bias and 5x5 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 24x24 hidden neurons.

Sharing weights and biases means that all the neurons in the first hidden layer detect exactly the same feature just at different locations in the input image. To see why this makes sense, consider the following convolution filter:

Let's take an example image and apply convolution on a receptive filed:

Basically, in the input image, if there is a shape that generally resembles the curve that this filter is representing, then all of the multiplications summed together will result in a large value! Now let's see what happens when we move our filter.

Therefore, this convolution picks up a right bending curve wherever it is on . 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

The network structure I've described so far can detect just a

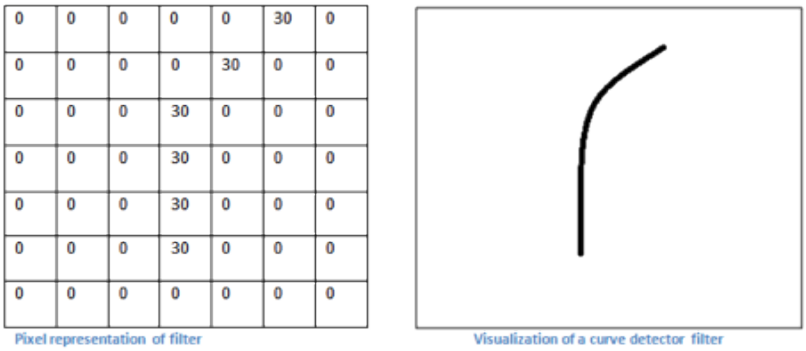


Figure 19: A convolutional filter. Source.

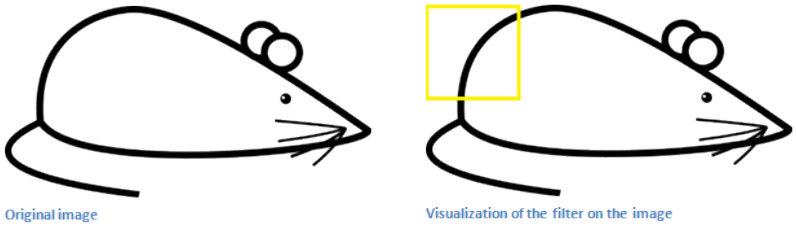


Figure 20: Convolution on an example image. Source.

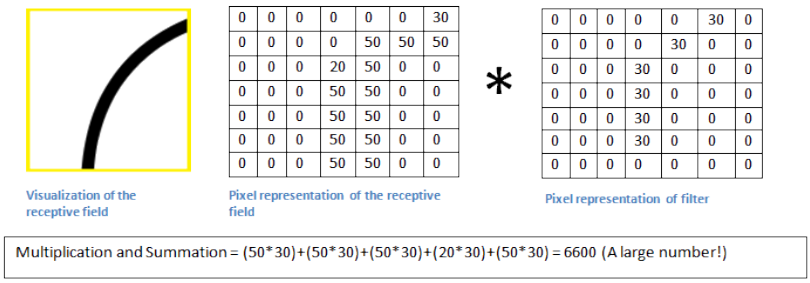


Figure 21: Apply convolution. Source.

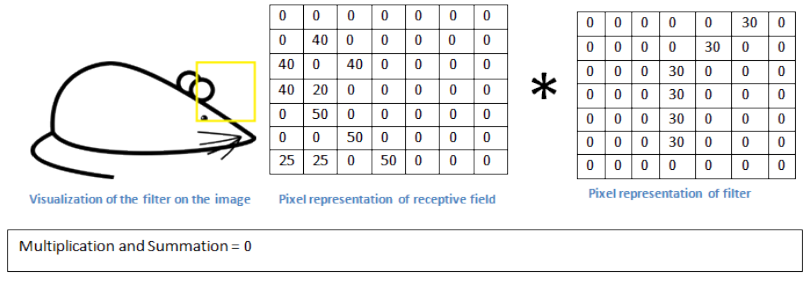


Figure 22: Apply convolution at a different receptive field. Source.

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

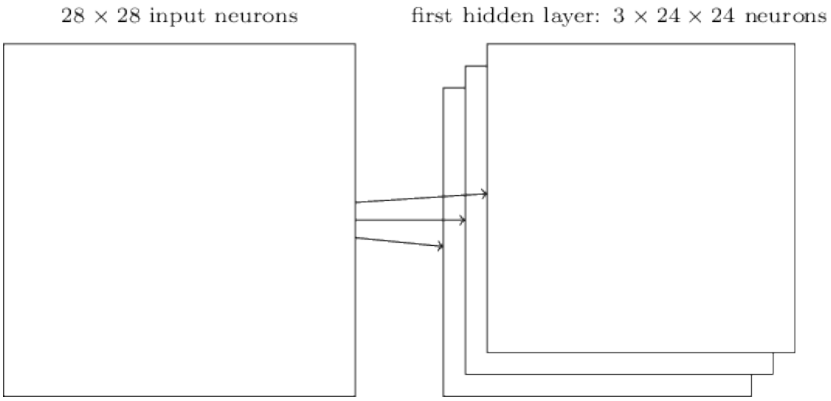


Figure 23: Apply convolution at a different receptive field. Source.

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=5x5 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 20x26=520 parameters defining the convolutional layer. By comparison, suppose we had a fully connected first layer, with 784=28x28 input neurons, and a relatively modest 30 hidden neurons. That's a total of 784x30 weights, plus an extra 30 biases, for a total of 23,550 parameters.

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) 2x2 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 2x2 input region, as illustrated in the following diagram:

Note that since we have 24ÃŮ24 neurons output from the convolutional layer, after pooling we have 12ÃŮ12 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

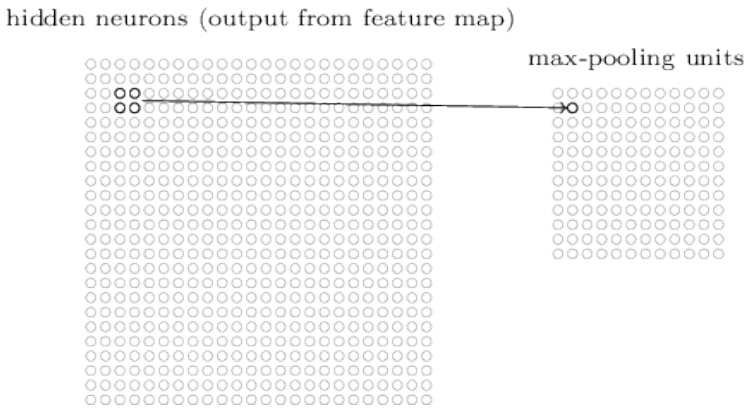


Figure 24: Pooling Layer. Source.

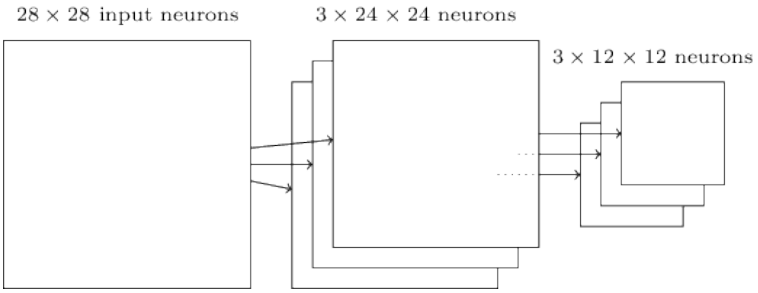


Figure 25: Convolution and pooling layer. Source.

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.

Case Study: LeNet

Let's put everything we've learnt together and analyze one of the very early successes [14] of convolutional networks: LeNet. This is the *architecture* of LeNet:

[14] LeNet is published in 1998! CNNs are not exactly new.

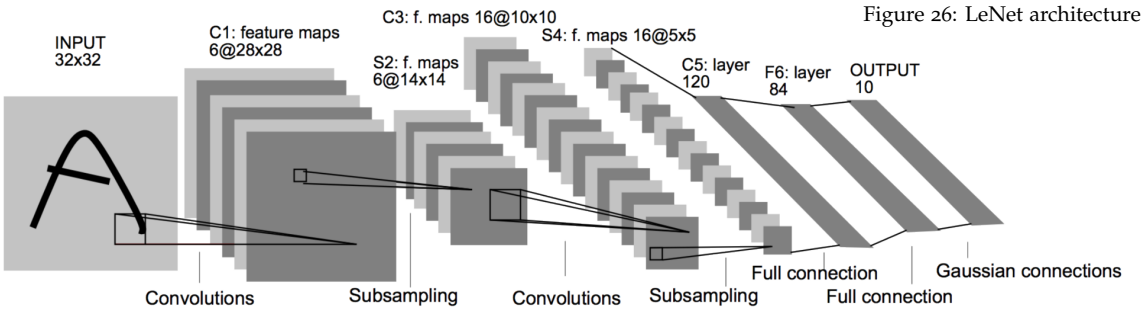


Figure 26: LeNet architecture Source.

Let's go over each of the component layers of LeNet: [15]

[15] I actually describe slightly modified version of LeNet.

- **Input**: Gray scale image of size 32 x 32.
- **C1**: Convolutional layer of 6 feature maps, kernel size (5, 5) and stride 1. Output size therefore is 6 X 28 x 28. Number of trainable parameters is $(5 * 5 + 1) * 6 = 156$.

- **S2**: Pooling/subsampling layer with kernel size (2, 2) and stride 2. Output size is 6x14x14. Number of trainable parameters = 0.
- **C3**: Convolutional layer of 16 feature maps. Each feature map is connected to all the 6 feature maps from the previous layer. Kernel size and stride are same as before. Output size is 16x10x10. Number of trainable parameters is $(6 * 5 * 5 + 1) * 16 = 2416$.
- **S4**: Pooling layer with same *hyperparameters* as above. Output size = 16 x 5 x 5.
- **C5**: Convolutional layer of 120 feature maps and kernel size (5, 5). This amounts to *full connection* with outputs of previous layer. Number of parameters are $(16 * 5 * 5 + 1) * 120 = 48120$.
- **F6**: *Fully connected layer*[16] of 84 units. i.e, All units in this layer are connected to previous layer's outputs. Number of parameters is $(120 + 1) * 84 = 10164$
- **Output**: Fully connected layer of 10 units with softmax activation[17].

[16] This is same as layers in MLP we've seen before.

[17] Ignore 'Gaussian connections'. It is for a older loss function no longer in use.

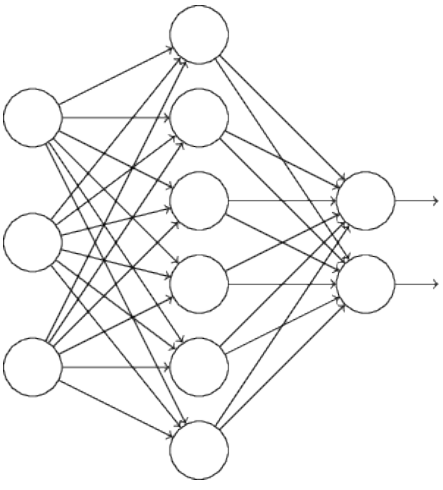
Dataset used was MNIST. It has 60,000 training images and 10,000 testing examples.

Tricks of the Trade

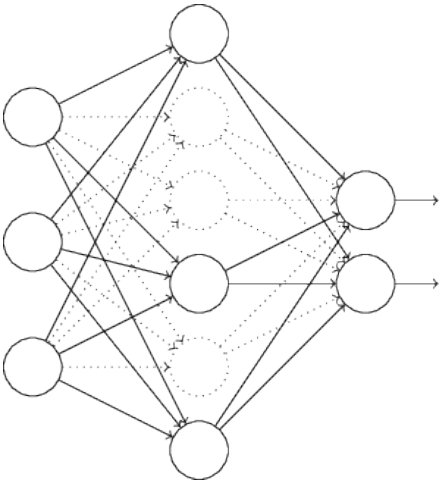
Dropout

With so many parameters in neural networks, overfitting is a real problem. For example, LeNet has about the same number of parameters as there are training examples. There are a few techniques like L_1/L_2 regularization you might be familiar with. These modify cost function by adding L_1/L_2 norm of parameters respectively.

Dropout is radically different for regularization. We modify the network itself instead of the cost function. Suppose we're trying to train a network:



With dropout, We start by randomly (and temporarily) deleting half the hidden neurons in the network, while leaving the input and output neurons untouched. After doing this, we'll end up with a network along the following lines.



We forward-propagate the input x 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.

The weights and biases will have been learnt under conditions in which half the hidden neurons were dropped out. When we actually run the full network that means that twice as many hidden neurons will be active. To compensate for that, we halve the weights outgoing from the hidden neurons.

Why would dropout help? Explanation from AlexNet paper [18]:

> 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.

[18] AlexNet is the paper which lead to renaissance of CNNs.

In other words, we can think of dropout as a way of making sure that the model is robust to the loss of any individual piece of evidence. Of course, the true measure of dropout is that it has been very successful in improving the performance of neural networks.

Data Augmentation

You probably already know that more data leads to better accuracy. It's not surprising that this is the case, since less training data means our network will be exposed to fewer variations in the way human beings write digits.

Obtaining more training data can be expensive, and so is not always possible in practice. However, there's another idea which can work nearly as well, and that's to artificially expand the training data. Suppose, for example, that we take an MNIST training image of a five,

and rotate it by a small amount, let's say 15 degrees:

It's still recognizably the same digit. And yet at the pixel level it's quite different to any image currently in the MNIST training data. 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.

We call such expansion as *data augmentation*. Rotation is not the only way to augment the data. A few examples are crop, zoom etc. The general principle is to expand the training data by applying operations that reflect real-world variation.

Weight initialization and Batch Normalization

Training a neural network is a highly non convex problem. Therefore, initialization of parameters to be optimized is important. To understand better, recall the unstable gradient problem. This is the equation of gradients for parameters in jth layer:

$$\frac{\partial C}{\partial \theta_l} = \frac{\partial y_L}{\partial u_L} * \frac{\partial y_{L-1}}{\partial u_{L-1}} * \cdots * \frac{\partial y_{l-1}}{\partial u_{l-1}} * \frac{\partial y_l}{\partial \theta_l}$$

If a layer is not properly initialized, it scales inputs by k, i.e $\frac{\partial y_m}{\partial u_m} \approx k$. Therefore gradients of parameters in l th layer is

$$\frac{\partial C}{\partial \theta_l} = k^{L-l}$$

Thus, $k > 1$ leads to extremely large gradients and $k < 1$ to very small gradients in initial layers. Therefore, we want

$$k \approx 1$$

This can be made sure with good weight initialization. Historically, bad weight inits are what prevented deep neural networks to be trained.

A recently developed technique called Batch Normalization alleviates a lot of headaches with initializations by explicitly forcing this throughout a network. The core observation is that this is possible because normalization is a simple differentiable operation.

It has become a very common practice to use Batch Normalization in neural networks. In practice, networks that use Batch Normalization are significantly more robust to bad initialization.

Practical Advice

ImageNet Dataset and ILSVRC

ImageNet is a *huge* dataset for visual recognition research. It currently has about *14 million* images tagged manually.

ImageNet project runs an annual contest, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), where algorithms compete to correctly classify and detect objects and scenes. ILSVRC recognition challenge is conducted on a subset of ImageNet: 1.2 million images with 1000 classes.

Here are example images and the recognition results:

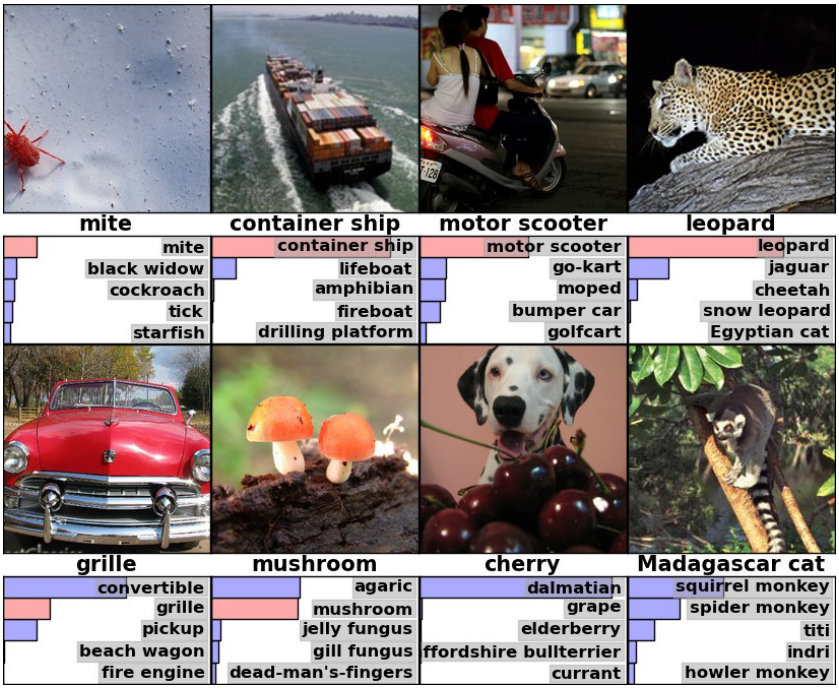


Figure 27: Images from imagenet. Source.

Alexnet was the first CNN to participate in ILSVRC and won the 2012 challenge by a significant margin. This lead to a renaissance of CNNs for visual recognition. This is the architecture:

It isn't too different from LeNet we discussed before. Over the years, newer CNN architectures won this challenge. Notable ones are

- VGGNet

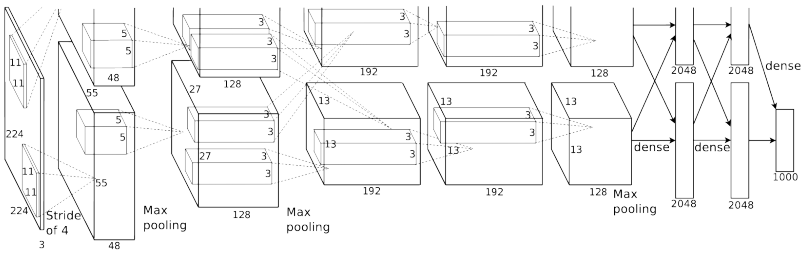


Figure 28: Alexnet architecture. Source.

- GoogLeNet
- Inception
- ResNet

You will keep on hearing these architectures if you work more on CNNs. Here are the accuracies from these networks:

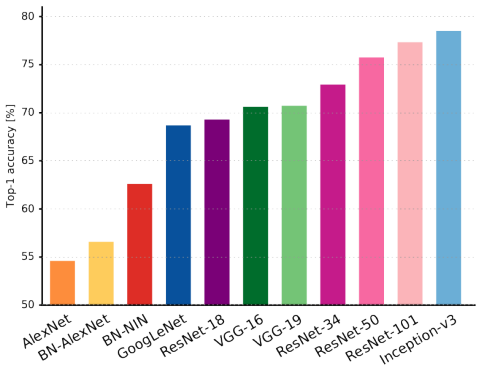


Figure 29: Top 1 accuracies on ILSVRC. Source.

Transfer Learning

To train a CNNs like above from scratch (i.e random initialization), you will need a huge dataset of the ImageNet scale. In practice, very few people do this.

Instead, you will pretrain your network on large dataset like imagenet and use the learned weights as initializations. Usually, you will just download the trained model of one of the above architectures from internet and use them as your weight initializations. This is called *transfer learning*.

This is a very powerful trick. For example, here pretrained weights of ResNet-18 are used to train a classifier to classify ants and bees.

This is the dataset used:

Training : 120 ants + 120 bees images

Testing : 75 ants + 75 bees images

With this small dataset, accuracy is around 95 % !

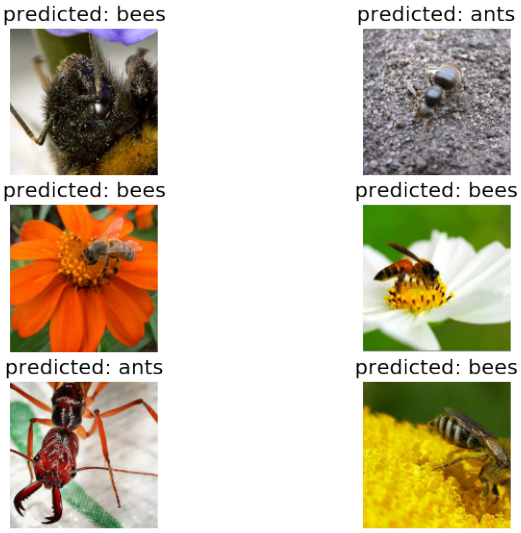


Figure 30: Ants/Bees classifier. Source.

Why does transfer learning work? 1.2 million images in imagenet cover wide diversity of real world images. Filters learned on imagenet will therefore be sufficiently general to apply on any similar problem. i.e, you will not overfit on your small dataset.

Moral of the story: you don't need a large dataset if you are working on real life images!

GPUs

GPUs dramatically speed up neural networks. This is because most of the neural network computation is just matrix multiplication and thus is readily vectorizable. GPUs excel at such computations.

For example, look at the times taken by the above transfer learning code to train:

CPU : ~ 20 min

GPU : ~ 1 min

And this was with meager dataset of 400 images. Imagine working with a dataset of the scale of ImageNet. GPUs are essential if you are serious about deep learning.

Other FAQ

What framework to use?

> Lot of frameworks are available: Keras, Tensorflow, PyTorch. I suggest using Keras if you are new to deep learning.

How do I know what architecture to use?

> Don't be a hero!

- Take whatever works best on ILSVRC (latest ResNet)
- Download a pretrained model
- Potentially add/delete some parts of it
- Finetune it on your application.

How do I know what hyperparameters to use?

Don't be a hero!

- Use whatever is reported to work best on ILSVRC.
- Play with the regularization strength (dropout rates)

But my model is not converging!

- Take a very small subset (like, 50 samples) of your dataset and train your network on this.
- Your network should completely overfit on this data. If not play with learning rates. If you couldn't get your network to overfit, something is either wrong with your code/initializations or you need to pick a more powerful model.