Chapter 6

Deep Feedforward Netwo

Deep feedforward networks, also called **feedforward multilayer perceptrons** (MLPs), are the quintessential The goal of a feedforward network is to approximate some fu for a classifier, $y = f^*(\boldsymbol{x})$ maps an input $\boldsymbol{x}$ to a category y. defines a mapping $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$ and learns the value of the pa in the best function approximation.

These models are called **feedforward** because informa function being evaluated from $\boldsymbol{x}$, through the intermediate define f, and finally to the output $\boldsymbol{y}$. There are no **feedbac** outputs of the model are fed back into itself. When feedfo are extended to include feedback connections, they are cal **networks**, as presented in chapter 10.

Feedforward networks are of extreme importance to m

overall length of the chain gives the **depth** of the model. Th
arose from this terminology. The final layer of a feedforwar
output layer. During neural network training, we drive
The training data provides us with noisy, approximate exam
at different training points. Each example $\boldsymbol{x}$ is accompanie
The training examples specify directly what the output layer
$\boldsymbol{x}$; it must produce a value that is close to y. The behavio
not directly specified by the training data. The learning a
how to use those layers to produce the desired output, bu
not say what each individual layer should do. Instead, the le
decide how to use these layers to best implement an approx
the training data does not show the desired output for eac
are called **hidden layers**.

Finally, these networks are called *neural* because they
neuroscience. Each hidden layer of the network is typical
dimensionality of these hidden layers determines the **widt**
element of the vector may be interpreted as playing a role
Rather than thinking of the layer as representing a single ve
we can also think of the layer as consisting of many **uni**
each representing a vector-to-scalar function. Each unit
the sense that it receives input from many other units a
activation value. The idea of using many layers of vector-
is drawn from neuroscience. The choice of the functions $f^{(}$
these representations is also loosely guided by neuroscienti
the functions that biological neurons compute. Modern ne
however, is guided by many mathematical and engineerin
goal of neural networks is not to perfectly model the brain

nonlinear transformation. Equivalently, we can apply the k
section 5.7.2, to obtain a nonlinear learning algorithm based
the ϕ mapping. We can think of ϕ as providing a set of fe
as providing a new representation for $\boldsymbol{x}$.

The question is then how to choose the mapping ϕ.

1. One option is to use a very generic ϕ, such as the infi
is implicitly used by kernel machines based on the I
of high enough dimension, we can always have enou
training set, but generalization to the test set ofte
generic feature mappings are usually based only on
smoothness and do not encode enough prior informa
problems.

2. Another option is to manually engineer ϕ. Until the a
this was the dominant approach. It requires decad
each separate task, with practitioners specializing in
as speech recognition or computer vision, and with
domains.

3. The strategy of deep learning is to learn ϕ. In this app
$y = f(\boldsymbol{x}; \boldsymbol{\theta}, \boldsymbol{w}) = \phi(\boldsymbol{x}; \boldsymbol{\theta})^\top \boldsymbol{w}$. We now have parameters
ϕ from a broad class of functions, and parameters $\boldsymbol{w}$
the desired output. This is an example of a deep feed
ϕ defining a hidden layer. This approach is the only
gives up on the convexity of the training problem, bu
the harms. In this approach, we parametrize the rep
and use the optimization algorithm to find the $\boldsymbol{\theta}$ that

deterministic mappings from $\boldsymbol{x}$ to $\boldsymbol{y}$ that lack feedback conn
presented later, apply these principles to learning stochast
with feedback, and probability distributions over a single v

We begin this chapter with a simple example of a feedf
we address each of the design decisions needed to deploy a
First, training a feedforward network requires making ma
decisions as are necessary for a linear model: choosing t
function, and the form of the output units. We review these b
learning, then proceed to confront some of the design dec
to feedforward networks. Feedforward networks have intro
hidden layer, and this requires us to choose the **activatio**
be used to compute the hidden layer values. We must also
of the network, including how many layers the network sho
layers should be connected to each other, and how ma
each layer. Learning in deep neural networks requires co
of complicated functions. We present the **back-propagat**
modern generalizations, which can be used to efficiently co
Finally, we close with some historical perspective.

6.1 Example: Learning XOR

To make the idea of a feedforward network more concre
example of a fully functioning feedforward network on a ver
the XOR function.

The XOR function ("exclusive or") is an operation on
and x_2. When exactly one of these binary values is equal t

appropriate cost function for modeling binary data. More a
are described in section 6.2.2.2.

Evaluated on our whole training set, the MSE loss func

$$J(\boldsymbol{\theta}) = \frac{1}{4} \sum_{\boldsymbol{x} \in \mathbb{X}} (f^*(\boldsymbol{x}) - f(\boldsymbol{x}; \boldsymbol{\theta}))^2 .$$

Now we must choose the form of our model, $f(\boldsymbol{x}; \boldsymbol{\theta})$. S
a linear model, with $\boldsymbol{\theta}$ consisting of $\boldsymbol{w}$ and b. Our model is

$$f(\boldsymbol{x}; \boldsymbol{w}, b) = \boldsymbol{x}^\top \boldsymbol{w} + b.$$

We can minimize $J(\boldsymbol{\theta})$ in closed form with respect to $\boldsymbol{w}$ a
equations.

After solving the normal equations, we obtain $\boldsymbol{w} = \mathbf{0}$
model simply outputs 0.5 everywhere. Why does this happ
how a linear model is not able to represent the XOR funct
this problem is to use a model that learns a different fea
linear model is able to represent the solution.

Specifically, we will introduce a simple feedforward net
layer containing two hidden units. See figure 6.2 for an illu
This feedforward network has a vector of hidden units $\boldsymbol{h}$ t
function $f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})$. The values of these hidden units are
for a second layer. The second layer is the output layer of th
layer is still just a linear regression model, but now it is applie
The network now contains two functions chained together,
$y = f^{(2)}(\boldsymbol{h}; \boldsymbol{w}, b)$, with the complete model being $f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}$

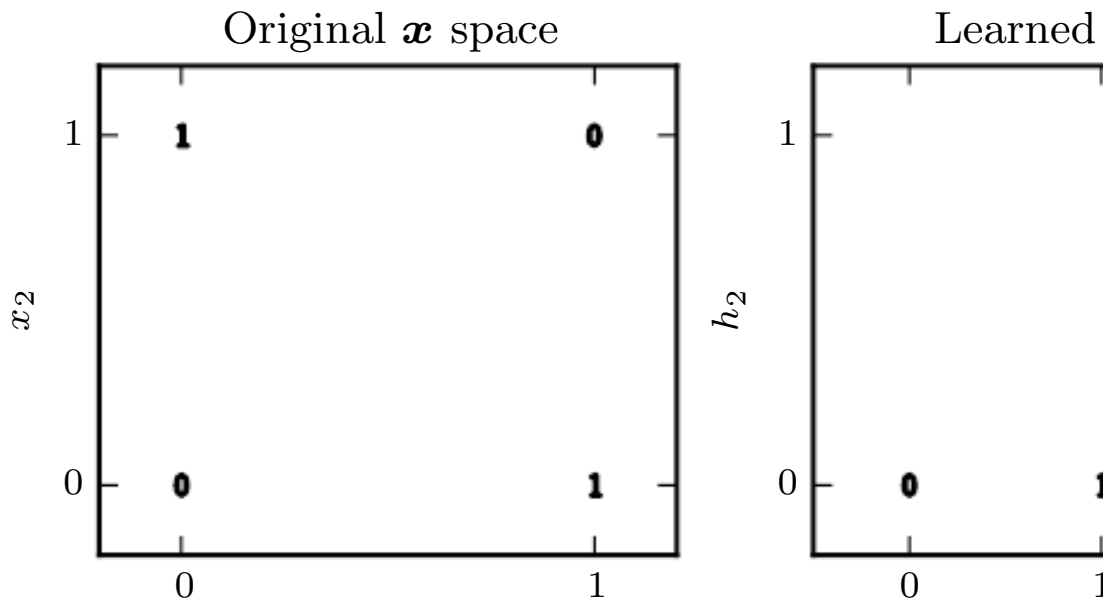


Figure 6.1: Solving the XOR problem by learning a representa
printed on the plot indicate the value that the learned function m
(Left) A linear model applied directly to the original input can
function. When $x_1 = 0$, the model's output must increase as x_2
the model's output must decrease as x_2 increases. A linear m
coefficient w_2 to x_2. The linear model therefore cannot use th
the coefficient on x_2 and cannot solve this problem. *(Right)* I
represented by the features extracted by a neural network, a lin
the problem. In our example solution, the two points that must
collapsed into a single point in feature space. In other words, th

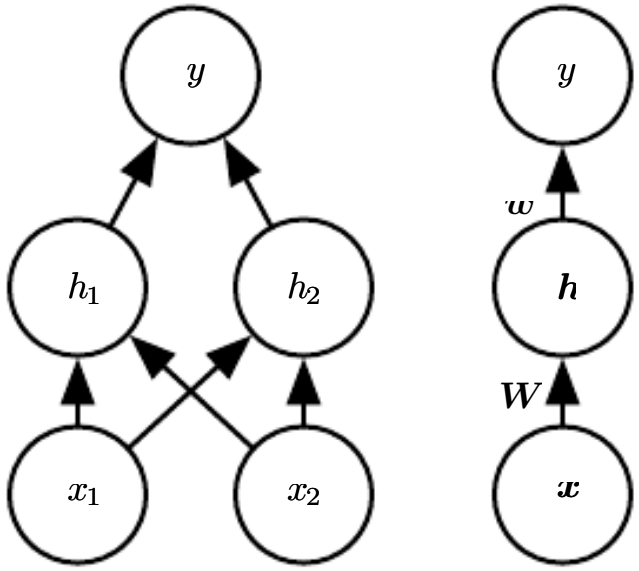
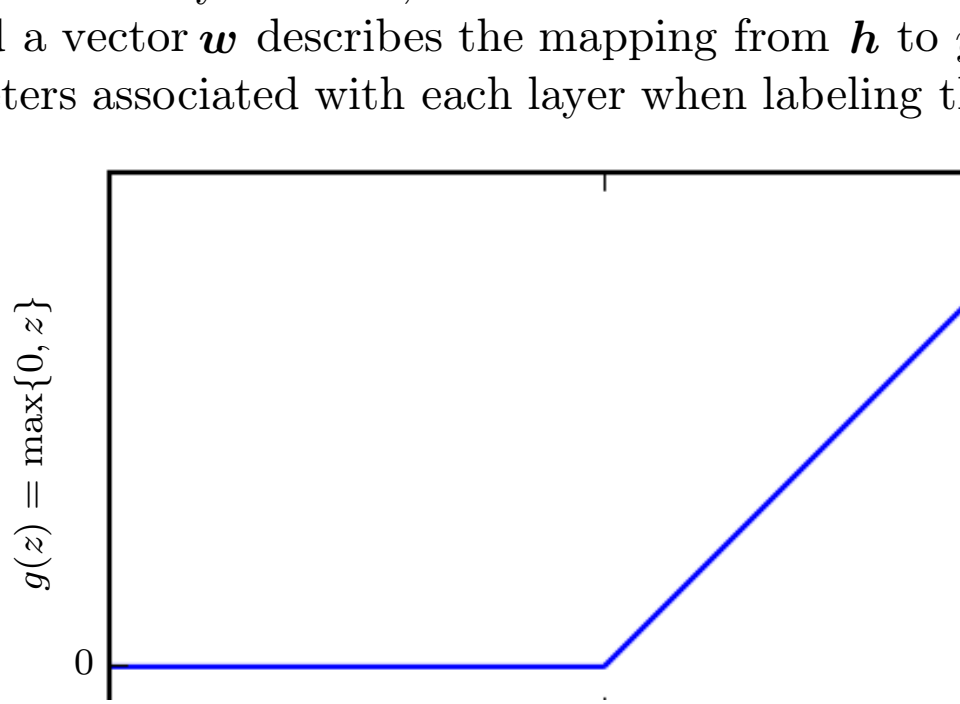


Figure 6.2: An example of a feedforward network, drawn in two dif
this is the feedforward network we use to solve the XOR exampl
layer containing two units. *(Left)* In this style, we draw every un
This style is explicit and unambiguous, but for networks larger t
consume too much space. *(Right)* In this style, we draw a node in
vector representing a layer's activations. This style is much mo
we annotate the edges in this graph with the name of the paran
relationship between two layers. Here, we indicate that a matrix $\boldsymbol{W}$
from $\boldsymbol{x}$ to $\boldsymbol{h}$, and a vector $\boldsymbol{w}$ describes the mapping from $\boldsymbol{h}$ to
intercept parameters associated with each layer when labeling t



affine transformation from an input vector to an output sc
an affine transformation from a vector $\boldsymbol{x}$ to a vector $\boldsymbol{h}$, so a
parameters is needed. The activation function g is typically
that is applied element-wise, with $h_i = g(\boldsymbol{x}^\top \boldsymbol{W}_{:,i} + c_i)$. In m
the default recommendation is to use the **rectified linear**
et al., 2009; Nair and Hinton, 2010; Glorot *et al.*, 2011a), de
function $g(z) = \max\{0, z\}$, depicted in figure 6.3.

We can now specify our complete network as

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^\top \max\{0, \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{c}\}$$

We can then specify a solution to the XOR problem. L

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix},$$

$$\boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix},$$

$$\boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix},$$

and $b = 0$.

We can now walk through how the model processes a b
be the design matrix containing all four points in the binary
example per row:

$$\boldsymbol{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}.$$

In this space, all the examples lie along a line with slope 1. line, the output needs to begin at 0, then rise to 1, then d linear model cannot implement such a function. To finish $\boldsymbol{h}$ for each example, we apply the rectified linear transform

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}.$$

This transformation has changed the relationship between t longer lie on a single line. As shown in figure 6.1, they now linear model can solve the problem.

We finish with multiplying by the weight vector $\boldsymbol{w}$:

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}.$$

The neural network has obtained the correct answer for ever

In this example, we simply specified the solution, then s zero error. In a real situation, there might be billions of billions of training examples, so one cannot simply guess here. Instead, a gradient-based optimization algorithm ca produce very little error. The solution we described to the global minimum of the loss function, so gradient descent point. There are other equivalent solutions to the XOR

The largest difference between the linear models we have networks is that the nonlinearity of a neural network cause functions to become nonconvex. This means that neural trained by using iterative, gradient-based optimizers that function to a very low value, rather than the linear equatio linear regression models or the convex optimization algorith gence guarantees used to train logistic regression or SVMs converges starting from any initial parameters (in theory— but can encounter numerical problems). Stochastic gradi nonconvex loss functions has no such convergence guarantee values of the initial parameters. For feedforward neural netw initialize all weights to small random values. The biases ma or to small positive values. The iterative gradient-based op used to train feedforward networks and almost all other dee in detail in chapter 8, with parameter initialization in p section 8.4. For the moment, it suffices to understand that is almost always based on using the gradient to descend th way or another. The specific algorithms are improvemen the ideas of gradient descent, introduced in section 4.3, and most often improvements of the stochastic gradient descent in section 5.9.

We can of course train models such as linear regressio machines with gradient descent too, and in fact this is com set is extremely large. From this point of view, training a much different from training any other model. Computing more complicated for a neural network but can still be done In Section 6.5 we describe how to obtain the gradient using

the same as those for other parametric models, such as lin

In most cases, our parametric model defines a distrib we simply use the principle of maximum likelihood. Th cross-entropy between the training data and the model's function.

Sometimes, we take a simpler approach, where rather tha probability distribution over $\boldsymbol{y}$, we merely predict some sta on $\boldsymbol{x}$. Specialized loss functions enable us to train a predic

The total cost function used to train a neural network of the primary cost functions described here with a regular already seen some simple examples of regularization appli section 5.2.2. The weight decay approach used for linear applicable to deep neural networks and is among the mo tion strategies. More advanced regularization strategies fo described in chapter 7.

6.2.1.1 Learning Conditional Distributions with M

Most modern neural networks are trained using maximum that the cost function is simply the negative log-likelihood, as the cross-entropy between the training data and the mo cost function is given by

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x}).$$

The specific form of the cost function changes from mod

mean squared error holds for a linear model, but in fact, regardless of the $f(\boldsymbol{x}; \boldsymbol{\theta})$ used to predict the mean of the G

An advantage of this approach of deriving the cost fu likelihood is that it removes the burden of designing cost fu Specifying a model $p(\boldsymbol{y} \mid \boldsymbol{x})$ automatically determines a cos

One recurring theme throughout neural network design the cost function must be large and predictable enough to for the learning algorithm. Functions that saturate (becom this objective because they make the gradient become very this happens because the activation functions used to proc hidden units or the output units saturate. The negative avoid this problem for many models. Several output units i that can saturate when its argument is very negative. T negative log-likelihood cost function undoes the exp of some discuss the interaction between the cost function and the c section 6.2.2.

One unusual property of the cross-entropy cost used likelihood estimation is that it usually does not have a minim to the models commonly used in practice. For discrete c models are parametrized in such a way that they cannot r of zero or one, but can come arbitrarily close to doing s is an example of such a model. For real-valued output v can control the density of the output distribution (for exa variance parameter of a Gaussian output distribution) the to assign extremely high density to the correct training se cross-entropy approaching negative infinity. Regularizatio in chapter 7 provide several different ways of modifying th

rather than by having a specific parametric form. From
can view the cost function as being a **functional** rather t
functional is a mapping from functions to real numbers.
learning as choosing a function rather than merely choosi
We can design our cost functional to have its minimum
function we desire. For example, we can design the cost
minimum lie on the function that maps $\boldsymbol{x}$ to the expect
Solving an optimization problem with respect to a function r
tool called **calculus of variations**, described in section 19
to understand calculus of variations to understand the cont
the moment, it is only necessary to understand that calcul
used to derive the following two results.

Our first result derived using calculus of variations is tha
tion problem

$$f^* = \arg\min_f \mathbb{E}_{\mathbf{x},\mathbf{y}\sim p_{\text{data}}} ||\boldsymbol{y} - f(\boldsymbol{x})||^2$$

yields

$$f^*(\boldsymbol{x}) = \mathbb{E}_{\mathbf{y}\sim p_{\text{data}}(\boldsymbol{y}|\boldsymbol{x})}[\boldsymbol{y}],$$

so long as this function lies within the class we optimize ove
could train on infinitely many samples from the true data g
minimizing the mean squared error cost function would give a
the mean of $\boldsymbol{y}$ for each value of $\boldsymbol{x}$.

Different cost functions give different statistics. A secon
calculus of variations is that

$$f^* = \arg\min_f \mathbb{E}_{\mathbf{x},\mathbf{y}\sim p_{\text{data}}} ||\boldsymbol{y} - f(\boldsymbol{x})||_1$$

yields a function that predicts the *median* value of $\boldsymbol{y}$ for ea
function may be described by the family of functions we op
function is commonly called **mean absolute error**.

Unfortunately, mean squared error and mean absolute e
results when used with gradient-based optimization. So
saturate produce very small gradients when combined wit
This is one reason that the cross-entropy cost function is m
squared error or mean absolute error, even when it is not ne
entire distribution $p(\boldsymbol{y} \mid \boldsymbol{x})$.

6.2.2 Output Units

The choice of cost function is tightly coupled with the choic

6.2.2.1 Linear Units for Gaussian Output Distribu

One simple kind of output unit is based on an affine tr
nonlinearity. These are often just called linear units.

Given features $\boldsymbol{h}$, a layer of linear output units produces

Linear output layers are often used to produce the
Gaussian distribution:

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; \hat{\boldsymbol{y}}, \boldsymbol{I}).$$

Maximizing the log-likelihood is then equivalent to minimi
error.

The maximum likelihood framework makes it straigh
covariance of the Gaussian too, or to make the covariance
function of the input. However, the covariance must be cons
definite matrix for all inputs. It is difficult to satisfy such c
output layer, so typically other output units are used to para
Approaches to modeling the covariance are described short

Because linear units do not saturate, they pose little
based optimization algorithms and may be used with a wide
algorithms.

6.2.2.2 Sigmoid Units for Bernoulli Output Distri

Many tasks require predicting the value of a binary vari
problems with two classes can be cast in this form.

The maximum likelihood approach is to define a Berno

outside the unit interval, the gradient of the output of the
its parameters would be $\mathbf{0}$. A gradient of $\mathbf{0}$ is typically pr
learning algorithm no longer has a guide for how to impr
parameters.

Instead, it is better to use a different approach that en
strong gradient whenever the model has the wrong answer.
on using sigmoid output units combined with maximum lik

A sigmoid output unit is defined by

$$\hat{y} = \sigma\left(\boldsymbol{w}^\top \boldsymbol{h} + b\right),$$

where σ is the logistic sigmoid function described in sectio

We can think of the sigmoid output unit as having two
uses a linear layer to compute $z = \boldsymbol{w}^\top \boldsymbol{h} + b$. Next, it uses
function to convert z into a probability.

We omit the dependence on $\boldsymbol{x}$ for the moment to di
probability distribution over y using the value z. The sign
by constructing an unnormalized probability distribution
sum to 1. We can then divide by an appropriate const
probability distribution. If we begin with the assumption tha
probabilities are linear in y and z, we can exponentiate to ob
probabilities. We then normalize to see that this yields a
controlled by a sigmoidal transformation of z:

$$\log \tilde{P}(y) = yz,$$
$$\tilde{P}(y) = \exp(yz),$$
$$\exp(yz)$$

likelihood learning of a Bernoulli parametrized by a sigmoi

$$\begin{aligned} J(\boldsymbol{\theta}) &= -\log P(y \mid \boldsymbol{x}) \\ &= -\log \sigma\left((2y-1)z\right) \\ &= \zeta\left((1-2y)z\right). \end{aligned}$$

This derivation makes use of some properties from sect
the loss in terms of the softplus function, we can see that
$(1-2y)z$ is very negative. Saturation thus occurs only w
has the right answer—when $y = 1$ and z is very positive,
negative. When z has the wrong sign, the argument to
$(1-2y)z$, may be simplified to $|z|$. As $|z|$ becomes large whi
the softplus function asymptotes toward simply returning
derivative with respect to z asymptotes to sign(z), so, in
incorrect z, the softplus function does not shrink the gradie
is useful because it means that gradient-based learning can
a mistaken z.

When we use other loss functions, such as mean squa
saturate anytime $\sigma(z)$ saturates. The sigmoid activation f
when z becomes very negative and saturates to 1 when z
The gradient can shrink too small to be useful for learnin
whether the model has the correct answer or the incorrect a
maximum likelihood is almost always the preferred approa
output units.

Analytically, the logarithm of the sigmoid is always defi
the sigmoid returns values restricted to the open interval (0
the entire closed interval of valid probabilities $[0, 1]$. In soft

can be used inside the model itself, if we wish the model to
n different options for some internal variable.

In the case of binary variables, we wished to produce a

$$\hat{y} = P(y = 1 \mid \boldsymbol{x}).$$

Because this number needed to lie between 0 and 1, and b
logarithm of the number to be well behaved for gradient-
the log-likelihood, we chose to instead predict a number
Exponentiating and normalizing gave us a Bernoulli distrib
sigmoid function.

To generalize to the case of a discrete variable with n
to produce a vector $\hat{\boldsymbol{y}}$, with $\hat{y}_i = P(y = i \mid \boldsymbol{x})$. We requ
element of $\hat{y}_i$ be between 0 and 1, but also that the entire v
it represents a valid probability distribution. The same app
the Bernoulli distribution generalizes to the multinoulli distr
layer predicts unnormalized log probabilities:

$$\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b},$$

where $z_i = \log \tilde{P}(y = i \mid \boldsymbol{x})$. The softmax function can t
normalize $\boldsymbol{z}$ to obtain the desired $\hat{\boldsymbol{y}}$. Formally, the softmax

$$\text{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}.$$

As with the logistic sigmoid, the use of the exp func
training the softmax to output a target value y using maxi
this case, we wish to maximize $\log P(y = i; \boldsymbol{z}) = \log \text{soft}$

that this term can be roughly approximated by $\max_j z_j$. based on the idea that $\exp(z_k)$ is insignificant for any z_k that $\max_j z_j$. The intuition we can gain from this approximatio log-likelihood cost function always strongly penalizes the prediction. If the correct answer already has the largest inpu the $-z_i$ term and the $\log \sum_j \exp(z_j) \approx \max_j z_j = z_i$ term This example will then contribute little to the overall train dominated by other examples that are not yet correctly cla

So far we have discussed only a single example. Overall, u likelihood will drive the model to learn parameters that drive the fraction of counts of each outcome observed in the trai

$$\text{softmax}(\boldsymbol{z}(\boldsymbol{x};\boldsymbol{\theta}))_i \approx \frac{\sum_{j=1}^{m} \mathbf{1}_{y^{(j)}=i,\boldsymbol{x}^{(j)}=}}{\sum_{j=1}^{m} \mathbf{1}_{\boldsymbol{x}^{(j)}=\boldsymbol{x}}}$$

Because maximum likelihood is a consistent estimator, this is as long as the model family is capable of representing the tr practice, limited model capacity and imperfect optimizati model is only able to approximate these fractions.

Many objective functions other than the log-likelihoo with the softmax function. Specifically, objective functions t undo the exp of the softmax fail to learn when the argume very negative, causing the gradient to vanish. In particular, loss function for softmax units and can fail to train the mod even when the model makes highly confident incorrect pre To understand why these other loss functions can fail, we softmax function itself.

Like the sigmoid, the softmax activation can saturate.

The reformulated version enables us to evaluate softmax wit
errors, even when z contains extremely large or extrem
Examining the numerically stable variant, we see that th
driven by the amount that its arguments deviate from max

An output $\text{softmax}(\boldsymbol{z})_i$ saturates to 1 when the correspo
($z_i = \max_i z_i$) and z_i is much greater than all the othe
$\text{softmax}(\boldsymbol{z})_i$ can also saturate to 0 when z_i is not maxima
much greater. This is a generalization of the way that sigm
can cause similar difficulties for learning if the loss funct
compensate for it.

The argument $\boldsymbol{z}$ to the softmax function can be produce
The most common is simply to have an earlier layer of the
every element of $\boldsymbol{z}$, as described above using the linear layer
straightforward, this approach actually overparametrizes
constraint that the n outputs must sum to 1 means that onl
necessary; the probability of the n-th value may be obtai
first $n-1$ probabilities from 1. We can thus impose a requir
of $\boldsymbol{z}$ be fixed. For example, we can require that $z_n = 0$.
what the sigmoid unit does. Defining $P(y = 1 \mid \boldsymbol{x}) = \sigma(z)$ is
$P(y = 1 \mid \boldsymbol{x}) = \text{softmax}(\boldsymbol{z})_1$ with a two-dimensional $\boldsymbol{z}$ and
argument and the n argument approaches to the softmax
set of probability distributions but have different learning
there is rarely much difference between using the overparam
restricted version, and it is simpler to implement the overp

From a neuroscientific point of view, it is interesting to t
a way to create a form of competition between the units tha
softmax outputs always sum to 1 so an increase in the value

It would perhaps be better to call the softmax function
current name is an entrenched convention.

6.2.2.4 Other Output Types

The linear, sigmoid, and softmax output units describe
common. Neural networks can generalize to almost any ki
we wish. The principle of maximum likelihood provides a g
a good cost function for nearly any kind of output layer.

In general, if we define a conditional distribution $p(\boldsymbol{y}$
maximum likelihood suggests we use $-\log p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$ as ou

In general, we can think of the neural network as represen
The outputs of this function are not direct predictions of
$f(\boldsymbol{x}; \boldsymbol{\theta}) = \boldsymbol{\omega}$ provides the parameters for a distribution ove
can then be interpreted as $-\log p(\mathbf{y}; \boldsymbol{\omega}(\boldsymbol{x}))$.

For example, we may wish to learn the variance of a cond
given $\mathbf{x}$. In the simple case, where the variance σ^2 is a con
form expression because the maximum likelihood estimator o
empirical mean of the squared difference between observation
value. A computationally more expensive approach that do
special-case code is to simply include the variance as one o
distribution $p(\mathbf{y} \mid \boldsymbol{x})$ that is controlled by $\boldsymbol{\omega} = f(\boldsymbol{x}; \boldsymbol{\theta})$. The
$-\log p(\boldsymbol{y}; \boldsymbol{\omega}(\boldsymbol{x}))$ will then provide a cost function with t
necessary to make our optimization procedure incrementally
the simple case where the standard deviation does not de
can make a new parameter in the network that is copied dir
parameter might be σ itself or could be a parameter v repr

and logarithm operations is well behaved. By comparison, output in terms of variance, we would need to use division. becomes arbitrarily steep near zero. While large gradients trarily large gradients usually result in instability. If we para terms of standard deviation, the log-likelihood would still i as squaring. The gradient through the squaring operation making it difficult to learn parameters that are squared. Re use standard deviation, variance, or precision, we must ensu matrix of the Gaussian is positive definite. Because the eige matrix are the reciprocals of the eigenvalues of the cova equivalent to ensuring that the precision matrix is positiv diagonal matrix, or a scalar times the diagonal matrix, th we need to enforce on the output of the model is positivity is the raw activation of the model used to determine the can use the softplus function to obtain a positive precision same strategy applies equally if using variance or standard precision or if using a scalar times identity rather than dia

It is rare to learn a covariance or precision matrix with diagonal. If the covariance is full and conditional, then a be chosen that guarantees positive definiteness of the predi This can be achieved by writing $\boldsymbol{\Sigma}(\boldsymbol{x}) = \boldsymbol{B}(\boldsymbol{x})\boldsymbol{B}^\top(\boldsymbol{x})$, where square matrix. One practical issue if the matrix is full rank likelihood is expensive, with a $d \times d$ matrix requiring $O(d^3$ determinant and inverse of $\boldsymbol{\Sigma}(\boldsymbol{x})$ (or equivalently, and mo eigendecomposition or that of $\boldsymbol{B}(\boldsymbol{x})$).

We often want to perform multimodal regression, th values from a conditional distribution $p(\boldsymbol{y} \mid \boldsymbol{x})$ that can

1. Mixture components $p(\mathrm{c} = i \mid \boldsymbol{x})$: these form a m
over the n different components associated with later
typically be obtained by a softmax over an n-dimensio
that these outputs are positive and sum to 1.

2. Means $\boldsymbol{\mu}^{(i)}(\boldsymbol{x})$: these indicate the center or mean as
Gaussian component and are unconstrained (typically
all for these output units). If $\mathbf{y}$ is a d-vector, then the
an $n \times d$ matrix containing all n of these d-dimensio
these means with maximum likelihood is slightly m
learning the means of a distribution with only one c
want to update the mean for the component that a
observation. In practice, we do not know which com
observation. The expression for the negative log-likeli
each example's contribution to the loss for each compo
that the component produced the example.

3. Covariances $\boldsymbol{\Sigma}^{(i)}(\boldsymbol{x})$: these specify the covariance mat
i. As when learning a single Gaussian component, we t
matrix to avoid needing to compute determinants. As w
of the mixture, maximum likelihood is complicated
partial responsibility for each point to each mixture
descent will automatically follow the correct proces
specification of the negative log-likelihood under the

It has been reported that gradient-based optimization of
mixtures (on the output of neural networks) can be unreliab
gets divisions (by the variance) which can be numerically

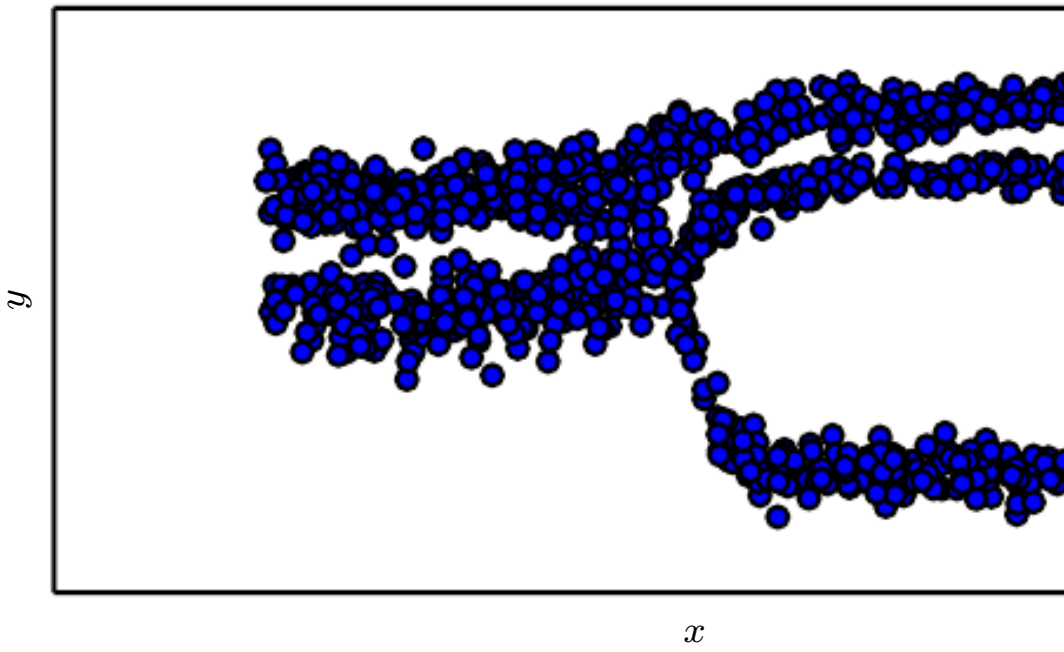


Figure 6.4: Samples drawn from a neural network with a mixtu
The input x is sampled from a uniform distribution, and the o
$p_{\text{model}}(y \mid x)$. The neural network is able to learn nonlinear ma
the parameters of the output distribution. These parameters i
governing which of three mixture components will generate th
parameters for each mixture component. Each mixture comp
predicted mean and variance. All these aspects of the output dist
with respect to the input x, and to do so in nonlinear ways.

a high degree of quality in these real-valued domains. An
density network is shown in figure 6.4.

In general, we may wish to continue to model larger vec
variables, and to impose richer and richer structures on thes
example, if we want our neural network to output a seque
forms a sentence, we might continue to use the principle o
applied to our model $p(\boldsymbol{y}; \omega(\boldsymbol{x}))$. In this case, the model we

networks: how to choose the type of hidden unit to use in t
model.

The design of hidden units is an extremely active area o
yet have many definitive guiding theoretical principles.

Rectified linear units are an excellent default choice of hi
types of hidden units are available. It can be difficult to d
which kind (though rectified linear units are usually an a
describe here some of the basic intuitions motivating eac
These intuitions can help decide when to try out which unit.
which will work best is usually impossible. The design pr
and error, intuiting that a kind of hidden unit may work v
a network with that kind of hidden unit and evaluating
validation set.

Some of the hidden units included in this list are not
at all input points. For example, the rectified linear func
is not differentiable at $z = 0$. This may seem like it inva
a gradient-based learning algorithm. In practice, gradient
well enough for these models to be used for machine lear
part because neural network training algorithms do not us
minimum of the cost function, but instead merely reduce it
shown in figure 4.3. (These ideas are described further in cha
not expect training to actually reach a point where the gradi
for the minima of the cost function to correspond to points w
Hidden units that are not differentiable are usually nond
small number of points. In general, a function $g(z)$ has a
by the slope of the function immediately to the left of z
defined by the slope of the function immediately to the r

these usually do not apply to neural network training. T
that in practice one can safely disregard the nondifferentiab
activation functions described below.

Unless indicated otherwise, most hidden units can be
a vector of inputs $\boldsymbol{x}$, computing an affine transformation
then applying an element-wise nonlinear function $g(\boldsymbol{z})$.
distinguished from each other only by the choice of the
function $g(\boldsymbol{z})$.

6.3.1 Rectified Linear Units and Their Genera

Rectified linear units use the activation function $g(z) = \max$

These units are easy to optimize because they are so simi
only difference between a linear unit and a rectified linear
linear unit outputs zero across half its domain. This makes
a rectified linear unit remain large whenever the unit is
are not only large but also consistent. The second deriv
operation is 0 almost everywhere, and the derivative of the
1 everywhere that the unit is active. This means that the g
more useful for learning than it would be with activation fu
second-order effects.

Rectified linear units are typically used on top of an af

$$\boldsymbol{h} = g(\boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b}).$$

When initializing the parameters of the affine transforma

rectification fixes $\alpha_i = -1$ to obtain $g(z) = |z|$. It is used from images (Jarrett *et al.*, 2009), where it makes sense to invariant under a polarity reversal of the input illumination of rectified linear units are more broadly applicable. A **leak** 2013) fixes α_i to a small value like 0.01, while a **parametr** treats α_i as a learnable parameter (He *et al.*, 2015).

Maxout units (Goodfellow *et al.*, 2013a) generalize further. Instead of applying an element-wise function $g(z)$, into groups of k values. Each maxout unit then outputs th one of these groups:

$$g(\boldsymbol{z})_i = \max_{j \in \mathbb{G}^{(i)}} z_j ,$$

where $\mathbb{G}^{(i)}$ is the set of indices into the inputs for group i This provides a way of learning a piecewise linear function th directions in the input $\boldsymbol{x}$ space.

A maxout unit can learn a piecewise linear, convex funct Maxout units can thus be seen as *learning the activation fun* just the relationship between units. With large enough k, a to approximate any convex function with arbitrary fidelity. layer with two pieces can learn to implement the same fu as a traditional layer using the rectified linear activation value rectification function, or the leaky or parametric Re implement a totally different function altogether. The max be parametrized differently from any of these other layer dynamics will be different even in the cases where maxout l same function of $\boldsymbol{x}$ as one of the other layer types.

Each maxout unit is now parametrized by k weight vect

the past (Goodfellow *et al.*, 2014a).

Rectified linear units and all these generalizations of t
principle that models are easier to optimize if their beha
This same general principle of using linear behavior to obt
also applies in other contexts besides deep linear networks. I
learn from sequences and produce a sequence of states and o
them, one needs to propagate information through several ti
easier when some linear computations (with some direction
magnitude near 1) are involved. One of the best-perform
architectures, the LSTM, propagates information through t
particular straightforward kind of linear activation. This
section 10.10.

6.3.2 Logistic Sigmoid and Hyperbolic Tanger

Prior to the introduction of rectified linear units, most neu
logistic sigmoid activation function

$$g(z) = \sigma(z)$$

or the hyperbolic tangent activation function

$$g(z) = \tanh(z).$$

These activation functions are closely related because tanh

We have already seen sigmoid units as output units
probability that a binary variable is 1. Unlike piecewise l
units saturate across most of their domain—they saturate

$\boldsymbol{w}^\top \boldsymbol{U}^\top \boldsymbol{V}^\top \boldsymbol{x}$ as long as the activations of the network ca
makes training the tanh network easier.

Sigmoidal activation functions are more common in se
forward networks. Recurrent networks, many probabilis
autoencoders have additional requirements that rule out
linear activation functions and make sigmoidal units more
drawbacks of saturation.

6.3.3 Other Hidden Units

Many other types of hidden units are possible but are used

In general, a wide variety of differentiable functions p
Many unpublished activation functions perform just as well a
provide a concrete example, we tested a feedforward network
on the MNIST dataset and obtained an error rate of less th
competitive with results obtained using more conventiona
During research and development of new techniques, it is
different activation functions and find that several variation
perform comparably. This means that usually new hidden u
only if they are clearly demonstrated to provide a significa
hidden unit types that perform roughly comparably to know
as to be uninteresting.

It would be impractical to list all the hidden unit types
the literature. We highlight a few especially useful and dis

One possibility is to not have an activation $g(z)$ at all.
this as using the identity function as the activation funct

these low-rank relationships are often sufficient. Linear hid
effective way of reducing the number of parameters in a ne

Softmax units are another kind of unit that is usually
described in section 6.2.2.3) but may sometimes be used as
units naturally represent a probability distribution over a
possible values, so they may be used as a kind of switch.
units are usually only used in more advanced architectures
manipulate memory, as described in section 10.12.

A few other reasonably common hidden unit types incl

- **Radial basis function** (RBF), unit: $h_i = \exp\left(-$
function becomes more active as $\boldsymbol{x}$ approaches a tem
saturates to 0 for most $\boldsymbol{x}$, it can be difficult to optim

- **Softplus**: $g(a) = \zeta(a) = \log(1+e^a)$. This is a smooth
introduced by Dugas *et al.* (2001) for function appro
and Hinton (2010) for the conditional distributions of
models. Glorot *et al.* (2011a) compared the softplus
better results with the latter. The use of the softplus i
The softplus demonstrates that the performance of
be very counterintuitive—one might expect it to ha
the rectifier due to being differentiable everywhere or
completely, but empirically it does not.

- **Hard tanh**. This is shaped similarly to the tanh and
the latter, it is bounded, $g(a) = \max(-1, \min(1, a$
by Collobert (2004).

layer being a function of the layer that preceded it. In this s
is given by

$$\boldsymbol{h}^{(1)} = g^{(1)}\left(\boldsymbol{W}^{(1)\top}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right);$$

the second layer is given by

$$\boldsymbol{h}^{(2)} = g^{(2)}\left(\boldsymbol{W}^{(2)\top}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}\right);$$

and so on.

In these chain-based architectures, the main architectu
choosing the depth of the network and the width of ea
see, a network with even one hidden layer is sufficient t
Deeper networks are often able to use far fewer units pe
parameters, as well as frequently generalizing to the test set
be harder to optimize. The ideal network architecture for
via experimentation guided by monitoring the validation se

6.4.1 Universal Approximation Properties and

A linear model, mapping from features to outputs via mat
by definition represent only linear functions. It has the adva
train because many loss functions result in convex optimi
applied to linear models. Unfortunately, we often want
nonlinear functions.

At first glance, we might presume that learning a nonli
designing a specialized model family for the kind of nonline
Fortunately, feedforward networks with hidden layers provi

to another. While the original theorems were first stated
activation functions that saturate for both very negative a
ments, universal approximation theorems have also been p
of activation functions, which includes the now commonl
unit (Leshno *et al.*, 1993).

The universal approximation theorem means that regar
we are trying to learn, we know that a large MLP will be
function. We are not guaranteed, however, that the train
able to *learn* that function. Even if the MLP is able to r
learning can fail for two different reasons. First, the optim
for training may not be able to find the value of the param
to the desired function. Second, the training algorithm m
function as a result of overfitting. Recall from section 5.2.1
theorem shows that there is no universally superior machi
Feedforward networks provide a universal system for repres
sense that, given a function, there exists a feedforward netw
the function. There is no universal procedure for exami
specific examples and choosing a function that will generali
training set.

According to the universal approximation theorem, there
enough to achieve any degree of accuracy we desire, but
say how large this network will be. Barron (1993) provide
size of a single-layer network needed to approximate a br
Unfortunately, in the worst case, an exponential number of
with one hidden unit corresponding to each input configur
distinguished) may be required. This is easiest to see in
number of possible binary functions on vectors $\boldsymbol{v} \in \{0, 1\}$

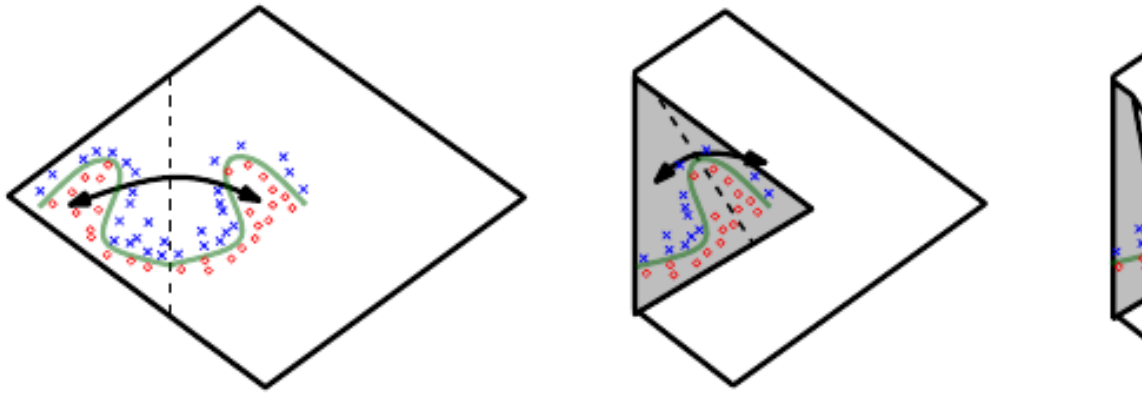


Figure 6.5: An intuitive, geometric explanation of the exponen
rectifier networks formally by Montufar *et al.* (2014). *(Left)* An ab
unit has the same output for every pair of mirror points in its
of symmetry is given by the hyperplane defined by the weights
function computed on top of that unit (the green decision surfac
of a simpler pattern across that axis of symmetry. *(Center)* The
by folding the space around the axis of symmetry. *(Right)* Anoth
be folded on top of the first (by another downstream unit) to o
(which is now repeated four times, with two hidden layers).
permission from Montufar *et al.* (2014).

were first proved for models that do not resemble the con
neural networks used for machine learning but have since b
models. The first results were for circuits of logic gates
work extended these results to linear threshold units with
(Håstad and Goldmann, 1991; Hajnal *et al.*, 1993), and t
continuous-valued activations (Maass, 1992; Maass *et al.*,
neural networks use rectified linear units. Leshno *et al.*
that shallow networks with a broad family of non-polynomi
including rectified linear units, have universal approximatio
results do not address the questions of depth or efficiency—
a sufficiently wide rectifier network could represent any fu

The main theorem in Montufar *et al.* (2014) states tha
regions carved out by a deep rectifier network with d input
per hidden layer is

$$O\left(\binom{n}{d}^{d(l-1)} n^d\right),$$

that is, exponential in depth l. In the case of maxout netw
unit, the number of linear regions is

$$O\left(k^{(l-1)+d}\right).$$

Of course, there is no guarantee that the kinds of functio
applications of machine learning (and in particular for AI)

We may also want to choose a deep model for statisti
we choose a specific machine learning algorithm, we are i
set of prior beliefs we have about what kind of function
learn. Choosing a deep model encodes a very general belie
want to learn should involve composition of several simpler
interpreted from a representation learning point of view as
the learning problem consists of discovering a set of underly
that can in turn be described in terms of other, simpler
variation. Alternately, we can interpret the use of a deep arc
a belief that the function we want to learn is a computer
multiple steps, where each step makes use of the previous
intermediate outputs are not necessarily factors of variati
analogous to counters or pointers that the network uses t
processing. Empirically, greater depth does seem to result i

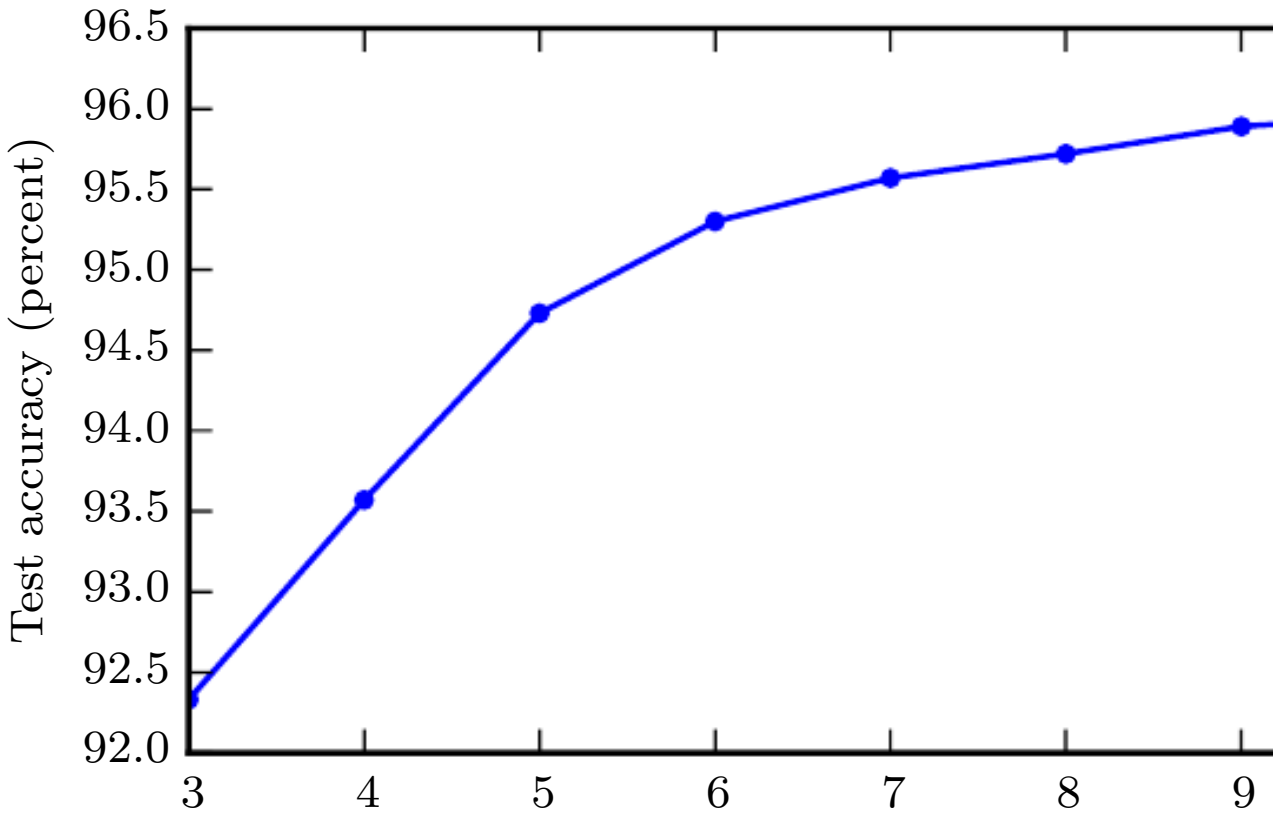


Figure 6.6: Effect of depth. Empirical results showing that dee
better when used to transcribe multidigit numbers from photogr
from Goodfellow *et al.* (2014d). The test set accuracy consistently
depth. See figure 6.7 for a control experiment demonstrating th
model size do not yield the same effect.

Many neural network architectures have been develo
Specialized architectures for computer vision called convo
described in chapter 9. Feedforward networks may also
recurrent neural networks for sequence processing, describe
have their own architectural considerations.

In general, the layers need not be connected in a chain,
most common practice. Many architectures build a main ch

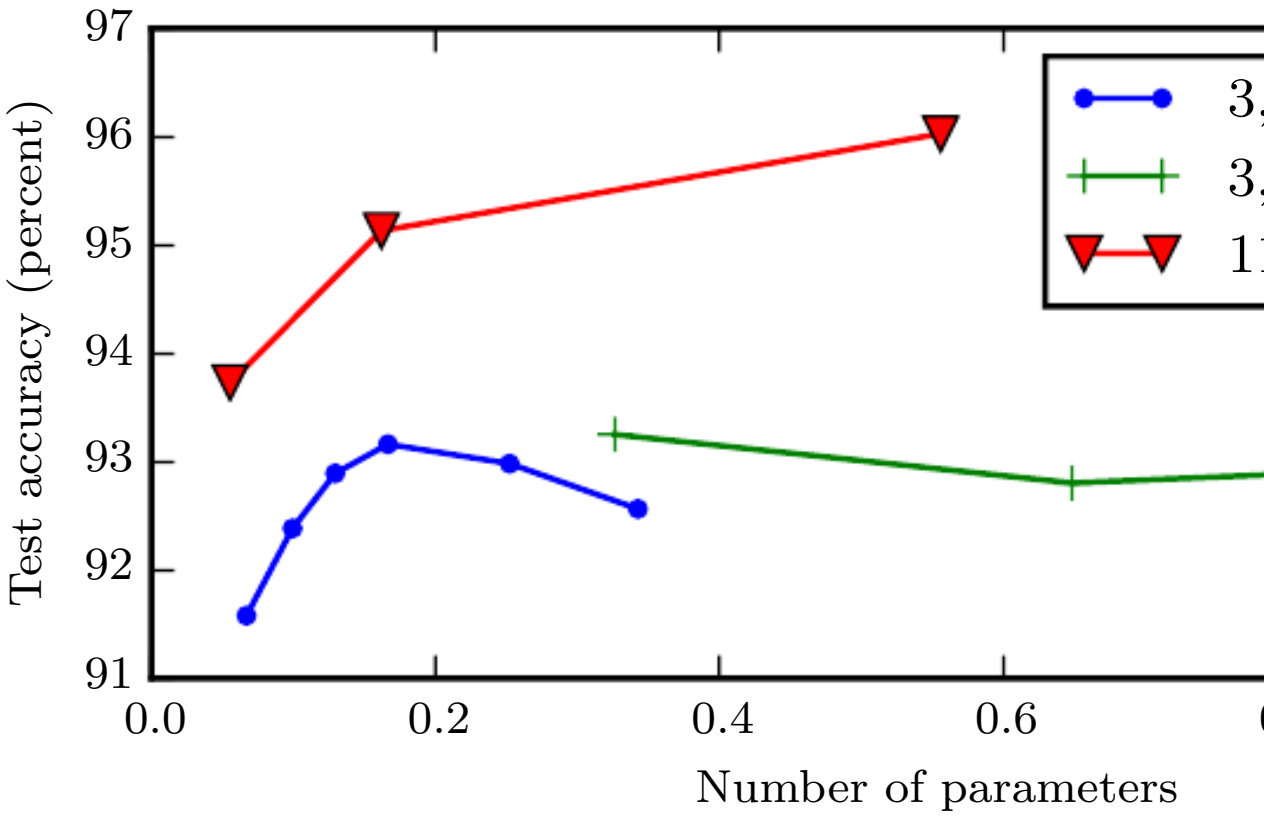


Figure 6.7: Effect of number of parameters. Deeper models
This is not merely because the model is larger. This experimen
(2014d) shows that increasing the number of parameters in layers
without increasing their depth is not nearly as effective at increas
as illustrated in this figure. The legend indicates the depth o
each curve and whether the curve represents variation in the
or the fully connected layers. We observe that shallow models
around 20 million parameters while deep ones can benefit from

networks, described in chapter 9, use specialized patterns
that are very effective for computer vision problems. In thi
to give more specific advice concerning the architecture of a
In subsequent chapters we develop the particular architectu
been found to work well for different application domains.

6.5 Back-Propagation and Other Differ Algorithms

When we use a feedforward neural network to accept an in
output $\hat{\boldsymbol{y}}$, information flows forward through the network.
the initial information that then propagates up to the hid
and finally produces $\hat{\boldsymbol{y}}$. This is called **forward propagat**
forward propagation can continue onward until it produ
The **back-propagation** algorithm (Rumelhart *et al.*, 1986
backprop, allows the information from the cost to then f
the network in order to compute the gradient.

Computing an analytical expression for the gradient i
numerically evaluating such an expression can be computat
back-propagation algorithm does so using a simple and ine

The term back-propagation is often misunderstood a
learning algorithm for multi layer neural networks. Actu
refers only to the method for computing the gradient, wh
such as stochastic gradient descent, is used to perform learn
Furthermore, back-propagation is often misunderstood as
layer neural networks, but in principle it can compute deri

f with multiple outputs. We restrict our description here
used case, where f has a single output.

6.5.1 Computational Graphs

So far we have discussed neural networks with a relatively in
To describe the back-propagation algorithm more precisely,
more precise **computational graph** language.

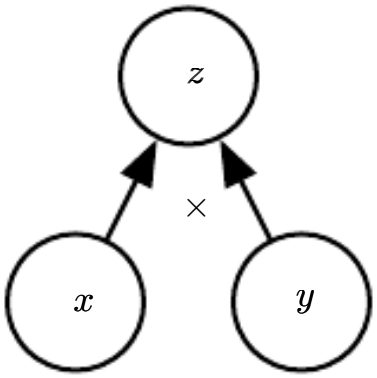
Many ways of formalizing computation as graphs are p

Here, we use each node in the graph to indicate a varia
be a scalar, vector, matrix, tensor, or even a variable of an

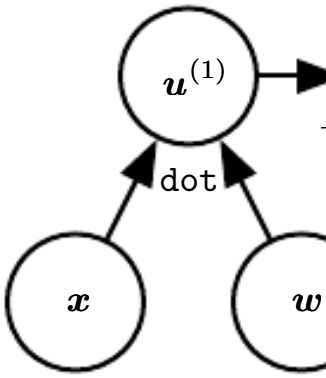
To formalize our graphs, we also need to introduce the
An operation is a simple function of one or more variables
is accompanied by a set of allowable operations. Functi
than the operations in this set may be described by compo
together.

Without loss of generality, we define an operation to
output variable. This does not lose generality because the ou
multiple entries, such as a vector. Software implementation
usually support operations with multiple outputs, but we
description because it introduces many extra details that
conceptual understanding.

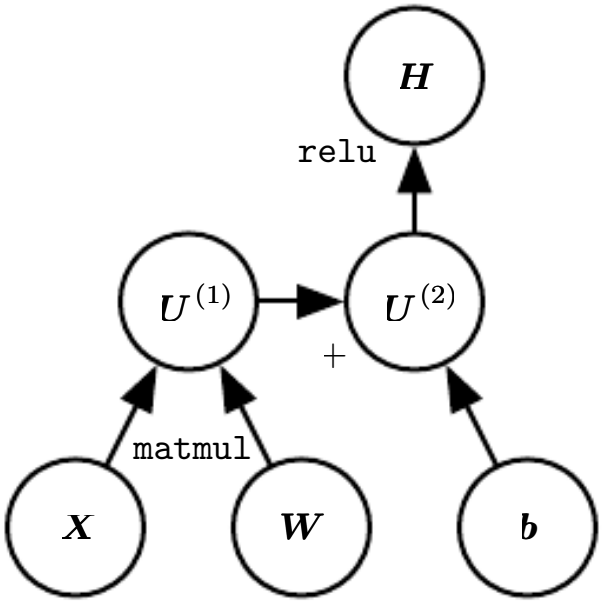
If a variable y is computed by applying an operation
we draw a directed edge from x to y. We sometimes ann
with the name of the operation applied, and other times on
operation is clear from context.



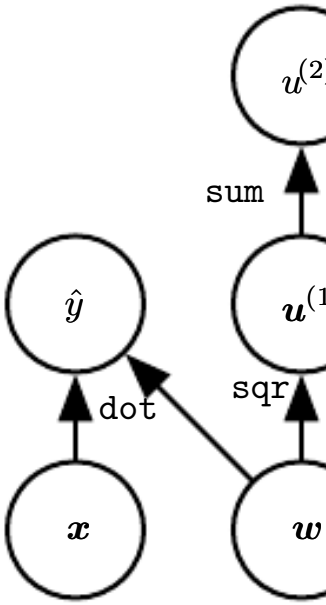
(a)



(b)



(c)



(d)

the chain rule states that

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}.$$

We can generalize this beyond the scalar case. Suppose g maps from $\mathbb{R}^m$ to $\mathbb{R}^n$, and f maps from $\mathbb{R}^n$ to $\mathbb{R}$. If $\boldsymbol{y} = g$

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}.$$

In vector notation, this may be equivalently written as

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{\top} \nabla_{\boldsymbol{y}} z,$$

where $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ is the $n \times m$ Jacobian matrix of g.

From this we see that the gradient of a variable $\boldsymbol{x}$ can be c a Jacobian matrix $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ by a gradient $\nabla_{\boldsymbol{y}} z$. The back-propaga of performing such a Jacobian-gradient product for each op

Usually we apply the back-propagation algorithm to t mensionality, not merely to vectors. Conceptually, this i back-propagation with vectors. The only difference is ho ranged in a grid to form a tensor. We could imagine flatt a vector before we run back-propagation, computing a v and then reshaping the gradient back into a tensor. In back-propagation is still just multiplying Jacobians by gra

To denote the gradient of a value z with respect to a te just as if $\mathbf{X}$ were a vector. The indices into $\mathbf{X}$ now have mu

6.5.3 Recursively Applying the Chain Rule to

Using the chain rule, it is straightforward to write down an a
the gradient of a scalar with respect to any node in the cor
produced that scalar. Actually evaluating that expression i
introduces some extra considerations.

Specifically, many subexpressions may be repeated se
overall expression for the gradient. Any procedure that c
will need to choose whether to store these subexpressions
several times. An example of how these repeated subexpre
figure 6.9. In some cases, computing the same subexpressi
be wasteful. For complicated graphs, there can be expon
wasted computations, making a naive implementation of th
In other cases, computing the same subexpression twice c
reduce memory consumption at the cost of higher runtime

We begin with a version of the back-propagation algori
actual gradient computation directly (algorithm 6.2 along wi
associated forward computation), in the order it will actually
to the recursive application of chain rule. One could either
computations or view the description of the algorithm as a
of the computational graph for computing the back-propa
formulation does not make explicit the manipulation and t
symbolic graph that performs the gradient computation.
presented in section 6.5.6, with algorithm 6.5, where we al
that contain arbitrary tensors.

First consider a computational graph describing how to c
$u^{(n)}$ (say, the loss on a training example). This scalar

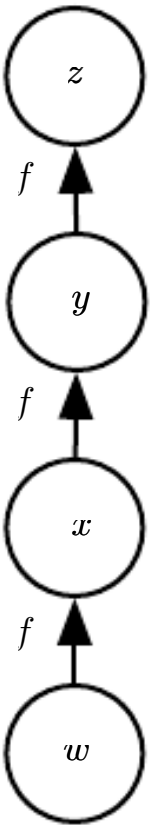


Figure 6.9: A computational graph that results in repeated subexp
the gradient. Let $w \in \mathbb{R}$ be the input to the graph. We use the s
as the operation that we apply at every step of a chain: $x = f$
To compute $\frac{\partial z}{\partial w}$, we apply equation 6.44 and obtain:

$$\begin{aligned}
&\frac{\partial z}{\partial w} \\
=&\frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \\
=&f'(y)f'(x)f'(w) \\
=&f'(f(f(w)))f'(f(w))f'(w).
\end{aligned}$$

That algorithm specifies the forward propagation compu
put in a graph $\mathcal{G}$. To perform back-propagation, we can con
graph that depends on $\mathcal{G}$ and adds to it an extra set of
subgraph $\mathcal{B}$ with one node per node of $\mathcal{G}$. Computation in
the reverse of the order of computation in $\mathcal{G}$, and each no
derivative $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ associated with the forward graph node u
the chain rule with respect to scalar output $u^{(n)}$:

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$$

as specified by algorithm 6.2. The subgraph $\mathcal{B}$ contains exa
edge from node $u^{(j)}$ to node $u^{(i)}$ of $\mathcal{G}$. The edge from $u^{(j)}$ to
the computation of $\frac{\partial u^{(i)}}{\partial u^{(j)}}$. In addition, a dot product is per
between the gradient already computed with respect to node
of $u^{(j)}$ and the vector containing the partial derivatives $\frac{\partial u^{(i}}{\partial u^{(j}}$
nodes $u^{(i)}$. To summarize, the amount of computation re
the back-propagation scales linearly with the number of
computation for each edge corresponds to computing a par
node with respect to one of its parents) as well as perform
and one addition. Below, we generalize this analysis to tens
is just a way to group multiple scalar values in the same

Algorithm 6.1 A procedure that performs the computati
$u^{(1)}$ to $u^{(n_i)}$ to an output $u^{(n)}$. This defines a computational
computes numerical value $u^{(i)}$ by applying a function $f^{(i)}$ t

efficient implementations.

The back-propagation algorithm is designed to reduce t
subexpressions without regard to memory. Specifically, it
of one Jacobian product per node in the graph. This can
that backprop (algorithm 6.2) visits each edge from noc
the graph exactly once in order to obtain the associated p
Back-propagation thus avoids the exponential explosion i
sions. Other algorithms may be able to avoid more subexp
simplifications on the computational graph, or may be abl
by recomputing rather than storing some subexpressions.
after describing the back-propagation algorithm itself.

Algorithm 6.2 Simplified version of the back-propagation a
the derivatives of $u^{(n)}$ with respect to the variables in the g
intended to further understanding by showing a simplified c
are scalars, and we wish to compute the derivatives with re
This simplified version computes the derivatives of all noc
computational cost of this algorithm is proportional to th
the graph, assuming that the partial derivative associated w
a constant time. This is of the same order as the numbe
the forward propagation. Each $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ is a function of the pa
linking the nodes of the forward graph to those added for
graph.

Run forward propagation (algorithm 6.1 for this exampl
tions of the network.

6.5.4 Back-Propagation Computation in Fully

To clarify the above definition of the back-propagation comp
the specific graph associated with a fully-connected multi l

Algorithm 6.3 first shows the forward propagation, whi
the supervised loss $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ associated with a single (input,ta
$(\boldsymbol{x}, \boldsymbol{y})$, with $\hat{\boldsymbol{y}}$ the output of the neural network when $\boldsymbol{x}$ is

Algorithm 6.4 then shows the corresponding compu
applying the back-propagation algorithm to this graph.

Algorithms 6.3 and 6.4 are demonstrations chosen to be
ward to understand. However, they are specialized to one s

Modern software implementations are based on the ge
propagation described in section 6.5.6 below, which can acc
tational graph by explicitly manipulating a data structure fo
computation.

Algorithm 6.3 Forward propagation through a typical de
the computation of the cost function. The loss $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ d
$\hat{\boldsymbol{y}}$ and on the target $\boldsymbol{y}$ (see section 6.2.1.1 for examples
obtain the total cost J, the loss may be added to a regu
contains all the parameters (weights and biases). Algorit
compute gradients of J with respect to parameters $\boldsymbol{W}$ and
demonstration uses only a single input example $\boldsymbol{x}$. Practic
use a minibatch. See section 6.5.7 for a more realistic dem

Require: Network depth, l
Require: $\boldsymbol{W}^{(i)}, i \in \{1, \ldots, l\}$, the weight matrices of the m

Algorithm 6.4 Backward computation for the deep ne
rithm 6.3, which uses, in addition to the input $\boldsymbol{x}$, a target
yields the gradients on the activations $\boldsymbol{a}^{(k)}$ for each laye
output layer and going backwards to the first hidden layer.
which can be interpreted as an indication of how each layer's
to reduce error, one can obtain the gradient on the parame
gradients on weights and biases can be immediately usec
tic gradient update (performing the update right after th
computed) or used with other gradient-based optimization

After the forward computation, compute the gradient or
$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$
for $k = l, l-1, \ldots, 1$ **do**
  Convert the gradient on the layer's output into a
  nonlinearity activation (element-wise multiplication if
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$
  Compute gradients on weights and biases (including t
  where needed):
  $\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\theta)$
  $\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\, \boldsymbol{h}^{(k-1)\top} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\theta)$
  Propagate the gradients w.r.t. the next lower-level hidd
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top}\, \boldsymbol{g}$
end for

6.5.5 Symbol-to-Symbol Derivatives

Algebraic expressions and computational graphs both op
variables that do not have specific values. These algeb

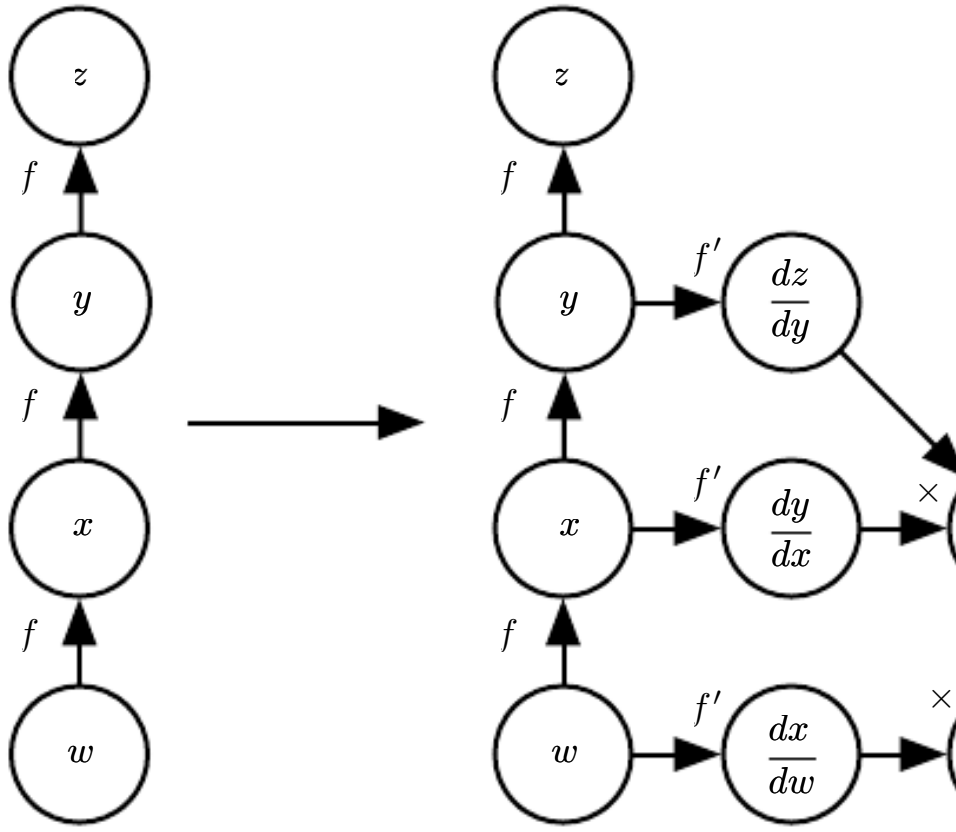


Figure 6.10: An example of the symbol-to-symbol approach to c
this approach, the back-propagation algorithm does not need t
specific numeric values. Instead, it adds nodes to a computation
to compute these derivatives. A generic graph evaluation engin
derivatives for any specific numeric values. *(Left)* In this exampl
representing $z = f(f(f(w)))$. *(Right)* We run the back-propagati
it to construct the graph for the expression corresponding to $\frac{dz}{dw}$
not explain how the back-propagation algorithm works. The pur
what the desired result is: a computational graph with a sym
derivative.

is the approach taken by Theano (Bergstra *et al.*, 2010; Ba
TensorFlow (Abadi *et al.*, 2015). An example of how it
figure 6.10. The primary advantage of this approach is th
described in the same language as the original expression. I

performing exactly the same computations as are done in
symbol-to-symbol approach. The key difference is that
approach does not expose the graph.

6.5.6 General Back-Propagation

The back-propagation algorithm is very simple. To comput
scalar z with respect to one of its ancestors $\boldsymbol{x}$ in the graph,
that the gradient with respect to z is given by $\frac{dz}{dz} = 1$.
the gradient with respect to each parent of z in the gra
current gradient by the Jacobian of the operation that pro
multiplying by Jacobians, traveling backward through the
we reach $\boldsymbol{x}$. For any node that may be reached by going ba
two or more paths, we simply sum the gradients arriving
that node.

More formally, each node in the graph $\mathcal{G}$ corresponds to
maximum generality, we describe this variable as being a
general can have any number of dimensions. They subsum
matrices.

We assume that each variable V is associated with the

- `get_operation`(V): This returns the operation th
  sented by the edges coming into V in the computation
  there may be a Python or C++ class representing the
  operation, and the `get_operation` function. Suppose
  is created by matrix multiplication, $\boldsymbol{C} = \boldsymbol{AB}$. The
  returns a pointer to an instance of the corresponding

of a scalar z with respect to $\boldsymbol{C}$ is given by $\boldsymbol{G}$. The matrix m
is responsible for defining two back-propagation rules, on
arguments. If we call the `bprop` method to request the gr
$\boldsymbol{A}$ given that the gradient on the output is $\boldsymbol{G}$, then the
matrix multiplication operation must state that the gradi
is given by $\boldsymbol{G}\boldsymbol{B}^\top$. Likewise, if we call the `bprop` method t
with respect to $\boldsymbol{B}$, then the matrix operation is responsibl
`bprop` method and specifying that the desired gradient is
back-propagation algorithm itself does not need to know any
only needs to call each operation's `bprop` rules with the righ
$\texttt{op.bprop}(\texttt{inputs}, \mathsf{X}, \mathsf{G})$ must return

$$\sum_i \left(\nabla_{\mathsf{X}} \texttt{op.f}(\texttt{inputs})_i\right) G_i,$$

which is just an implementation of the chain rule as expre
Here, `inputs` is a list of inputs that are supplied to the
mathematical function that the operation implements, X is th
we wish to compute, and G is the gradient on the output o

The `op.bprop` method should always pretend that all
from each other, even if they are not. For example, if the
two copies of x to compute x^2, the `op.bprop` method shou
derivative with respect to both inputs. The back-propagati
add both of these arguments together to obtain $2x$, whi
derivative on x.

Software implementations of back-propagation usually p
tions and their `bprop` methods, so that users of deep learnin
able to back-propagate through graphs built using common

Algorithm 6.5 The outermost skeleton of the back-propag
portion does simple setup and cleanup work. Most of the in
in the `build_grad` subroutine of algorithm 6.6

Require: $\mathbb{T}$, the target set of variables whose gradients m
Require: $\mathcal{G}$, the computational graph
Require: z, the variable to be differentiated
  Let $\mathcal{G}'$ be $\mathcal{G}$ pruned to contain only nodes that are ancesto
  of nodes in $\mathbb{T}$.
  Initialize `grad_table`, a data structure associating tensc
  `grad_table`$[z] \leftarrow 1$
  for $\mathbf{V}$ in $\mathbb{T}$ **do**
    `build_grad`$(\mathbf{V}, \mathcal{G}, \mathcal{G}', \texttt{grad\_table})$
  end for
  Return `grad_table` restricted to $\mathbb{T}$

Algorithm 6.6 The inner loop subroutine `build_grad`($\mathbf{V}$,
the back-propagation algorithm, called by the back-propaga
in algorithm 6.5.

Require: $\mathbf{V}$, the variable whose gradient should be added
Require: $\mathcal{G}$, the graph to modify
Require: $\mathcal{G}'$, the restriction of $\mathcal{G}$ to nodes that participate
Require: `grad_table`, a data structure mapping nodes to
  if $\mathbf{V}$ is in `grad_table` **then**
    Return `grad_table`$[\mathbf{V}]$
  end if
  $i \leftarrow 1$
  for $\mathbf{C}$ in `get_consumers`$(\mathbf{V}, \mathcal{G}')$ **do**

of operations executed. Keep in mind here that we refer t
fundamental unit of our computational graph, which mig
several arithmetic operations (for example, we might have a g
multiplication as a single operation). Computing a gradient i
will never execute more than $O(n^2)$ operations or store th
$O(n^2)$ operations. Here we are counting operations in the co
individual operations executed by the underlying hardware
remember that the runtime of each operation may be highly
multiplying two matrices that each contain millions of entri
a single operation in the graph. We can see that computing t
most $O(n^2)$ operations because the forward propagation stag
all n nodes in the original graph (depending on which value
we may not need to execute the entire graph). The back-p
adds one Jacobian-vector product, which should be expresse
edge in the original graph. Because the computational grap
graph it has at most $O(n^2)$ edges. For the kinds of graphs t
in practice, the situation is even better. Most neural netw
roughly chain-structured, causing back-propagation to have
better than the naive approach, which might need to execu
nodes. This potentially exponential cost can be seen by ex
the recursive chain rule (equation 6.53) nonrecursively:

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{\substack{\text{path } (u^{(\pi_1)}, u^{(\pi_2)}, \ldots, u^{(\pi_t)}), \\ \text{from } \pi_1 = j \text{ to } \pi_t = n}} \prod_{k=2}^{t} \frac{\partial u^{(\pi}}{\partial u^{(\pi_k}}$$

Since the number of paths from node j to node n can gro
length of these paths, the number of terms in the above sur

6.5.7 Example: Back-Propagation for MLP Tr

As an example, we walk through the back-propagation alg
train a multilayer perceptron.

Here we develop a very simple multilayer perceptron
layer. To train this model, we will use minibatch stocha
The back-propagation algorithm is used to compute the gr
single minibatch. Specifically, we use a minibatch of exam
set formatted as a design matrix $\boldsymbol{X}$ and a vector of ass
The network computes a layer of hidden features $\boldsymbol{H} =$
simplify the presentation we do not use biases in this mode
graph language includes a `relu` operation that can compu
wise. The predictions of the unnormalized log probabilitie
given by $\boldsymbol{H}\boldsymbol{W}^{(2)}$. We assume that our graph language incl
operation that computes the cross-entropy between the targe
distribution defined by these unnormalized log probabilitie
entropy defines the cost J_{MLE}. Minimizing this cross-entro
likelihood estimation of the classifier. However, to make this
we also include a regularization term. The total cost

$$J = J_{\text{MLE}} + \lambda \left(\sum_{i,j} \left(W_{i,j}^{(1)} \right)^2 + \sum_{i,j} \left(W_{i,j}^{(2)} \right.$$

consists of the cross-entropy and a weight decay term w
computational graph is illustrated in figure 6.11.

The computational graph for the gradient of this examp
it would be tedious to draw or to read. This demonstrat
of the back-propagation algorithm, which is that it can a

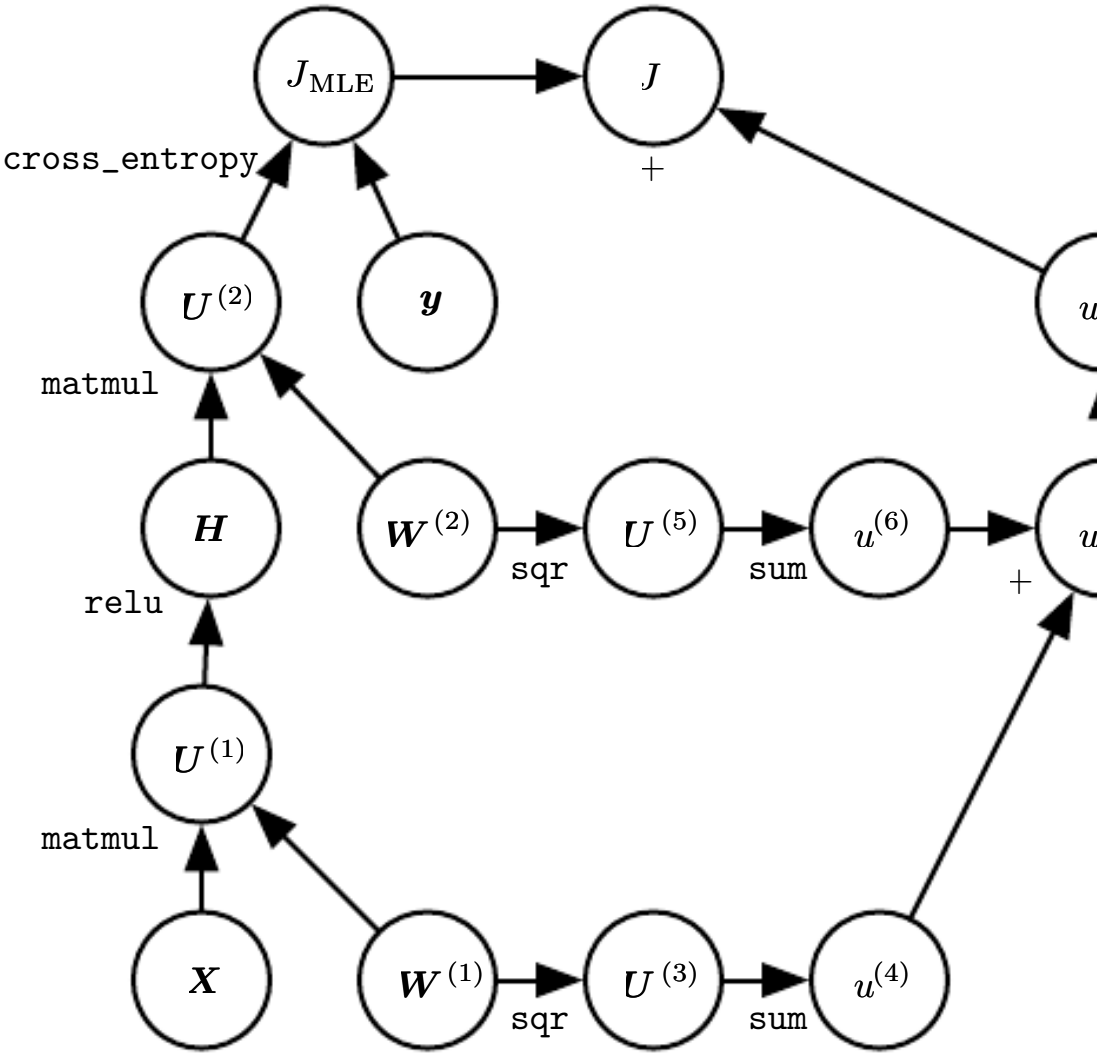


Figure 6.11: The computational graph used to compute the cost
single-layer MLP using the cross-entropy loss and weight decay.

explore two different branches. On the shorter branch,
gradient on $\boldsymbol{W}^{(2)}$, using the back-propagation rule for th
the matrix multiplication operation. The other branch cor
chain descending further along the network. First, the back
computes $\nabla_{\boldsymbol{H}} J = \boldsymbol{G}\boldsymbol{W}^{(2)\top}$ using the back-propagation rule
to the matrix multiplication operation. Next, the `relu` o
propagation rule to zero out components of the gradient co

This value is stored from the time it is computed until t
returned to the same point. The memory cost is thus $O($
number of examples in the minibatch and n_h is the numbe

6.5.8 Complications

Our description of the back-propagation algorithm here is s
mentations actually used in practice.

As noted above, we have restricted the definition of
function that returns a single tensor. Most software imp
support operations that can return more than one tensor. F
to compute both the maximum value in a tensor and the in
best to compute both in a single pass through memory, so
implement this procedure as a single operation with two ou

We have not described how to control the memory
propagation. Back-propagation often involves summation of
In the naive approach, each of these tensors would be com
all of them would be added in a second step. The naive a
high memory bottleneck that can be avoided by maintain
adding each value to that buffer as it is computed.

Real-world implementations of back-propagation also n
data types, such as 32-bit floating point, 64-bit floating po
The policy for handling each of these types takes special ca

Some operations have undefined gradients, and it is im
cases and determine whether the gradient requested by the

Various other technicalities make real-world differentia

mode accumulation. Other approaches evaluate the subexpressions of the chain rule in different orders. In general, determining the order of operations that results in the lowest computational cost is a difficult problem. Finding the optimal sequence of operations to compute the gradient is NP-complete (Naumann, 2008), in the sense that it may require simplifying algebraic expressions into their least expensive form.

For example, suppose we have variables $p_1, p_2, \ldots, p_n$ representing probabilities and variables $z_1, z_2, \ldots, z_n$ representing unnormalized log probabilities. Suppose we define

$$q_i = \frac{\exp(z_i)}{\sum_i \exp(z_i)},$$

where we build the softmax function out of exponentiation, summation and division operations, and construct a cross-entropy loss $J = -\sum$ mathematician can observe that the derivative of J with respect to z_i has a very simple form: $q_i - p_i$. The back-propagation algorithm is not designed to simplify the gradient this way and will instead explicitly propagate gradients through the logarithm and exponentiation operations in the original graph. Software libraries such as Theano (Bergstra *et al.*, 2010; Bastien *et al.*, 2012) are able to perform some kinds of algebraic substitution to improve over the graph produced by the pure back-propagation algorithm.

When the forward graph $\mathcal{G}$ has a single output node and each partial derivative $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ can be computed with a constant amount of computation, guarantees that the number of computations for the gradient is at most the same order as the number of computations for the forward graph. This can be seen in algorithm 6.2, because each local partial derivative needs to be computed only once along with an associated multiplication for the recursive chain-rule formulation (equation 6.53). The

accumulation has been proposed for obtaining real-time co
in recurrent networks, for example (Williams and Zipser, 198
avoids the need to store the values and gradients for the w
computational efficiency for memory. The relationship betw
backward mode is analogous to the relationship between l
right-multiplying a sequence of matrices, such as

$$\boldsymbol{ABCD},$$

where the matrices can be thought of as Jacobian. For exa
vector while $\boldsymbol{A}$ has many rows, the graph will have a single o
and starting the multiplications from the end and going b
matrix-vector products. This order corresponds to the bac
starting to multiply from the left would involve a series of m
which makes the whole computation much more expensive
than $\boldsymbol{D}$ has columns, however, it is cheaper to run the mult
corresponding to the forward mode.

In many communities outside machine learning, it is more
differentiation software that acts directly on traditional p
code, such as Python or C code, and automatically ge
differentiate functions written in these languages. In the de
computational graphs are usually represented by explicit d
by specialized libraries. The specialized approach has the
the library developer to define the `bprop` methods for every
the user of the library to only those operations that have
specialized approach also has the benefit of allowing custom
rules to be developed for each operation, enabling the devel
or stability in nonobvious ways that an automatic procedure

These libraries use the same kind of data structure to descr
derivatives as they use to describe the original function bei
means that the symbolic differentiation machinery can be

In the context of deep learning, it is rare to compute a s
of a scalar function. Instead, we are usually interested in pr
matrix. If we have a function $f : \mathbb{R}^n \to \mathbb{R}$, then the Hessian
In typical deep learning applications, n will be the numbe
model, which could easily number in the billions. The en
thus infeasible to even represent.

Instead of explicitly computing the Hessian, the typical
is to use **Krylov methods**. Krylov methods are a set of i
performing various operations, such as approximately invert
approximations to its eigenvectors or eigenvalues, withou
other than matrix-vector products.

To use Krylov methods on the Hessian, we only nee
pute the product between the Hessian matrix $\boldsymbol{H}$ and an
straightforward technique (Christianson, 1992) for doing so

$$\boldsymbol{H}\boldsymbol{v} = \nabla_{\boldsymbol{x}} \left[\left(\nabla_{\boldsymbol{x}} f(x) \right)^\top \boldsymbol{v} \right].$$

Both gradient computations in this expression may be comp
the appropriate software library. Note that the outer gradie
gradient of a function of the inner gradient expression.

If $\boldsymbol{v}$ is itself a vector produced by a computational gra
specify that the automatic differentiation software should no
the graph that produced $\boldsymbol{v}$.

While computing the Hessian is usually not advisable, i

have long been used to solve optimization problems in clo
descent was not introduced as a technique for iteratively app
to optimization problems until the nineteenth century (Ca

Beginning in the 1940s, these function approximation t
motivate machine learning models such as the perceptron.
models were based on linear models. Critics including Marv
several of the flaws of the linear model family, such as its
XOR function, which led to a backlash against the entire ne

Learning nonlinear functions required the developme
ceptron and a means of computing the gradient through s
applications of the chain rule based on dynamic program
in the 1960s and 1970s, mostly for control applications (Ke
Denham, 1961; Dreyfus, 1962; Bryson and Ho, 1969; Drey
sensitivity analysis (Linnainmaa, 1976). Werbos (1981) pr
techniques to training artificial neural networks. The idea
in practice after being independently rediscovered in differe
Parker, 1985; Rumelhart *et al.*, 1986a). The book **Paral**
cessing presented the results of some of the first succes
back-propagation in a chapter (Rumelhart *et al.*, 1986b) th
to the popularization of back-propagation and initiated a v
search in multilayer neural networks. The ideas put forward
book, particularly by Rumelhart and Hinton, go much bey
They include crucial ideas about the possible computatio
several central aspects of cognition and learning, which
"connectionism" because of the importance this school of
connections between neurons as the locus of learning and
these ideas include the notion of distributed representation

number of algorithmic changes have also improved the p
networks noticeably.

One of these algorithmic changes was the replacement
with the cross-entropy family of loss functions. Mean square
the 1980s and 1990s but was gradually replaced by cross-
principle of maximum likelihood as ideas spread between th
and the machine learning community. The use of cross-
improved the performance of models with sigmoid and so
had previously suffered from saturation and slow learning
squared error loss.

The other major algorithmic change that has greatly imp
of feedforward networks was the replacement of sigmoid hidd
linear hidden units, such as rectified linear units. Rectificati
function was introduced in early neural network models and d
as the cognitron and neocognitron (Fukushima, 1975, 1980).
not use rectified linear units but instead applied rectification
Despite the early popularity of rectification, it was largely
in the 1980s, perhaps because sigmoids perform better whe
very small. As of the early 2000s, rectified linear units we
a somewhat superstitious belief that activation functions
points must be avoided. This began to change in about 200
observed that "using a rectifying nonlinearity is the single
in improving the performance of a recognition system," a
factors of neural network architecture design.

For small datasets, Jarrett *et al.* (2009) observed tha
linearities is even more important than learning the weigh
Random weights are sufficient to propagate useful informat

are completely inactive. (2) For some inputs, a biologic
proportional to its input. (3) Most of the time, biological
regime where they are inactive (i.e., they should have **spa**

When the modern resurgence of deep learning began
networks continued to have a bad reputation. From abou
widely believed that feedforward networks would not perfor
assisted by other models, such as probabilistic models. T
that with the right resources and engineering practices,
perform very well. Today, gradient-based learning in feedfo
as a tool to develop probabilistic models, such as the va
and generative adversarial networks, described in chapter
viewed as an unreliable technology that must be supporte
gradient-based learning in feedforward networks has been
powerful technology that can be applied to many other mac
2006, the community used unsupervised learning to suppo
and now, ironically, it is more common to use supervise
unsupervised learning.

Feedforward networks continue to have unfulfilled pote
expect they will be applied to many more tasks, and that ad
algorithms and model design will improve their performa
chapter has primarily described the neural network fam
subsequent chapters, we turn to how to use these models—
train them.