FROM: "LINEAR ALGEBRA AND LEARNING FROM DATA"
BY GILBERT STRANG,
WELLESLEY-CAMBRIDGE PRESS,
2019

# Part VII

# Learning from Data

VII.1 The Construction of Deep Neural Networks

VII.2 Convolutional Neural Nets

VII.3 Backpropagation and the Chain Rule

VII.4 Hyperparameters : The Fateful Decisions

VII.5 The World of Machine Learning

# Part VII : Learning from Data

This part of the book is a great adventure—hopefully for the reader, certainly for the author, and it involves the whole science of thought and intelligence. You could call it Machine Learning (ML) or Artificial Intelligence (AI). Human intelligence created it (but we don't fully understand what we have done). Out of some combination of ideas and failures, *attempting at first to imitate the neurons in the brain*, a successful approach has emerged to finding **patterns in data**.

What is important to understand about deep learning is that those data-fitting computations, of almost unprecedented size, are often heavily underdetermined. There are a great many points in the training data, but there are far more weights to be computed in a deep network. The art of deep learning is to find, among many possible solutions, one that will **generalize to new data.**

It is a remarkable observation that learning on deep neural nets with many weights leads to a successful tradeoff: $F$ is accurate on the training set *and* the unseen test set. This is the good outcome from minibatch gradient descent with momentum and the hyperparameters from Section VII.4 (including stepsize selection and early stopping).

This chapter is organized in an irregular order. **Deep learning comes first**. Earlier models like Support Vector Machines and Kernel Methods are briefly described in VII.5. The order is anhistorical, and the reader will know why. Neural nets have become the primary architecture for the most interesting (and the most difficult) problems of machine learning. That multi-layer architecture often succeeds, but by no means always! This book has been preparing for deep learning and we simply give it first place.

Sections VII.1-2 describe the learning function $F(\boldsymbol{x}, \boldsymbol{v})$ for *fully connected nets and convolutional nets*. The training data is given by a set of feature vectors $\boldsymbol{v}$. The weights that allow $F$ to classify that data are in the vector $\boldsymbol{x}$. To optimize $F$, gradient descent needs its derivatives $\partial F/\partial \boldsymbol{x}$. The weights $\boldsymbol{x}$ are the matrices $A_1, \ldots, A_L$ and bias vectors $\boldsymbol{b}_1, \ldots, \boldsymbol{b}_L$ that take the sample data $\boldsymbol{v} = \boldsymbol{v}_0$ to the output $\boldsymbol{w} = \boldsymbol{v}_L$.

Formulas for $\partial F/\partial A$ and $\partial F/\partial \boldsymbol{b}$ are not difficult. Those formulas are useful to see. But real codes use *automatic differentiation (AD) for backpropagation* (Section VII.3). Each hidden layer with its optimized weights learns more about the data and the population from which it comes—in order to classify new and unseen data from the same population.

## The Functions of Deep Learning

Suppose one of the digits $0, 1, \ldots, 9$ is drawn in a square. How does a person recognize which digit it is ? That neuroscience question is not answered here. How can a computer recognize which digit it is ? This is a machine learning question. Probably both answers begin with the same idea : *Learn from examples.*

So we start with $M$ different images (the training set). An image will be a set of $p$ small pixels—or a vector $\boldsymbol{v} = (v_1, \ldots, v_p)$. The component $v_i$ tells us the "grayscale" of the $i$th pixel in the image : how dark or light it is. So we have $M$ images each with $p$ features : $M$ vectors $\boldsymbol{v}$ in $p$-dimensional space. For every $\boldsymbol{v}$ in that training set we know the digit it represents.

In a way, we know a function. We have $M$ inputs in $\mathbf{R}^p$ each with an output from 0 to 9. But we don't have a "rule". We are helpless with a new input. Machine learning proposes to create a rule that succeeds on (most of) the training images. But "succeed" means much more than that : The rule should give the correct digit for a much wider set of test images, taken from the same population. This essential requirement is called *generalization.*

*What form shall the rule take* ? Here we meet the fundamental question. Our first answer might be : $F(\boldsymbol{v})$ could be a linear function from $\mathbf{R}^p$ to $\mathbf{R}^{10}$ (a 10 by $p$ matrix). The 10 outputs would be probabilities of the numbers 0 to 9. We would have $10p$ entries and $M$ training samples to get mostly right.

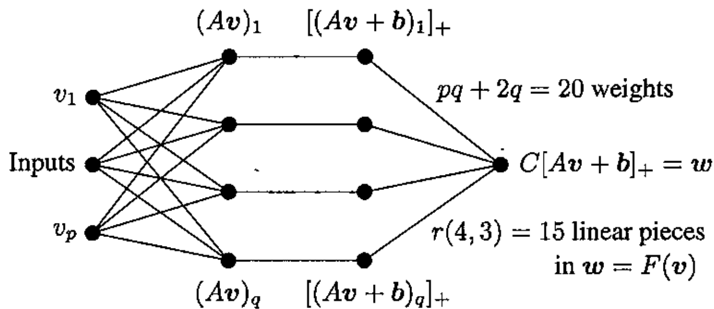
The difficulty is : Linearity is far too limited. Artistically, two zeros could make an 8. 1 and 0 could combine into a handwritten 9 or possibly 6. Images don't add. In recognizing faces instead of numbers, we will need a lot of pixels—and the input-output rule is nowhere near linear.

Artificial intelligence languished for a generation, waiting for new ideas. There is no claim that the absolutely best class of functions has now been found. That class needs to allow a great many parameters (called weights). And it must remain feasible to compute all those weights (in a reasonable time) from knowledge of the training set.

The choice that has succeeded beyond expectation—and has turned shallow learning into deep learning—is *Continuous Piecewise Linear (CPL) functions.* **Linear** for simplicity, **continuous** to model an unknown but reasonable rule, and **piecewise** to achieve the nonlinearity that is an absolute requirement for real images and data.

This leaves the crucial question of computability. What parameters will quickly describe a large family of CPL functions ? Linear finite elements start with a triangular mesh. But specifying many individual nodes in $\mathbf{R}^p$ is expensive. Much better if those nodes are the *intersections* of a smaller number of lines (or hyperplanes). Please know that a regular grid is too simple.

Here is a first construction of a piecewise linear function of the data vector $\boldsymbol{v}$. Choose a matrix $A_1$ and vector $\boldsymbol{b}_1$. Then set to zero (this is the nonlinear step) all negative components of $A_1\boldsymbol{v} + \boldsymbol{b}_1$. Then multiply by a matrix $A_2$ to produce 10 outputs in $\boldsymbol{w} = F(\boldsymbol{v}) = A_2(A_1\boldsymbol{v} + \boldsymbol{b}_1)_+$. That vector $(A_1\boldsymbol{v} + \boldsymbol{b}_1)_+$ forms a "hidden layer" between the input $\boldsymbol{v}$ and the output $\boldsymbol{w}$.




Actually the nonlinear function called ReLU $(x) = x_+ = \max(x, 0)$ was originally smoothed into a logistic curve like $1/(1+e^{-x})$. It was reasonable to think that continuous derivatives would help in optimizing the weights $A_1, b_1, A_2$. That proved to be wrong.

The graph of each component of $(A_1 v + b_1)_+$ has two halfplanes (one is flat, from the zeros where $A_1 v + b_1$ is negative). If $A_1$ is $q$ by $p$, the input space $\mathbf{R}^p$ is sliced by $q$ hyperplanes into $r$ pieces. We can count those pieces! This measures the "expressivity" of the overall function $F(v)$. The formula from combinatorics uses the binomial coefficients (see Section VII.1):

$$r(q,p) = \binom{q}{0} + \binom{q}{1} + \cdots + \binom{q}{p}$$

This number gives an impression of the graph of $F$. But our function is not yet sufficiently expressive, and one more idea is needed.

Here is the indispensable ingredient in the learning function $F$. The best way to create complex functions from simple functions is by **composition**. Each $F_i$ is linear (or affine) followed by the nonlinear ReLU : $F_i(v) = (A_i v + b_i)_+$. Their composition is $F(v) = F_L(F_{L-1}(\ldots F_2(F_1(v))))$. We now have $L-1$ hidden layers before the final output layer. The network becomes deeper as $L$ increases. That depth can grow quickly for convolutional nets (with banded Toeplitz matrices $A$).

The great optimization problem of deep learning is to compute weights $A_i$ and $b_i$ that will make the outputs $F(v)$ nearly correct—close to the digit $w(v)$ that the image $v$ represents. This problem of minimizing some measure of $F(v) - w(v)$ is solved by following a gradient downhill. The gradient of this complicated function is computed by *backpropagation*—the workhorse of deep learning that executes the chain rule.

A historic competition in 2012 was to identify the 1.2 million images collected in ImageNet. The breakthrough neural network in AlexNet had 60 million weights. Its accuracy (after 5 days of stochastic gradient descent) cut in half the next best error rate. Deep learning had arrived.

Our goal here was to identify continuous piecewise linear functions as powerful approximators. That family is also convenient—closed under addition and maximization and composition. The magic is that the learning function $F(A_i, b_i, v)$ gives accurate results on images $v$ that $F$ has never seen.

*This two-page essay was written for SIAM News* (December 2018).

## Bias vs. Variance : Underfit vs. Overfit

A **training set** contains $N$ vectors $v_1, \ldots, v_N$ with $m$ components each (the $m$ features of each sample). For each of those $N$ points in $\mathbf{R}^m$, we are given a value $y_i$. We assume there is an unknown function $f(x)$ so that $y_i = f(x_i) + \epsilon_i$, where the noise $\epsilon$ has zero mean and variance $\sigma^2$. That is the function $f(x)$ that our algorithms try to learn.

Our learning algorithm actually finds a function $F(x)$ close to $f(x)$. For example, $F$ from our learning algorithm could be linear (not that great) or piecewise linear (much better) – this depends on the algorithm we use. We fervently hope that $F(x)$ will be close to the correct $f(x)$ **not only on the training samples but also for later test samples.**

The warning is often repeated, and always the same: **Don't overfit the data**. The option is there, to reproduce all known observations. It is more important to prevent large swings in the learning function (which is built from the weights). This function is going to be applied to new data. Implicitly or explicitly, we need to **regularize** this function $F$.

Ordinarily, we regularize by adding a penalty term like $\lambda||x||$ to the function that we are minimizing. This gives a smoother and more stable solution as the minimum point. For deep learning problems this isn't always necessary! We don't fully understand why steepest descent or stochastic steepest descent will find a near minimum that generalizes well to unseen test data—with no penalty term. Perhaps the success comes from following this rule: *Stop the minimization early before you overfit.*

If $F$ does poorly on the training samples with large error **(bias)**, that is *underfitting*

If $F$ does well on the training samples but not well on test samples, that is *overfitting*.

*This is the* **bias-variance tradeoff**. *High bias from underfitting, high variance from* overfitting. Suppose we scale $f$ and $F$ so that $\mathrm{E}[F(x)] = 1$.

$$\textbf{Bias} = \mathrm{E}[f(x) - F(x)] \qquad \textbf{Variance} = \mathrm{E}[(F(x))^2] - (\mathrm{E}[F(x)])^2$$

We are forced into this tradeoff by the following identity for $(\text{Bias})^2 + (\text{Variance}) + (\text{Noise})^2$:

$$\mathrm{E}[(y - F(x))^2] = (\mathrm{E}[f(x) - F(x)])^2 + \mathrm{E}[(F(x))^2] - (\mathrm{E}[F(x)])^2 + \mathrm{E}[(y - f(x))^2]$$

Again, *bias* comes from allowing less freedom and using fewer parameters (weights). *Variance* is large when we provide too much freedom and too many parameters for $F$. Then the learned function $F$ can be super-accurate on the training set but out of control on an unseen test set. **Overfitting produces an $F$ that does not generalize.**

Here are links to six sites that support codes for machine learning:

| | |
|---|---|
| Caffe : arXiv:1408.5093 | Keras : http://keras.io/ |
| MatConvNet : www.vlfeat.org/matconvnet | Theano : arXiv : 1605.02688 |
| Torch : torch.ch | TensorFlow : www.tensorflow.org |

# VII.1 The Construction of Deep Neural Networks

Deep neural networks have evolved into a major force in machine learning. Step by step, the structure of the network has become more resilient and powerful—and more easily adapted to new applications. One way to begin is to describe essential pieces in the structure. Those pieces come together into a **learning function** $F(x, v)$ *with weights* $x$ *that capture information from the training data* $v$—to prepare for use with new test data.

Here are important steps in creating that function $F$:

| | | |
|---|---|---|
| 1 | Key operation | **Composition** $F = F_3(F_2(F_1(x, v)))$ |
| 2 | Key rule | **Chain rule for $x$-derivatives of $F$** |
| 3 | Key algorithm | **Stochastic gradient descent to find the best weights $x$** |
| 4 | Key subroutine | **Backpropagation to execute the chain rule** |
| 5 | Key nonlinearity | **ReLU**$(y)$ = **max**$(y, 0)$ = **ramp function** |

Our first step is to describe the pieces $F_1, F_2, F_3, \ldots$ for one layer of neurons at a time. The weights $x$ that connect the layers $v$ are optimized in creating $F$. The vector $v = v_0$ comes from the training set, and the function $F_k$ produces the vector $v_k$ at layer $k$. The whole success is to build the power of $F$ from those pieces $F_k$ in equation (1).

### $F_k$ is a Piecewise Linear Function of $v_{k-1}$

The input to $F_k$ is a vector $v_{k-1}$ of length $N_{k-1}$. The output is a vector $v_k$ of length $N_k$, ready for input to $F_{k+1}$. This function $F_k$ has two parts, first linear and then nonlinear:

1. The linear part of $F_k$ yields $A_k v_{k-1} + b_k$ (that bias vector $b_k$ makes this "affine")

2. A fixed nonlinear function like ReLU is applied to *each component* of $A_k v_{k-1} + b_k$

$$\boxed{v_k = F_k(v_{k-1}) = \text{ReLU}\,(A_k v_{k-1} + b_k)} \quad (1)$$

The training data for each sample is in a feature vector $v_0$. The matrix $A_k$ has shape $N_k$ by $N_{k-1}$. The column vector $b_k$ has $N_k$ components. **These $A_k$ and $b_k$ are weights** constructed by the optimization algorithm. Frequently *stochastic gradient descent* computes optimal weights $x = (A_1, b_1, \ldots, A_L, b_L)$ in the central computation of deep learning. It relies on backpropagation to find the $x$-derivatives of $F$, to solve $\nabla F = 0$.

The activation function ReLU$(y)$ = max$(y, 0)$ gives flexibility and adaptability. Linear steps alone were of limited power and ultimately they were unsuccessful.

ReLU is applied to every "neuron" in every internal layer. There are $N_k$ neurons in layer $k$, containing the $N_k$ outputs from $A_k v_{k-1} + b_k$. Notice that ReLU itself is continuous and piecewise linear, as its graph shows. (The graph is just a ramp with slopes 0 and 1. Its derivative is the usual step function.) When we choose ReLU, the composite function $F = F_L(F_2(F_1(x, v)))$ has an important and attractive property:

**The learning function $F$ is continuous and piecewise linear in $v$.**

## One Internal Layer ($L = 2$)

Suppose we have measured $m = 3$ features of one sample point in the training set. Those features are the 3 components of the input vector $\boldsymbol{v} = \boldsymbol{v}_0$. Then the first function $F_1$ in the chain multiplies $\boldsymbol{v}_0$ by a matrix $A_1$ and adds an offset vector $\boldsymbol{b}_1$ (bias vector). If $A_1$ is 4 by 3 and the vector $\boldsymbol{b}_1$ is 4 by 1, we have 4 components of $A_0\boldsymbol{v}_0 + \boldsymbol{b}_0$.

That step found 4 combinations of the 3 original features in $\boldsymbol{v} = \boldsymbol{v}_0$. The 12 weights in the matrix $A_1$ were optimized over many feature vectors $\boldsymbol{v}_0$ in the training set, to choose a 4 by 3 matrix (and a 4 by 1 bias vector) that would find 4 insightful combinations.

The final step to reach $\boldsymbol{v}_1$ is to apply the nonlinear "activation function" to each of the 4 components of $A_1\boldsymbol{v}_0 + \boldsymbol{b}_1$. Historically, the graph of that nonlinear function was often given by a smooth "*S-curve*". Particular choices then and now are in Figure VII.1.

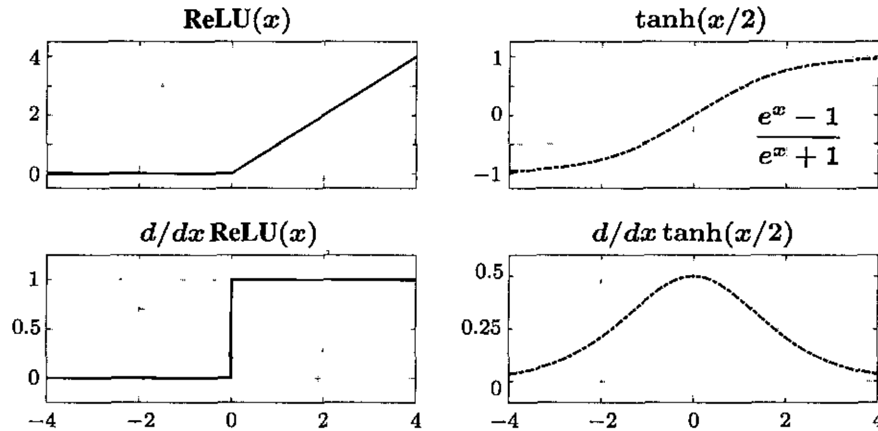



Figure VII.1: The **Rectified Linear Unit** and a sigmoid option for nonlinearity.

Previously it was thought that a sudden change of slope would be dangerous and possibly unstable. But large scale numerical experiments indicated otherwise! A better result was achieved by the **ramp function** $\text{ReLU}(y) = \max(y, 0)$. We will work with ReLU:

$$\textbf{Substitute } A_1\boldsymbol{v}_0 + \boldsymbol{b}_1 \textbf{ into ReLU to find } \boldsymbol{v}_1 \qquad (\boldsymbol{v}_1)_k = \max((A_1\boldsymbol{v}_0 + \boldsymbol{b}_1)_k, 0). \tag{2}$$

Now we have the components of $\boldsymbol{v}_1$ at the four "neurons" in layer 1. The input layer held the three components of this particular sample of training data. We may have thousands or millions of samples. The optimization algorithm found $A_1$ and $\boldsymbol{b}_1$, possibly by stochastic gradient descent using backpropagation to compute gradients of the overall loss.

Suppose our neural net is shallow instead of deep. It only has this first layer of 4 neurons. Then the final step will multiply the 4-component vector $\boldsymbol{v}_1$ by a 1 by 4 matrix $A_2$ (a row vector). It can add a single number $\boldsymbol{b}_2$ to reach the value $\boldsymbol{v}_2 = A_2\boldsymbol{v}_1 + \boldsymbol{b}_2$. The nonlinear function ReLU is not applied to the output.

$$\begin{array}{l} \text{Overall we compute } \boldsymbol{v}_2 = F(\boldsymbol{x}, \boldsymbol{v}_0) \text{ for each feature vector } \boldsymbol{v}_0 \text{ in the training set.} \\ \text{The steps are } \boldsymbol{v}_2 = A_2\boldsymbol{v}_1 + \boldsymbol{b}_2 = A_2\left(\text{ReLU}\left(A_1\boldsymbol{v}_0 + \boldsymbol{b}_1\right)\right) + \boldsymbol{b}_2 = F(\boldsymbol{x}, \boldsymbol{v}_0). \end{array} \tag{3}$$

The goal in optimizing $x = A_1, b_1, A_2, b_2$ is that the output values $v_\ell = v_2$ at the last layer $\ell = 2$ should correctly capture the important features of the training data $v_0$.

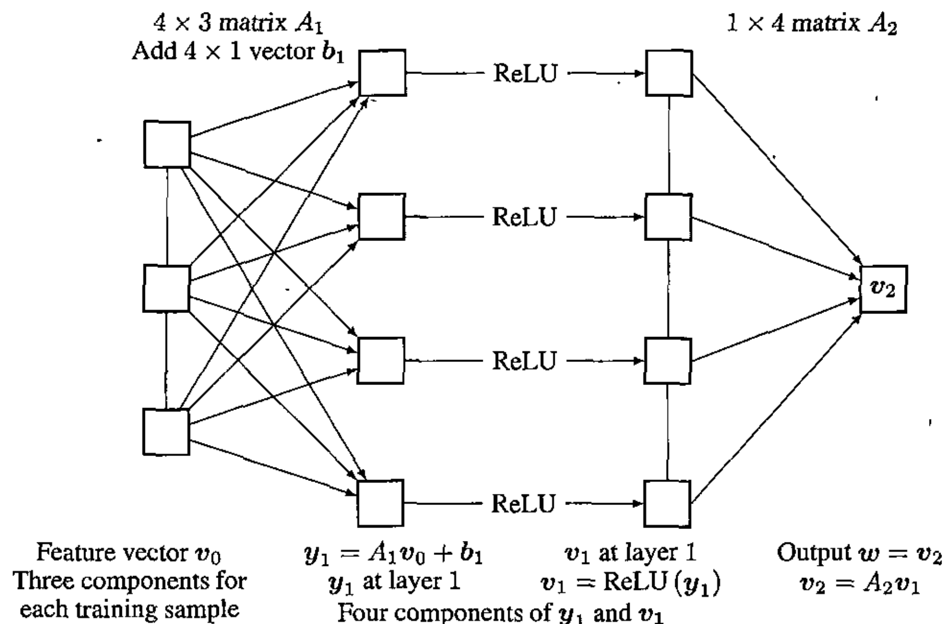



Figure VII.2: A feed-forward neural net with 4 neurons on **one internal layer.** The output $v_2$ (plus or minus) classifies the input $v_0$ (dog or cat). Then $v_2$ is a composite measure of the 3-component feature vector $v_0$. This net has 20 weights in $A_k$ and $b_k$.

> For a **classification problem** each sample $v_0$ of the training data is assigned **1 or −1**. We want the output $v_2$ to have that correct sign (most of the time). For a **regression problem** we use the numerical value (**not just the sign**) of $v_2$. We do not choose enough weights $A_k$ and $b_k$ to get every sample correct. And we do *not* necessarily want to ! That would probably be ***overfitting the training data.*** It could give erratic results when $F$ is applied to new and unknown test data.
>
> Depending on our choice of loss function $L(x, v_2)$ to minimize, this problem can be like least squares or entropy minimization. We are choosing $x$ = weight matrices $A_k$ and bias vectors $b_k$ to minimize $L$. Those two loss functions—square loss and cross-entropy loss—are compared in Section VII.4.

Our hope is that **the function $F$ has "learned" the data.** This is machine learning. We don't want to choose so many weights in $x$ that every input sample is sure to be correctly classified. *That is not learning.* That is simply fitting (overfitting) the data.

We want a balance where the function $F$ has learned what is important in recognizing *dog versus cat—or identifying an oncoming car versus a turning car.*

Machine learning doesn't aim to capture every detail of the numbers $0, 1, 2 \ldots, 9$. It just aims to capture enough information to decide correctly *which number it is.*

## The Initial Weights $x_0$ in Gradient Descent

The architecture in a neural net decides the form of the learning function $F(x, v)$. The training data goes into $v$. Then we *initialize* the weights $x$ in the matrices $A$ and vectors $b$. From those initial weights $x_0$, the optimization algorithm (normally a form of gradient descent) computes weights $x_1$ and $x_2$ and onward, aiming to minimize the total loss.

The question is : *What weights* $x_0$ *to start with* ? Choosing $x_0 = 0$ would be a disaster. Poor initialization is an important cause of failure in deep learning. A proper choice of the net and the initial $x_0$ has random (and independent) weights that meet two requirements:

1. $x_0$ has a carefully chosen variance $\sigma^2$.
2. The hidden layers in the neural net have enough neurons (not too narrow).

Hanin and Rolnick show that the initial variance $\sigma^2$ controls the mean of the computed weights. The layer widths control the variance of the weights. The key point is this: *Many-layered depth can reduce the loss on the training set. But if* $\sigma^2$ *is wrong or width is sacrificed, then gradient descent can lose control of the weights. They can explode to infinity or implode to zero.*

The danger controlled by the variance $\sigma^2$ of $x_0$ is exponentially large or exponentially small weights. The good choice is $\sigma^2 = 2$/fan-in. The fan-in is the maximum number of inputs to neurons (Figure VII.2 has fan-in $= 4$ at the output). The initialization "He uniform" in Keras makes this choice of $\sigma^2$.

The danger from narrow hidden layers is exponentially large variance of $x$ for deep nets. If layer $j$ has $n_j$ neurons, the quantity to control is the sum of 1/(layer widths $n_j$).

Looking ahead, convolutional nets (ConvNets) and residual networks (ResNets) can be very deep. Exploding or vanishing weights is a constant danger. Ideas from physics (*mean field theory*) have become powerful tools to explain and also avoid these dangers. Pennington and coauthors proposed a way to stay on the edge between fast growth and decay, even for $10,000$ layers. A key is to use orthogonal transformations: Exactly as in matrix multiplication $Q_1Q_2Q_3$, orthogonality leaves the size unchanged.

For ConvNets, fan-in becomes the number of features times the kernel size (and not the full size of $A$). For ResNets, a correct $\sigma^2$ normally removes both dangers. Very deep networks can produce very impressive learning.

The key point: Deep learning can go wrong if it doesn't start right.

K. He, X.Zhang, S. Ren, and J. Sun, *Delving deep into rectifiers*, arXiv : 1502.01852.

B. Hanin and D. Rolnick, *How to start training : The effect of initialization and architecture*, arXiv : 1803.01719, 19 Jun 2018.

L. Xiao, Y. Bahri, J. Sohl-Dickstein, S. Schoenholz, and J. Pennington, *Dynamical isometry and a mean field theory of CNNs : How to train* $10,000$ *layers*, arXiv : 1806.05393, 2018.

## Stride and Subsampling

Those words represent two ways to achieve the same goal: *Reduce the dimension.* Suppose we start with a 1D signal of length 128. We want to filter that signal—multiply that vector by a weight matrix $A$. We also want to reduce the length to 64. Here are two ways to reach that goal.

**In two steps** Multiply the 128-component vector $\boldsymbol{v}$ by $A$, and then discard the odd-numbered components of the output. This is filtering followed by subsampling. The output is $(\downarrow 2)\, A\boldsymbol{v}$.

**In one step** Discard the odd-numbered rows of the matrix $A$. The new matrix $A_2$ becomes short and wide: 64 rows and 128 columns. The "**stride**" of the filter is now 2. Now multiply the 128-component vector $\boldsymbol{v}$ by $A_2$. Then $A_2\boldsymbol{v}$ is the same as $(\downarrow 2)\, A\boldsymbol{v}$. A stride of 3 would keep every third component.

Certainly the one-step striding method is more efficient. If the stride is 4, the dimension is divided by 4. In two dimensions (for images) it is reduced by 16.

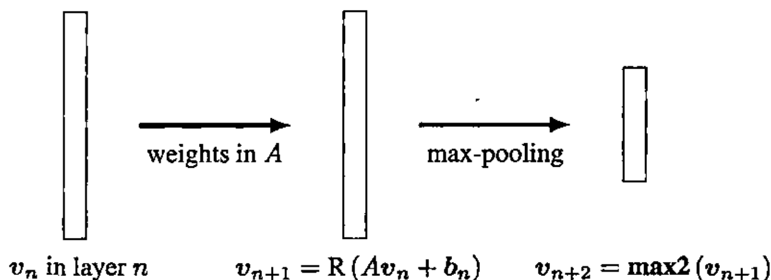
The two-step method makes clear that half or three-fourths of the information is lost. Here is a way to reduce the dimension from 128 to 64 as before, but to run less risk of destroying important information: *Max-pooling.*

## Max-pooling

Multiply the 128-component vector $\boldsymbol{v}$ by $A$, as before. Then from each even-odd pair of outputs like $(A\boldsymbol{v})_2$ and $(A\boldsymbol{v})_3$, *keep the maximum.* Please notice right away: Max-pooling is simple and fast, **but taking the maximum is not a linear operation.** It is a sensible route to dimension reduction, pure and simple.

For an image (a 2-dimensional signal) we might use max-pooling over every 2 by 2 square of pixels. Each dimension is reduced by 2, The image dimension is reduced by 4. This speeds up the training, when the number of neurons on a hidden layer is divided by 4.

Normally a max-pooling step is given its own separate place in the overall architecture of the neural net. Thus a part of that architecture might look like this:




$\boldsymbol{v}_n$ in layer $n$ $\quad \boldsymbol{v}_{n+1} = \mathrm{R}\,(A\boldsymbol{v}_n + \boldsymbol{b}_n) \quad \boldsymbol{v}_{n+2} = \mathbf{max2}\,(\boldsymbol{v}_{n+1})$

Dimension reduction has another important advantage, in addition to reducing the computation. *Pooling also reduces the possibility of overfitting.* Average pooling would keep the *average* of the numbers in each pool: now the pooling layer is linear.

## The Graph of the Learning Function $F(v)$

The graph of $F(v)$ is a surface made up of many, many flat pieces—they are planes or hyperplanes that fit together along all the folds where ReLU produced a change of slope. This is like origami except that this graph has flat pieces going to infinity. And the graph might not be in $\mathbf{R}^3$—the feature vector $v = v_0$ has $N_0 = m$ components.

Part of the *mathematics of deep learning* is to estimate the number of flat pieces and to visualize how they fit into one piecewise linear surface. That estimate comes after an example of a neural net with one internal layer. Each feature vector $v_0$ contains $m$ measurements like height, weight, age of a sample in the training set.

In the example, $F$ had three inputs in $v_0$ and one output $v_2$. Its graph will be a piecewise flat surface in 4-dimensional space. The height of the graph is $v_2 = F(v_0)$, over the point $v_0$ in 3-dimensional space. Limitations of space in the book (and severe limitations of imagination in the author) prevent us from drawing that graph in $\mathbf{R}^4$. Nevertheless we can try to count the flat pieces, based on 3 inputs and 4 neurons and 1 output.

*Note* 1 With only $m = 2$ inputs (2 features for each training sample) the graph of $F$ is a surface in 3D. We can and will make an attempt to describe it.

*Note* 2 You actually see points on the graph of $F$ when you run examples on **playground.tensorflow.org**. This is a very instructive website.

That website offers four options for the training set of points $v_0$. You choose the number of layers and neurons. Please choose the ReLU activation function ! Then the program counts epochs as gradient descent optimizes the weights. (An *epoch* sees all samples on average once.) If you have allowed enough layers and neurons to correctly classify the blue and orange training samples, you will see a polygon separating them. **That polygon shows where $F = 0$.** It is the cross-section of the graph of $z = F(v)$ at height $z = 0$.

That polygon separating blue from orange (or *plus* from *minus*: this is classification) is the analog of a separating hyperplane in a Support Vector Machine. If we were limited to linear functions and a straight line between a blue ball and an orange ring around it, separation would be impossible. But for the deep learning function $F$ this is not difficult...

We will discuss experiments on this **playground.tensorflow** site in the Problem Set.

**Important Note: Fully Connected versus Convolutional**

We don't want to mislead the reader. Those "fully connected" nets are often not the most effective. If the weights around one pixel in an image can be repeated around all pixels (why not ?), then one row of $A$ is all we need. The row can assign zero weights to faraway pixels. Local **convolutional neural nets (CNN's)** are the subject of Section VII.2.

You will see that the count grows exponentially with the number of neurons and layers. That is a useful insight into the power of deep learning. We badly need insight because the size and depth of the neural network make it difficult to visualize in full detail.

## Counting Flat Pieces in the Graph : One Internal Layer

It is easy to count entries in the weight matrices $A_k$ and the bias vectors $b_k$. Those numbers determine the function $F$. But it is far more interesting to count the number of flat pieces in the graph of $F$. This number measures the **expressivity** of the neural network. $F(x, v)$ is a more complicated function than we fully understand (at least so far). The system is deciding and acting on its own, without explicit approval of its "thinking". For driverless cars we will see the consequences fairly soon.

Suppose $v_0$ has $m$ components and $A_1 v_0 + b_1$ has $N$ components. We have $N$ functions of $v_0$. Each of those linear functions is zero along a hyperplane (dimension $m - 1$) in $\mathbf{R}^m$. When we apply ReLU to that linear function it becomes piecewise linear, with a fold along that hyperplane. On one side of the fold its graph is sloping, on the other side the function changes from negative to zero.

Then the next matrix $A_2$ combines those $N$ piecewise linear functions of $v_0$, so we now have folds along *$N$ different hyperplanes in $\mathbf{R}^m$*. This describes each piecewise linear component of the next layer $A_2(\text{ReLU}(A_1 v_0 + b_1))$ in the typical case.

You could think of $N$ straight folds in the plane (the folds are actually along $N$ hyperplanes in $m$-dimensional space). The first fold separates the plane in two pieces. The next fold from ReLU will leave us with four pieces. The third fold is more difficult to visualize, but the following figure shows that there are seven (*not eight*) pieces.

In combinatorial theory, we have a **hyperplane arrangement**—and a theorem of Tom Zaslavsky counts the pieces. The proof is presented in Richard Stanley's great textbook on *Enumerative Combinatorics* (2001). But that theorem is more complicated than we need, because it allows the fold lines to meet in all possible ways. Our task is simpler because we assume that the fold lines are in "general position"—$m + 1$ folds don't meet. For this case we now apply the neat counting argument given by Raghu, Poole, Kleinberg, Gangul, and Dickstein : *On the Expressive Power of Deep Neural Networks*, arXiv : 1606.05336v6 : See also *The Number of Response Regions* by Pascanu, Montufar, and Bengio on arXiv 1312.6098.

**Theorem** For $v$ in $\mathbf{R}^m$, suppose the graph of $F(v)$ has folds along $N$ hyperplanes $H_1, \ldots, H_N$. Those come from $N$ linear equations $a_i^{\mathrm{T}} v + b_i = 0$, in other words from ReLU at $N$ neurons. Then the number of linear pieces of $F$ and regions bounded by the $N$ hyperplanes is $r(N, m)$ :

$$r(N,m) = \sum_{i=0}^{m} \binom{N}{i} = \binom{N}{0} + \binom{N}{1} + \cdots + \binom{N}{m}. \qquad (4)$$

These binomial coefficients are

$$\binom{N}{i} = \frac{N!}{i!(N-i)!} \quad \text{with } 0! = 1 \text{ and } \binom{N}{0} = 1 \text{ and } \binom{N}{i} = 0 \text{ for } i > N.$$

**Example** The function $F(x, y, z) = \text{ReLU}(x) + \text{ReLU}(y) + \text{ReLU}(z)$ has 3 folds along the 3 planes $x = 0, y = 0, z = 0$. Those planes divide $\mathbf{R}^3$ into $r(3, 3) = 8$ pieces where $F = x + y + z$ and $x + z$ and $x$ and $0$ (and 4 more). Adding $\text{ReLU}(x + y + z - 1)$ gives a fourth fold and $r(4, 3) = 15$ pieces of $\mathbf{R}^3$. Not 16 because the new fold plane $x + y + z = 1$ does not meet the 8th original piece where $x < 0, y < 0, z < 0$.

George Polya's famous YouTube video *Let Us Teach Guessing* cut a cake by 5 planes. He helps the class to find $r(5,3) = 26$ pieces. Formula (4) allows $m$-dimensional cakes.

One hyperplane in $\mathbf{R}^m$ produces $\binom{1}{0} + \binom{1}{1} = 2$ regions. And $N = 2$ hyperplanes will produce $r(2,m) = 1 + 2 + 1 = 4$ regions provided $m > 1$. When $m = 1$ we have two folds in a line, which only separates the line into $r(2,1) = 3$ pieces.

The count $r$ of linear pieces will follow from the recursive formula

$$\boxed{r(N,m) = r(N-1,m) + r(N-1,m-1).} \tag{5}$$

To understand that recursion, start with $N-1$ hyperplanes in $\mathbf{R}^m$ and $r(N-1,m)$ regions. *Add one more hyperplane* $H$ (dimension $m-1$). The established $N-1$ hyperplanes cut $H$ into $r(N-1,m-1)$ regions. Each of those pieces of $H$ divides one existing region into two, adding $r(N-1,m-1)$ regions to the original $r(N-1,m)$; see Figure VII.3. So the recursion is correct, and we now apply equation (5) to compute $r(N,m)$.

The count starts at $r(1,0) = r(0,1) = 1$. Then (4) is proved by induction on $N+m$:

$$\begin{aligned} r(N-1,m) + r(N-1,m-1) &= \sum_0^m \binom{N-1}{i} + \sum_0^{m-1} \binom{N-1}{i} \\ &= \binom{N-1}{0} + \sum_0^{m-1} \left[\binom{N-1}{i} + \binom{N-1}{i+1}\right] \\ &= \binom{N}{0} + \sum_0^{m-1} \binom{N}{i+1} = \sum_0^m \binom{N}{i}. \end{aligned} \tag{6}$$

The two terms in brackets (second line) became one term because of a useful identity:

$$\binom{N-1}{i} + \binom{N-1}{i+1} = \binom{N}{i+1} \quad \text{and the induction is complete.}$$

Mike Giles made that presentation clearer, and he suggested Figure VII.3 to show the effect of the last hyperplane $H$. There are $r = 2^N$ linear pieces of $F(v)$ for $N \le m$ and $r \approx N^m/m!$ pieces for $N >> m$, when the hidden layer has many neurons.

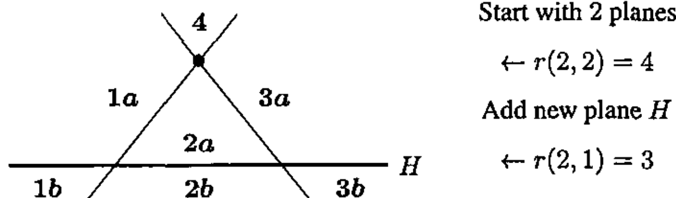



Figure VII.3: The $r(2,1) = 3$ pieces of $H$ create 3 new regions. Then the count becomes $r(3,2) = 4 + 3 = 7$ flat regions in the continuous piecewise linear surface $v_2 = F(v_0)$. A fourth fold will cross all 3 existing folds and create 4 new regions, so $r(4,2) = 11$.

## Flat Pieces of $F(v)$ with More Hidden Layers

Counting the linear pieces of $F(\boldsymbol{v})$ is much harder with 2 internal layers in the network. Again $\boldsymbol{v}_0$ and $\boldsymbol{v}_1$ have $m$ and $N_1$ components. Now $A_1\boldsymbol{v}_1 + \boldsymbol{b}_1$ will have $N_2$ components before ReLU. Each one is like the function $F$ for one layer, described above. Then application of ReLU will create new folds in its graph. Those folds are along the lines where a component of $A_1\boldsymbol{v}_1 + \boldsymbol{b}_1$ is zero.

Remember that each component of $A_1\boldsymbol{v}_1 + \boldsymbol{b}_1$ is piecewise linear, not linear. So it *crosses zero (if it does) along a piecewise linear surface, not a hyperplane.* The straight lines in Figure VII.3 for the folds in $\boldsymbol{v}_1$ will change to *piecewise straight lines* for the folds in $\boldsymbol{v}_2$. In $m$ dimensions they are connected pieces of hyperplanes. So the count becomes variable, depending on the details of $\boldsymbol{v}_0, A_1, \boldsymbol{b}_1, A_2$, and $\boldsymbol{b}_2$.

Still we can estimate the number of linear pieces. We have $N_2$ piecewise straight lines (or piecewise hyperplanes in $\mathbf{R}^m$) from $N_2$ ReLU's at the second hidden layer. If those lines were actually straight, we would have a total of $N_1 + N_2$ folds in each component of $\boldsymbol{v}_3 = F(\boldsymbol{v}_0)$. *Then the formula* (4) *to count the pieces would have* $N_1 + N_2$ *in place of* $N$. This is our estimate (open for improvement) with two layers between $\boldsymbol{v}_0$ and $\boldsymbol{v}_3$.

## Composition $F_3(F_2(F_1(v)))$

The word "composition" would simply represent "matrix multiplication" if all our functions were linear: $F_k(\boldsymbol{v}) = A_k\boldsymbol{v}$. Then $F(\boldsymbol{v}_0) = A_3A_2A_1\boldsymbol{v}_0$: just one matrix. For nonlinear $F_k$ the meaning is the same: Compute $\boldsymbol{v}_1 = F_1(\boldsymbol{v}_0)$, then $\boldsymbol{v}_2 = F_2(\boldsymbol{v}_1)$, and finally $\boldsymbol{v}_3 = F_3(\boldsymbol{v}_2)$. This operation of **composition** $\boldsymbol{F_3(F_2(F_1(v_0)))}$ is far more powerful in creating functions than addition!

For a neural network, composition produces continuous piecewise linear functions $F(\boldsymbol{v}_0)$. The 13th problem on Hilbert's list of 23 unsolved problems in 1900 asked a question about *all* continuous functions. A famous generalization of his question was this:

> Is every continuous function $F(x, y, z)$ of three variables the composition of continuous functions $G_1, \ldots, G_N$ of two variables? The answer is *yes*.

Hilbert seems to have expected the answer *no*. But a positive answer was given in 1957 by Vladimir Arnold (age 19). His teacher Andrey Kolmogorov had previously created multivariable functions out of 3-variable functions.

Related questions have negative answers. If $F(x, y, z)$ has continuous derivatives, it may be impossible for all the 2-variable functions to have continuous derivatives (Vitushkin). And to construct 2-variable continuous functions $F(x, y)$ as compositions of 1-variable continuous functions (the ultimate 13th problem) you must allow *addition*. The 2-variable functions $xy$ and $x^y$ use 1-variable functions exp, log, and log log:

$$xy = \exp(\log x + \log y) \quad \text{and} \quad x^y = \exp(\exp(\log y + \log\log x)). \tag{7}$$

So much to learn from the Web. A chapter of *Kolmogorov's Heritage in Mathematics* (Springer, 2007) connects these questions explicitly to neural networks.

*Is the answer to Hilbert still yes for continuous piecewise linear functions on* $\mathbf{R}^m$ ?

## Neural Nets Give Universal Approximation

The previous paragraphs wandered into the analysis of functions $f(v)$ of several variables. For deep learning a key question is the approximation of $f$ by a neural net—when the weights $x$ are chosen to bring $F(x, v)$ close to $f(v)$.

There is a qualitative question and also a quantitative question:

1 For any continuous function $f(v)$ with $v$ in a cube in $\mathbf{R}^d$, can a net with enough layers and neurons and weights $x$ give uniform approximation to $f$ within any desired accuracy $\epsilon > 0$? This property is called **universality**.

$$\boxed{\textbf{If } f(v) \textbf{ is continuous there exists } x \textbf{ so that } |F(x,v) - f(v)| < \epsilon \textbf{ for all } v.} \quad (8)$$

2 If $f(v)$ belongs to a normed space $S$ of smooth functions, how quickly does the approximation error improve as the net has more weights?

$$\boxed{\textbf{Accuracy of approximation to } f \qquad \min_{x} \|F(x,v) - f(v)\| \le C\|f\|_S} \quad (9)$$

Function spaces $S$ often use the $L^2$ or $L^1$ or $L^\infty$ norm of the function $f$ and its partial derivatives up to order $r$. Functional analysis gives those spaces a meaning even for non-integer $r$. $C$ usually decreases as the smoothness parameter $r$ is increased. For continuous piecewise linear approximation over a uniform grid with meshwidth $h$ we often find $C = O(h^2)$.

The response to Question 1 is *yes*. Wikipedia notes that one hidden layer (with enough neurons!) is sufficient for approximation within $\epsilon$. The 1989 proof by George Cybenko used a sigmoid function rather than ReLU, and the theorem is continually being extended. Ding-Xuan Zhou proved that we can require the $A_k$ to be convolution matrices (the structure becomes a CNN). Convolutions have many fewer weights than arbitrary matrices—and universality allows many convolutions.

The response to Question **2** by Mhaskar, Liao, and Poggio begins with the degree of approximation to functions $f(v_1, \ldots, v_d)$ with continuous derivatives of order $r$. *For $n$ weights the usual error bound is $Cn^{-r/d}$.* The novelty is their introduction of **composite functions** built from 2-variable functions, as in $f(v_1, v_2, v_3, v_4) = f_3(f_1(v_1, v_2), f_2(v_3, v_4))$. For a composite function, the approximation by a hierarchical net is much more accurate. *The error bound becomes $Cn^{-r/2}$.*

The proof applies the standard result for $d = 2$ variables to each function $f_1, f_2, f_3$. A difference of composite functions is a composite of 2-variable differences.

1 G. Cybenko, Approximation by superpositions of a sigmoidal function, *Mathematics of Control, Signals, and Systems* 7 (1989) 303-314.

2 K. Hornik, Approximation capabilities of multilayer feedforward networks, *Neural Networks* 4 (1991) 251-257.

3 H. Mhaskar, Q. Liao, and T. Poggio, *Learning functions: When is deep better than shallow*, arXiv: 01603.00988v4, 29 May 2016.

4 D.-X. Zhou, *Universality of deep convolutional neural networks*, arXiv : 1805. 10769, 20 Jul 2018.

5 D. Rolnick and M. Tegmark, *The power of deeper networks for expressing natural functions*, arXiv : 1705.05502, 27 Apr 2018.

## Problem Set VII.1

**1** In the example $F = \text{ReLU}(x) + \text{ReLU}(y) + \text{ReLU}(z)$ that follows formula (4) for $r(N, m)$, suppose the 4th fold comes from $\text{ReLU}(x + y + z)$. Its fold plane $x + y + z = 0$ now meets the 3 original fold planes $x = 0, y = 0, z = 0$ at a single point $(0,0,0)$—*an exceptional case*. Describe the 16 (not 15) linear pieces of $F =$ sum of these four ReLU's.

**2** Suppose we have $m = 2$ inputs and $N$ neurons on a hidden layer, so $F(x, y)$ is a linear combination of $N$ ReLU's. Write out the formula for $r(N, 2)$ to show that the count of linear pieces of $F$ has leading term $\frac{1}{2}N^2$.

**3** Suppose we have $N = 18$ lines in a plane. If 9 are vertical and 9 are horizontal, how many pieces of the plane? Compare with $r(18, 2)$ when the lines are in general position and no three lines meet.

**4** What weight matrix $A_1$ and bias vector $b_1$ will produce $\text{ReLU}(x + 2y - 4)$ and $\text{ReLU}(3x - y + 1)$ and $\text{ReLU}(2x + 5y - 6)$ as the $N = 3$ components of the first hidden layer? (The input layer has 2 components $x$ and $y$.) If the output $w$ is the sum of those three ReLU's, how many pieces in $w(x, y)$?

**5** Folding a line four times gives $r(4, 1) = 5$ pieces. Folding a plane four times gives $r(4; 2) = 11$ pieces. According to formula (4), how many flat subsets come from folding $\mathbf{R}^3$ four times? The flat subsets of $\mathbf{R}^3$ meet at 2D planes (like a door frame).

**6** The binomial theorem finds the coefficients $\binom{N}{k}$ in $(a + b)^N = \sum_0^N \binom{N}{k} a^k b^{N-k}$.

For $a = b = 1$ what does this reveal about those coefficients and $r(N, m)$ for $m \geq N$?

**7** In Figure VII.3, one more fold will produce 11 flat pieces in the graph of $z = F(x, y)$. Check that formula (4) gives $r(4, 2) = 11$. How many pieces after five folds?

**8** Explain with words or show with graphs why each of these statements about Continuous Piecewise Linear functions (CPL functions) is true:

| | |
|---|---|
| $M$ | The maximum $M(x, y)$ of two CPL functions $F_1(x, y)$ and $F_2(x, y)$ is CPL. |
| $S$ | The sum $S(x, y)$ of two CPL functions $F_1(x, y)$ and $F_2(x, y)$ is CPL. |
| $C$ | If the one-variable functions $y = F_1(x)$ and $z = F_2(y)$ are CPL, so is the composition $C(x) = z = (F_2(F_1(x))$. |

**9** How many weights and biases are in a network with $m = N_0 = 4$ inputs in each feature vector $\boldsymbol{v}_0$ and $N = 6$ neurons on each of the 3 hidden layers? How many activation functions (ReLU) are in this network, before the final output?

**10** (Experimental) In a neural network with two internal layers and a total of 10 neurons, should you put more of those neurons in layer **1** or layer **2**?

**Problems 11–13 use the blue ball, orange ring example on playground.tensorflow.org with one hidden layer and activation by ReLU (not Tanh). When learning succeeds, a white polygon separates blue from orange in the figure that follows.**

**11** Does learning succeed for $N = 4$? What is the count $r(N, 2)$ of flat pieces in $F(\boldsymbol{x})$? The white polygon shows where flat pieces in the graph of $F(\boldsymbol{x})$ change sign as they go through the base plane $z = 0$. How many sides in the polygon?

**12** Reduce to $N = 3$ neurons in one layer. Does $F$ still classify blue and orange correctly? How many flat pieces $r(3, 2)$ in the graph of $F(\boldsymbol{v})$ and how many sides in the separating polygon?

**13** Reduce further to $N = 2$ neurons in one layer. Does learning still succeed? What is the count $r(2, 2)$ of flat pieces? How many folds in the graph of $F(\boldsymbol{v})$? How many sides in the white separator?

**14** Example 2 has blue and orange in two quadrants each. With one layer, do $N = 3$ neurons and even $N = 2$ neurons classify that training data correctly? How many flat pieces are needed for success? Describe the unusual graph of $F(\boldsymbol{v})$ when $N = 2$.

**15** Example 4 with blue and orange spirals is much more difficult! With one hidden layer, can the network learn this training data? Describe the results as $N$ increases.

**16** Try that difficult example with two hidden layers. Start with $4 + 4$ and $6 + 2$ and $2 + 6$ neurons. Is $2 + 6$ better or worse or more unusual than $6 + 2$?

**17** How many neurons bring complete separation of the spirals with two hidden layers? Can three layers succeed with fewer neurons than two layers?

I found that $4 + 4 + 2$ and $4 + 4 + 4$ neurons give very unstable iterations for that spiral graph. There were spikes in the training loss until the algorithm stopped trying. *playground.tensorflow.org (on our back cover!) was a gift from Daniel Smilkov.*

**18** What is the smallest number of pieces that 20 fold lines can produce in a plane?

**19** How many pieces are produced from 10 vertical and 10 horizontal folds?

**20** What is the maximum number of pieces from 20 fold lines in a plane?

# VII.2 Convolutional Neural Nets

This section is about networks with a different architecture. Up to now, each layer was fully connected to the next layer. If one layer had $n$ neurons and the next layer had $m$ neurons, then the matrix $A$ connecting those layers is $m$ by $n$. *There were $mn$ independent weights in $A$.* The weights from all layers were chosen to give a final output that matched the training data. The derivatives needed in that optimization were computed by backpropagation. Now we might have only 3 or 9 independent weights per layer.

That fully connected net will be extremely inefficient for image recognition. First, the weight matrices $A$ will be huge. If one image has 200 by 300 pixels, then its input layer has $60,000$ components. The weight matrix $A_1$ for the first hidden layer has $60,000$ columns. The problem is: We are looking for connections between faraway pixels. Almost always, the important connections in an image are **local**.

Text and music have a 1D local structure: a time series

Images have a 2D local structure: 3 copies for red-green-blue

Video has a 3D local structure: Images in a time series

More than this, the search for structure is essentially the same everywhere in the image. There is normally no reason to process one part of a text or image or video differently from other parts. We can use the same weights in all parts: *Share the weights.* The neural net of local connections between pixels is **shift-invariant**: the same everywhere.

The result is a big reduction in the number of independent weights. Suppose each neuron is connected to only $E$ neurons on the next layer, and those connections are the same for all neurons. Then the matrix $A$ between those layers has only $E$ independent weights $x$. The optimization of those weights becomes enormously faster. In reality we have time to create several different channels with their own $E$ or $E^2$ weights. They can look for edges in different directions (horizontal, vertical, and diagonal).

In one dimension, a banded shift-invariant matrix is a **Toeplitz matrix** or a **filter**. Multiplication by that matrix $A$ is a **convolution** $x * v$. The network of connections between all layers is a **Convolutional Neural Net** (**CNN** or **ConvNet**). Here $E = 3$.

$$A = \begin{bmatrix} x_1 & x_0 & x_{-1} & 0 & 0 & 0 \\ 0 & x_1 & x_0 & x_{-1} & 0 & 0 \\ 0 & 0 & x_1 & x_0 & x_{-1} & 0 \\ 0 & 0 & 0 & x_1 & x_0 & x_{-1} \end{bmatrix} \qquad \begin{array}{c} v = (v_0, v_1, v_2, v_3, v_4, v_5) \\ y = Av = \quad (y_1, y_2, y_3, y_4) \\ N+2 \text{ inputs and } N \text{ outputs} \end{array}$$

It is valuable to see $A$ as *a combination of shift matrices* $\boldsymbol{L}, \boldsymbol{C}, \boldsymbol{R}$: *Left, Center, Right.*

$$\textbf{Each shift has a diagonal of 1's} \qquad A = x_1 L + x_0 C + x_{-1} R$$

Then the derivatives of $\boldsymbol{y} = \boldsymbol{Av} = \boldsymbol{x_1 Lv} + \boldsymbol{x_0 Cv} + \boldsymbol{x_{-1} Rv}$ are exceptionally simple:

$$\boxed{\frac{\partial y}{\partial x_1} = Lv \qquad \frac{\partial y}{\partial x_0} = Cv \qquad \frac{\partial y}{\partial x_{-1}} = Rv} \tag{1}$$

## Convolutions in Two Dimensions

When the input $\boldsymbol{v}$ is an image, the convolution with $\boldsymbol{x}$ becomes two-dimensional. The numbers $x_{-1}, x_0, x_1$ change to $E^2 = 3^2$ independent weights. The inputs $v_{ij}$ have two indices and $\boldsymbol{v}$ represents $(N+2)^2$ pixels. The outputs have only $N^2$ pixels unless we pad with zeros at the boundary. The 2D convolution $\boldsymbol{x} * \boldsymbol{v}$ is a *linear combination of* 9 *shifts.*

**Weights** $\begin{bmatrix} x_{11} & x_{01} & x_{-11} \\ x_{10} & x_{00} & x_{-10} \\ x_{1-1} & x_{0-1} & x_{-1-1} \end{bmatrix}$

**Input image** $v_{ij}$ $i, j$ from$(0,0)$ to$(N+1, N+1)$
**Output image** $y_{ij}$ $i, j$ from $(1,1)$ to $(N, N)$
**Shifts L, C, R, U, D** = Left, Center, Right, Up, Down

$$A = x_{11}LU + x_{01}CU + x_{-11}RU + x_{10}L + x_{00}C + x_{-10}R + x_{1-1}LD + x_{0-1}CD + x_{-1-1}RD$$

This expresses the convolution matrix $A$ as a combination of 9 shifts. The derivatives of the output $\boldsymbol{y} = A\boldsymbol{v}$ are again exceptionally simple. We use these nine derivatives to create the gradients $\nabla \boldsymbol{F}$ and $\nabla \boldsymbol{L}$ that are needed in stochastic gradient descent to improve the weights $\boldsymbol{x}_k$. The next iteration $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - s\nabla \boldsymbol{L}_k$ has weights that better match the correct outputs from the training data.

These nine derivatives of $\boldsymbol{y} = A\boldsymbol{v}$ are computed inside backpropagation:

$$\frac{\partial \boldsymbol{y}}{\partial x_{11}} = LU\boldsymbol{v} \quad \frac{\partial \boldsymbol{y}}{\partial x_{01}} = CU\boldsymbol{v} \quad \frac{\partial \boldsymbol{y}}{\partial x_{-11}} = RU\boldsymbol{v} \quad \ldots \quad \frac{\partial \boldsymbol{y}}{\partial x_{-1-1}} = RD\boldsymbol{v} \qquad (2)$$

CNN's can readily afford to have $\boldsymbol{B}$ **parallel channels** (and that number $B$ can vary as we go deeper into the net). The count of weights in $\boldsymbol{x}$ is so much reduced by weight sharing and weight locality, that we don't need and we can't expect one set of $E^2 = 9$ weights to do all the work of a convolutional net.

Let me highlight the operational meaning of convolution. In 1 dimension, the formal algebraic definition $y_j = \sum x_i v_{j-i} = \sum x_{j-k} v_k$ involves a "flip" of the $v$'s or the $x$'s. This is a source of confusion that we do not need. We look instead at left-right shifts $L$ and $R$ of the whole signal (in 1D) and also up-down shifts $U$ and $D$ in two dimensions. Each shift is a matrix with a diagonal full of 1's. That saves us from the complication of remembering flipped subscripts.

A convolution is a combination of shift matrices (producing a filter or Toeplitz matrix)

A cyclic convolution is a combination of cyclic shifts (producing a circulant matrix)

A continuous convolution is a continuous combination (an integral) of shifts

In deep learning, the coefficients in the combination will be the "weights" to be learned.

## Two-dimensional Convolutional Nets

Now we come to the real success of CNN's: **Image recognition.** ConvNets and deep learning have produced a small revolution in computer vision. The applications are to self-driving cars, drones, medical imaging, security, robotics—there is nowhere to stop. Our interest is in the algebra and geometry and intuition that makes all this possible.

In two dimensions (for images) the matrix $A$ is **block Toeplitz.** Each small block is $E$ by $E$. This is a familiar structure in computational engineering. The count $E^2$ of independent weights to be optimized is far smaller than for a fully connected network.

The same weights are used around all pixels (*shift-invariance*). The matrix produces a 2D convolution $\boldsymbol{x} * \boldsymbol{v}$. Frequently $A$ is called a **filter.**

To understand an image, look to see where it changes. *Find the edges.* Our eyes look for sharp cutoffs and steep gradients. Our computer can do the same by creating a filter. The dot products between a smooth function and a moving filter window will be smooth. But when an edge in the image lines up with a diagonal wall, *we see a spike.* Those dot products (fixed image) · (moving image) are exactly the "**convolution**" of the two images.

The difficulty with two or more dimensions is that edges can have many directions. We will need horizontal and vertical and diagonal filters for the test images. And filters have many purposes, including smoothing, gradient detection, and edge detection.

**1 Smoothing** For a 2D function $f$, the natural smoother is *convolution with a Gaussian*:

$$Gf(x,y) = \frac{1}{2\pi\sigma^2}\, e^{-(x^2+y^2)/2\sigma^2} * f = \frac{1}{\sqrt{2\pi}\,\sigma}\, e^{-x^2/2\sigma^2} * \frac{1}{\sqrt{2\pi}\,\sigma}\, e^{-y^2/2\sigma^2} * f(x,y)$$

This shows $G$ as a product of 1D smoothers. The Gaussian is everywhere positive, so it is *averaging*: $Gf$ cannot have a larger maximum than $f$. The filter removes noise (at a price in sharp edges). For small variance $\sigma^2$, details become clearer.

For a 2D vector (a matrix $f_{ij}$ instead of a function $f(x,y)$) the Gaussian must become discrete. The perfection of radial symmetry will be lost because the matrix $G$ is square. Here is a 5 by 5 discrete Gaussian $G$ $(E = 5)$:

$$G = \frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix} \approx \frac{1}{289}\begin{bmatrix} 1 \\ 4 \\ 7 \\ 4 \\ 1 \end{bmatrix}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \end{bmatrix} \qquad (3)$$

We also lost our exact product of 1D filters. To come closer, use a larger matrix $G = \boldsymbol{x}\boldsymbol{x}^{\mathrm{T}}$ with $\boldsymbol{x} = (.006, .061, .242, .383, .242, .061, .006)$ and discard the small outside pixels.

**2 Gradient detection** Image processing (as distinct from learning by a CNN) needs filters that detect the gradient. They contain specially chosen weights. We mention some simple filters just to indicate how they can find gradients—the first derivatives of $f$.

**One dimension** $(x_1, x_0, x_{-1}) = \left(-\frac{1}{2}, 0, \frac{1}{2}\right)$ $\left[\left(\frac{1}{2}, 0, -\frac{1}{2}\right) \text{ in convolution form}\right]$
$E = 3$

In this case the components of $Av$ are centered differences: $(Av)_i = \frac{1}{2} v_{i+1} - \frac{1}{2} v_{i-1}$. When the components of $v$ are increasing linearly from left to right, as in $v_i = 3i$, the output from the filter is $\frac{1}{2} 3(i+1) - \frac{1}{2} 3(i-1) = 3 =$ *correct gradient.*

The flip to $\left(\frac{1}{2}, 0, -\frac{1}{2}\right)$ comes from the definition of convolution as $\sum x_{i-k} v_k$.

**Two dimensions** These $3 \times 3$ *Sobel operators* approximate $\partial/\partial x$ and $\partial/\partial y$ :

$$E = 3 \qquad \frac{\partial}{\partial x} \approx \frac{1}{2} \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad \frac{\partial}{\partial y} \approx \frac{1}{2} \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{4}$$

For functions, the gradient vector $g = \text{grad}\, f$ has $||g||^2 = |\partial f/\partial x|^2 + |\partial f/\partial y|^2$.

Those weights were created for image processing, to locate the most important features of a typical image: *its edges.* These would be candidates for $E$ by $E$ filters inside a 2D convolutional matrix $A$. But remember that in deep learning, weights like $\frac{1}{2}$ and $-\frac{1}{2}$ are not chosen by the user. They are created from the training data.

Sections IV.2 and IV.5 of this book studied cyclic convolutions and Toeplitz matrices. Shift-invariance led to the application of discrete Fourier transforms. But in a CNN, ReLU is likely to act on each neuron. The network may include zero-padding—as well as max-pooling layers. So we cannot expect to apply the full power of Fourier analysis.

**3 Edge detection** After the gradient direction is estimated, we look for edges—the most valuable features to know. "*Canny Edge Detection*" is a highly developed process. Now we don't want smoothing, which would blur the edge. The good filters become *Laplacians of Gaussians*:

$$E\, f(x,y) = \nabla^2[g(x,y) * f(x,y)] = [\nabla^2 g(x,y)] * f(x,y). \tag{5}$$

The Laplacian $\nabla^2 G$ of a Gaussian is $(x^2 + y^2 - 2\sigma^2)\, e^{-(x^2+y^2)/2\sigma^2}/\pi\sigma^4$.

## The Stride of a Convolutional Filter

**Important** The filters described so far all have a *stride* $S = 1$. For a larger stride, the *moving window takes longer steps* as it moves across the image. Here is the matrix $A$ for a 1-dimensional 3-weight filter with a stride of 2. Notice especially that the length of the output $y = Av$ is reduced by that factor of 2 (previously four outputs and now two):

$$\textbf{Stride } S = 2 \qquad A = \begin{bmatrix} x_1 & x_0 & x_{-1} & 0 & 0 \\ 0 & 0 & x_1 & x_0 & x_{-1} \end{bmatrix} \tag{6}$$

Now the nonzero weights like $x_1$ in $L$ are two columns apart ($S$ columns apart for stride $S$). In 2D, a stride $S = 2$ reduces each direction by 2 and the whole output by 4.

## Extending the Signal

Instead of losing neurons at the edges of the image when $A$ is not square, we can *extend the input layer*. We are "inventing" components beyond the image boundary. Then the output $y = Av$ fits the image block: equal dimensions for input and output.

The simplest and most popular approach is **zero-padding**: Choose all additional components to be *zeros*. The extra columns on the left and right of $A$ multiply those zeros. In between, we have a square Toeplitz matrix as in Section IV.5. It is still determined by a much smaller set of weights than the number of entries in $A$.

For periodic signals, zero-padding is replaced by *wraparound*. The Toeplitz matrix becomes a circulant (Section IV.2). The Discrete Fourier Transform tells its eigenvalues. The eigenvectors are always the columns of the Fourier matrix. The multiplication $Av$ is a *cyclic convolution* and the Convolution Rule applies.

A more accurate choice is to go beyond the boundary by *reflection*. If the last component of the signal is $v_N$, and the matrix is asking for $v_{N+1}$ and $v_{N+2}$, we can reuse $v_N$ and $v_{N-1}$ (or else $v_{N-1}$ and $v_{N-2}$). Whatever the length of $v$ and the size of $A$, all the matrix entries in $A$ come from the same $E$ weights $x_{-1}$ to $x_1$ or $x_{-2}$ to $x_2$ (and $E^2$ weights in 2D).

*Note* Another idea. We might accept the original dimension (128 in our example) and use the reduction to 64 as a way to apply **two filters $C_1$ and $C_2$**. Each filter output is downsampled from 128 to 64. The total sample count remains 128. If the filters are suitably independent, no information is lost and the original 128 values can be recovered.

This process is linear. Two 64 by 128 matrices are combined into 128 by 128: square. If that matrix is invertible, as we intend, the filter bank is *lossless*.

*This is what CNN's usually do: Add more channels of weight matrices A in order to capture more features of the training sample.* The neural net has a bank of $B$ filters.

## Filter Banks and Wavelets

The idea in those last paragraphs produces a **filter bank**. This is just a set of $B$ different filters (convolutions). In signal processing, an important case combines a lowpass filter $C_1$ with a highpass filter $C_2$. The output of $C_1v$ is a smoothed signal (dominated by low frequencies). The output $C_2v$ is dominated by high frequencies. A perfect cutoff by ideal filters cannot be achieved by finite matrices $C_1$ and $C_2$.

From two filters we have a total of 256 output components. Then both outputs are subsampled. The result is 128 components, separated approximately into *averages and differences*—low frequencies and high frequencies. The matrix is 128 by 128.

**Wavelets** The wavelet idea is to repeat the same steps on the 64 components of the lowpass output $(\downarrow 2)\,C_1x$. *Then* $(\downarrow 2)\,C_1\,(\downarrow 2)\,C_1x$ *is an average of averages.* Its frequencies are concentrated in the lowest quarter ($|\omega| \leq \pi/4$) of all frequencies. The mid-frequency output $(\downarrow 2)\,C_2\,(\downarrow 2)\,C_1x$ with 32 components will not be subdivided. Then $128 = 64 + 32 + 16 + 16$.

In the limit of infinite subdivision, *wavelets* enter. This low-high frequency separation is an important theme in signal processing. It has not been so important for deep learning. But with multiple channels in a CNN, frequency separation could be effective.

## Counting the Number of Inputs and Outputs

In a one-dimensional problem, suppose a layer has $N$ neurons. We apply a convolutional matrix with $E$ nonzero weights. The stride is $S$, and we pad the input signal by $P$ zero's at each end. How many outputs ($M$ numbers) does this filter produce?

$$\boxed{\textbf{Karpathy's formula} \qquad M = \frac{N - E + 2P}{S} + 1} \tag{7}$$

In a 2D or 3D problem, this 1D formula applies in each direction.

Suppose $E = 3$ and the stride is $S = 1$. If we add one zero ($P = 1$) at each end, then

$$M = N - 3 + 2 + 1 = N \qquad \text{(input length = output length)}$$

This case $2P = E - 1$ with stride $S = 1$ is the most common architecture for CNN's.

If we don't pad the input with zeros, then $P = 0$ and $M = N - 2$ (as in the 4 by 6 matrix $A$ at the start of this section). In 2 dimensions this becomes $M^2 = (N-2)^2$. We lose neurons this way, but we avoid zero-padding.

Now suppose the stride is $S = 2$. Then $N - E$ must be an even number. Otherwise the formula (4) produces a fraction. Here are two examples of success for stride $S = 2$, with $N - E = 5 - 3$ and padding $P = 0$ or $P = 1$ at both ends of the five inputs:

$$\begin{array}{c}\textbf{Stride}\\ \textbf{2}\end{array} \quad \begin{bmatrix} x_{-1} & x_0 & x_1 & 0 & 0 \\ 0 & 0 & x_{-1} & x_0 & x_1 \end{bmatrix} \begin{bmatrix} x_{-1} & x_0 & x_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & x_{-1} & x_0 & x_1 & 0 & 0 \\ 0 & 0 & 0 & 0 & x_{-1} & x_0 & x_1 \end{bmatrix}$$

Again, our counts apply in each direction to an image in 2D or a tensor.

## A Deep Convolutional Network

Recognizing images is a major application of deep learning (and a major success). The success came with the creation of AlexNet and the development of convolutional nets. This page will describe a deep network of local convolutional matrices for image recognition. We follow the prize-winning paper of Simonyan and Zisserman from ICLR 2015. That paper recommends a deep architecture of $L = 16$–$19$ layers with small ($3 \times 3$) filters. The network has a breadth of $B$ parallel channels ($B$ images on each layer).

If the breadth $B$ were to stay the same at all layers, and all filters had $E$ by $E$ local weights, a straightforward formula would estimate the number $W$ of weights in the net:

$$\boxed{W \approx LBE^2 \qquad \textbf{$L$ layers, $B$ channels, $E$ by $E$ local convolutions}} \tag{8}$$

Notice that $W$ does not depend on the count of neurons on each layer. This is because $A$ has $E^2$ weights, whatever its size. Pooling will change that size without changing $E^2$.

But the count of $B$ channels can change—*and it is very common to end a CNN with fully-connected layers*. This will radically change the weight count $W$!

It is valuable to discuss the decisions taken by Simonyan and Zisserman, together with other options. Their choices led to $W \approx 135,000,000$ weights. The computations were on four NVIDIA GPU's, and training one net took 2-3 weeks. The reader may have less computing power (and smaller problems). So the network hyperparameters $L$ and $B$ will be reduced. We believe that the important principles remain the same.

A key point here is the recommendation to reduce the size $E$ of the local convolutions. 5 by 5 and 7 by 7 filters were rejected. In fact a 1 by 1 convolutional layer can be a way to introduce an extra bank of ReLU's—as in the ResNets coming next.

The authors compare three convolution layers, each with 3 by 3 filters, to a single layer of less local 7 by 7 convolutions. They are comparing 27 weights with 49 weights, and three nonlinear layers with one. In both cases the influence of a single data point spreads to three neighbors vertically and horizontally in the image or the RGB images ($B = 3$). Preference goes to the 3 by 3 filters with extra nonlinearities from more neurons per layer.

## Softmax Outputs for Multiclass Networks

In recognizing digits, we have 10 possible outputs. For letters and other symbols, 26 or more. With multiple output classes, we need an appropriate way to decide the very last layer (the output layer $w$ in the neural net that started with $v$). "Softmax" replaces the two-output case of logistic regression. **We are turning $n$ numbers into probabilities.**

The outputs $w_1, \ldots, w_n$ are converted to probabilities $p_1, \ldots, p_n$ that add to 1 :

$$\textbf{Softmax} \qquad p_j = \frac{1}{S}\, e^{w_j} \quad \text{where} \quad S = \sum_{k=1}^{n} e^{w_k} \tag{9}$$

Certainly softmax assigns the largest probability $p_j$ to the largest output $w_j$. But $e^w$ is a nonlinear function of $w$. So the softmax assignment is not invariant to scale: If we double all the outputs $w_j$, softmax will produce different probabilities $p_j$. For small $w$'s softmax actually deemphasizes the largest number $w_{\max}$.

In the CNN example of **teachyourmachine.com** to recognize digits, you will see how softmax produces the probabilities displayed in a pie chart—an excellent visual aid.

**CNN** We need a lot of weights to fit the data, and we are proud that we can compute them (with the help of gradient descent). But there is no justification for the number of weights to be uselessly large—*if weights can be reused.* For long signals in 1D and especially images in 2D, we may have no reason to change the weights from pixel to pixel.

1. cs231n.github.io/convolutional–networks/ (karpathy@cs.stanford.edu)

2. K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, ICLR (2015), arXiv : 1409.1556v6, 10 Apr 2015.

3. A. Krizhevsky, I. Sutskever, and G. Hinton, *ImageNet classification with deep convolutional neural networks*, NIPS (2012) 1106–1114.

4. Y. LeCun and Y. Bengio, *Convolutional networks for images, speech, and time-series*, *Handbook of Brain Theory and Neural Networks*, MIT Press (1998).

## Support Vector Machine in the Last Layer

For CNN's in computer vision, the final layer often has a special form. If the previous layers used ReLU and max-pooling (both piecewise linear), the last step can become a difference-of-convex program, and eventually a multiclass Support Vector Machine (SVM). Then optimization of the weights in a piecewise linear CNN can be one layer at a time.

L. Berrada, A. Zisserman, and P. Kumar, *Trusting SVM for piecewise linear CNNs*, arXiv: 1611.02185, 6 Mar 2017.

## The World Championship at the Game of Go

A dramatic achievement by a deep convolutional network was to defeat the (human) world champion at Go. This is a difficult game played on a 19 by 19 board. In turn, two players put down "stones" in attempting to surround those of the opponent. When a group of one color has no open space beside it (left, right, up, or down), those stones are removed from the board. Wikipedia has an animated game.

AlphaGo defeated the leading player Lee Sedol by 4 games to 1 in 2016. It had trained on thousands of human games. This was a convincing victory, but not overwhelming. Then the neural network was deepened and improved. Google's new version AlphaGo Zero learned to play without any human intervention—simply by playing against itself. Now it defeated its former self AlphaGo by 100 to 0.

The key point about the new and better version is that **the machine learned by itself**. It was told the rules and nothing more. The first version had been fed earlier games, aiming to discover why winners had won and losers had lost. The outcome from the new approach was parallel to the machine translation of languages. To master a language, special cases from grammar seemed essential. How else to learn all those exceptions? The translation team at Google was telling the system what it needed to know.

Meanwhile another small team was taking a different approach: Let the machine figure it out. *In both cases, playing Go and translating languages, success came with a deeper neural net and more games and no coaching.*

It is the depth and the architecture of AlphaGo Zero that interest us here. The hyperparameters will come in Section VII.4: the fateful decisions. The parallel history of Google Translate must wait until VII.5 because Recurrent Neural Networks **(RNN's)** *are needed—to capture the sequential structure of text.*

It is interesting that the machine often makes opening moves that have seldom or never been chosen by humans. The input to the network is a board position and its history. The output vector gives the probability of selecting each move—and also a scalar that estimates the probability of winning from that position. Every step communicates with a Monte Carlo tree search, to produce *reinforcement learning*.

## Residual Networks (ResNets)

Networks are becoming seriously deeper with more and more hidden layers. Mostly these are convolutional layers with a moderate number of independent weights. But depth brings dangers. Information can jam up and never reach the output. The problem of "vanishing gradients" can be serious: so many multiplications in propagating so far, with the result that computed gradients are exponentially small. When it is well designed, depth is a good thing—**but you must create paths for learning to move forward.**

The remarkable thing is that those fast paths can be very simple: "*skip connections*" *that go directly to the next layer*—*bypassing the usual step* $v_n = (A_n v_{n-1} + b_n)_+$. An efficient proposal of Veit, Wilber, and Belongie is to allow either a *skip* or a normal convolution, with a ReLU step every time. If the net has $L$ layers, there will be $2^L$ possible routes— fast or normal from each layer to the next.

One result is that entire layers can be removed without significant impact. The $n$th layer is reached by $2^{n-1}$ possible paths. Many paths have length well below $n$, not counting the skips.

It is hard to predict whether deep ConvNets will be replaced by ResNets.

K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, arXiv: 1512.03385, 10 Dec 2015. This paper works with extremely deep neural nets by adding shortcuts that skip layers, with weights $A = I$. Otherwise depth can degrade performance.

K. He, X. Zhang, S. Ren, and J. Sun, *Identity mappings in deep residual networks*, arXiv: 1603.05027, 25 Jul 2016.

A. Veit, M. Wilber, and S. Belongie, *Residual networks behave like ensembles of relatively shallow networks*, arXiv: 1605.06431, 27 Oct 2016.

## A Simple CNN: Learning to Read Letters

One of the class projects at MIT was a convolutional net. The user begins by drawing multiple copies (not many) of $A$ and $B$. On this training set, the correct classification is part of the input from the user. Then comes the mysterious step of *learning this data*—*creating a continuous piecewise linear function* $F(v)$ *that gives high probability to the* correct answer (the letter that was intended).

For learning to read digits, 10 probabilities appear in a pie chart. You quickly discover that too small a training set leads to frequent errors. If the examples had centered numbers or letters, and the test images are not centered, the user understands why those errors appear.

One purpose of **teachyourmachine.com** is education in machine learning at all levels (schools included). It is accessible to every reader.

These final references apply highly original ideas from signal processing to CNN's:

R. Balestriero and R. Baraniuk, *Mad Max: Affine spline insights into deep learning*, arXiv: 1805.06576.

S. Mallat, *Understanding deep convolutional networks*, Phil. Trans. Roy. Soc. **374** (2016); arXiv: 1601.04920.

C.-C. J. Kuo, *The CNN as a guided multilayer RECOS transform*, IEEE Signal Proc. Mag. **34** (2017) 81-89; arXiv: 1701.08481.

## Problem Set VII.2

**1** Wikipedia proposes a $5 \times 5$ matrix (different from equation (3)) to approximate a Gaussian. Compare the two filters acting on a horizontal edge (all 1's above all 0's) and a diagonal edge (lower triangle of 1's, upper triangle of 0's).

**2** What matrix—corresponding to the Sobel matrices in equation (4)—would you use to find gradients in the 45 ° diagonal direction?

**3** (Recommended) For image recognition, remember that the input sample $\boldsymbol{v}$ is a matrix (say 3 by 3). Pad it with zeros on all sides to be 5 by 5. Now apply a convolution as in the text (before equation (2)) to produce a 3 by 3 output $A\boldsymbol{v}$. What are the $1,1$ and $2,2$ entries of $A\boldsymbol{v}$?

**4** Here are two matrix approximations $L$ to the Laplacian $\partial^2 u/\partial x^2 + \partial^2 u/\partial y^2 = \nabla^2 u$:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 1 & 4 & 1 \\ 4 & -20 & 4 \\ 1 & 4 & 1 \end{bmatrix}.$$

What are the responses $\boldsymbol{LV}$ and $\boldsymbol{LD}$ to a vertical or diagonal step edge?

$$V = \begin{bmatrix} 2 & 2 & 2 & 6 & 6 & 6 \\ 2 & 2 & 2 & 6 & 6 & 6 \\ 2 & 2 & 2 & 6 & 6 & 6 \\ 2 & 2 & 2 & 6 & 6 & 6 \end{bmatrix} \qquad D = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

**5** *Could a convolutional net learn calculus?* Start with the derivatives of fourth degree polynomials $p(x)$. The inputs could be graphs of $p = a_0 + a_1 x + \cdots + a_4 x^4$ for $0 \le x \le 1$ and a training set of $a$'s. The correct outputs would be the coefficients $0, a_1, 2a_2, 3a_3, 4a_4$ from $dp/dx$. Using softmax with 5 classes, could you design and create a CNN to learn differential calculus?

**6** Would it be easier or harder to learn integral calculus? With the same inputs, the six outputs would be $0, a_0, \frac{1}{2}a_1, \frac{1}{3}a_2, \frac{1}{4}a_3, \frac{1}{5}a_4$.

**7** How difficult is addition of polynomials, with two graphs as inputs? The training set outputs would be the correct sums $a_0 + b_0, \ldots, a_4 + b_4$ of the coefficients. Is multiplication of polynomials difficult with 9 outputs $a_0 b_0, a_0 b_1 + a_1 b_0, \ldots, a_4 b_4$?

The inputs in 5-7 are pictures of the graphs. Cleve Moler reported on experiments:

**https://blogs.mathworks.com/cleve/2018/08/06/teaching–calculus–to–a–deep–learner**

Also **..2018/10/22/teaching–a–newcomer–about–teaching–calculus–to–a–deep–learner**

A theory of deep learning for hidden physics is emerging: for example see arXiv: 1808.04327.

# VII.3 Backpropagation and the Chain Rule

*Deep learning is fundamentally a giant problem in optimization.* We are choosing numerical "weights" to minimize a loss function $L$ (which depends on those weights). $L(\boldsymbol{x})$ adds up all the losses $\ell(\boldsymbol{w} - \textbf{true}) = \ell(F(\boldsymbol{x}, \boldsymbol{v}) - \text{true})$ between the computed outputs $\boldsymbol{w} = F(\boldsymbol{x}, \boldsymbol{v})$ and the true classifications of the inputs $\boldsymbol{v}$. Calculus tells us the system of equations to solve for the weights that minimize $L$:

**The partial derivatives of $L$ with respect to the weights $x$ should be zero.**

Gradient descent in all its variations needs to *compute derivatives* (components of the gradient of $F$) at the current values of the weights. The derivatives $\partial F/\partial \boldsymbol{x}$ lead to $\partial L/\partial \boldsymbol{x}$. From that information we move to new weights that give a smaller loss. Then we recompute derivatives of $F$ and $L$ at the new values of the weights, and repeat.

*Backpropagation is a method to compute derivatives quickly, using the chain rule*:

**Chain rule**
$$\frac{dF}{dx} = \frac{d}{dx}(F_3(F_2(F_1(x)))) = \left(\frac{dF_3}{dF_2}(F_2(F_1(x)))\right)\left(\frac{dF_2}{dF_1}(F_1(x))\right)\left(\frac{dF_1}{dx}(x)\right)$$

One goal is a way to visualize how the function $F$ is computed from the weights $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N$. A neat way to do this is a **computational graph**. It separates the big computation into small steps, and we can find the derivative of each step (each computation) on the graph. Then the chain rule from calculus gives the derivatives of the final output $\boldsymbol{w} = F(\boldsymbol{x}, \boldsymbol{v})$ with respect to all the weights $\boldsymbol{x}$. For a standard net, the steps in the chain rule can correspond to layers in the neural net.

This is an incredibly efficient improvement on the separate computation of each derivative $\partial F/\partial x_i$. At first it seems unbelievable, that reorganizing the computations can make such an enormous difference. In the end (the doubter might say) you have to compute derivatives for each step and multiply by the chain rule. But the method does work—and $N$ derivatives are computed in far less than $N$ times the cost of one derivative $\partial F/\partial x_1$.

Backpropagation has been discovered many times. Another name is **automatic differentiation (AD)**. You will see that the steps can be arranged in two basic ways: **forward-mode** and **backward-mode**. The right choice of mode can make a large difference in the cost (a factor of thousands). That choice depends on whether you have many functions $F$ depending on a few inputs, or few functions $F$ depending on many inputs.

Deep learning has basically one loss function depending on many weights. The right choice is "backward-mode AD". This is what we call **backpropagation**. It is the computational heart of deep learning. We will illustrate computational graphs and backpropagation by a small example.

The computational graphs were inspired by the brilliant exposition of Christopher Olah, posted on his blog (colah.github.io). Since 2017 he has published on (https://distill.pub). And the new paper by Catherine and Desmond Higham (arXiv: 1801.05894, to appear in *SIAM Review*) gives special attention to backpropagation, with very useful codes.

## Derivatives $\partial F/\partial x$ of the Learning Function $F(x, v)$

The weights $x$ consist of all the matrices $A_1, \ldots, A_L$ and the bias vectors $b_1, \ldots, b_L$. The inputs $v = v_0$ are the training data. The outputs $w = F(x, v_0)$ appear in layer $L$. Thus $w = v_L$ is the last step in the neural net, after $v_1, \ldots, v_{L-1}$ in the hidden layers.

Each new layer $v_n$ comes from the previous layer by $\mathrm{R}(b_n + A_n v_{n-1})$. Here R is the nonlinear activation function (usually ReLU) applied one component at a time.

Thus deep learning carries us from $v = v_0$ to $w = v_L$. Then we substitute $w$ into the loss function to measure the error for that sample $v$. It may be a classification error: 0 instead of 1, or 1 instead of 0. It may be a least squares regression error $||g - w||^2$, with $w$ instead of a desired output $g$. Often it is a "cross-entropy". The total loss $L(x)$ is the sum of the losses on all input vectors $v$.

The giant optimization of deep learning aims to find the weights $x$ that minimize $L$. *For full gradient descent the loss is* $L(x)$. *For stochastic gradient descent the loss at each* iteration is $\ell(x)$—from a single input or a minibatch of inputs. **In all cases we need the derivatives $\partial w/\partial x$ of the outputs $w$** (the components of the last layer) with respect to the weights $x$ (the $A$'s and $b$'s that carry us from layer to layer).

This is one reason that deep learning is so expensive and takes so long—even on GPU's. For convolutional nets the derivatives were found quickly and easily in Section VII.2.

## Computation of $\partial F/\partial x$ : Explicit Formulas

We plan to compute the derivatives $\partial F/\partial x$ in two ways. The first way is to present the explicit formulas: the derivative with respect to each and every weight. The second way is to describe the **backpropagation algorithm** that is constantly used in practice.

Start with the last bias vector $b_L$ and weight matrix $A_L$ that produce the final output $v_L = w$. There is no nonlinearity at this layer, and we drop the layer index $L$:

$$\boxed{v_L = b_L + A_L v_{L-1} \quad \text{or simply} \quad w = b + Av.} \tag{1}$$

Our goal is to find the derivatives $\partial w_i/\partial b_j$ and $\partial w_i/\partial A_{jk}$ for all components of $b + Av$. When $j$ is different from $i$, *the $i$th output $w_i$ is not affected by $b_j$ or $A_{jk}$*. Multiplying $A$ times $v$, row $j$ of $A$ produces $w_j$ and not $w_i$. We introduce the symbol $\delta$ which is 1 or 0:

$$\delta_{ij} = 1 \text{ if } i = j \qquad \delta_{ij} = 0 \text{ if } i \neq j \qquad \text{The identity matrix } I \text{ has entries } \delta_{ij}.$$

Columns of $I$ are **1-hot vectors!** The derivatives are 1 or 0 or $v_k$ (Section I.12):

**Fully connected layer**
**Independent weights $A_{jk}$**

$$\boxed{\frac{\partial w_i}{\partial b_j} = \delta_{ij} \quad \text{and} \quad \frac{\partial w_i}{\partial A_{jk}} = \delta_{ij} v_k} \tag{2}$$

*Example* There are six $b$'s and $a$'s in $\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} + \begin{bmatrix} a_{11}v_1 + a_{12}v_2 \\ a_{21}v_1 + a_{22}v_2 \end{bmatrix}$.

**Derivatives of $w_1$** $\dfrac{\partial w_1}{\partial b_1} = 1, \dfrac{\partial w_1}{\partial b_2} = 0, \dfrac{\partial w_1}{\partial a_{11}} = v_1, \dfrac{\partial w_1}{\partial a_{12}} = v_2, \dfrac{\partial w_1}{\partial a_{21}} = \dfrac{\partial w_1}{\partial a_{22}} = 0.$

## Combining Weights $b$ and $A$ into $M$

It is often convenient to combine the bias vector $b$ and the matrix $A$ into one matrix $M$:

$$\textbf{Matrix of weights} \qquad M = \begin{bmatrix} 1 & 0^{\mathrm{T}} \\ b & A \end{bmatrix} \quad \text{has} \quad M \begin{bmatrix} 1 \\ v \end{bmatrix} = \begin{bmatrix} 1 \\ b + Av \end{bmatrix}. \tag{3}$$

For each layer of the neural net, the top entry (*the zeroth entry*) is now **fixed at** 1. After multiplying that layer by $M$, *the zeroth component on the next layer is still* 1. Then $\mathrm{ReLU}(1) = 1$ preserves that entry in every hidden layer.

At the beginning of this book (page v), the big image of a neural net had squares for zeroth entries and circles for all other entries. *Every square now contains a* 1.

This block matrix $M$ produces a compact derivative formula for the last layer $\boldsymbol{w} = M\boldsymbol{v}$.

$$M = \begin{bmatrix} 1 & 0^{\mathrm{T}} \\ b & A \end{bmatrix} \quad \textbf{and} \quad \frac{\partial w_i}{\partial M_{jk}} = \delta_{ij} v_k \quad \textbf{for} \quad i > 0. \tag{4}$$

Both $\boldsymbol{v}$ and $\boldsymbol{w}$ begin with 1's. Then $k = 0$ correctly gives $\partial w_0 / \partial M_{j0} = 0$ for $j > 0$.

## Derivatives for Hidden Layers

Now suppose there is one hidden layer, so $L = 2$. The output is $\boldsymbol{w} = \boldsymbol{v}_L = \boldsymbol{v}_2$, the hidden layer contains $\boldsymbol{v}_1$, and the input is $\boldsymbol{v}_0 = \boldsymbol{v}$. The nonlinear R is probably ReLU.

$$\boldsymbol{v}_1 = \mathrm{R}(\boldsymbol{b}_1 + A_1 \boldsymbol{v}_0) \quad \text{and} \quad \boldsymbol{w} = \boldsymbol{b}_2 + A_2 \boldsymbol{v}_1 = \boldsymbol{b}_2 + A_2 \mathrm{R}(\boldsymbol{b}_1 + A_1 \boldsymbol{v}_0).$$

Equation (2) still gives the derivatives of $\boldsymbol{w}$ with respect to the last weights $\boldsymbol{b}_2$ and $A_2$. The function R is absent at the output and $\boldsymbol{v}$ is $\boldsymbol{v}_1$. But the derivatives of $\boldsymbol{w}$ with respect to $\boldsymbol{b}_1$ and $A_1$ do involve the nonlinear function R acting on $\boldsymbol{b}_1 + A_1 \boldsymbol{v}_0$.

So the derivatives in $\partial \boldsymbol{w} / \partial A_1$ need the chain rule $\partial f / \partial x = (\partial f / \partial g)(\partial g / \partial x)$:

$$\textbf{Chain rule} \quad \frac{\partial \boldsymbol{w}}{\partial A_1} = \frac{\partial [A_2 \mathrm{R}(\boldsymbol{b}_1 + A_1 \boldsymbol{v}_0)]}{\partial A_1} = A_2 \mathrm{R}'(\boldsymbol{b}_1 + A_1 \boldsymbol{v}_0) \frac{\partial (\boldsymbol{b}_1 + A_1 \boldsymbol{v}_0)}{\partial A_1}. \tag{5}$$

That chain rule has three factors. Starting from $\boldsymbol{v}_0$ at layer $L - 2 = 0$, the weights $\boldsymbol{b}_1$ and $A_1$ bring us toward the layer $L - 1 = 1$. The derivatives of that step are exactly like equation (2). But the output of that partial step is not $\boldsymbol{v}_{L-1}$. To find that hidden layer we first have to apply R. So the chain rule includes its derivative $\mathrm{R}'$. Then the final step (to $\boldsymbol{w}$) multiplies by the last weight matrix $A_2$.

The Problem Set extends these formulas to $L$ layers. They could be useful. But with pooling and batch normalization, automatic differentiation seems to defeat hard coding.

**Very important** Notice how formulas like (2) and (5) go **backwards from $\boldsymbol{w}$ to $\boldsymbol{v}$.** Automatic backpropagation will do this too. **"Reverse mode"** starts with the output.

## Details of the Derivatives $\partial w/\partial A_1$

We feel some responsibility to look more closely at equation (5). Its nonlinear part $R'$ comes from the derivative of the nonlinear activation function. The usual choice is the ramp function ReLU $(x) = (x)_+$, and we see ReLU as the limiting case of an $S$-shaped sigmoid function. Here is ReLU together with its first two derivatives:

$$\text{ReLU}(x) = \max(0, x) = (x)_+ \qquad \textbf{Ramp function } R(x)$$

$$dR/dx = \begin{cases} 0 & x < 0 \\ 1 & x > 0 \end{cases} \qquad \begin{array}{l}\textbf{Step function} \\ H(x) = dR/dx\end{array}$$

$$d^2R/dx^2 = \begin{cases} 0 & x \neq 0 \\ 1 & \text{integral over all } x \end{cases} \qquad \begin{array}{l}\textbf{Delta function} \\ \delta(x) = d^2R/dx^2\end{array}$$

The delta function represents an *impulse*. It models a finite change in an infinitesimal time. It is physically impossible but mathematically convenient. It is defined, not at every point $x$, but by its effect on integrals from $-\infty$ to $\infty$ of a continuous function $g(x)$:

$$\int \delta(x)\,dx = 1 \qquad \int \delta(x)\,g(x)\,dx = g(0) \qquad \int \delta(x-a)\,g(x)\,dx = g(a)$$

With ReLU, a neuron could stay at zero through all the steps of deep learning. This "dying ReLU" can be avoided in several ways—it is generally not a major problem. One way that firmly avoids it is to change to a Leaky ReLU with a nonzero gradient:

$$\textbf{Leaky ReLU}\,(x) = \begin{cases} x & x \geq 0 \\ .01x & x \leq 0 \end{cases} \quad \text{Always ReLU}\,(ax) = a\text{ReLU}\,(x) \tag{6}$$

Geoffrey Hinton pointed out that if all the bias vectors $\boldsymbol{b}_1, \ldots, \boldsymbol{b}_L$ are set to zero at every layer of the net, the scale of the input $\boldsymbol{v}$ passes straight through to the output $\boldsymbol{w} = F(\boldsymbol{v})$. Thus $F(A\boldsymbol{v}) = aF(\boldsymbol{v})$. (A final softmax would lose this scale invariance.)

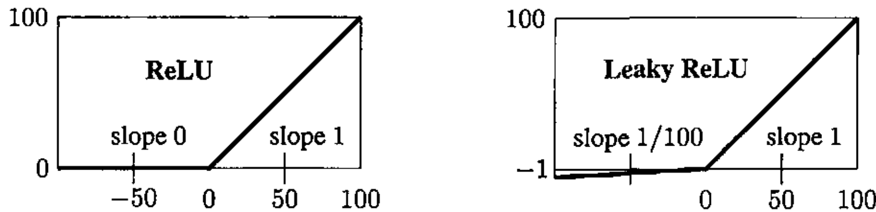



Figure VII.4: The graphs of ReLU and Leaky ReLU (two options for nonlinear activation).

Returning to formula (5), write $A$ and $\boldsymbol{b}$ for the matrix $A_{L-1}$ and the vector $\boldsymbol{b}_{L-1}$ that produce the last hidden layer. Then ReLU and $A_L$ and $\boldsymbol{b}_L$ produce the final output $\boldsymbol{w} = \boldsymbol{v}_L$. Our interest is in $\partial w/\partial A$, the dependence of $\boldsymbol{w}$ on the next to last matrix of weights.

$$\boxed{\boldsymbol{w} = A_L(R(A\boldsymbol{v} + \boldsymbol{b})) + \boldsymbol{b}_L \quad \text{and} \quad \frac{\partial \boldsymbol{w}}{\partial A} = A_L R'(A\boldsymbol{v} + \boldsymbol{b})\frac{\partial(A\boldsymbol{v} + \boldsymbol{b})}{\partial A}} \tag{7}$$

We think of $R$ as a diagonal matrix of ReLU functions acting component by component on $Av + b$. Then $J = R'(Av + b)$ is a diagonal matrix with 1's for positive components and 0's for negative components. (Don't ask about zeros.) Formula (7) has become (8):

$$w = A_L\, R\,(Av + b) \quad \text{and} \quad \frac{\partial w}{\partial A} = A_L\, J\, \frac{\partial (Av + b)}{\partial A} \tag{8}$$

We know every component ($v_k$ or zero) of the third factor from the derivatives in (2).

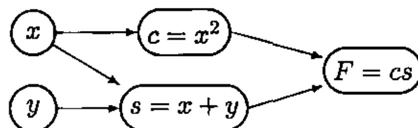
When the sigmoid function $R_a$ replaces the ReLU function, the diagonal matrix $J = R_a'(Av + b)$ no longer contains 1's and 0's. Now we evaluate the derivative $dR_a/dx$ at each component of $Av + b$.

In practice, backpropagation finds the derivatives with respect to all $A$'s and $b$'s. It creates those derivatives automatically (and effectively).

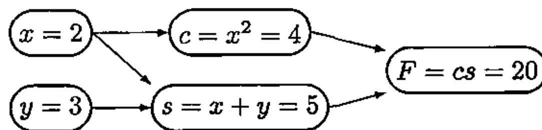
## Computational Graphs

Suppose $F(x, y)$ is a function of two variables $x$ and $y$. Those inputs are the first two nodes in the computational graph. A typical step in the computation—**an edge in the graph**—is one of the operations of arithmetic (addition, subtraction, multiplication,...). The final output is the function $F(x, y)$. Our example will be $\boldsymbol{F = x^2(x + y)}$.

Here is the graph that computes $F$ with intermediate nodes $c = x^2$ and $s = x + y$:




When we have inputs $x$ and $y$, for example $\boldsymbol{x = 2}$ and $\boldsymbol{y = 3}$, the edges lead to $\boldsymbol{c = 4}$ and $\boldsymbol{s = 5}$ and $\boldsymbol{F = 20}$. This agrees with the algebra that we normally crowd into one line: $F = x^2(x + y) = 2^2(2 + 3) = 4(5) = 20$.



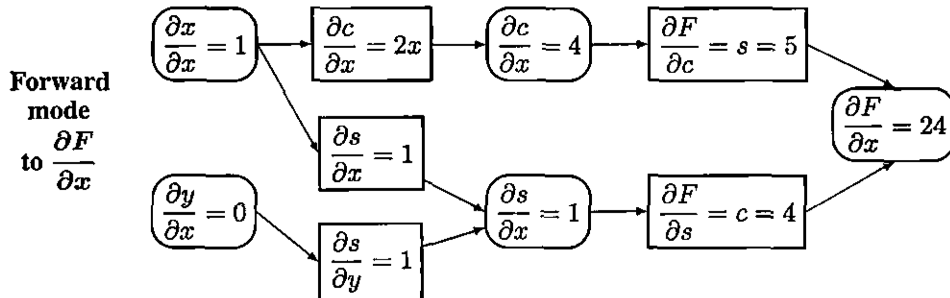

Now we compute the derivative of each step—each edge in the graph. Begin with the $x$-derivative. At first we choose *forward-mode*, starting with the input $x$ and moving toward the output function $x^2(x + y)$. So the first steps use the power rule for $c = x^2$ and the sum rule for $s = x + y$. The last step applies the product rule to $F = c$ times $s$.

$$\frac{\partial c}{\partial x} = 2x \qquad \frac{\partial s}{\partial x} = 1 \qquad \frac{\partial F}{\partial c} = s \qquad \frac{\partial F}{\partial s} = c$$

Moving through the graph produces the chain rule!

$$\frac{\partial F}{\partial x} = \frac{\partial F}{\partial c}\frac{\partial c}{\partial x} + \frac{\partial F}{\partial s}\frac{\partial s}{\partial x}$$
$$= (s)(2x) + (c)(1) = (5)(4) + (4)(1) = \mathbf{24}$$

The result is to compute the derivative of the output $F$ with respect to *one input* $x$. You can see those $x$-derivatives on the computational graph.




There will be a similar graph for $y$-derivatives—the forward mode leading to $\partial F/\partial y$. Here is the chain rule and the numbers that would appear in that graph for $x = 2$ and $y = 3$ and $c = x^2 = 2^2$ and $s = x + y = 2 + 3$ and $F = cs$:

$$\frac{\partial F}{\partial y} = \frac{\partial F}{\partial c}\frac{\partial c}{\partial y} + \frac{\partial F}{\partial s}\frac{\partial s}{\partial y}$$
$$= (s)(0) + (c)(1) = (5)(0) + (4)(1) = \mathbf{4}$$

The computational graph for $\partial F/\partial y$ is not drawn but the point is important: *Forward mode requires a new graph for each input $x_i$, to compute the partial derivative $\partial F/\partial x_i$.*

## Reverse Mode Graph for One Output

The reverse mode starts with the output $F$. **It computes the derivatives with respect to both inputs**. The computations go **backward** through the graph.

That means it does not follow the empty line that started with $\partial y/\partial x = 0$ in the forward graph for $x$-derivatives. And it would not follow the empty line $\partial x/\partial y = 0$ in the forward graph (not drawn) for $y$-derivatives. A larger and more realistic problem with $N$ inputs will have $N$ forward graphs, each with $N - 1$ empty lines (because the $N$ inputs are independent). The derivative of $x_i$ with respect to every other input $x_j$ is $\partial x_i/\partial x_j = 0$.

Instead of $N$ forward graphs from $N$ inputs, we will have **one backward graph from one output**. Here is that reverse-mode computational graph. It finds the derivative of $F$ with respect to every node. It starts with $\partial F/\partial F = 1$ and goes in reverse.

A computational graph executes the chain rule to find derivatives. The reverse mode finds all derivatives $\partial F/\partial x_i$ by following all chains **backward from output to input.** Those chains all appear as paths **on one graph**—not as separate chain rules for exponentially many possible paths. This is the success of reverse mode.

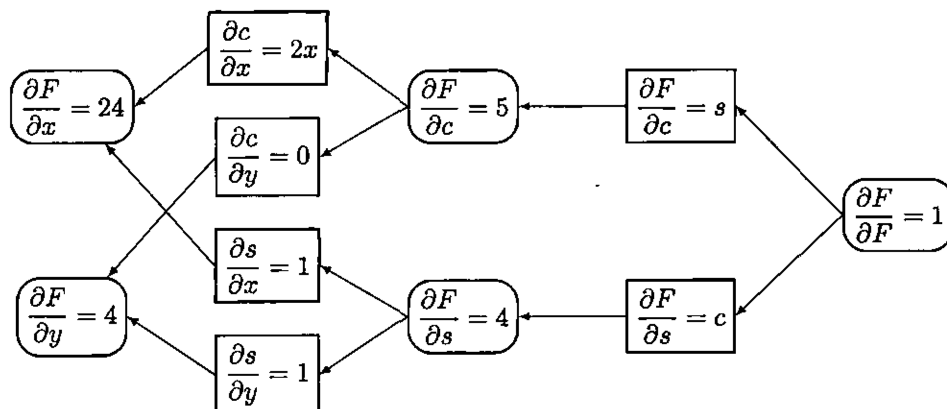



Figure VII.5: Reverse-mode computation of the gradient $\left(\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}\right)$ at $x = 2, y = 3$.

## Product of Matrices $ABC$ : Which Order ?

The decision between forward and reverse order also appears in matrix multiplication! If we are asked to multiply $A$ times $B$ times $C$, the associative law offers two choices for the multiplication order:

**$AB$ first or $BC$ first?** **Compute $(AB)\,C$ or $A(BC)$?**

The result is the same but the number of individual multiplications can be very different. Suppose the matrix $A$ is $m$ by $n$, and $B$ is $n$ by $p$, and $C$ is $p$ by $q$.

**First way**

$$AB = (m \times n)\,(n \times p) \text{ has } mnp \text{ multiplications}$$
$$(AB)C = (m \times p)\,(p \times q) \text{ has } mpq \text{ multiplications}$$

**Second way**

$$BC = (n \times p)\,(p \times q) \text{ has } npq \text{ multiplications}$$
$$A(BC) = (m \times n)\,(n \times q) \text{ has } mnq \text{ multiplications}$$

So the comparison is between $mp(n+q)$ and $nq(m+p)$. Divide both numbers by $mnpq$:

| The first way is faster when $\frac{1}{q} + \frac{1}{n}$ is smaller than $\frac{1}{m} + \frac{1}{p}$. |
|---|

Here is an extreme case (extremely important). *Suppose $C$ is a column vector*: $p$ by 1. Thus $q = 1$. Should you multiply $BC$ to get another column vector ($n$ by 1) and then $A(BC)$ to find the output ($m$ by 1)? Or should you multiply $AB$ first?

The question almost answers itself. The correct $A(BC)$ produces a vector at each step. The matrix-vector multiplication $BC$ has $np$ steps. The next matrix-vector multiplication $A(BC)$ has $mn$ steps. Compare those $np + mn$ steps to the cost of starting with the matrix-matrix option $AB$ ($mnp$ steps!). Nobody in their right mind would do that.

*But if $A$ is a row vector, $(AB)C$ is better.* Row times matrix each time.

This will match the central computation of deep learning: **Training the network = optimizing the weights.** The output $F(\boldsymbol{v})$ from a deep network is a chain starting with $\boldsymbol{v}$:

$$F(\boldsymbol{v}) = A_L \boldsymbol{v}_{L-1} = A_L(\mathrm{R}\, A_{L-1}(\ldots(\mathrm{R}\, A_2(\mathbf{R}\, A_1 \boldsymbol{v})))) \text{ is forward through the net.}$$

The derivatives of $F$ with respect to the matrices $A$ (and the bias vectors $\boldsymbol{b}$) are easiest for the *last matrix* $A_L$ in $A_L \boldsymbol{v}_{L-1}$. The derivative of $A\boldsymbol{v}$ with respect to $A$ contains $\boldsymbol{v}$'s:

$\dfrac{\partial F_i}{\partial A_{jk}} = \delta_{ij}\, v_k$. Next is the derivative of $A_L\, \mathrm{ReLU}\,(A_{L-1} v_{L-1})$ with respect to $A_{L-1}$.

We can explain how that same reverse mode also appears in the comparison of direct methods *versus* adjoint methods for optimization (choosing good weights).

## Adjoint Methods

The same question of the best order for matrix multiplication $ABC$ comes up in a big class of optimization problems. We are solving a square system of $N$ linear equations $E\boldsymbol{v} = \boldsymbol{b}$. The vector $\boldsymbol{b}$ depends on **design variables** $\boldsymbol{p} = (p_1, \ldots, p_M)$. Therefore the solution vector $\boldsymbol{v} = E^{-1}\boldsymbol{b}$ depends on $\boldsymbol{p}$. **The matrix $\partial \boldsymbol{v}/\partial \boldsymbol{p}$ containing the derivatives $\partial v_i/\partial p_j$ will be $N$ by $M$.**

To repeat: We are minimizing $F(\boldsymbol{v})$. The vector $\boldsymbol{v}$ depends on the design variables $\boldsymbol{p}$. So we need a chain rule that multiplies derivatives $\partial F/\partial v_i$ times derivatives $\partial v_i/\partial p_j$. Let me show how this becomes a product of **three matrices**—and the multiplication order is decisive. **Three sets of derivatives** control how $F$ depends on the input variables $p_i$:

| | |
|---|---|
| $A = \boldsymbol{\partial F/\partial v_i}$ | The derivatives of $F$ with respect to $v_1, \ldots, v_N$ |
| $B = \boldsymbol{\partial v_i/\partial b_k}$ | The derivative of each $v_i$ with respect to each $b_k$ |
| $C = \boldsymbol{\partial b_k/\partial p_j}$ | The derivative of each $b_k$ with respect to each $p_j$ |

To see $\partial v_i/\partial p_j$ we take derivatives of the equation $E\boldsymbol{v} = \boldsymbol{b}$ with respect to the $p_j$:

$$E\,\frac{\partial \boldsymbol{v}}{\partial p_j} = \frac{\partial \boldsymbol{b}}{\partial p_j}, \quad j = 1, \ldots, M \quad \text{so} \quad \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{p}} = E^{-1}\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{p}}. \tag{9}$$

It seems that we have $M$ linear systems of size $N$. Those will be expensive to solve over and over, as we search for the choice of $\boldsymbol{p}$ that minimizes $F(\boldsymbol{v})$. The matrix $\partial \boldsymbol{v}/\partial \boldsymbol{p}$ contains the derivatives of $v_1, \ldots, v_N$ with respect to the design variables $p_1, \ldots, p_M$.

Suppose for now that the cost function $F(\boldsymbol{v}) = \boldsymbol{c}^{\mathrm{T}}\boldsymbol{v}$ is linear (so $\partial F/\partial \boldsymbol{v} = \boldsymbol{c}^{\mathrm{T}}$). Then what optimization actually needs is the gradient of $F(\boldsymbol{v})$ with respect to the $p$'s. The first set of derivatives $\partial F/\partial \boldsymbol{v}$ is just the vector $\boldsymbol{c}^{\mathrm{T}}$:

$$\boxed{\frac{\partial F}{\partial \boldsymbol{p}} = \frac{\partial F}{\partial \boldsymbol{v}}\,\frac{\partial \boldsymbol{v}}{\partial \boldsymbol{p}} = \boldsymbol{c}^{\mathrm{T}}\, E^{-1}\,\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{p}} \quad \text{has three factors to be multiplied.}} \tag{10}$$

This is the key equation. It ends with a product of a **row vector** $\mathbf{c}^{\mathrm{T}}$ times an $N$ **by** $N$ **matrix** $E^{-1}$ times an $N$ **by** $M$ **matrix** $\partial b/\partial p$. How should we compute that product?

Again the question almost answers itself. We do *not* want to multiply two matrices. So we are *not* computing $\partial v/\partial p$ after all. Instead the good first step is to find $\boldsymbol{c}^{\mathbf{T}}E^{-1}$. This produces a row vector $\boldsymbol{\lambda}^{\mathbf{T}}$. In other words we solve the adjoint equation $E^{\mathrm{T}}\boldsymbol{\lambda} = \boldsymbol{c}$:

$$\textbf{Adjoint equation} \qquad E^{\mathrm{T}}\boldsymbol{\lambda} = \boldsymbol{c} \quad \text{gives} \quad \boldsymbol{\lambda}^{\mathrm{T}}E = \boldsymbol{c}^{\mathrm{T}} \quad \text{and} \quad \boldsymbol{\lambda}^{\mathrm{T}} = \boldsymbol{c}^{\mathrm{T}}E^{-1}. \tag{11}$$

Substituting $\boldsymbol{\lambda}^{\mathbf{T}}$ for $\boldsymbol{c}^{\mathbf{T}}E^{-1}$ in equation (10), the final step multiplies that row vector times the derivatives of the vector $\boldsymbol{b}$ (its gradient):

$$\textbf{Gradient of the cost } \boldsymbol{F} \qquad \frac{\partial F}{\partial \boldsymbol{p}} = \boldsymbol{\lambda}^{\mathbf{T}}\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{p}} \qquad (1 \text{ by } N \text{ times } N \text{ by } M). \tag{12}$$

**The optimal order is $(AB)C$ because the first factor $A$ is actually the row vector $\boldsymbol{\lambda}^{\mathbf{T}}$.**

This example of an adjoint method started with $E\boldsymbol{x} = \boldsymbol{b}$. The right hand side $\boldsymbol{b}$ depended on design parameters $\boldsymbol{p}$. So the solution $\boldsymbol{x} = E^{-1}\boldsymbol{b}$ depended on $\boldsymbol{p}$. Then the cost function $F(\boldsymbol{x}) = \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x}$ depended on $\boldsymbol{p}$.

The adjoint equation $A^{\mathrm{T}}\boldsymbol{\lambda} = \boldsymbol{c}$ found the vector $\boldsymbol{\lambda}$ that efficiently combined the last two steps. "Adjoint" has a parallel meaning to "transpose" and we can apply it also to differential equations. The design variables $p_1, \ldots, p_M$ might appear in the matrix $E$, or in an eigenvalue problem or a differential equation.

Our point here is to emphasize and reinforce the key idea of backpropagation:

**The reverse mode can order the derivative computations in a faster way.**

## Adjoints and Sensitivity for Deep Layers

Coming closer to the problem of deep learning, what are the derivatives $\partial w/\partial x_j$ of the outputs $\boldsymbol{w} = (w_1, \ldots w_M)$ at layer $L$ with respect to the parameters $\boldsymbol{x} = (x_1, \ldots, x_N)$? That output $\boldsymbol{w} = \boldsymbol{v}_L$ is seen after $L$ steps from the input $\boldsymbol{v}_0$. We write step $n$ as

$$\boldsymbol{v}_n = F_n(\boldsymbol{v}_{n-1}, \boldsymbol{x}_n) \quad \text{where} \quad F_n \quad \text{depends on the weights (parameters)} \quad \boldsymbol{x}_n. \tag{13}$$

(13) is a **recurrence relation**. And the same $P$ parameters $\boldsymbol{x}$ could be used at every step. Deep learning has new parameters for each new layer—which gives it "learning power" that an ordinary recurrence relation cannot hope for. In fact a typical recurrence (13) is just a finite difference analog of a differential equation $d\boldsymbol{v}/dt = \boldsymbol{f}(\boldsymbol{v}, \boldsymbol{x}, t)$.

The analogy is not bad. In this case too we may be aiming for a desired output $\boldsymbol{v}(T)$, and we are choosing parameters $\boldsymbol{x}$ to bring us close. The problem is to find the matrix of derivatives $J = \partial \boldsymbol{v}_N/\partial \boldsymbol{x}_M$. We have to apply the chain rule to equation (13), all the way back from $N$ to 0. Here is a step of the chain:

$$v_N = F_N(v_{N-1}, x_N) = F_N(F_{N-1}(v_{N-2}, x_{N-1}), x_N). \tag{14}$$

Take its derivatives with respect to $x_{N-1}$, to see the rule over the last two layers:

$$\frac{\partial v_N}{\partial x_{N-1}} = \frac{\partial F_N}{\partial v_{N-1}} \frac{\partial v_{N-1}}{\partial x_{N-1}} \qquad \frac{\partial v_N}{\partial x_{N-2}} = \frac{\partial F_N}{\partial v_{N-1}} \frac{\partial v_{N-1}}{\partial x_{N-2}} = \frac{\partial F_N}{\partial v_{N-1}} \frac{\partial v_{N-1}}{\partial v_{N-2}} \frac{\partial v_{N-2}}{\partial x_{N-2}}$$

That last expression is a triple product $ABC$. The calculation requires a decision: *Start with $AB$ or start with $BC$?* Both the adjoint method for optimization and the reverse mode of backpropagation would counsel: *Begin with $AB$.*

The last two pages developed from class notes by Steven Johnson: *Adjoint methods and sensitivity analysis for recurrence relations*, **http://math.mit.edu/~stevenj/18.336/recurrence2.pdf**. Also online: *Notes on adjoint methods for* 18.335.

For deep learning, the recurrence relation is between layers of the net.

## Problem Set VII.3

**1** If $\boldsymbol{x}$ and $\boldsymbol{y}$ are column vectors in $\mathbf{R}^n$, is it faster to multiply $\boldsymbol{x}(\boldsymbol{y}^{\mathrm{T}}\boldsymbol{x})$ or $(\boldsymbol{x}\boldsymbol{y}^{\mathrm{T}})\boldsymbol{x}$?

**2** If $A$ is an $m$ by $n$ matrix with $m > n$, is it faster to multiply $A(A^{\mathrm{T}}A)$ or $(AA^{\mathrm{T}})A$?

**3** (a) If $A\boldsymbol{x} = \boldsymbol{b}$, what are the derivatives $\partial x_i/\partial b_j$ with $A$ fixed?

(b) What are the derivatives of $\partial x_i/\partial A_{jk}$ with $\boldsymbol{b}$ fixed?

**4** For $\boldsymbol{x}$ and $\boldsymbol{y}$ in $\mathbf{R}^n$, what are $\partial(\boldsymbol{x}^{\mathrm{T}}\boldsymbol{y})/\partial x_i$ and $\partial(\boldsymbol{x}\boldsymbol{y}^{\mathrm{T}})/\partial x_i$?

**5** Draw a computational graph to compute the function $f(x, y) = x^3(x - y)$. Use the graph to compute $f(2, 3)$.

**6** Draw a reverse mode graph to compute the derivatives $\partial f/\partial x$ and $\partial f/\partial y$ for $f = x^3(x - y)$. Use the graph to find those derivatives at $x = 2$ and $x = 3$.

**7** Suppose $A$ is a Toeplitz matrix in a convolutional neural net (**CNN**). The number $a_k$ is on diagonal $k = 1 - n, \ldots, n - 1$. If $\boldsymbol{w} = A\boldsymbol{v}$, what is the derivative $\partial w_i/\partial a_k$?

**8** In a max-pooling layer, suppose $w_i = \max(v_{2i-1}, v_{2i})$. Find all $\partial w_i/\partial v_j$.

**9** To understand the chain rule, start from this identity and let $\Delta x \to 0$:

$$\frac{f(g(x + \Delta x)) - f(g(x))}{\Delta x} = \frac{f(g(x + \Delta x)) - f(g(x))}{g(x + \Delta x) - g(x)} \quad \frac{g(x + \Delta x) - g(x)}{\Delta x}$$

Then the derivative at $x$ of $f(g(x))$ equals $df/dg$ at $g(x)$ times $dg/dx$ at $x$.

Question: Find the derivative at $x = 0$ of $\sin(\cos(\sin x))$.

Backpropagation is essentially equivalent to **AD** (automatic differentiation) in reverse mode:

A. Griewank and A. Walther, *Evaluating Derivatives*, SIAM (2008).

# VII.4 Hyperparameters : The Fateful Decisions

After the loss function is chosen and the network architecture is decided, there are still critical decisions to be made. We must choose the *hyperparameters*. They govern the algorithm itself—the computation of the weights. Those weights represent what the computer has learned from the training set: how to predict the output from the features in the input. In machine learning, the decisions include those hyperparameters and the loss function and dropout and regularization.

The goal is to find patterns that distinguish 5 from 7 and 2—by looking at pixels. The hyperparameters decide how quickly and accurately those patterns are discovered. The **stepsize** $s_k$ in gradient descent is first and foremost. That number appears in the iteration $x_{k+1} = x_k - s_k \nabla L(x_k)$ or one of its variants: *accelerated* (by momentum) or *adaptive* (ADAM) or *stochastic* with a random minibatch of training data at each step $k$.

The words **learning rate** are often used in place of stepsize. Depending on the author, the two might be identical or differ by a normalizing factor. Also: $\eta_k$ *often replaces* $s_k$. First we ask for the optimal stepsize when there is only one unknown. Then we point to a general approach. Eventually we want a faster decision.

1. **Choose** $s_k = 1/L''(x_k)$. Newton uses the second derivative of $L$. That choice accounts for the quadratic term in the Taylor series for $L(x)$ around the point $x_k$. As a result, Newton's method is *second order*: The error in $x_{k+1}$ is proportional to the *square* of the error in $x_k$. Near the minimizing $x^*$, convergence is fast.

   In more dimensions, the second derivative becomes the Hessian matrix $H(x) = \nabla^2 L(x_k)$. Its size is the number of weights (components of $x$). To find $x_{k+1}$, Newton solves a large system of equations $H(x_k)(x_{k+1} - x_k) = -\nabla L(x_k)$. Gradient descent replaces $H$ by a single number $1/s_k$.

2. **Decide** $s_k$ **from a line search**. The gradient $\nabla L(x_k)$ sets the direction of the line. The current point $x_k$ is the start of the line. By evaluating $L(x)$ at points on the line, we find a nearly minimizing point—which becomes $x_{k+1}$.

   Line search is a practical idea. One algorithm is *backtracking*, as described in Section VI.4. This reduces the stepsize $s$ by a constant factor until the decrease in $L$ is consistent with the steepness of the gradient (again within a chosen factor). Optimizing a line search is a carefully studied 1-dimensional problem.

   But no method is perfect. We look next at the effect of a poor stepsize $s$.

## Too Small or Too Large

We need to identify the difficulties with a poor choice of learning rate:

**$s_k$ is too small** Then gradient descent takes too long to minimize $L(x)$
Many steps $x_{k+1} - x_k = -s_k \nabla L(x_k)$ with small improvement

**$s_k$ is too large** We are overshooting the best choice $x_{k+1}$ in the descent direction
Gradient descent will jump around the minimizing $x^*$.

Suppose the first steps $s_0$ and $s_1$ are found by line searches, and work well. We may want to stay with that learning rate for the early iterations. **Normally we reduce** $s$ as the minimization of $L(x)$ continues.

| | |
|---|---|
| **Larger steps at the start** | Get somewhere close to the optimal weights $x^*$ |
| **Smaller steps at the end** | Aim for convergence without overshoot |

A learning rate schedule $s_k = s_0/\sqrt{k}$ or $s_k = s_0/k$ systematically reduces the steps.

After reaching weights $x$ that are close to minimizing the loss function $L(x, v)$ we may want to bring new $v$'s from a **validation set**. This is not yet production mode. The purpose of **cross-validation** is to confirm that the computed weights $x$ are capable of producing accurate outputs from new data.

## Cross-validation

Cross-validation aims to estimate the validity of our model and the strength of our learning function. Is the model too weak or too simple to give accurate predictions and classifications? Are we overfitting the training data and therefore at risk with new test data? You could say that cross-validation works more carefully with a relatively small data set, so that testing and production can go forward quickly on a larger data set.

*Note* Another statistical method—for another purpose—also reuses the data. This is the *bootstrap* introduced by Brad Efron. It is used (and needed) when the sample size is small or its distribution is not known. We aim for maximum understanding by returning to the (small) sample and *reusing that data* to extract new information. Normally small data sets are not the context for applications to deep learning.

A first step in cross-validation is to divide the available data into $K$ subsets. If $K = 2$, these would essentially be the training set and test set—but we are usually aiming for more information from smaller sets before working with a big test set. *$K$-fold cross-validation* uses each of $K$ subsets separately as a test set. In every trial, the other $K - 1$ subsets form the training set. We are reworking the same data (moderate size) to learn more than one optimization can teach us.

Cross-validation can make a learning rate adaptive: changing as descent proceeds.

There are many variants, like "double cross-validation". In a standardized $m$ by $n$ least squares problem $Ax = b$, Wikipedia gives the expected value $(m - n - 1)/(m + n - 1)$ for the mean square error. Higher errors normally indicate overfitting. The corresponding test in deep learning warns us to consider earlier stopping.

This section on hyperparameters was influenced and improved by Bengio's long chapter in a remarkable book. The book title is *Neural Networks: Tricks of the Trade* (2nd edition), edited by G. Montavon, G. Orr, and K.-R. Müller. It is published by Springer (2012) with substantial contributions from leaders in the field.

## Batch Normalization of Each Layer

As training goes forward, the mean and variance of the original population can change at every layer of the network. This change in the distribution of inputs is "covariate shift". We often have to adjust the stepsize and other hyperparameters, due to this shift in the statistics of layers. A good plan is to *normalize the input to each layer.*

Normalization makes the training safer and faster. The need for dropout often disappears. Fewer iterations can now give more accurate weights. And the cost can be very moderate. *Often we just train two additional parameters on each layer.*

The problem is greatest when the nonlinear function is a sigmoid rather than ReLU. The sigmoid "saturates" by approaching a limit like 1 (while ReLU increases forever as $x \to \infty$). The nonlinear sigmoid becomes virtually linear and even constant when $x$ becomes large. Training slows down because the nonlinearity is barely used.

It remains to decide the point at which inputs will be normalized. Ioffe and Szegedy avoid computing covariance matrices (far too expensive). Their normalizing transform acts on each input $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_B$ in a minibatch of size $B$ :

**mean** $\quad \boldsymbol{\mu} = (\boldsymbol{v}_1 + \cdots + \boldsymbol{v}_B)\,/B$

**variance** $\quad \sigma^2 = \left(||\boldsymbol{v}_1 - \boldsymbol{\mu}||^2 + \cdots + ||\boldsymbol{v}_B - \boldsymbol{\mu}||^2\right)/B$

**normalize** $\quad \boldsymbol{V}_i = (\boldsymbol{v}_i - \boldsymbol{\mu})\,/\sqrt{\sigma^2 + \epsilon}$ for small $\epsilon > 0$

**scale/shift** $\quad \boldsymbol{y}_i = \boldsymbol{\gamma} \boldsymbol{V}_i + \boldsymbol{\beta}$ $\quad$ ($\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are trainable parameters)

The key point is to **normalize the inputs $\boldsymbol{y}_i$ to each new layer.** What was good for the original batch of vectors (at layer zero) is also good for the inputs to each hidden layer.

S. Ioffe and C. Szegedy, *Batch normalization*, arXiv : 1502.03167v3, 2 Mar 2015.

## Dropout

*Dropout is the removal of randomly selected neurons in the network.* Those are components of the input layer $\boldsymbol{v}_0$ or of hidden layers $\boldsymbol{v}_n$ before the output layer $\boldsymbol{v}_L$. All weights in the $A$'s and $\boldsymbol{b}$'s connected to those dropped neurons disappear from the net (Figure VII.6). Typically hidden layer neurons might be given probability $p = 0.5$ of surviving, and input components might have $p = 0.8$ or higher. *The main objective of random dropout is to avoid overfitting.* It is a relatively inexpensive averaging method compared to combining predictions from many networks.

Dropout was proposed by five leaders in the development of deep learning algorithms : N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Their paper "*Dropout*" appears in: *Journal of Machine Learning Research* 15 (2014) 1929-1958. For recent connections of dropout to physics and uncertainty see arXiv : 1506.02142 and 1809.08327.

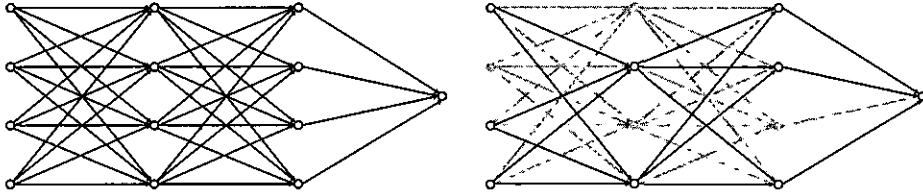


Figure VII.6: Crossed neurons have dropped out in the thinned network.

Dropout offers a way to compute with many different neural architectures at once. In training, each new $v_0$ (the feature vector of an input sample) leads to a new thinned network. Starting with $N$ neurons there are $2^N$ possible thinned networks.

At test time, we use the full network (no dropout) with weights rescaled from the training weights. The outgoing weights from an undropped neuron are multiplied by $p$ in the rescaling. This approximate averaging at test time led the five authors to reduced generalization errors—more simply than from other regularization methods.

One inspiration for dropout was genetic reproduction—where half of each parent's genes are dropped and there is a small random mutation. That dropout for a child seems more unforgiving and permanent than dropout for deep learning—which averages over many thinned networks. (True, we see some averaging over siblings. But the authors conjecture that over time, our genes are forced to be robust in order to survive.)

The dropout model uses a zero-one random variable $r$ (a Bernoulli variable). Then $r = 1$ with probability $p$ and $r = 0$ with probability $1 - p$. The usual feed-forward step to layer $n$ is $\boldsymbol{y}_n = A_n \boldsymbol{v}_{n-1} + \boldsymbol{b}_n$, followed by the nonlinear $\boldsymbol{v}_n = \mathrm{R}\boldsymbol{y}_n$. *Now a random $r$ multiplies each component of $\boldsymbol{v}_{n-1}$* to drop that neuron when $r = 0$. Component by component, $\boldsymbol{v}_{n-1}$ is multiplied by 0 or 1 to give $\boldsymbol{v}^*_{n-1}$. Then $\boldsymbol{y}_n = A_n \boldsymbol{v}^*_{n-1} + \boldsymbol{b}_n$.

To compute gradients, use backpropagation for each training example in the minibatch. Then average those gradients. Stochastic gradient descent can still include acceleration (momentum added) and adaptive descent and weight decay. The authors highly recommend regularizing the weights, for example by a maximum norm requirement $||\boldsymbol{a}|| \leq c$ on the columns of all weight matrices $A$.

## Exploring Hyperparameter Space

Often we optimize hyperparameters using experiments or experience. To decide the learning rate, we may try three possibilities and measure the drop in the loss function. A geometric sequence like $.1, .01, .001$ would make more sense than an arithmetic sequence $.05, .03, .01$. And if the smallest or largest choice gives the best results, then continue the experiment to the next number in the series. In this stepsize example, you would be considering computational cost as well as validation error.

LeCun emphasizes that for a multiparameter search, *random sampling* is the way to cover many possibilities quickly. Grid search is too slow in multiple dimensions.

## Loss Functions

The loss function *measures the difference between the correct and the computed output* for each sample. The correct output—often a classification $y = 0, 1$ or $y = 1, 2, \ldots, n$—is part of the training data. The computed output at the final layer is $w = F(x, v)$ from the learning function with weights $x$ and input $v$.

Section VI.5 defined three familiar loss functions. Then this chapter turned to the structure of the neural net and the function $F$. Here we come back to compare square loss with cross-entropy loss.

**1. Quadratic cost (square loss)**: $\ell(y, w) = \frac{1}{2}||y - w||^2$.

This is the loss function for least squares—always a possible choice. But it is not a favorite choice for deep learning. One reason is the parabolic shape for the graph of $\ell(y, w)$, as we approach zero loss at $w = y$. The derivative also approaches zero.

A zero derivative at the minimum is normal for a smooth loss function, but it frequently leads to an unwanted result: *The weights $A$ and $b$ change very slowly near the optimum.* Learning slows down and many iterations are needed.

$$\textbf{2. Cross-entropy loss} : \ell(y, w) = -\frac{1}{n}\sum_{1}^{n}\left[y_i \log z_i + (1 - y_i)\log(1 - z_i)\right] \qquad (1)$$

Here we allow and expect that the $N$ outputs $w_i$ from training the neural net have been normalized to $z(w)$, with $0 < z_i < 1$. Often those $z_i$ are probabilities. Then $1 - z_i$ is also between 0 and 1. So both logarithms in (1) are negative, and the minus sign assures that the overall loss is positive: $\ell > 0$.

More than that, the logarithms give a different and desirable approach to $z = 0$ or 1. For this calculation we refer to Nielsen's online book *Neural Networks and Deep Learning*, which focuses on sigmoid activation functions instead of ReLU. The price of those smooth functions is that they *saturate* (lose their nonlinearity) near their endpoints.

Cross-entropy has good properties, but where do the logarithms come from? The first *point is Shannon's formula for entropy (a measure of information). If message $i$ has prob*ability $p_i$, you should allow $-\log p_i$ bits for that message. Then the expected (average) number of bits per message is best possible:

$$\boxed{\textbf{Entropy} = -\sum_{1}^{m} p_i \log p_i. \text{ For } m = 2 \text{ this is } -p\log p - (1-p)\log(1-p).} \qquad (2)$$

Cross-entropy comes in when we don't know the $p_i$ and we use $\widehat{p}_i$ instead:

$$\boxed{\textbf{Cross-entropy} = -\sum_{1}^{m} p_i \log \widehat{p}_i. \text{ For } m = 2 \text{ this is } -p\log \widehat{p} - (1-p)\log(1-\widehat{p}).} \qquad (3)$$

(3) is always larger than (2). The true $p_i$ are not known and the $\widehat{p}_i$ cost more. The difference is a very useful but not symmetric function called **Kullback-Leibler (KL) divergence.**

## Regularization : $\ell^2$ or $\ell^1$ (or none)

Regularization is a voluntary but well-advised decision. It adds a *penalty term* to the loss function $L(x)$ that we minimize : an $\ell^2$ penalty in ridge regression and $\ell^1$ in LASSO.

**RR** Minimize $||b-Ax||_2^2+\lambda_2\,||x||_2^2$ **LASSO** Minimize $||b-Ax||_2^2+\lambda_1 \sum |x_i|$

The penalty controls the size of $x$. *Regularization is also called weight decay.*

The coefficient $\lambda_2$ or $\lambda_1$ is a hyperparameter. Its value can be based on cross-validation. The purpose of the penalty terms is to avoid overfitting (sometimes expressed as *fitting the noise*). Cross-validation for a given $\lambda$ finds the minimizing $x$ on a test set. Then it checks by using those weights on a training set. If it sees errors from overfitting, $\lambda$ is increased.

A small value of $\lambda$ tends to increase the variance of the error : overfitting. Large $\lambda$ will increase the bias : underfitting because the fitting term $||b - Ax||^2$ is less important.

**A different viewpoint !** Recent experiments on MNIST make it unclear if explicit regularization is always necessary. The best test performance is often seen with $\lambda = 0$ (then $x^*$ is the minimum norm solution $A^+b$). The analysis by Liang and Rakhlin identifies matrices for which this good result can be expected—provided the data leads to fast decay of the spectrum of the sample covariance matrix and the kernel matrix.

In many cases these are the matrices of Section III.3 : *Effectively low rank.* Similar ideas are increasingly heard, that deep learning with many extra weights and good hyperparameters will find solutions that generalize, without penalty.

T. Liang and A. Rakhlin, *Just interpolate : Kernel "ridgeless" regression can generalize*, arXiv : 1808.00387, 1 Aug 2018.

## The Structure of AlphaGo Zero

It is interesting to see the sequence of operations in AlphaGo Zero, learning to play Go :

1. A convolution of 256 filters of kernel size $3 \times 3$ with stride 1 : $E = 3, S = 1$
2. Batch normalization
3. ReLU
4. A convolution of 256 filters of kernel size $3 \times 3$ with stride 1
5. Batch normalization
6. A skip connection as in ResNets that adds the input to the block
7. ReLU
8. A fully connected linear layer to a hidden layer of size 256
9. ReLU

Training was by stochastic gradient descent on a fixed data set that contained the final 2 million games of self-played data from a previous run of AlphaGo Zero.

The CNN includes a fully connected layer that outputs a vector of size $19^2 + 1$. This accounts for all positions on the $19 \times 19$ board, plus a pass move allowed in Go.

# VII.5 The World of Machine Learning

Fully connected nets and convolutional nets are parts of a larger world. From training data they lead to a learning function $F(x, v)$. That function produces a close approximation to the correct output $w$ for each input $v$ ($v$ is the vector of features of that sample). But machine learning has developed a multitude of other approaches—some long established—to the problem of learning from data.

This book cannot do justice to all those ideas. It does seem useful to describe Recurrent Neural Nets (and Support Vector Machines). We also include key words to indicate the scope of machine learning. (A *glossary* is badly needed! That would be a tremendous contribution to this field.) At the end is a list of books on topics in machine learning.

## Recurrent Neural Networks (RNNs)

These networks are appropriate for data that comes in a definite order. This includes time series and natural language: *speech or text or handwriting*. In the network of connections from inputs $v$ to outputs $w$, the new feature is the *input from the previous time* $t-1$. This recurring input is determined by the function $h(t-1)$.

Figure VII.7 shows an outline of that new step in the architecture of the network.

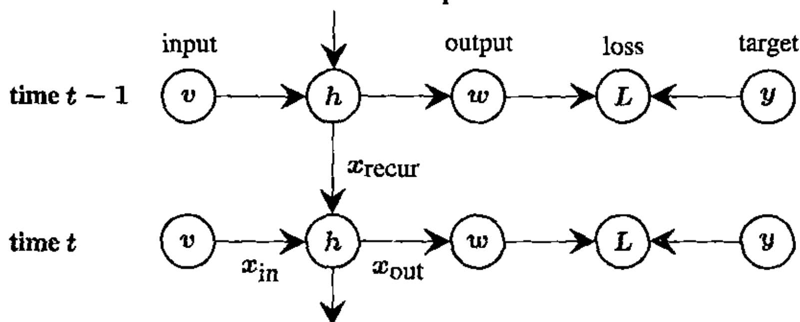



Figure VII.7: The computational graph for a recurrent network finds loss-minimizing outputs $w$ at each time $t$. The inputs to $h(t)$ are the new data $v(t)$ and the recurrent data $h(t-1)$ from the previous time. The weights multiplying the data are $x_{\text{in}}$ and $x_{\text{recur}}$ and $x_{\text{out}}$, chosen to minimize the loss $L(y-w)$. This network architecture is *universal*: It will compute any formula that is computable by a Turing machine.

## Key Words and Ideas

1 Kernel learning (next page)

2 Support Vector Machines (next page)

3 Generative Adversarial Networks

4 Independent Component Analysis

5 Graphical models

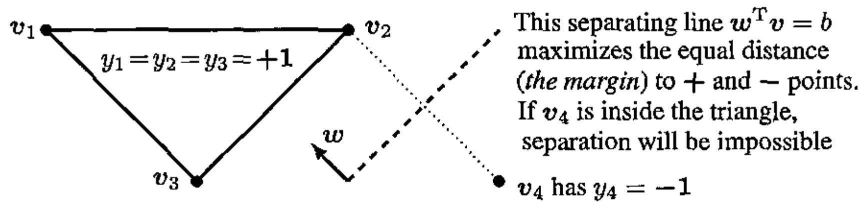
6 Bayesian statistics

7 Random forests

8 Reinforcement learning

## Support Vector Machines

Start with $n$ points $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n$ in $m$-dimensional space. Each $\boldsymbol{v}_i$ comes with a *classification* $y_i = 1$ *or* $y_i = -1$. The goal proposed by Vapnik is to find a plane $\boldsymbol{w}^{\mathrm{T}}\boldsymbol{v} = b$ in $m$ dimensions that *separates the plus points from the minus points*—if this is possible. That vector $\boldsymbol{w}$ will be perpendicular to the plane. The number $b$ tells us the distance $|b|/||\boldsymbol{w}||$ from the line or plane or hyperplane in $\mathbf{R}^m$ to the point $(0, \ldots, 0)$.




**Problem** Find $\boldsymbol{w}$ and $b$ so that $\boldsymbol{w}^{\mathrm{T}}\boldsymbol{v}_i - b$ has the correct sign $y_i$ for all points $i = 1, \ldots, n$.

If $\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3$ are plus points ($y = +1$) in a plane, then $\boldsymbol{v}_4$ *must be outside the triangle of* $\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3$. The picture shows the line of maximum separation (*maximum margin*).

**Maximum margin** Minimize $||\boldsymbol{w}||$ under the conditions $y_i(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{v}_i - b) \geq 1$.

This is a "hard margin". That inequality requires $\boldsymbol{v}_i$ to be on its correct side of the separator. If the points can't be separated, then no $\boldsymbol{w}$ and $b$ will succeed. For a "soft margin" we go ahead to choose the best available $\boldsymbol{w}$ and $b$, based on hinge loss + penalty:

$$\textbf{Soft margin} \quad \text{Minimize} \left[\frac{1}{n}\sum_{1}^{n} \max\left(0, 1 - y_i(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{v}_i - b)\right)\right] + \lambda\, ||\boldsymbol{w}||^2. \qquad (1)$$

That hinge loss (the maximum term) is zero when $\boldsymbol{v}_i$ is on the correct side of the separator. If separation is impossible, the penalty $\lambda||\boldsymbol{w}||^2$ balances hinge losses with margin sizes.

If we introduce a variable $h_i$ for that hinge loss we are minimizing a quadratic function of $\boldsymbol{w}$ with linear inequalities connecting $\boldsymbol{w}, b, y_i$ and $h_i$. This is quadratic programming in high dimensions—well understood in theory but challenging in practice.

## The Kernel Trick

SVM is linear separation. A plane separates + points from − points. The kernel trick allows a nonlinear separator, when feature vectors $\boldsymbol{v}$ are transformed to $N(\boldsymbol{v})$. Then the dot product of transformed vectors gives us the **kernel function** $\boldsymbol{K}(\boldsymbol{v}_i, \boldsymbol{v}_j) = N(\boldsymbol{v}_i)^{\mathrm{T}} N(\boldsymbol{v}_j)$.

The key is to work entirely with $K$ and not at all with the function $N$. *In fact we never see or need* $N$. In the linear case, this corresponds to choosing a positive definite $K$ and not seeing the matrix $A$ in $K = A^{\mathrm{T}}A$. The RBF kernel $\exp(-||\boldsymbol{v}_i - \boldsymbol{v}_j||^2/2\sigma^2)$ is in III.3.

M. Belkin, S. Ma, and S. Mandal, *To understand deep learning we need to understand kernel learning*, arXiv:1802.01396. "Non-smooth Laplacian kernels defeat smooth Gaussians"

T. Hofmann, B. Schölkopf, and A. J. Smola, *Kernel methods in machine learning*, Annals of Statistics **36** (2008) 1171-1220 (with extensive references).

## Google Translate

An exceptional article about deep learning and the development of Google Translate appeared in the New York Times Magazine on Sunday, 14 December 2016. It tells how Google suddenly jumped from a conventional translation to a recurrent neural network. The author Gideon Lewis-Kraus describes that event as three stories in one: the work of the development team, and the group inside Google that saw what was possible, and the worldwide community of scientists who gradually shifted our understanding of how to learn: https://www.nytimes.com/2016/12/14/magazine/the-great-AI-awakening.html

The development took less than a year. Google Brain and its competitors conceived the idea in five years. The worldwide story of machine learning is an order of magnitude longer in time and space. The key point about the recent history is the earthquake it produced in the approach to learning a language:

> Instead of programming every word and grammatical rule and exception in both languages, *let the computer find the rules.* Just give it enough correct translations.
>
> If we were recognizing images, the inputs would be many examples with correct labels (the training set). The machine creates the function $F(\boldsymbol{x}, \boldsymbol{v})$.

This is closer to how children learn. And it is closer to how we learn. If you want to teach checkers or chess, the best way is to get a board and make the moves. Play the game.

The steps from this vision to neural nets and deep learning did not come easily. Marvin Minsky was certainly one of the leaders. But his book with Seymour Papert was partly about what "Perceptrons" could not do. With only one layer, the XOR function ($A$ or $B$ but not both) was unavailable. Depth was missing and it was needed.

The lifework of Geoffrey Hinton has made an enormous difference to this subject. For machine translation, he happened to be at Google at the right time. For image recognition, he and his students won the visual recognition challenge in 2012 (with AlexNet). Its depth changed the design of neural nets. Equally impressive is a 1986 article in *Nature*, in which Rumelhart, Hinton, and Williams foresaw that backpropagation would become crucial in optimizing the weights: *Learning representations by back-propagating errors.*

These ideas led to great work worldwide. The "cat paper" in 2011-2012 described training a face detector without labeled images. The leading author was Quoc Le: *Building high-level features using large scale unsupervised learning*: **arxiv.org/abs/1112.6209.** A large data set of 200 by 200 images was sampled from YouTube. The size was managed by localizing the receptive fields. The network had one billion weights to be trained—this is still a million times smaller than the number of neurons in our visual cortex. Reading this paper, you will see the arrival of deep learning.

A small team was quietly overtaking the big team that used rules. Eventually the paper with 31 authors arrived on **arxiv.org/abs/1609.08144**. And Google had to switch to the deep network that didn't start with rules.

# Books on Machine Learning


1 Y. S. Abu-Mostafa *et al, Learning from Data*, AMLBook (2012).

2 C. C. Aggarwal, *Neural Networks and Deep Learning: A Textbook*, Springer (2018).

3 E. Alpaydim, *Introduction to Machine Learning*, MIT Press (2016).

4 E. Alpaydim, *Machine Learning : The New AI*, MIT Press (2016).

5 C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer (2006).

6 F. Chollet, *Deep Learning with Python* and *Deep Learning with R*, Manning (2017).

7 B. Efron and T. Hastie, *Computer Age Statistical Inference*, Cambridge (2016). https://web.stanford.edu/~hastie/CASI_files/PDF/casi.pdf

8 A. Géron, *Hands-On Machine Learning with Scikit-Learn and TensorFlow*, O'Reilly (2017).

9 I. Goodfellow, Y. Bengio , and A. Courville, *Deep Learning*, MIT Press (2016).

10 T. Hastie, R. Tibshirani , and J. Friedman, *The Elements of Statistical Learning : Data Mining, Inference, and Prediction*, Springer (2011).

11 M. Mahoney, J. Duchi, and A. Gilbert, editors, *The Mathematics of Data*, American Mathematical Society (2018).

12 M. Minsky and S. Papert, *Perceptrons*, MIT Press (1969).

13 A. Moitra, *Algorithmic Aspects of Machine Learning*, Cambridge (2014).

14 G. Montavon, G. Orr, and K. R. Müller, eds, *Neural Networks : Tricks of the Trade*, 2nd edition, Springer (2012).

15 M. Nielsen, *Neural Networks and Deep Learning*, (title).com (2017).

16 B. Recht and S. Wright, *Optimization for Machine Learning*, to appear.

17 A. Rosebrock, *Deep Learning for Computer Vision with Python*, pyimagsearch (2018).

18 S. Shalev-Schwartz and S. Ben-David, *Understanding Machine Learning : From Theory to Algorithms*, Cambridge (2014).

19 S. Sra, S. Nowozin, and S. Wright, eds. *Optimization for Machine Learning*, MIT Press (2012).

20 G. Strang, *Linear Algebra and Learning from Data*, Wellesley-Cambridge Press (2019).

21 V. N. Vapnik, *Statistical Learning Theory*, Wiley (1998).