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 $A_k v_{k-1} + b_k$

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

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 $\text{ReLU}(y) = \max(y, 0)$ 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.

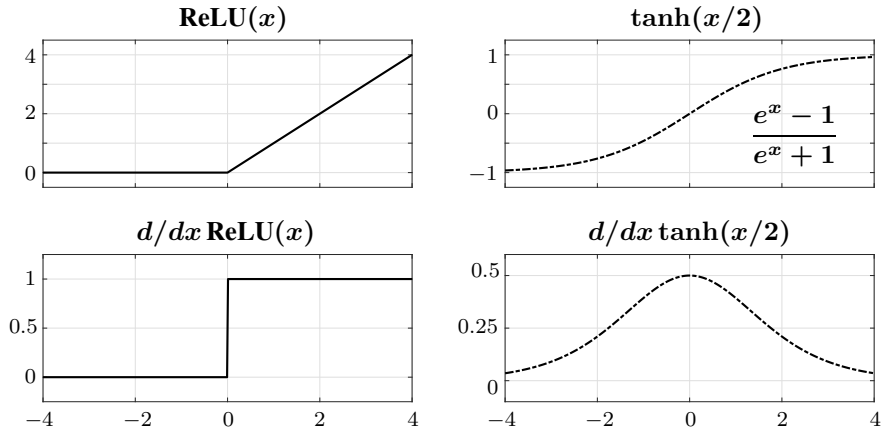


Figure VII.1: The **Re**ctified **L**inear **U**nit and a sigmoid option for nonlinearity.

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 :

$$\boxed{\textbf{Substitute } \boldsymbol{A_1 v_0 + b_1} \textbf{ into ReLU to find } \boldsymbol{v_1} \qquad (\boldsymbol{v_1})_k = \max((A_1\boldsymbol{v_0} + \boldsymbol{b}_1)_k, 0).} \quad (2)$$

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.

$$\boxed{\begin{array}{l}\text{Overall we compute } \boldsymbol{v}_2 = F(\boldsymbol{x}, \boldsymbol{v_0}) \text{ for each feature vector } \boldsymbol{v}_0 \text{ in the training set.} \\ \text{The steps are } \boldsymbol{v}_2 = A_2\boldsymbol{v}_1 + \boldsymbol{b}_2 = A_2\,(\text{ReLU}\,(A_1\boldsymbol{v}_0 + \boldsymbol{b}_1)) + \boldsymbol{b}_2 = F(\boldsymbol{x}, \boldsymbol{v_0}).\end{array}} \quad (3)$$

The goal in optimizing $\boldsymbol{x} = A_1, \boldsymbol{b}_1, A_2, \boldsymbol{b}_2$ is that the output values $\boldsymbol{v}_\ell = \boldsymbol{v}_2$ at the last layer $\ell = 2$ should correctly capture the important features of the training data $\boldsymbol{v}_0$.

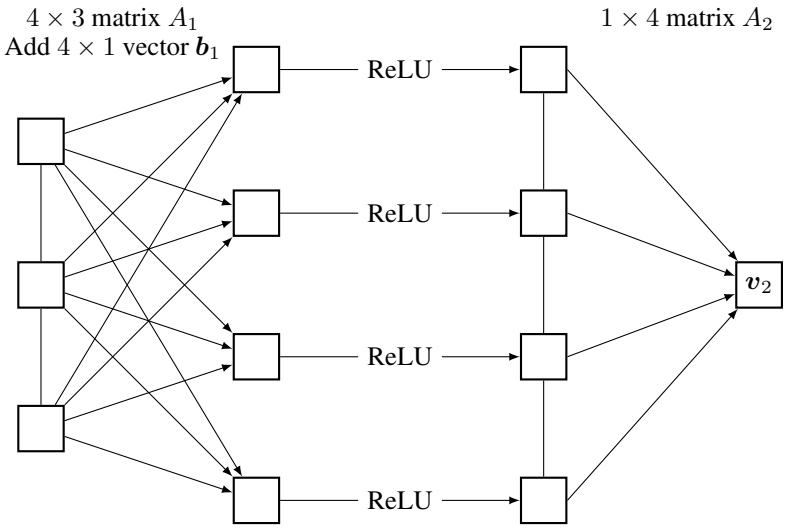


Figure VII.2: A feed-forward neural net with 4 neurons on **one internal layer**. The output $\boldsymbol{v}_2$ (plus or minus) classifies the input $\boldsymbol{v}_0$ (dog or cat). Then $\boldsymbol{v}_2$ is a composite measure of the 3-component feature vector $\boldsymbol{v}_0$. This net has 20 weights in A_k and $\boldsymbol{b}_k$.

For a **classification problem** each sample $\boldsymbol{v}_0$ of the training data is assigned $\mathbf{1}$ **or** $\mathbf{-1}$. We want the output $\boldsymbol{v}_2$ to have that correct sign (most of the time). For a **regression problem** we use the numerical value (**not just the sign**) of $\boldsymbol{v}_2$. We do not choose enough weights A_k and $\boldsymbol{b}_k$ 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 $L(x, \boldsymbol{v}_2)$ to minimize, this problem can be like least squares or entropy minimization. We are choosing $\boldsymbol{x} =$ weight matrices A_k and bias vectors $\boldsymbol{b}_k$ 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 $\boldsymbol{F}$ has "learned" the data**. This is machine learning. We don't want to choose so many weights in $\boldsymbol{x}$ 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 $F(\boldsymbol{x}, \boldsymbol{v})$. The training data goes into $\boldsymbol{v}$. Then we *initialize* the weights $\boldsymbol{x}$ in the matrices A and vectors $\boldsymbol{b}$. From those initial weights $\boldsymbol{x}_0$, the optimization algorithm (normally a form of gradient descent) computes weights $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ and onward, aiming to minimize the total loss.

The question is : *What weights $\boldsymbol{x}_0$ to start with* ? Choosing $\boldsymbol{x}_0 = 0$ would be a disaster. Poor initialization is an important cause of failure in deep learning. A proper choice of the net and the initial $\boldsymbol{x}_0$ has random (and independent) weights that meet two requirements :

1. $\boldsymbol{x}_0$ has a carefully chosen variance σ^2.

2. The hidden layers in the neural net have enough neurons (not too narrow).

Hanin and Rolnick show that the initial variance σ^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 σ^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 σ^2 of $\boldsymbol{x}_0$ 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 σ^2.

The danger from narrow hidden layers is exponentially large variance of $\boldsymbol{x}$ 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 $Q_1 Q_2 Q_3$, 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 σ^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.

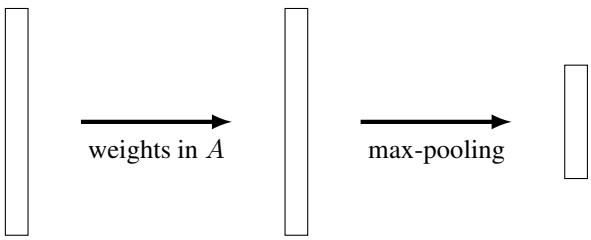
The two-step method makes clear that half or three-fourths of the information is lost. Here is a way to reduce the dimension from 128 to 64 as before, but to run less risk of destroying important information : *Max-pooling*.

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 :



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 $F(\boldsymbol{v})$ 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 $\boldsymbol{v} = \boldsymbol{v}_0$ 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 $\boldsymbol{v}_0$ contains m measurements like height, weight, age of a sample in the training set.

In the example, F had three inputs in $\boldsymbol{v}_0$ and one output $\boldsymbol{v}_2$. Its graph will be a piecewise flat surface in 4-dimensional space. The height of the graph is $\boldsymbol{v}_2 = F(\boldsymbol{v}_0)$, over the point $\boldsymbol{v}_0$ 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 $\boldsymbol{v}_0$. 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 $\boldsymbol{F = 0}$**. It is the cross-section of the graph of $z = F(\boldsymbol{v})$ 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 $\boldsymbol{b}_k$. 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. $F(\boldsymbol{x}, \boldsymbol{v})$ 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 $\boldsymbol{v}_0$ has m components and $A_1\boldsymbol{v}_0+\boldsymbol{b}_1$ has N components. We have N functions of $\boldsymbol{v}_0$. 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 $A_2(\text{ReLU}(A_1\boldsymbol{v}_0 + \boldsymbol{b}_1))$ 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 $\boldsymbol{v}$ in $\mathbf{R}^m$, suppose the graph of $F(\boldsymbol{v})$ has folds along N hyperplanes $H_1, \dots, H_N$. Those come from N linear equations $\boldsymbol{a}_i^{\mathrm{T}}\boldsymbol{v} + b_i = 0$, in other words from ReLU at N neurons. Then the number of linear pieces of F and regions bounded by the N hyperplanes is $r(N, m)$:

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

These binomial coefficients are

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

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 $\begin{pmatrix} 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \end{pmatrix} = 2$ 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 $\boldsymbol{r}$ 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 $r(N,m)$.

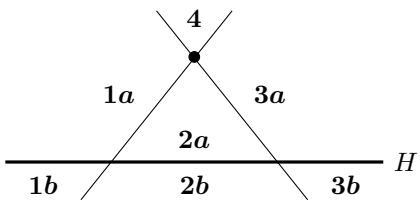
The count starts at $r(1,0) = r(0,1) = \mathbf{1}$. Then (4) is proved by induction on $N+m$:

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

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

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

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 $F(\boldsymbol{v})$ for $N \le m$ and $r \approx N^m/m!$ pieces for $N >> m$, when the hidden layer has many neurons.



Start with 2 planes

$\leftarrow r(2,2) = 4$

Add new plane H

$\leftarrow r(2,1) = 3$

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 $\boldsymbol{v}_2 = F(\boldsymbol{v}_0)$. 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 $\log\log$:

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

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 $f(\boldsymbol{v})$ of several variables. For deep learning a key question is the approximation of f by a neural net—when the weights $\boldsymbol{x}$ are chosen to bring $F(\boldsymbol{x}, \boldsymbol{v})$ close to $f(\boldsymbol{v})$.

There is a qualitative question and also a quantitative question :

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

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

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

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

Function spaces S often use the $\boldsymbol{L^2}$ or $\boldsymbol{L^1}$ or $\boldsymbol{L^\infty}$ 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 ϵ. 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 $\boldsymbol{Cn^{-r/2}}$*.

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 $\boldsymbol{b}_1$ 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 $\begin{pmatrix} N \\ k \end{pmatrix}$ in $(\boldsymbol{a} + \boldsymbol{b})^N = \sum_0^N \begin{pmatrix} N \\ k \end{pmatrix} a^k b^{N-k}$.

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 :

$\boldsymbol{M}$	The maximum $M(x, y)$ of two CPL functions $F_1(x, y)$ and $F_2(x, y)$ is CPL.
$\boldsymbol{S}$	The sum $S(x, y)$ of two CPL functions $F_1(x, y)$ and $F_2(x, y)$ is CPL.
$\boldsymbol{C}$	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 ?