EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

MATHEMATISCH-
NATURWISSENSCHAFTLICHE FAKULTÄT

Deep Neural Networks

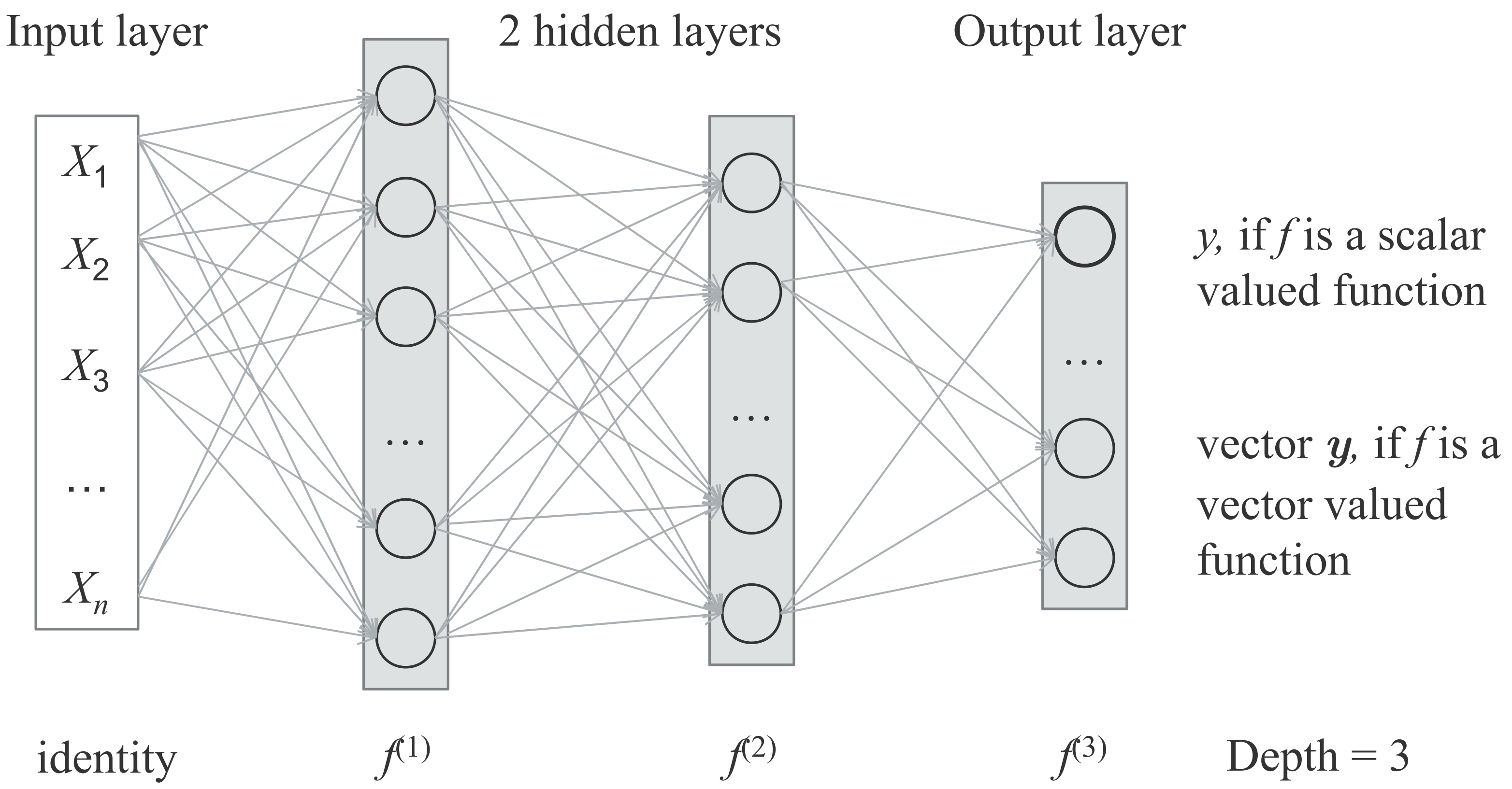
Chapter 6: Deep Feedforward Networks

Deep Feedforward Networks

- Deep feedforward networks, also often called feedforward neural networks, or multilayer perceptrons (MLPs), are now popular deep learning models.
- A feedforward network defines a mapping $\boldsymbol{y} = f(\boldsymbol{x};\boldsymbol{\theta})$ and learns the value of the parameters $\boldsymbol{\theta}$ that result in the best function approximation.
- Feedforward models: information only flows from input $\boldsymbol{x}$ through intermediate computations to output $\boldsymbol{y}$. There are no feedback connections.
- The network consists of different layers. Each layer may be described by a function $f^{(i)}$. $f^{(1)}$ is the first layer of the network, $f^{(2)}$ the second, etc.

$$\boldsymbol{y} = f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$$

Deep Feedforward Networks



$$\boldsymbol{y} = f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$$

Deep Feedforward Networks

- The overall length of the chain (highest index) gives the depth of the model.
- During neural network training, we drive $f(x)$ to match $f^*(x)$. Each example x is accompanied by a label y, which gives the output value of the network for input x.
- The learning algorithm must decide how to use these layers to best implement an approximation of f^*
- Because the training data does not show the desired output for each of these layers, these layers are called hidden layers.
- These networks are called *neural* because they are loosely inspired by neuroscience.

Deep Feedforward Networks

- To extend linear models to represent nonlinear functions of x, we can apply the linear model to a transformed input $\phi(x)$, where ϕ is a nonlinear transformation.
- Options how to choose the mapping ϕ:
  1. Use a very generic ϕ, such as the infinite-dimensional ϕ that is implicitly used by kernel machines based on the RBF kernel. If $\phi(x)$ is of high enough dimension, we always have enough capacity to fit the training set, but generalization to the test set often remains poor.
  2. Manually engineer ϕ. Before deep learning, this was the dominant approach. This requires a lot of human effort for each separate task, with experts in different domains such as speech recognition or computer vision.

Deep Feedforward Networks

3. The strategy of deep learning is to learn ϕ. In this approach, we have a model $y = f(x;\boldsymbol{\theta},\boldsymbol{w}) = \phi(x,\boldsymbol{\theta})^T \boldsymbol{w}$. We now have parameters $\boldsymbol{\theta}$ that we use to learn ϕ from a broad class of functions, and parameters $\boldsymbol{w}$ that map from $\phi(\boldsymbol{x})$ to the desired output. This is an example of a deep feedforward network, with ϕ defining a hidden layer. We parametrize the representation as $\phi(\boldsymbol{x};\ \boldsymbol{\theta})$ and use the optimization algorithm to find the $\boldsymbol{\theta}$ that corresponds to a good representation. This approach can be highly generic by using a broad family $\phi(\boldsymbol{x};\ \boldsymbol{\theta})$. But also human practitioners can encode their knowledge to help generalization by designing families $\phi(\boldsymbol{x};\ \boldsymbol{\theta})$ that they expect will perform well.

6.1 Example: Learning XOR

- We learn the XOR function with a feedforward network.
- The XOR function is the target function $y = f^*(\boldsymbol{x})$ that we want to learn.
- Our model provides a function $y = f(\boldsymbol{x}; \boldsymbol{\theta})$, our learning algorithm will adapt the parameters $\boldsymbol{\theta}$ to make f as similar as possible to f^*.

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

- We are not concerned with statistical generalization.
- We only want our network to perform correctly on the points $\mathbb{X} = \{[0, 0]^T, [0,1]^\mathrm{T},[1, 0]^\mathrm{T}, \text{and } [1, 1]^\mathrm{T}\}$.

  We will train the network on all four of these points. The only challenge is to fit the training set.

Example: Learning XOR

- Evaluated on the whole training set, the MSE loss function is $$J(\boldsymbol{\theta}) = \frac{1}{4}\sum_{\boldsymbol{x}\in\mathbb{X}}(f^*(\boldsymbol{x}) - f(\boldsymbol{x};\boldsymbol{\theta}))^2$$
- Now we must choose the form of our model, $f(\boldsymbol{x};\ \boldsymbol{\theta})$.
- Suppose that we choose a linear model, with $\boldsymbol{\theta}$ consisting of $\boldsymbol{w}$ and b. Our model is defined to be

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

- We can minimize $J(\boldsymbol{\theta})$ in closed form with respect to $\boldsymbol{w}$ and b using the normal equations.
- After solving the normal equations, we obtain $\boldsymbol{w}$ = 0 and b = 1/2. The linear model simply outputs 0.5 everywhere.

Example: Learning XOR

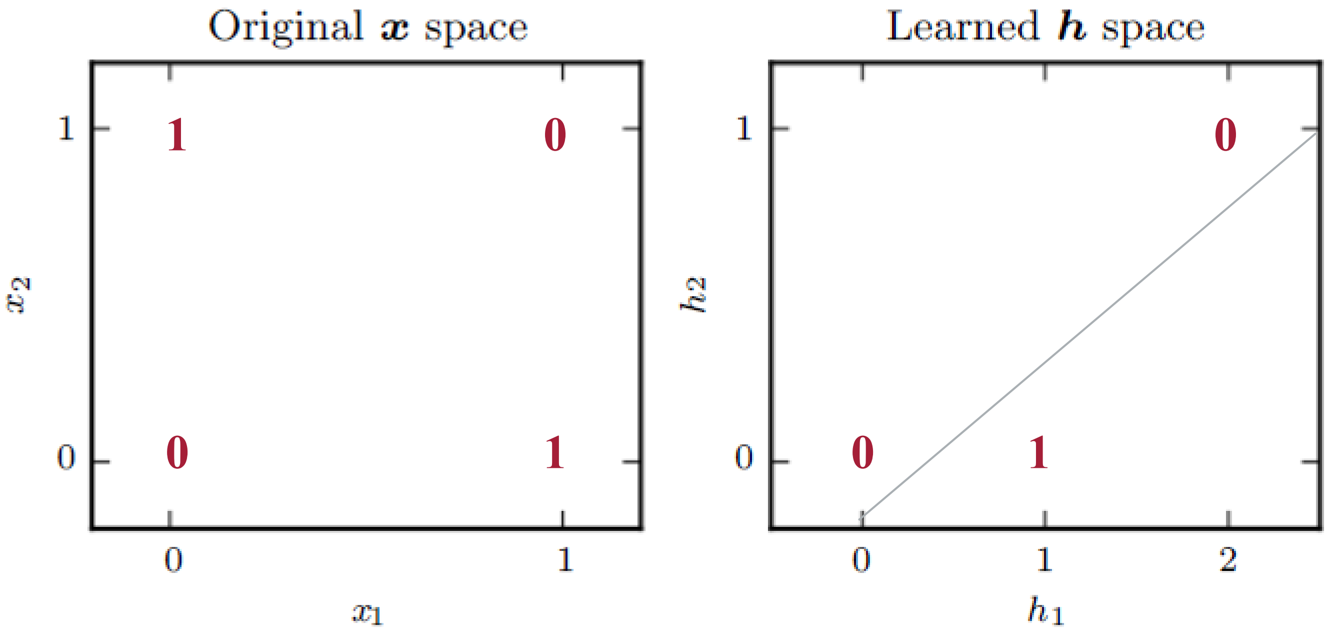


Fig. 6.1: Solving the XOR problem. The red numbers indicate y_i.
(Left) A linear model applied directly to the original input cannot implement the XOR function. When $x_1 = 0$, the model's output must increase as x_2 increases, thus $w_2 > 0$. When $x_1 = 1$, the model's output must decrease as x_2 increases, thus $w_2 < 0$. This is a contradiction.
(Right) In the transformed space represented by the features extracted by a neural network, a linear model can now solve the problem.

- So the linear model is not able to represent the XOR function.
- We introduce a simple feedforward network with one hidden layer containing two hidden units. See Fig. 6.2.
- The hidden units $\boldsymbol{h}$ are computed by a function

$$\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})$$

- The output layer is computed by a function

$$y = f^{(2)}(\boldsymbol{h}; \boldsymbol{w}, b)$$

- The complete model is computed by

$$y = f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = f^{(2)} f^{(1)}(\boldsymbol{x})$$

- $f^{(1)}$ and $f^{(2)}$ should not both be linear, else the resulting function would be linear, and could not represent XOR.

Example: Learning XOR

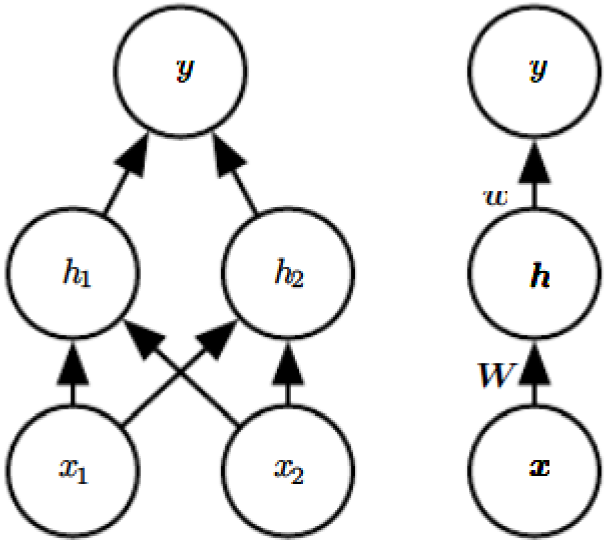


Fig. 6.2: The XOR network, drawn in two different styles. Its hidden layer contains two units. *(Left)* In this style, we draw every unit as a node in the graph. This is very clear, but for larger networks it consumes too much space. *(Right)* In this style, we draw a node in the graph for each entire vector representing a layer's activations. This is much more compact. Often we annotate the edges with the name of the parameters that connect the two layers. Here, a matrix $\boldsymbol{W}$ maps $\boldsymbol{x}$ to $\boldsymbol{h}$, and a vector $\boldsymbol{w}$ maps $\boldsymbol{h}$ to y.

- Most neural networks use an affine transformation controlled by learned parameters, followed by a fixed, nonlinear function called an activation function. We define

$$\boldsymbol{h} = g(\boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{c})$$

  where $\boldsymbol{W}$ provides the weights of a linear transformation and $\boldsymbol{c}$ the biases.

- The activation function is applied element-wise, e.g.

$$h_i = g(\boldsymbol{x}^T \boldsymbol{W}_{:,i} + c_i)$$

- In modern neural networks most often the rectified linear unit or ReLU is used, as shown in Fig. 6.3.

Rectified Linear Activation Function

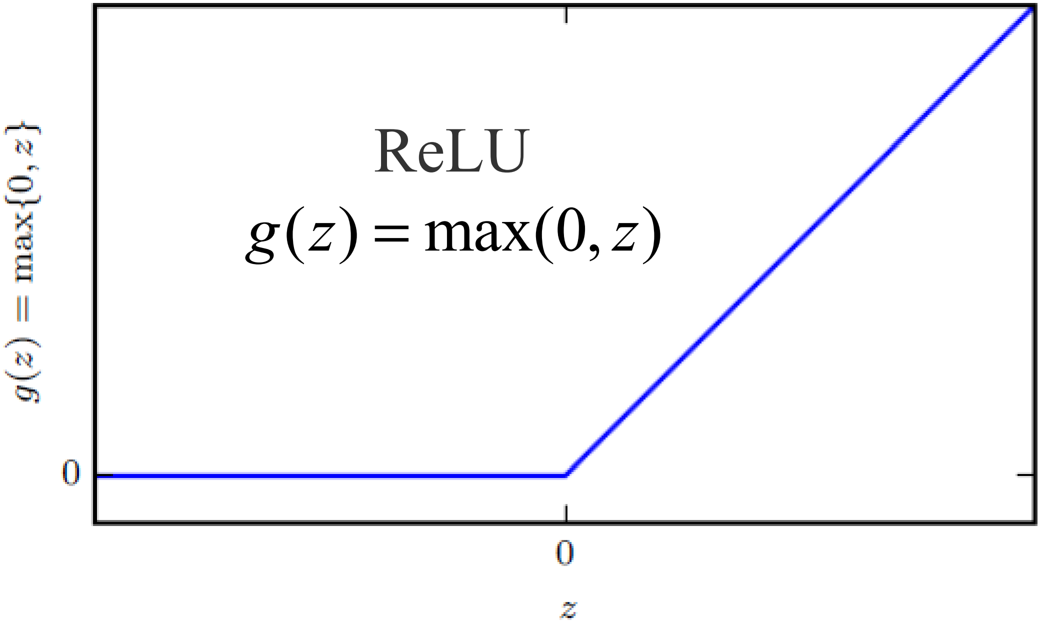


Fig. 6.3: The rectified linear activation function. This is now used in most feedforward neural networks. The function is nonlinear, but remains very close to linear, as it is a piecewise linear function with two linear pieces. Because ReLUs are nearly linear, they preserve many of the properties that make linear models easy to optimize with gradient based methods. They also preserve many of the properties that make linear models generalize well.

Example: Learning XOR

- We can now specify the complete network as

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

- We now can specify a solution to the XOR problem. Let

$$\boldsymbol{W}=\begin{bmatrix}1 & 1\\ 1 & 1\end{bmatrix} \qquad \boldsymbol{c}=\begin{bmatrix}0\\ -1\end{bmatrix}$$

$$\boldsymbol{w}=\begin{bmatrix}1\\ -2\end{bmatrix} \qquad b=0$$

- Let $\boldsymbol{X}$ be the design matrix containing all four input points, with one example per row

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

Example: Learning XOR

- The first step in the neural network is to multiply the input matrix $\boldsymbol{X}$ by the first layers weight matrix $\boldsymbol{W}$:

$$\boldsymbol{X}\boldsymbol{W} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}$$

- Next we add the bias vector $\boldsymbol{c}$ to the hidden units, to obtain

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

$$\boldsymbol{X}\boldsymbol{W} + \boldsymbol{C} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \qquad \boldsymbol{C} = \begin{bmatrix} 0 & -1 \\ 0 & -1 \\ 0 & -1 \\ 0 & -1 \end{bmatrix}$$

Example: Learning XOR

- We now apply the ReLU transformation with function g

$$f^{(1)}(\boldsymbol{X}) = g(\boldsymbol{X}\boldsymbol{W} + \boldsymbol{C}) = g\left(\begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}\right) = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

- We finish by multiplying with the weight vector w and adding bias b = 0

$$f^{(2)}\left(f^{(1)}(\boldsymbol{X})\right) = f^{(2)}\left(\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}\right) = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}\begin{bmatrix} 1 \\ -2 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \mathrm{XOR}$$

- The NN obtains the correct answer for each example.

6.2 Gradient-Based Learning

- Designing and training a neural network is not much different from training any other ML model with gradient descent.
- However, the nonlinearity of a neural network causes most loss functions to become non-convex. This means that NNs are usually trained by using iterative, gradient-based optimizers that drive the cost function to a local minimum.
- The linear equation solvers used to train linear regression models or the convex optimization algorithms used to train logistic regression or SVMs converge to a global optimum, starting from any initial parameters.

Gradient-Based Learning

- Stochastic gradient descent applied to non-convex loss functions has no such convergence guarantee, and is sensitive to the values of the initial parameters.
- For feedforward neural networks, it is important to initialize all weights to small random values. The biases may be initialized to zero or to small positive values.
- We can also train linear regression models and SVMs with gradient descent, and this is common when the training set is extremely large.
- Computing the gradient is slightly more complicated for a neural network, but can still be done efficiently and exactly.

6.2.1 Cost Functions

- In most cases, our parametric model defines a distribution $p(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$ and we simply use the principle of maximum likelihood.
- This means we use the cross-entropy between the training data and the model's predictions as the cost function.
- Sometimes, we take a simpler approach, where we merely predict some statistic of $\boldsymbol{y}$ conditioned on $\boldsymbol{x}$. Specialized loss functions allow us to train a predictor of these estimates.
- The total cost function to train a neural network will often combine one of the primary cost functions with a regularization term.

6.2.1.1. Learning Conditional Distributions with Maximum Likelihood

- Most modern neural networks are trained using maximum likelihood.
- This means that the cost function is simply the negative log-likelihood, equivalently described as the cross-entropy between the training data and the model distribution. It is given by

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

- The form of the cost function changes from model to model, depending on the specific form of $\log p_{\mathrm{model}}$.
- The expansion of the above equation typically yields some terms that do not depend on the model parameters and may be discarded.

Learning Conditional Distributions

- For example, if $p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x}) = N(\boldsymbol{y}; f(\boldsymbol{x};\boldsymbol{\theta}), \boldsymbol{I})$ then we recover the mean squared error (MSE) cost,

$$J(\boldsymbol{\theta}) = \frac{1}{2}\mathbb{E}_{\mathrm{x,y}\sim\hat{p}_{\text{data}}} \left\|\boldsymbol{y} - f(x;\boldsymbol{\theta})\right\|^2 + \text{const}$$

  up to a scaling factor of $1/2$ and a term that does not depend on θ. The discarded constant is based on the variance of the Gaussian distribution.

- For the above formula to hold we do not require that f is a linear function.

6.2.2 Output Units

- The choice of cost function is tightly coupled with the choice of output unit.
- Most of the time, we simply use the cross-entropy between the data distribution and the model distribution.
- The choice of how to represent the output then determines the form of the cross-entropy function.
- Throughout this section, we suppose that the feedforward network provides a set of hidden features defined by $\boldsymbol{h} = f(\boldsymbol{x}; \boldsymbol{\theta})$.
- The role of the output layer is then to provide some additional transformation from the features to complete the task that the network must perform.

6.2.2.1 Linear Units for Gaussian Output Distributions

- One simple kind of output unit is an affine transformation with no nonlinearity.
- These are often just called linear units.
- Given features h, a layer of linear output units produces a vector $\hat{y} = W^T \boldsymbol{h} + b$.
- Linear output layers are often used to produce the mean of a conditional 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 minimizing the mean squared error.
- Because linear units do not saturate, they pose little difficulty for gradient based optimization algorithms.

6.2.2.2 Sigmoid Units for Bernoulli Output Distributions

- Many tasks require predicting the value of a binary variable y (classification problems with two classes).
- The maximum-likelihood approach is to define a Bernoulli distribution over y conditioned on x.
- A Bernoulli distribution is defined by just a single number. The neural net needs to predict only $P(y=1 \mid \boldsymbol{x})$.
- For this number to be a valid probability, it must lie in the interval [0, 1].
- Suppose we were to use a linear unit, and threshold its value to obtain a valid probability:

$$P(y=1 \mid \boldsymbol{x}) = \max\left\{0, \min\left\{1, \boldsymbol{w}^T \boldsymbol{h} + b\right\}\right\}$$

- This would define a valid conditional distribution, but …

Sigmoid Units for Bernoulli Output Distributions

- …but we could not train it efficiently with gradient descent, as the gradient outside the unit interval is zero.
- A better approach is based on using sigmoid output units

$$\hat{y} = \sigma(\boldsymbol{w}^T \boldsymbol{h} + b)$$

where σ is the logistic sigmoid function.

- The sigmoid output unit has two steps.
  1. It uses a linear layer to compute $z = \boldsymbol{w}^T \boldsymbol{h} + b$.
  2. Next, it uses the sigmoid activation function to convert z into a probability.
- Let us discuss how to define a probability distribution over y using the value z:

Sigmoid Units

- We begin with the assumption that the unnormalized log probabilities are linear in y and z,

Why can we assume this?

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

unnormalized

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

Normalization yields

$$P(y) = \frac{\exp(yz)}{\sum_{y'=0}^{1} \exp(y'z)},$$

It shows that distribution follows sigmoid

$$P(y) = \sigma((2y-1)z).$$

How is this last step done?

- Probability distributions based on exponentiation and normalization are common throughout the statistical modeling literature. The z variable defining such a distribution over binary variables is called a logit.

Sigmoid Units

- Because the cost function used with maximum likelihood is $-\log P(y \mid \boldsymbol{x})$, the log in the cost function undoes the exp of the sigmoid.
- Without this effect, the saturation of the sigmoid could prevent gradient based learning from making progress.
- The loss function for maximum likelihood learning of a Bernoulli parametrized by a sigmoid is

J: loss

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

$$-\log \sigma(x) = \varsigma(-x)$$

Using the softplus function ζ

This derivation uses properties from section 3.10

Sigmoid Units

- By rewriting the loss in terms of the softplus function, we see that it saturates only when $(1-2y)z$ is very negative.
- Saturation thus occurs only when the model already has the right answer: when $y=1$ and $z \gg 0$ or y = 0 and $z \ll 0$
- When z has the wrong sign, the argument to the softplus function, $(1-2y)z$, may be simplified to $|z|$. As $|z|$ becomes large while z has the wrong sign, the softplus function asymptotes toward simply returning its argument $|z|$. The derivative with respect to z asymptotes to sign(z), so, in the limit of extremely incorrect z, the softplus function does not shrink the gradient at all.
- This property is very useful because gradient-based learning can act to quickly correct a mistaken z.

6.2.2.3 Softmax Units for Categorical Output Distributions

- Any time we wish to represent a probability distribution over a discrete variable with n possible values, we may use the softmax function. This can be seen as a generalization of the sigmoid function which was used to represent a probability distribution over a binary variable.
- Softmax functions are most often used as the output of a classifier, to represent the probability distribution over n different classes.
- In the case of binary variables, we wished to produce a single number

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

- This number needed to lie between 0 and 1

Softmax Units

- As we wanted the logarithm of the number to be well-behaved for gradient-based optimization of the log-likelihood, we chose to instead predict a number

  $$z = \log \tilde{P}(y = 1 \mid \boldsymbol{x})$$

  Exponentiating and normalizing gave us a Bernoulli distribution controlled by the sigmoid function.
- To generalize to the case of a discrete variable with n values, we now need to produce a vector $\hat{\boldsymbol{y}}$, with $\hat{y}_i = P(y = i \mid \boldsymbol{x})$. We require not only that each element of $\hat{y}_i$ be between 0 and 1, but also that the entire vector sums to 1 so that it represents a valid probability distribution.

Softmax Units

- The same approach that worked for the Bernoulli distribution generalizes to the categorical distribution:
- First, a linear layer predicts unnormalized log probabilities:

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

  where $z_i = \log \tilde{P}(y = i \mid \boldsymbol{x})$.
- Second, the softmax function exponentiates and normalizes $\boldsymbol{z}$ to obtain the desired $\hat{\boldsymbol{y}}$.

$$\operatorname{soft}\max(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

- As with the logistic sigmoid, the use of the exp function works very well when training the softmax to output a target value y using maximum log-likelihood.

Softmax Units

- In this case, we maximize $\log P(y = i; \boldsymbol{z}) = \log \operatorname{softmax}(\boldsymbol{z})_i$.
- Defining the softmax in terms of $\exp$ is natural, as the $\log$ in the log-likelihood can undo the $\exp$ of the softmax:

$$\log \operatorname{softmax}(\boldsymbol{z})_i = z_i - \log \sum_j \exp(z_j)$$

- the input z_i always has a direct contribution to the cost function. Because this term cannot saturate, learning always can proceed.
- When maximizing the log-likelihood, the first term encourages z_i to be pushed up, while the second term encourages all of $\boldsymbol{z}$ to be pushed down.
- The second term can be approximated as $\max_j z_j$. So the negative log-likelihood cost function always strongly penalizes the most active incorrect prediction.

Softmax Units

- So far we have discussed only a single example.
- Overall, unregularized maximum likelihood will drive the model to learn parameters that drive the softmax to predict the fraction of counts of each outcome observed in the training set:

↙ 1 if ...

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

← number of observations

- Because maximum likelihood is a consistent estimator, this is guaranteed to happen so long as the model family is capable of representing the training distribution.
- Many objective functions other than log-likelihood do not work as well with the softmax function. In particular, MSE is a poor loss function for softmax units.

Softmax Units

- The output values of the softmax can saturate when the differences between input values become extreme.
- To see that the softmax function responds to the difference between its inputs, observe that its output is invariant to adding the same scalar to all of its inputs:

$$\operatorname{softmax}(\boldsymbol{z}) = \operatorname{softmax}(\boldsymbol{z} + c)$$

- Using this property, we can derive a numerically stable variant of the softmax:

$$\operatorname{softmax}(\boldsymbol{z}) = \operatorname{softmax}(\boldsymbol{z} - \max_i z_i)$$

- We see that this numerically stable softmax function is driven by the amount that its arguments deviate from $\max_i z_i$.

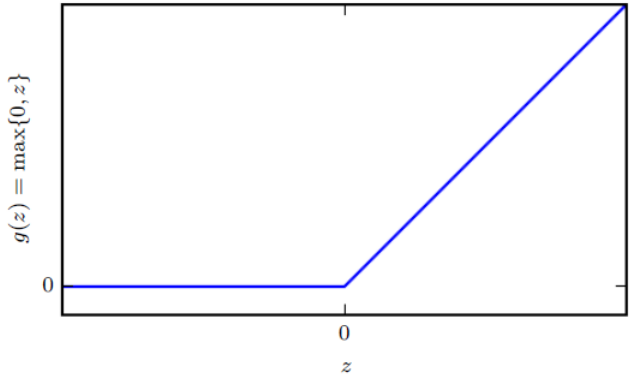
Softmax Units

- The argument z to the softmax function is usually produced by an earlier layer of the neural network, which outputs every element of z, using a linear layer $\boldsymbol{z} = \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{b}$.
- One may think of the softmax as a way to create competition between the units: the softmax outputs always sum to 1, so an increase in the value of one unit necessarily decreases the value of others.
- This is analogous to the lateral inhibition between nearby neurons in the cortex.
- At the extreme, it becomes a form of winner-takes-all (one of the outputs is nearly 1, the others are nearly 0).

6.3 Hidden Units

- Issue: How to choose the type of hidden unit to use in the hidden layers of the model.
- Rectified linear units (ReLUs) are an excellent default choice of hidden unit.
- Some of the hidden units discussed are not differentiable at all input points. For example, the ReLU $g(z) = \max(0, z)$ is not differentiable at $z = 0$.
- In practice, gradient descent still performs well enough for these models to be used for machine learning tasks.
- Most hidden units can be described as accepting a vector of inputs $\boldsymbol{x}$, computing an affine transformation $\boldsymbol{z} = \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{b}$ and then applying a nonlinear function $g(\boldsymbol{z})$.

6.3.1 Rectified Linear Units and Their Generalizations



- Rectified linear units (ReLUs) use $g(z) = \max(0, z)$.
- ReLUs are easy to optimize because they are half linear units.
- $$g'(z) = \begin{cases} 1 & \text{if the unit is active} \\ 0 & \text{else} \end{cases}$$
- So the gradient information is not damped or skewed.
- ReLUs are typically used after an affine transformation:
  $$\boldsymbol{h} = g(\boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{b})$$
- It is good practice to set all elements of $\boldsymbol{b}$ to a small, positive value, such as 0.1, making it likely that most ReLUs are initially active for most inputs.

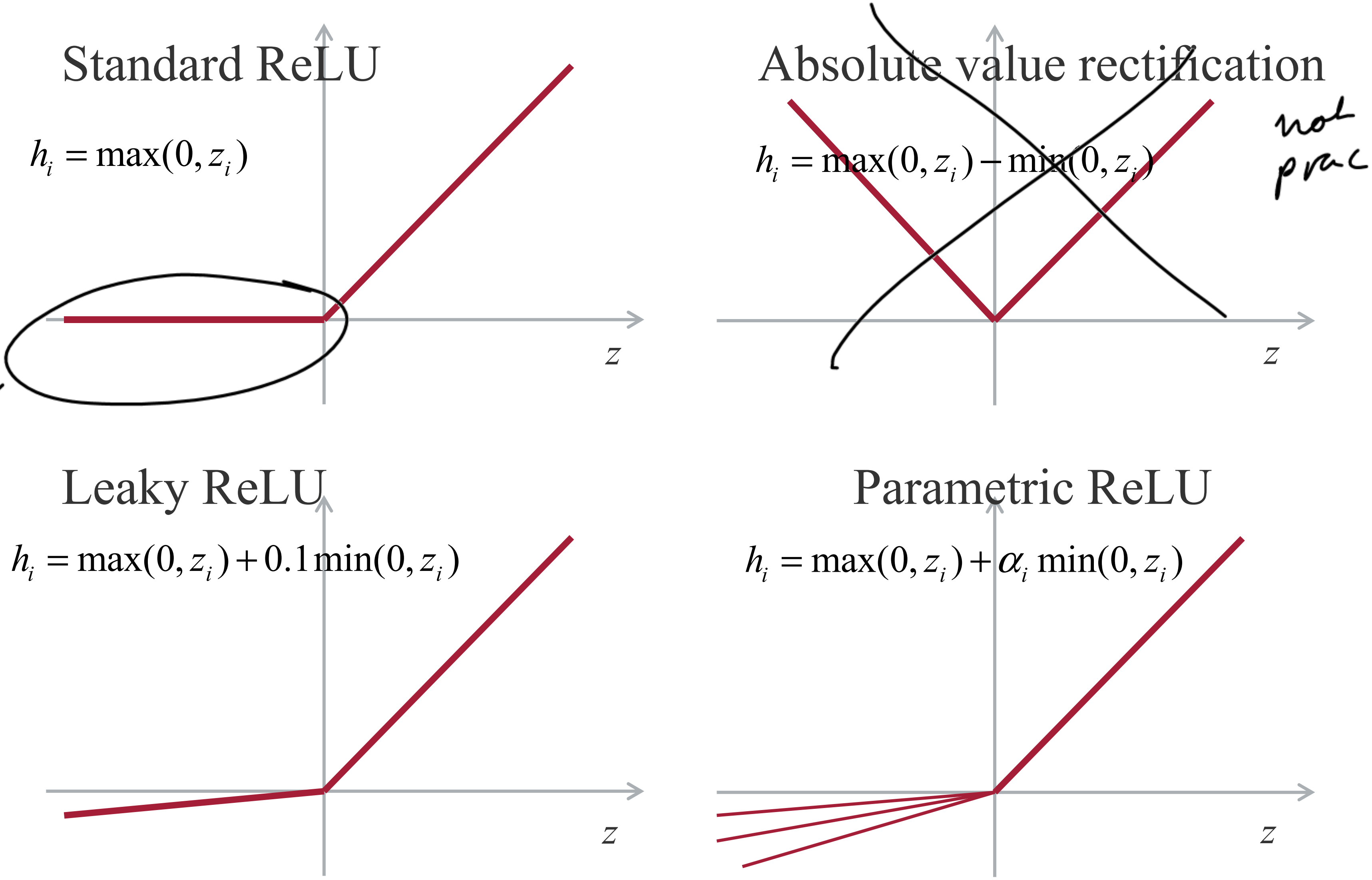
ReLUs and Their Generalizations

- One drawback to rectified linear units is that they cannot learn via gradient based methods on examples for which their activation is zero.
- Three generalizations of rectified linear units are based on using a non-zero slope α_i when $z_i < 0$.

$$h_i = g(\boldsymbol{z},\boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i)$$

- Absolute value rectification sets $\alpha_i = -1$ to obtain $g(z) = |z|$. It is used for object recognition of images.
- Leaky ReLU fixes α_i to a small value like 0.01.
- parametric ReLU or PReLU treats α_i as a learnable parameter, individual for each unit.

ReLU Variants



Standard ReLU

$$h_i = \max(0, z_i)$$

cannot learn if activation is negative

Absolute value rectification

$$h_i = \max(0, z_i) - \min(0, z_i)$$

not practical

Leaky ReLU

$$h_i = \max(0, z_i) + 0.1 \min(0, z_i)$$

Parametric ReLU

$$h_i = \max(0, z_i) + \alpha_i \min(0, z_i)$$

Maxout Units

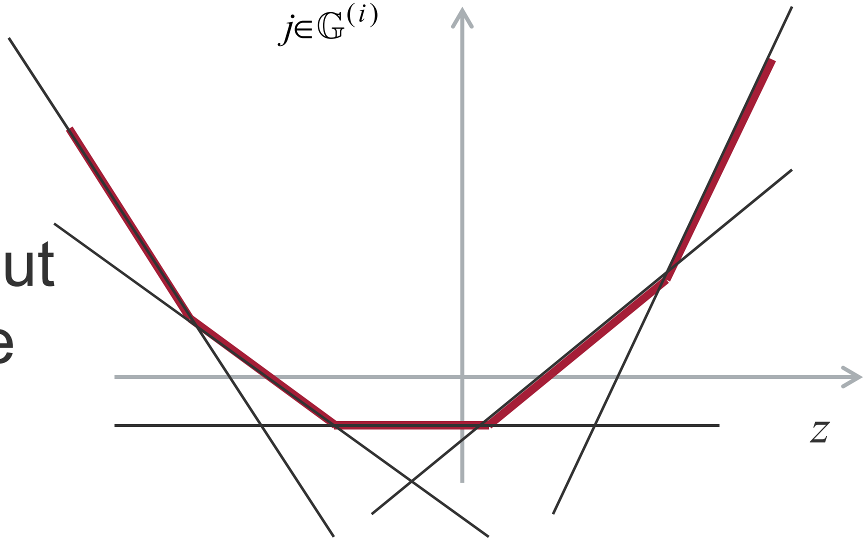
not popular

- Maxout units generalize rectified linear units further. Instead of applying an element-wise function $g(z)$, maxout units divide z into groups of k values.

  Each maxout unit then outputs the maximum element of one of these groups:

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

- A maxout unit can learn a piecewise linear, convex function with up to k pieces.
- With large enough k, a maxout unit can learn to approximate any convex function with arbitrary fidelity.



- Each maxout unit is parametrized by k weight vectors.

6.3.2 Logistic Sigmoid and tanh

- Prior to ReLUs, most neural networks used the logistic sigmoid activation function

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

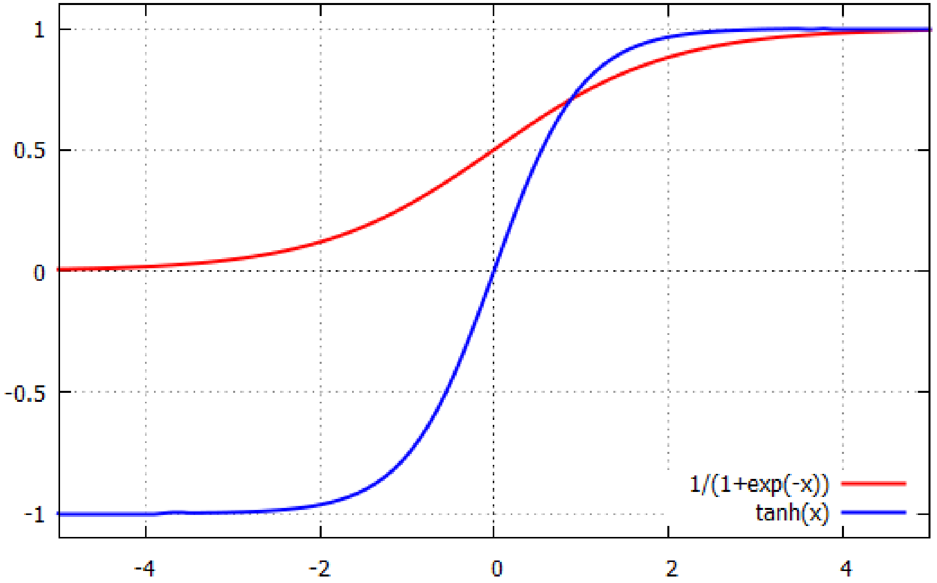
- Or the hyperbolic tangent activation function

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

- These functions are related, because

$$\tanh(z) = 2 \cdot \sigma(2z) - 1$$

- Sigmoidal units saturate when z is very high or very low.



Logistic Sigmoid and tanh

- The hyperbolic tangent activation function often performs better than the logistic sigmoid.
- It resembles the identity function more closely, as $\tanh(0) = 0$ while $\sigma(0) = 1/2$.
- Because $\tanh$ is similar to the identity function near 0, training a deep neural network
$$\hat{y} = \boldsymbol{w}^T \tanh(\boldsymbol{U}^T \tanh(\boldsymbol{V}^T \boldsymbol{x}))$$
resembles training a linear model $\hat{y} = \boldsymbol{w}^T \boldsymbol{U}^T \boldsymbol{V}^T \boldsymbol{x}$
as long as the activations of the network are small.
- This makes training the $\tanh$ network easier.

6.3.3 Other Hidden Units

- Many other types of hidden units are possible, but are used less frequently.
- Using $\cos(z)$ as activation function works as well (tested on the MNIST dataset with competitive error rate).
- Having no activation function $g(z)$ at all (just the identity) is also possible, but
- If every layer of the neural network consists of only linear transformations, then the network as a whole will be linear.
- However, it is acceptable for some layers of the neural network to be purely linear. This may factor a weight matrix $\boldsymbol{W}$ into two smaller weight matrices $\boldsymbol{U}$ and $\boldsymbol{V}$.

Linear Hidden Units

- Consider a neural network layer with n inputs and p outputs, $\boldsymbol{h} = g(\boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{b})$.
- We may replace $\boldsymbol{W}$ with two linear layers, with weight matrices $\boldsymbol{U}$ and $\boldsymbol{V}$.
- The factored approach is to compute $\boldsymbol{h} = g(\boldsymbol{V}^T \boldsymbol{U}^T \boldsymbol{x} + \boldsymbol{b})$.
- If $\boldsymbol{U}$ produces q outputs, then $\boldsymbol{U}$ and $\boldsymbol{V}$ together contain $(n + p)q$ parameters, while $\boldsymbol{W}$ contains np parameters. For small q, this can be a saving in parameters.
- It comes at the cost of constraining the linear transformation to be low-rank, but these low-rank relationships are often sufficient.
- A narrow layer of linear hidden units thus offers a way of reducing the number of parameters in a network.

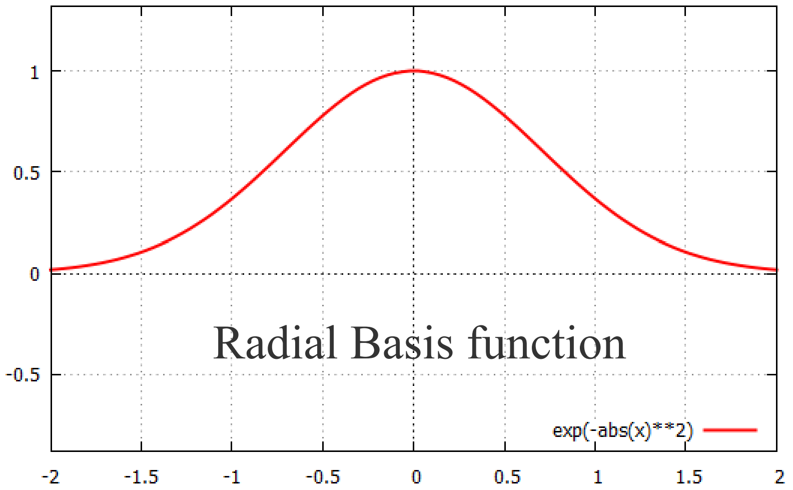
Other Hidden Unit Types

- Radial basis function (RBF) units:
- This function becomes more active as x approaches a template $\boldsymbol{W}_{:,i}$.

$$h_i = \exp\left(-\frac{1}{\sigma_i^2}\left\|\boldsymbol{W}_{:,i} - \boldsymbol{x}\right\|^2\right)$$

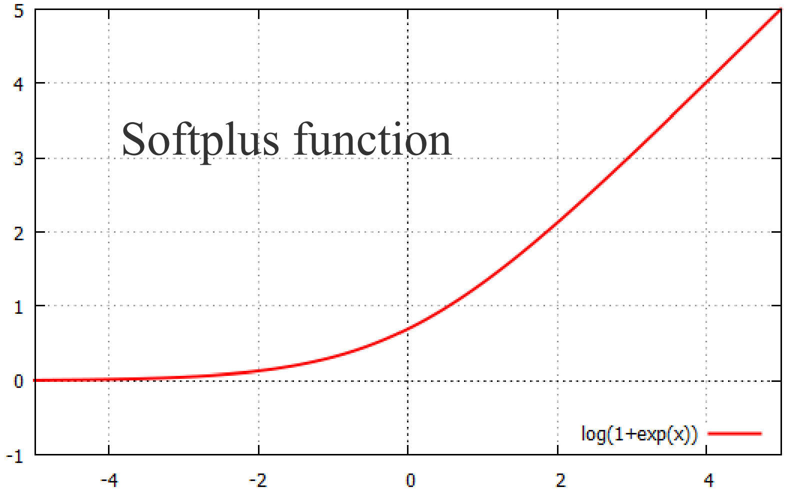
- It is heavily used in SVMs and in older neural networks.
- However, because it saturates to 0 for most x, it can be difficult to optimize.

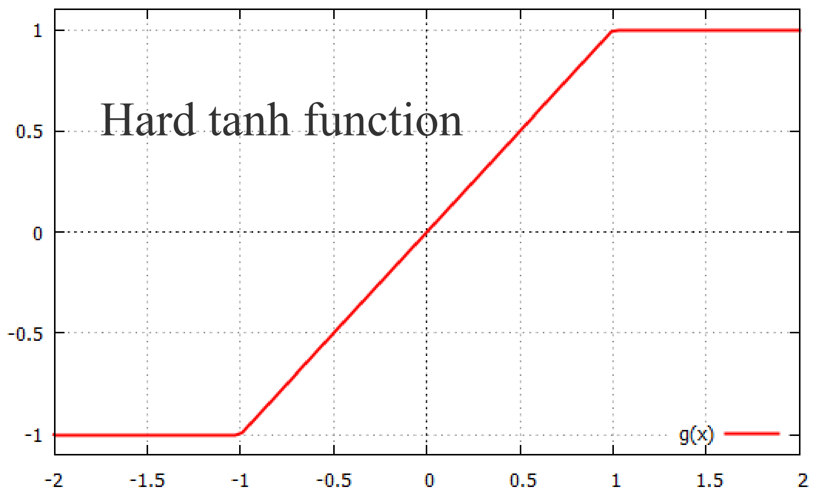
benefit: says 0 for inputs it has never seen before, vgl. Nearest Neighbour



Other Hidden Unit Types

- Softplus function: $g(z) = \zeta(z) = \log(1 + \exp(z))$
  This is a smooth version of the ReLU. It is differentiable everywhere. The use of the softplus is generally discouraged, as the ReLU empirically performs better.
- Hard tanh: $g(z) = \max(-1, \min(1, z))$





6.4 Architecture Design

- A key design consideration for neural networks is determining the architecture.
- This refers to the overall structure of the network: how many units it should have and how these units should be connected to each other.
- Most neural networks are organized into groups of units called layers. The first layer is given by

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

the second layer is given by

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

and so on.

6.4.1 Universal Approximation Properties and Depth

- Feedforward networks with hidden layers are universal approximators.
- Specifically, the universal approximation theorem (Hornik *et al.*, 1989; Cybenko, 1989) states that a feedforward network with a linear output layer and at least one hidden layer with any “squashing” activation function (such as the logistic activation function) can approximate any Borel measurable function from a finite-dim. space to another with any desired non-zero amount of error, provided that the network is given enough hidden units.
- The derivatives of the feedforward network can also approximate the derivatives of the function arbitrarily well (Hornik *et al.*, 1990).

Universal Approximation Properties, Depth

- Universal approximation theorems have later been proved for a wider class of activation functions, which includes the ReLU (Leshno *et al.*, 1993).
- The universal approximation theorem means that regardless of what function we are trying to learn, we know that a large MLP will be able to *represent* this function. However, we are not guaranteed that the training algorithm will be able to *learn* that function.
- Learning can fail for two reasons.
  1. The training algorithm may not find the parameter values that correspond to the desired function.
  2. The training algorithm might choose the wrong function due to overfitting.

Universal Approximation Properties, Depth

- The universal approximation theorem says that there exists a network large enough to achieve any degree of accuracy we desire, but the theorem does not say how large this network will be.
- It might have an exponential number of hidden units (possibly with one hidden unit corresponding to each input configuration that needs to be distinguished).
- So, a feedforward network with a single layer is sufficient to represent any function, but the layer may be infeasibly large and may fail to learn and generalize correctly.
- Often, using deeper models can reduce the number of units required to represent the desired function and can reduce the amount of generalization error.

- We may also want to choose a deep model for statistical reasons.
- Choosing a deep model encodes a very general belief that the function we want to learn should involve composition of several simpler functions.
- Empirically, greater depth does seem to result in better generalization for a wide variety of tasks (Bengio *et al.*, 2007; Erhan *et al.*, 2009; Bengio, 2009; Mesnil *et al.*, 2011; Ciresan *et al.*, 2012; Krizhevsky *et al.*, 2012; Sermanet *et al.*, 2013; Farabet *et al.*, 2013; Couprie *et al.*, 2013; Kahou *et al.*, 2013; Goodfellow *et al.*, 2014d; Szegedy *et al.*, 2014a).

Test Set Accuracy as function of Depth

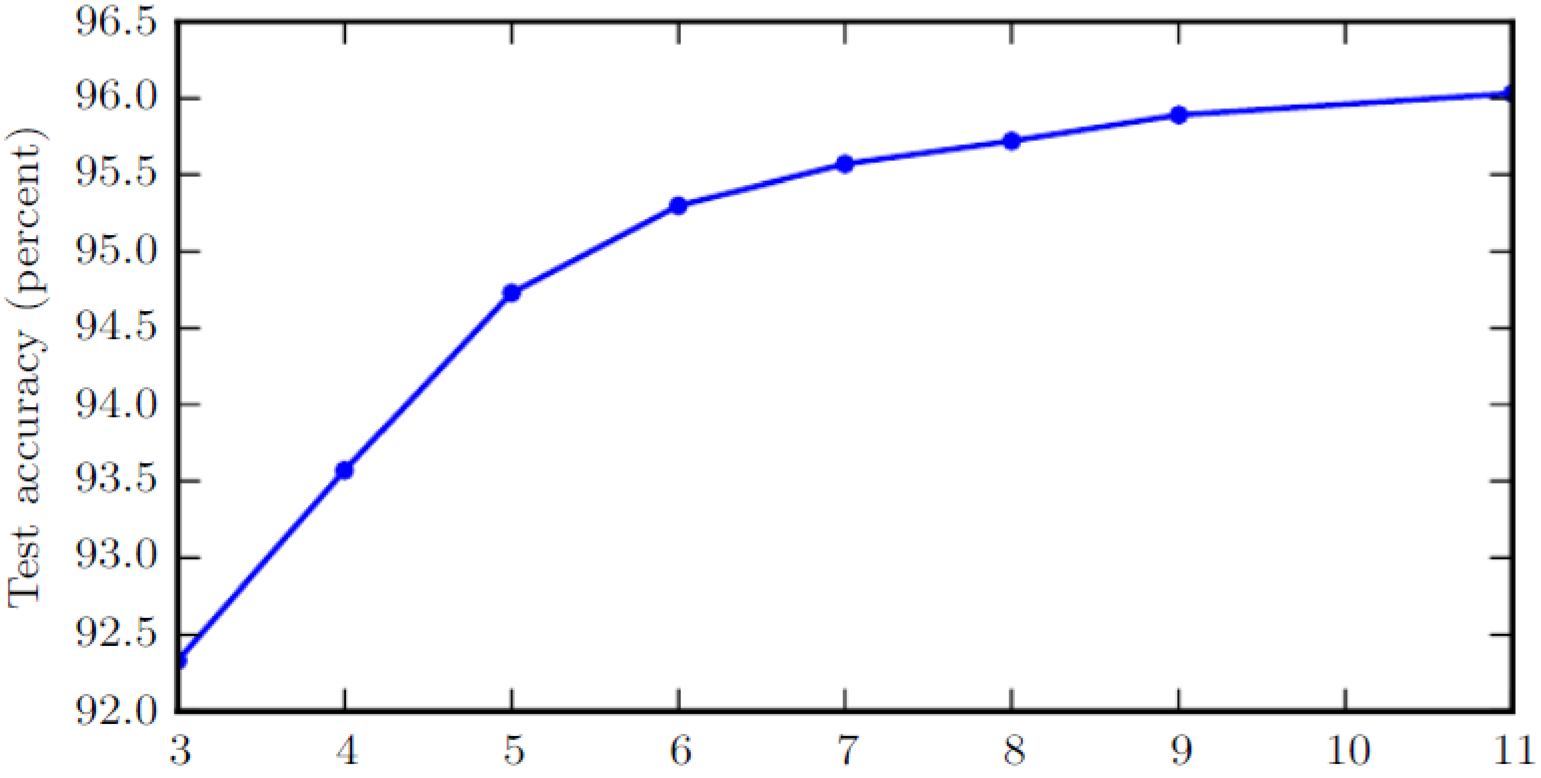


Fig. 6.6: Empirical results showing that deeper networks generalize better when used to transcribe multi-digit numbers from photographs of addresses. Data from Goodfellow *et al.* (2014d).

Test Set Accuracy as function of Depth

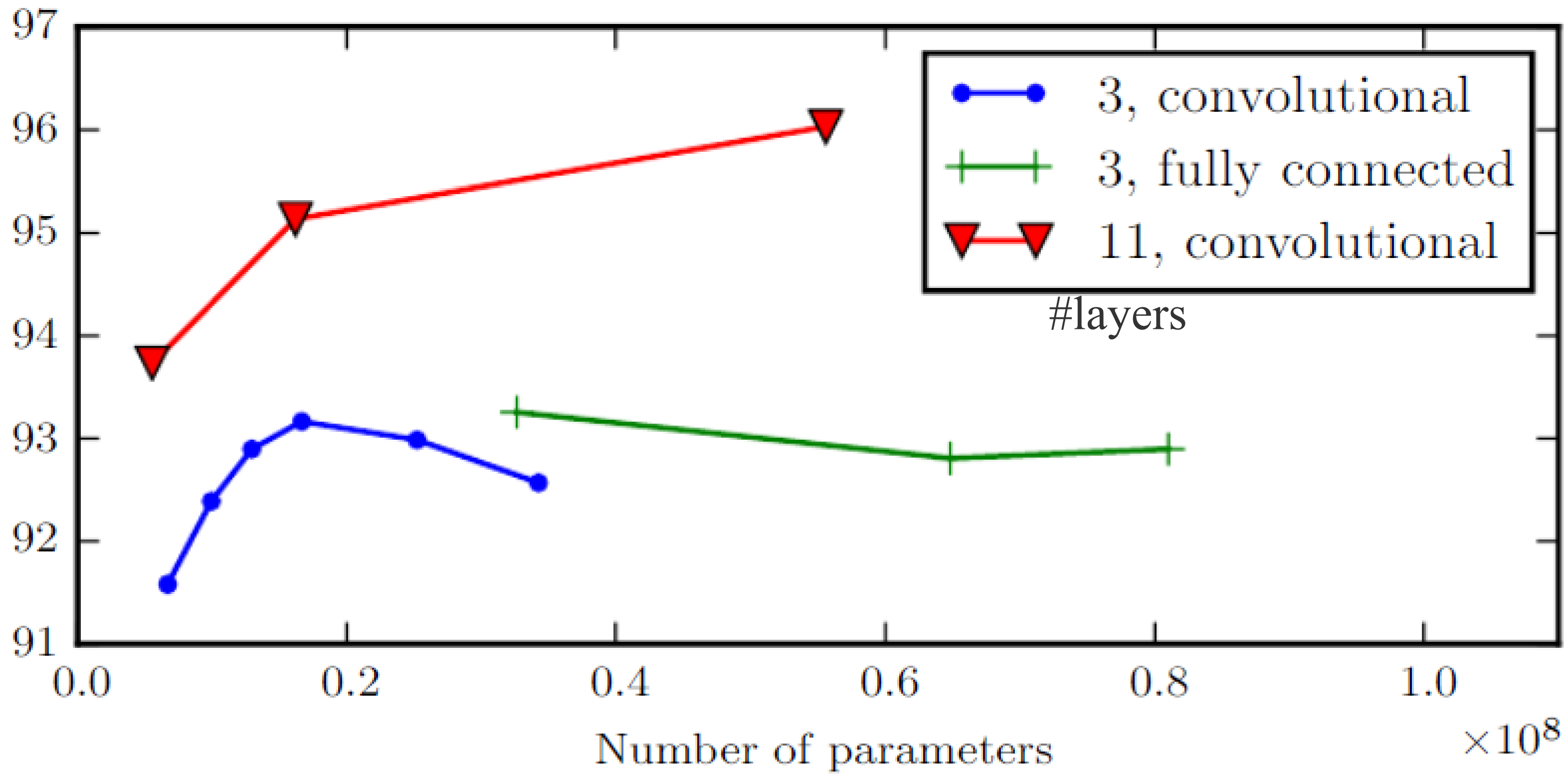


Fig. 6.7: Deeper models tend to perform better. This is not merely because the model is larger. This experiment shows that increasing the number of parameters in layers of convolutional networks without increasing their depth is not nearly as effective at increasing test set performance. This suggests that the function should consist of many simpler functions composed together.

6.4.2 Other Architectural Considerations

- Many neural network architectures have been developed for specific tasks:
- Specialized architectures for computer vision called convolutional networks are described in chapter 9.
- (Feedforward networks may also be generalized to recurrent neural networks for sequence processing, described in chapter 10. not in this lecture)
- Many architectures build a main chain but then add extra architectural features to it, such as skip connections or shortcut connections, going from layer i to layer $i + 2$ or higher. These make it easier for the gradient to flow from output layers to layers nearer the input.

6.5 Backpropagation and Variants

- When we use a feedforward neural network to accept an input x and produce an output $\hat{y}$, information flows forward through the network. This is called forward propagation. It produces a scalar cost $J(\theta)$.
- The backpropagation algorithm (Rumelhart *et al.*, 1986a), often simply called backprop, allows the information from the cost to then flow backwards through the network, in order to compute the gradient.
- Backpropagation is also computationally efficient.
- Backpropagation is often misunderstood as being specific to multilayer NNs, but in principle it can compute derivatives of any function. (has to be smooth)

Backpropagation and Variants

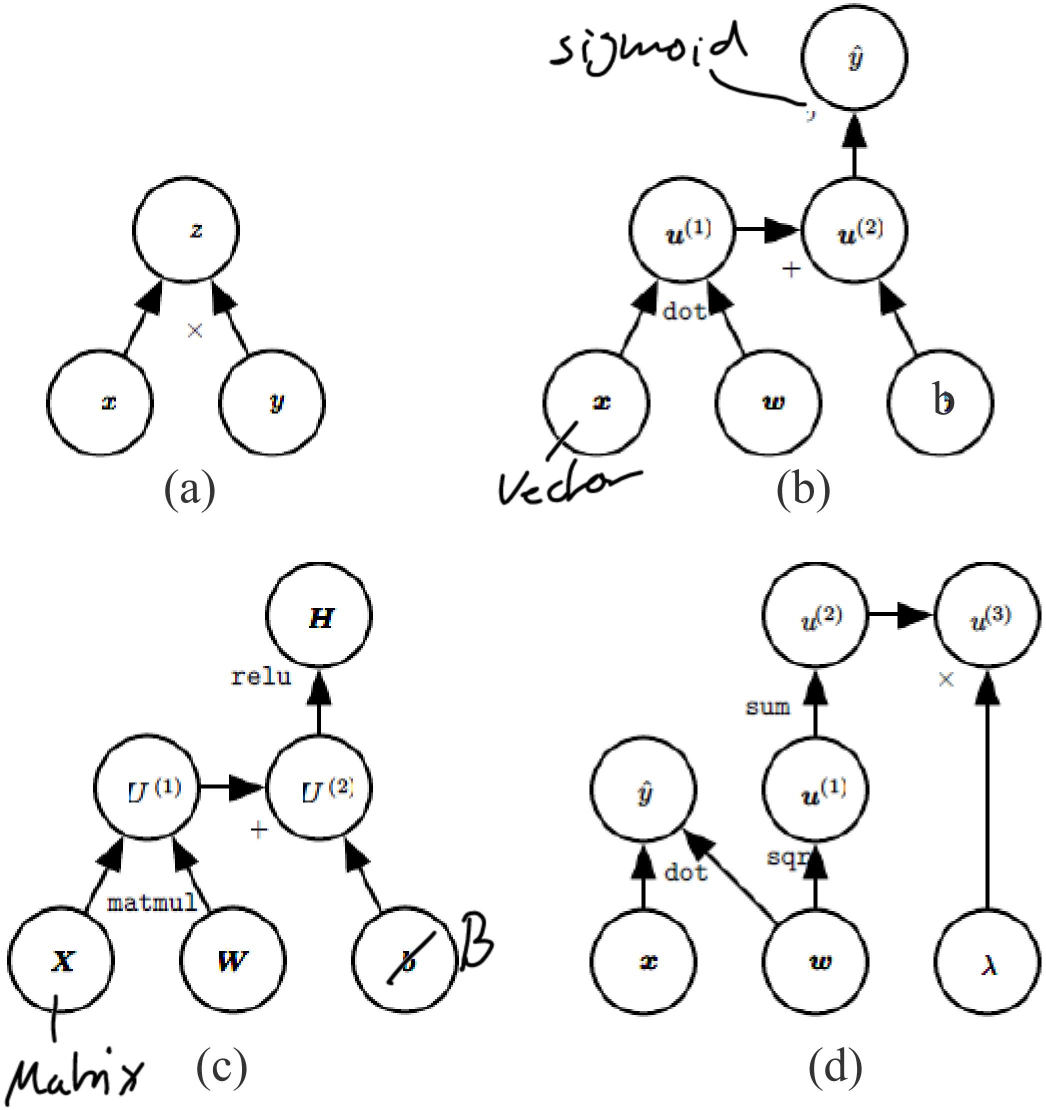
- We will describe how to compute the gradient $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}, \boldsymbol{y})$ for an arbitrary function f, where $\boldsymbol{x}$ is a set of variables whose derivatives are desired, and $\boldsymbol{y}$ are inputs to the function but whose derivatives are not required.
- In learning algorithms, we usually require the gradient of the cost function with respect to the parameters, $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$.

6.5.1 Computational Graphs

- To describe backpropagation more precisely, it is helpful to have a more precise computational graph language.
- Here, we use each node in the graph to indicate a variable. The variable may be a scalar, vector, matrix, tensor, or even a variable of another type.
- An operation is a simple function of one or more variables. We have a set of allowable operations.
- More complicated functions may be described by composing many operations together.
- If a variable y is computed by applying an operation to a variable x, then we draw a directed edge from x to y.
- We sometimes annotate the output node with the name of the operation applied.

Examples of Computational Graphs

Fig. 6.8: Computational graphs. *(a)*The graph to compute $z = xy$. *(b)*The graph for the logistic regression prediction. We name intermediate expressions $\boldsymbol{u}^{(i)}$. *(c)*The graph for the expression $\boldsymbol{H} = \max\{0, \boldsymbol{X}\boldsymbol{W} + \boldsymbol{b}\}$, which computes a design matrix of ReLU activations $\boldsymbol{H}$ given a design matrix containing a minibatch of inputs $\boldsymbol{X}$. *(d)* A computation graph that applies two operations to the weights $\boldsymbol{w}$ of a linear regression model. They are used to make both the prediction $\hat{y}$ and the weight decay penalty $\lambda \sum_i w_i^2$.



6.5.2 Chain Rule of Calculus

- Let x be a real number, and let f and g both be functions mapping from a real number to a real number. Suppose that $y = g(x)$ and $z = f(g(x)) = f(y)$. Then the chain rule states that
$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

dz: total derivative (just one variable)

∂z: partial derivative

- We can generalize this: Suppose that $\boldsymbol{x} \in \mathbb{R}^m, \boldsymbol{y} \in \mathbb{R}^n$, g maps from $\mathbb{R}^m$ to $\mathbb{R}^n$, f maps from $\mathbb{R}^n$ to $\mathbb{R}$. If $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f(\boldsymbol{y})$ then
$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$

- In vector notation this may be written as

Vector of all partial derivatives

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

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

Chain Rule of Calculus

- From this we see that the gradient of a variable x can be obtained by multiplying a transpose Jacobian matrix $\partial \boldsymbol{y}/\partial \boldsymbol{x}$ by a gradient $\nabla_{\boldsymbol{y}} z$.
- Backpropagation consists of performing such a Jacobian-gradient product for each operation in the graph.
- We may apply the backpropagation algorithm not only to vectors, but rather to tensors of arbitrary dimension. We may imagine flattening the tensor into a vector, computing a vector-valued gradient, then reshaping the gradient back into a tensor.
- To denote the gradient of a value z with respect to a tensor **X**, we write $\nabla_{\mathbf{X}} z$, just as if **X** were a vector.

Chain Rule of Calculus

- The indices into **X** now have multiple coordinates—for example, a 3D tensor is indexed by three coordinates. We can abstract this away by using a single variable i to represent the complete tuple of indices. For all possible index tuples i, $(\nabla_{\mathbf{X}} z)_i$ gives $\partial z/\partial X_i$. This is exactly the same as how for all possible integer indices i into a vector, $(\nabla_{x} z)_i$ gives $\partial z/\partial x_i$. Using this notation, we can write the chain rule as it applies to tensors. If $\mathbf{Y} = g(\mathbf{X})$ and $z = f(\mathbf{Y})$, then

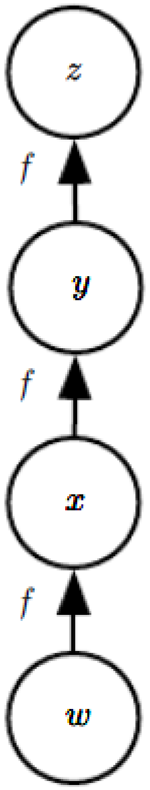
$$\nabla_{\mathbf{X}} z = \sum_j \left(\nabla_{\mathbf{X}} Y_j\right) \frac{\partial z}{\partial Y_j}$$

6.5.3 Recursively Applying the Chain Rule to Obtain Backprop

- Using the chain rule, it is straightforward to write down an algebraic expression for the gradient of a scalar with respect to any node in the computational graph that produced that scalar.
- However, actually evaluating that expression in a computer introduces some extra considerations:
- Subexpressions may be repeated several times within the computation of the gradient. We need to choose
  - store these subexpressions (needs more memory)
  - recompute them multiple times (needs more time)

Recursively Applying the Chain Rule

- Figure 6.9: A computational graph that results in repeated subexpressions when computing the gradient. Let $w \in \mathbb{R}$ be the input to the graph. We apply the same function $f : \mathbb{R} \rightarrow \mathbb{R}$ at every step of a chain: $x = f(w)$, $y = f(x)$, $z = f(y)$.
- To compute $\partial z/\partial w$, we obtain:



$$\frac{\partial z}{\partial w}$$

$$= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w}$$

$$= f'(y)f'(x)f'(w)$$

Needs more memory

$$= f'(f(\underline{f(w)}))f'(\underline{f(w)})f'(w)$$

Needs multiple computations of $f(w)$

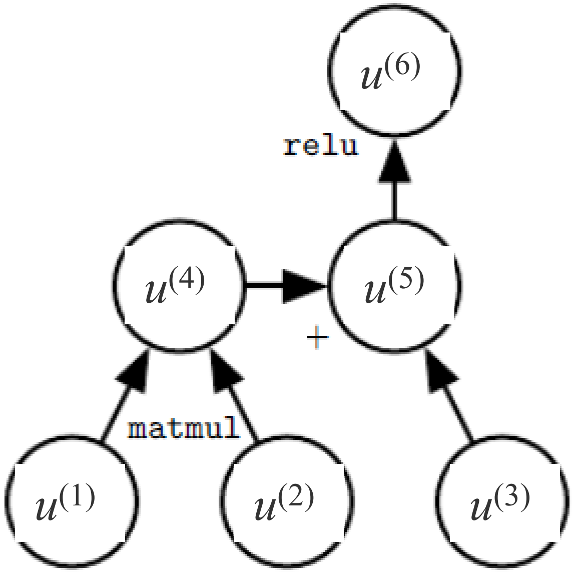
has to be computed multiple times

Forward Propagation

- We assume that the nodes of the computational graph have been ordered topologically, i.e. in such a way that we can compute their output one after the other, starting at $u^{(ni+1)}$ and going up to $u^{(n)}$.
- Each node $u^{(i)}$ is associated with an operation $f^{(i)}$ and is computed by evaluating the function $u^{(i)} = f(\mathbb{A}^{(i)})$ where $\mathbb{A}^{(i)}$ is the set of all nodes that are parents of $u^{(i)}$.

$A^{(i)}$: parents from $u^{(i)}$

^ direct

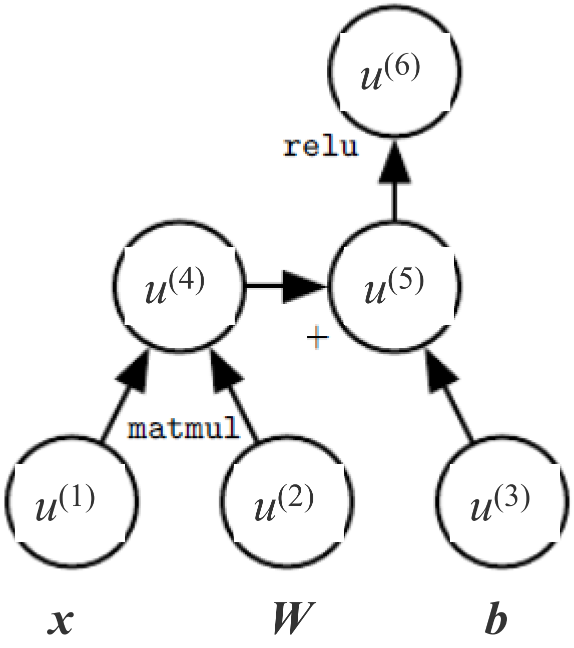


Forward Propagation

Algorithm 6.1 A procedure that performs the computations mapping n_i inputs $u^{(1)}$ to $u^{(n_i)}$ to an output $u^{(n)}$. This defines a computational graph where each node computes numerical value $u^{(i)}$ by applying a function $f^{(i)}$ to the set of arguments $\mathbb{A}^{(i)}$ that comprises the values of previous nodes $u^{(j)}$, $j < i$, with $j \in Pa(u^{(i)})$. The input to the computational graph is the vector $\boldsymbol{x}$, and is set into the first n_i nodes $u^{(1)}$ to $u^{(n_i)}$. The output of the computational graph is read off the last (output) node $u^{(n)}$.

```
for $i = 1, \ldots, n_i$ do
  $u^{(i)} \leftarrow x_i$
end for
for $i = n_i + 1, \ldots, n$ do
  $\mathbb{A}^{(i)} \leftarrow \{u^{(j)} \mid j \in Pa(u^{(i)})\}$      Parents of $u^{(i)}$
  $u^{(i)} \leftarrow f^{(i)}(\mathbb{A}^{(i)})$
end for
return $u^{(n)}$
```

n_i: input layer nodes



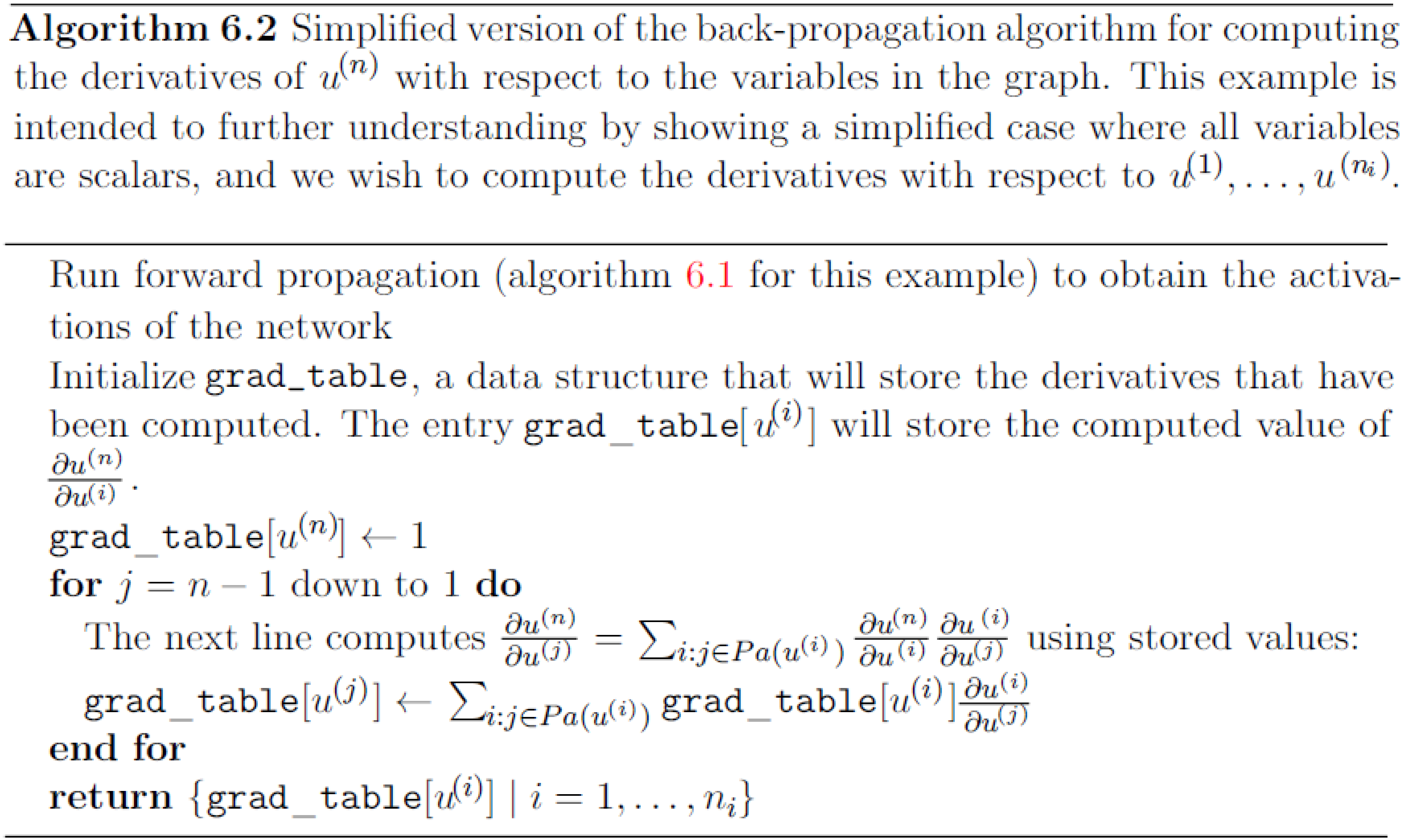
Backward Propagation

- To perform backpropagation, we can construct a computational graph that depends on $\mathcal{G}$ and adds to it an extra set of nodes. These form a subgraph $\mathcal{B}$ with one node per node of $\mathcal{G}$. Computation in $\mathcal{B}$ proceeds in exactly the reverse of the order of computation in $\mathcal{G}$, and each node of $\mathcal{B}$ computes the derivative $\partial u^{(n)}$ / $\partial u^{(i)}$ associated with the forward graph node $u^{(i)}$. This is done using the chain rule with respect to scalar output $u^{(n)}$:

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

- The subgraph $\mathcal{B}$ contains exactly one edge for each edge from node $u^{(j)}$ to node $u^{(i)}$ of $\mathcal{G}$. The edge from $u^{(j)}$ to $u^{(i)}$ is associated with the computation of $\partial u^{(i)}$ / $\partial u^{(j)}$.

Backward Propagation



Algorithm 6.2 Simplified version of the back-propagation algorithm for computing the derivatives of $u^{(n)}$ with respect to the variables in the graph. This example is intended to further understanding by showing a simplified case where all variables are scalars, and we wish to compute the derivatives with respect to $u^{(1)}, \dots, u^{(n_i)}$.

```
Run forward propagation (algorithm 6.1 for this example) to obtain the activations of the network
Initialize grad_table, a data structure that will store the derivatives that have been computed. The entry grad_table[$u^{(i)}$] will store the computed value of $\frac{\partial u^{(n)}}{\partial u^{(i)}}$.
grad_table[$u^{(n)}$] $\leftarrow 1$
for $j = n - 1$ down to $1$ do
    The next line computes $\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$ using stored values:
    grad_table[$u^{(j)}$] $\leftarrow \sum_{i:j \in Pa(u^{(i)})}$ grad_table[$u^{(i)}$] $\frac{\partial u^{(i)}}{\partial u^{(j)}}$
end for
return {grad_table[$u^{(i)}$] $\mid i = 1, \dots, n_i$}
```

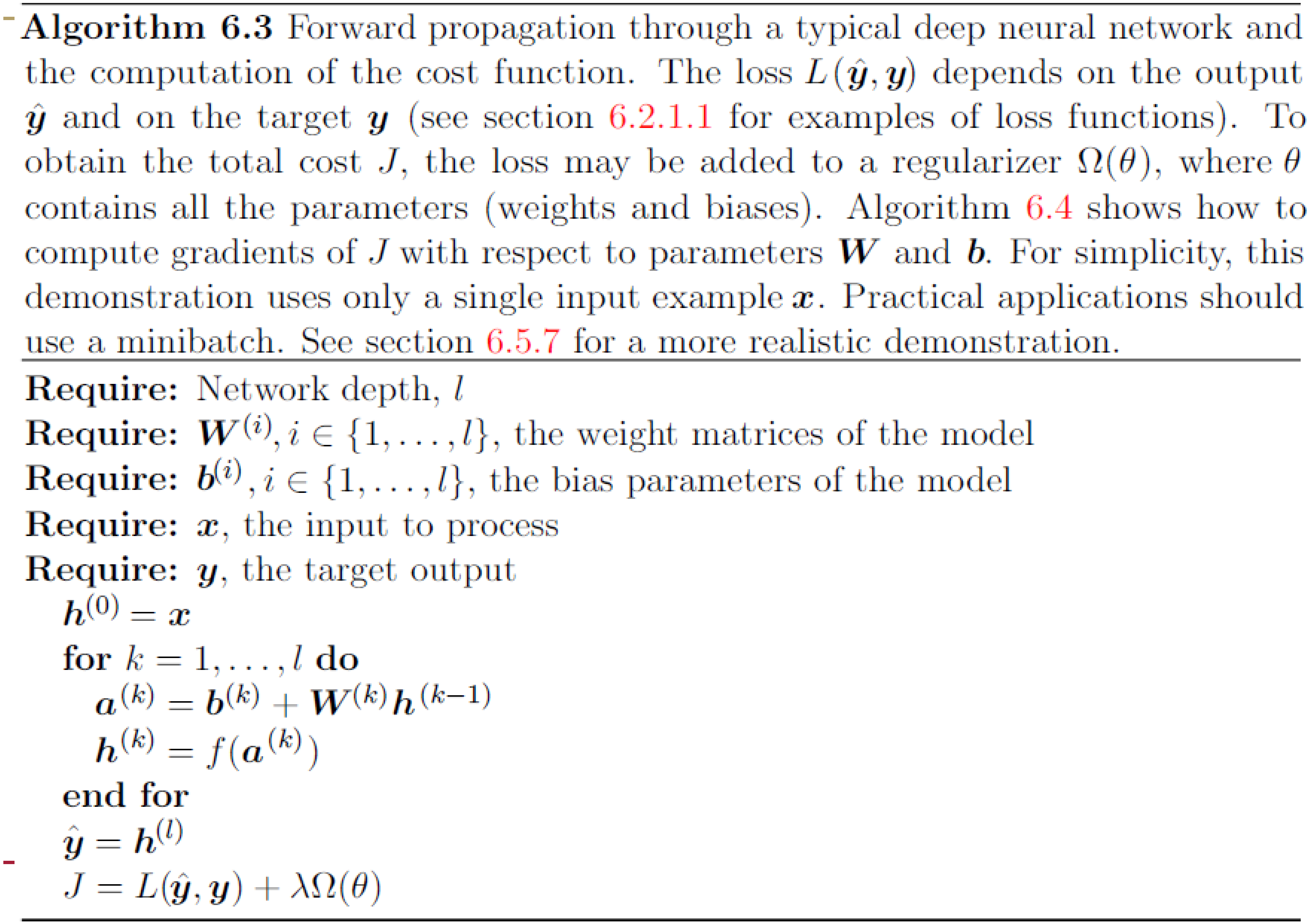
Backward Propagation

- To summarize, the amount of computation required for performing backpropagation scales linearly with the number of edges in $\mathcal{G}$.
- The computation for each edge corresponds to computing a partial derivative (of one node with respect to one of its parents) as well as performing one multiplication and one addition.
- Backpropagation is designed to reduce the number of common subexpressions without regard to memory.
- It thus avoids the exponential explosion in repeated subexpressions.

6.5.4 Backpropagation in Fully-Connected MLPs

- Let us consider the specific graph associated with a fully-connected multi-layer MLP.
- Algorithm 6.3 first shows the forward propagation, which maps parameters to the supervised loss $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ associated with a single (input, target) training example ($\boldsymbol{x}$, $\boldsymbol{y}$), with $\hat{\boldsymbol{y}}$ the output of the neural network when $\boldsymbol{x}$ is provided as input.
- Algorithm 6.4 then shows the corresponding computation to be done for applying the back-propagation algorithm to this graph.

Forward Propagation

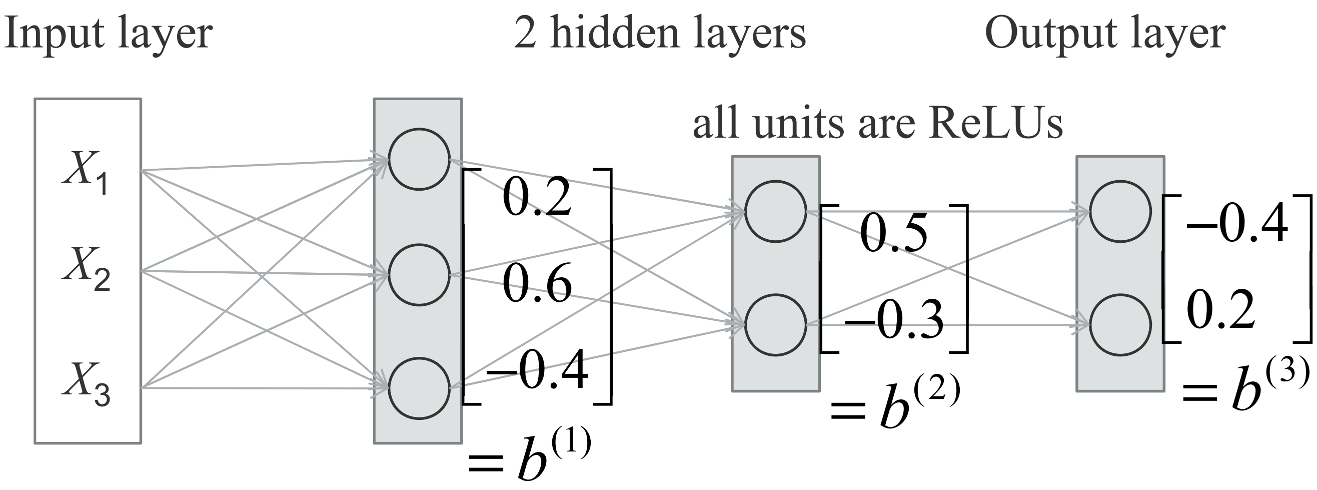


Algorithm 6.3 Forward propagation through a typical deep neural network and the computation of the cost function. The loss $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ depends on the output $\hat{\boldsymbol{y}}$ and on the target $\boldsymbol{y}$ (see section 6.2.1.1 for examples of loss functions). To obtain the total cost J, the loss may be added to a regularizer $\Omega(\theta)$, where θ contains all the parameters (weights and biases). Algorithm 6.4 shows how to compute gradients of J with respect to parameters $\boldsymbol{W}$ and $\boldsymbol{b}$. For simplicity, this demonstration uses only a single input example $\boldsymbol{x}$. Practical applications should use a minibatch. See section 6.5.7 for a more realistic demonstration.

```
Require: Network depth, $l$
Require: $\boldsymbol{W}^{(i)}, i \in \{1, \ldots, l\}$, the weight matrices of the model
Require: $\boldsymbol{b}^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
Require: $\boldsymbol{x}$, the input to process
Require: $\boldsymbol{y}$, the target output
  $\boldsymbol{h}^{(0)} = \boldsymbol{x}$
  for $k = 1, \ldots, l$ do
    $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)}$
    $\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
  end for
  $\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
  $J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \Omega(\theta)$
```

Example Forward Propagation

$$\boldsymbol{x} = \begin{bmatrix} 0 \\ 1 \\ 0.5 \end{bmatrix}$$



$$\begin{bmatrix} 0.2 \\ 0.6 \\ -0.4 \end{bmatrix} = b^{(1)} \qquad \begin{bmatrix} 0.5 \\ -0.3 \end{bmatrix} = b^{(2)} \qquad \begin{bmatrix} -0.4 \\ 0.2 \end{bmatrix} = b^{(3)}$$

$$y = \begin{bmatrix} \quad \end{bmatrix}$$

$$\boldsymbol{W}^{(1)} = \begin{bmatrix} 0.5 & 0.6 & -0.4 \\ 0.4 & 0.1 & 0.2 \\ -0.7 & 0.5 & -0.8 \end{bmatrix} \qquad \boldsymbol{W}^{(2)} = \begin{bmatrix} -0.5 & 0.5 & -0.4 \\ 0.2 & 0.6 & 0.8 \end{bmatrix} \qquad \boldsymbol{W}^{(3)} = \begin{bmatrix} 0.3 & -0.6 \\ 0.9 & 0.2 \end{bmatrix}$$

$$a^{(1)} = \begin{bmatrix} \\ \\ \\ \end{bmatrix} \qquad h^{(1)} = \begin{bmatrix} \\ \\ \\ \end{bmatrix} \qquad a^{(2)} = \begin{bmatrix} \\ \\ \end{bmatrix} \qquad h^{(2)} = \begin{bmatrix} \\ \\ \end{bmatrix} \qquad a^{(3)} = \begin{bmatrix} \\ \\ \end{bmatrix} \qquad h^{(3)} = \begin{bmatrix} \\ \\ \end{bmatrix}$$

Example Forward Propagation

Input layer

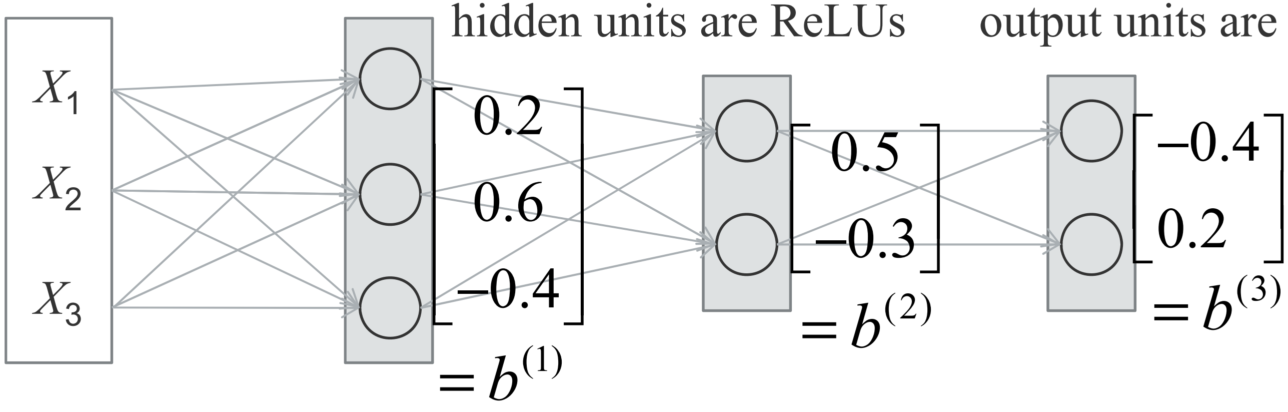
2 hidden layers

hidden units are ReLUs

Output layer

output units are linear

$$\boldsymbol{x} = \begin{bmatrix} 0 \\ 1 \\ 0.5 \end{bmatrix}$$



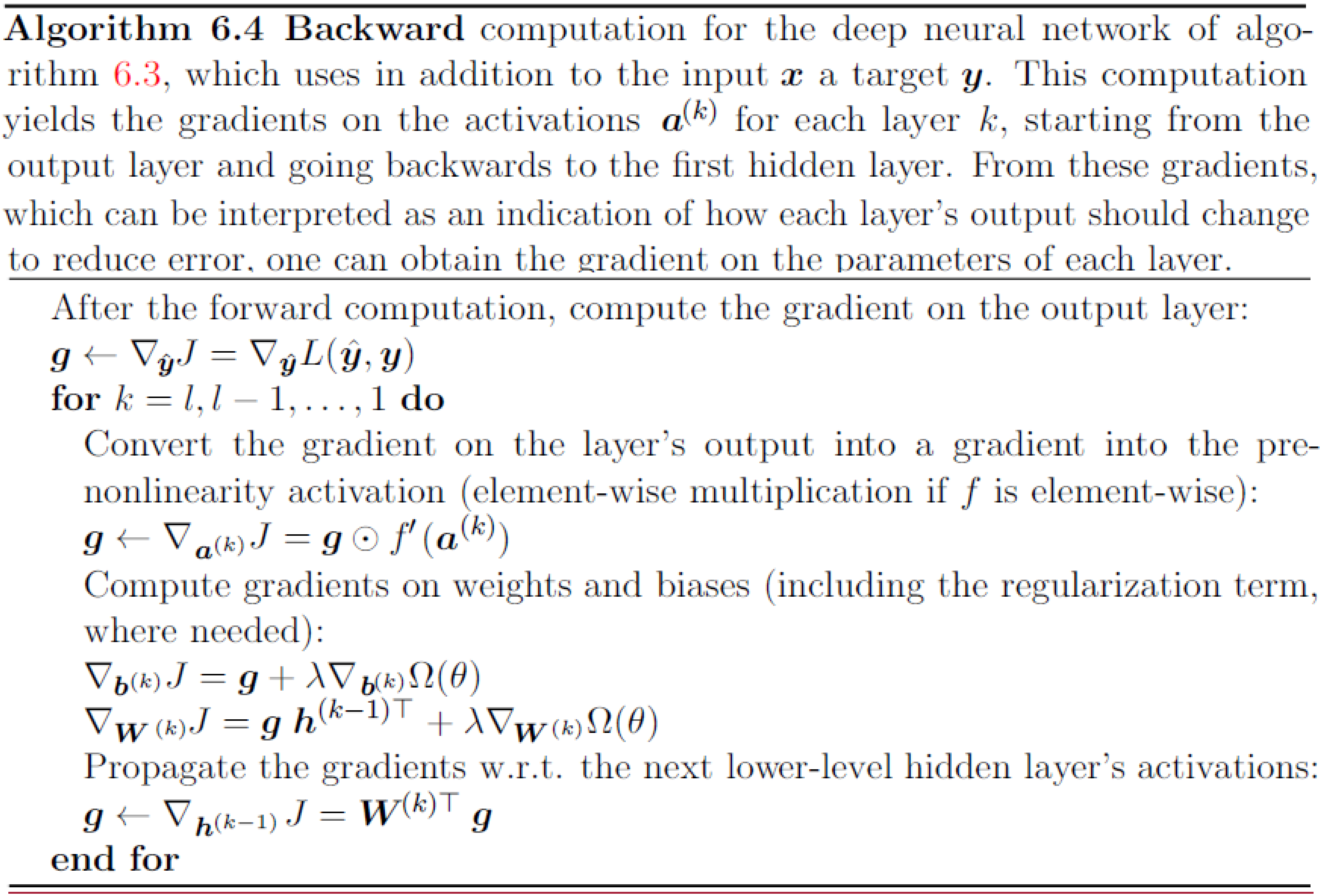
$$\begin{bmatrix} 0.2 \\ 0.6 \\ -0.4 \end{bmatrix} = b^{(1)} \qquad \begin{bmatrix} 0.5 \\ -0.3 \end{bmatrix} = b^{(2)} \qquad \begin{bmatrix} -0.4 \\ 0.2 \end{bmatrix} = b^{(3)}$$

$$y = \begin{bmatrix} -0.4 \\ 0.8 \end{bmatrix}$$

$$\boldsymbol{W}^{(1)} = \begin{bmatrix} 0.5 & 0.6 & -0.4 \\ 0.4 & 0.1 & 0.2 \\ -0.7 & 0.5 & -0.8 \end{bmatrix} \quad \boldsymbol{W}^{(2)} = \begin{bmatrix} -0.5 & 0.5 & -0.4 \\ 0.2 & 0.6 & 0.8 \end{bmatrix} \quad \boldsymbol{W}^{(3)} = \begin{bmatrix} 0.3 & -0.6 \\ 0.9 & 0.2 \end{bmatrix}$$

$$a^{(1)} = \begin{bmatrix} 0.6 \\ 0.8 \\ -0.3 \end{bmatrix} \quad h^{(1)} = \begin{bmatrix} 0.6 \\ 0.8 \\ 0 \end{bmatrix} \quad a^{(2)} = \begin{bmatrix} 0.6 \\ 0.3 \end{bmatrix} h^{(2)} = \begin{bmatrix} 0.6 \\ 0.3 \end{bmatrix} \quad a^{(3)} = \begin{bmatrix} -0.4 \\ 0.8 \end{bmatrix} h^{(3)} = \begin{bmatrix} -0.4 \\ 0.8 \end{bmatrix}$$

Backward Propagation



Algorithm 6.4 Backward computation for the deep neural network of algorithm 6.3, which uses in addition to the input $\boldsymbol{x}$ a target $\boldsymbol{y}$. This computation yields the gradients on the activations $\boldsymbol{a}^{(k)}$ for each layer k, starting from the output layer and going backwards to the first hidden layer. From these gradients, which can be interpreted as an indication of how each layer's output should change to reduce error, one can obtain the gradient on the parameters of each layer.

```
After the forward computation, compute the gradient on the output layer:
$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$
for $k = l, l-1, \ldots, 1$ do
  Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation (element-wise multiplication if $f$ is element-wise):
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$
  Compute gradients on weights and biases (including the regularization term, 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 hidden layer's activations:
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top} \boldsymbol{g}$
end for
```

Example

6.5.5 Symbol-to-Symbol Derivatives

- Algebraic expressions and computational graphs both operate on symbols (variables that do not have specific values).
- When we use or train a neural network, we must assign specific values to these symbols. We replace a symbolic input to the network x with a specific numeric value.
- “Symbol-to-number” approach: take a computational graph and a set of numerical values for the inputs to the graph, then return a set of numerical values describing the gradient at those input values.
- This approach is used by Torch (Collobert *et al.*, 2011b) and Caffe (Jia, 2013).

Symbol-to-Symbol Derivatives

- Symbol-to-symbol approach: take a computational graph and add additional nodes to the graph that provide a symbolic description of the desired derivatives.
- This approach is used by Theano (Bergstra *et al.*, 2010) and TensorFlow (Abadi *et al.*, 2015).
- The primary advantage of this approach is that the derivatives are described in the same language as the original expression.
- Because the derivatives are just another computational graph, it is possible to run back-propagation again, differentiating the derivatives in order to obtain higher derivatives.

Symbol-to-Symbol Approach

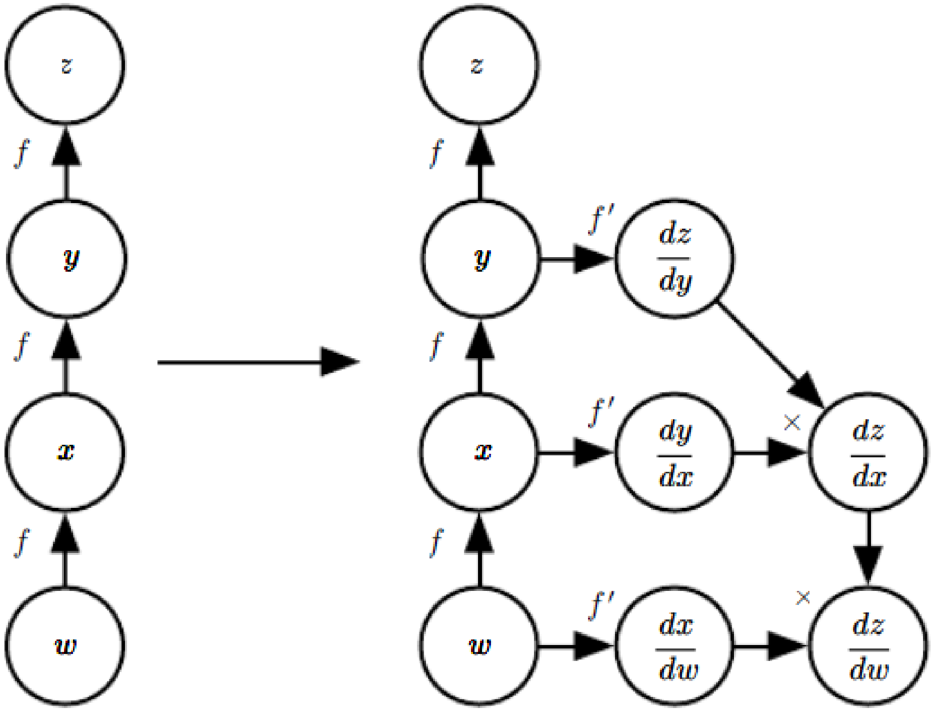


Fig. 6.10: Example of symbol-to-symbol approach: Backpropagation adds nodes to a computational graph how to compute these derivatives. A generic graph evaluation engine can later compute the derivatives for any specific numeric values. *(Left)* graph of $z = f(f(f(w)))$.
(Right) graph for the expression corresponding to dz/dw .

6.5.6 General Back-Propagation

- Backpropagation is very simple: To compute the gradient of some scalar z with respect to one of its ancestors $\boldsymbol{x}$ in the graph, we begin by observing that the gradient with respect to z is given by $dz/dz = 1$.
- We can then compute the gradient with respect to each parent of z by multiplying the current gradient by the Jacobian of the operation that produced z.
- We continue multiplying by Jacobians going backwards through the graph in this way until we reach $\boldsymbol{x}$.
- For any node that may be reached by going backwards from z through two or more paths, we simply sum the gradients arriving from different paths at that node.

General Backpropagation

- More formally, each node in the graph $\mathcal{G}$ corresponds to a variable. For maximum generality, we describe this variable as being a tensor **V**. They subsume scalars, vectors, and matrices.
- We assume that each variable **V** is associated with the following subroutines:
  - get_operation(**V**): returns the operation that computes **V**, represented by the edges coming into **V** in the graph.
  - get_consumers(**V**, $\mathcal{G}$): returns the list of variables that are children of **V** in the computational graph $\mathcal{G}$.
  - get_inputs(**V**, $\mathcal{G}$): returns the list of variables that are parents of **V** in the computational graph $\mathcal{G}$.

Example for bprop operation

- Each operation op is also associated with a bprop operation. This bprop operation can compute a Jacobian-vector product as described by equation 6.47.
- This is how the back-propagation algorithm is able to achieve great generality. Each operation is responsible for knowing how to back-propagate through the edges in the graph that it participates in.

Example for bprop operation

- Assume $\boldsymbol{C} = \boldsymbol{AB}$ (matrix multiplication). Suppose that the gradient of a scalar z with respect to $\boldsymbol{C}$ is given by $\boldsymbol{G}$.
- The matrix multiplication operation is responsible for defining two back-propagation rules, one for each of its input arguments. If we call the bprop method to request the gradient with respect to $\boldsymbol{A}$ given that the gradient on the output is $\boldsymbol{G}$, then the bprop method of the matrix multiplication operation must state that the gradient with respect to $\boldsymbol{A}$ is given by $\boldsymbol{GB}^T$.
- Likewise, if we call the bprop method to request the gradient with respect to $\boldsymbol{B}$, then the matrix operation is responsible for implementing the bprop method and specifying that the desired gradient is given by $\boldsymbol{A}^T\boldsymbol{G}$.

- The back-propagation algorithm itself does not need to know any differentiation rules. It only needs to call each operation’s bprop rules with the right arguments.
- Formally, op.bprop(inputs,**X**,**G**) must return

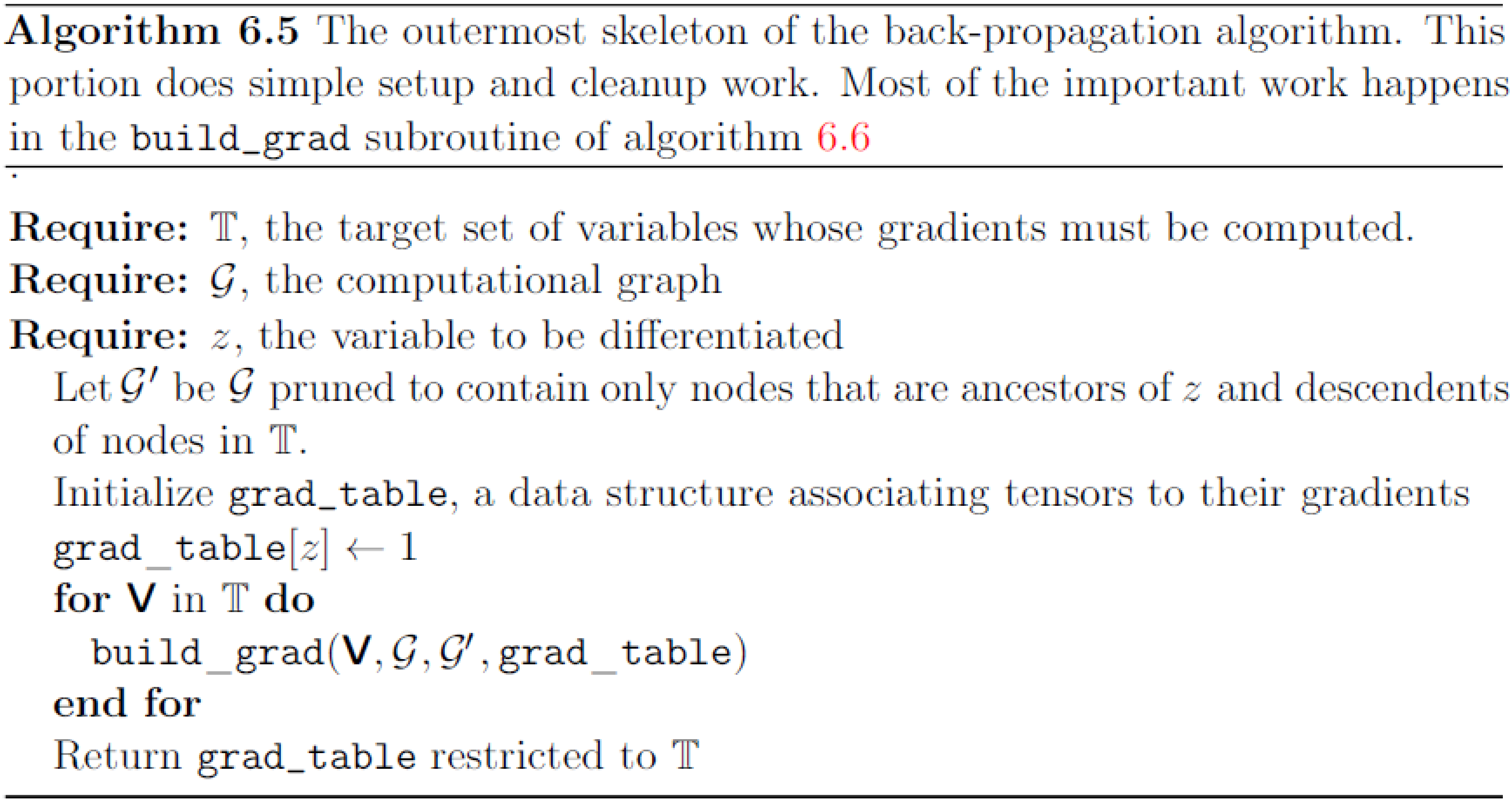
$$\sum_i (\nabla_{\mathbf{X}} \mathrm{op.f}\ (\mathrm{inputs})_i)G_i$$

which is just an implementation of the chain rule as expressed in equation 6.47.

- Here, inputs is a list of inputs that are supplied to the operation, op.f is the mathematical function that the operation implements, **X** is the input whose gradient we wish to compute, and **G** is the gradient on the output of the operation.

- The op.bprop method should always pretend that all of its inputs are distinct, even if they are not.
- For example, if the mul operator is passed two copies of x to compute x^2, the op.bprop method should still return x as the derivative with respect to both inputs. The back-propagation algorithm will later add both of these arguments together to obtain $2x$, which is the correct total derivative on x.
- Software implementations of back-propagation usually provide both the operations and their bprop methods, so that users of deep learning software libraries are able to back-propagate through graphs built using common operations like matrix multiplication, exponents, etc..

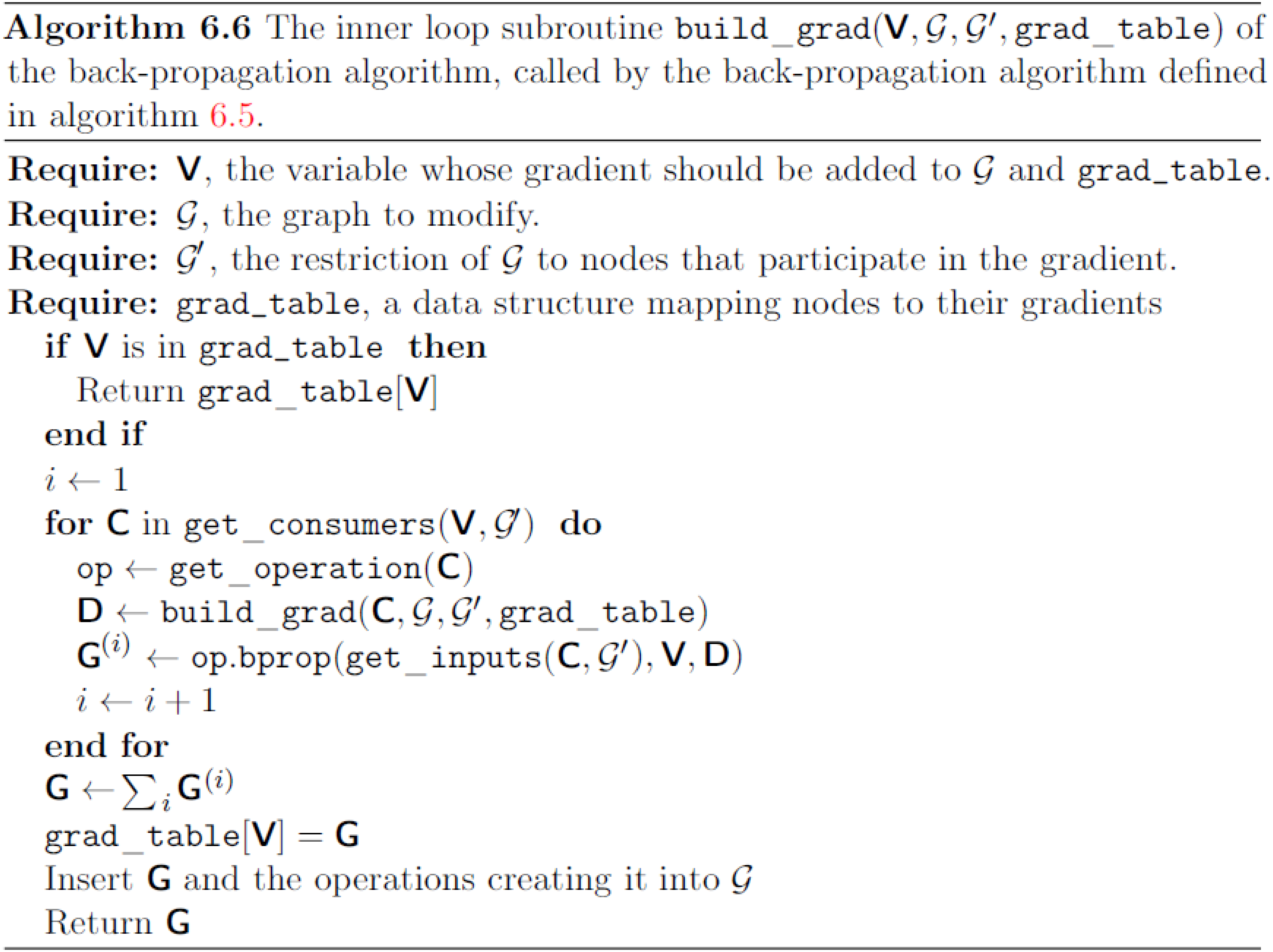
General Backpropagation Algorithm



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

.

```
Require: $\mathbb{T}$, the target set of variables whose gradients must be computed.
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 ancestors of $z$ and descendents
  of nodes in $\mathbb{T}$.
  Initialize grad_table, a data structure associating tensors to their gradients
  grad_table[$z$] $\leftarrow 1$
  for $\mathbf{V}$ in $\mathbb{T}$ do
    build_grad($\mathbf{V}, \mathcal{G}, \mathcal{G}'$, grad_table)
  end for
  Return grad_table restricted to $\mathbb{T}$
```



Algorithm 6.6 The inner loop subroutine build_grad($\mathbf{V}, \mathcal{G}, \mathcal{G}'$, grad_table) of the back-propagation algorithm, called by the back-propagation algorithm defined in algorithm 6.5.

```
Require: V, the variable whose gradient should be added to $\mathcal{G}$ and grad_table.
Require: $\mathcal{G}$, the graph to modify.
Require: $\mathcal{G}'$, the restriction of $\mathcal{G}$ to nodes that participate in the gradient.
Require: grad_table, a data structure mapping nodes to their gradients
  if V is in grad_table then
    Return grad_table[V]
  end if
  $i \leftarrow 1$
  for C in get_consumers(V, $\mathcal{G}'$) do
    op ← get_operation(C)
    D ← build_grad(C, $\mathcal{G}$, $\mathcal{G}'$, grad_table)
    $\mathbf{G}^{(i)} \leftarrow$ op.bprop(get_inputs(C, $\mathcal{G}'$), V, D)
    $i \leftarrow i + 1$
  end for
  $\mathbf{G} \leftarrow \sum_i \mathbf{G}^{(i)}$
  grad_table[V] = G
  Insert G and the operations creating it into $\mathcal{G}$
  Return G
```

Runtime analysis of bprop

- Back-propagation was developed in order to avoid computing the same subexpression in the chain rule multiple times, which might require exponential runtime.
- If we assume that each operation evaluation has roughly the same cost, then we may analyze the computational cost in terms of the number of operations in the computational graph executed.
- Computing a gradient in a graph with n nodes will never execute more than $O(n^2)$ operations or store the output of more than $O(n^2)$ operations. Here we are counting operations in the comp. graph, not individual operations of the underlying hardware, so the runtime of each operation may be highly variable.

- For example, multiplying two matrices that each contain millions of entries might correspond to a single operation in the graph. We can see that computing the gradient requires as most $O(n^2)$ operations because the forward propagation stage will at worst execute all n nodes in the original graph.
- The back-propagation algorithm adds one Jacobian-vector product, which should be expressed with $O(1)$ nodes, per edge in the original graph. Because the computational graph is a directed acyclic graph it has at most $O(n^2)$ edges.
- For the kinds of graphs that are commonly used in practice, the situation is even better.

- Most neural network cost functions are roughly chain-structured, causing back-propagation to have $O(n)$ cost. This is far better than the naive approach, which might need exponentially many nodes.
- This potentially exponential cost can be seen by rewriting the recursive chain rule non-recursively:

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{\substack{\text{path}(u^{(\pi_1)}, u^{(\pi_2)}, \ldots, u^{(\pi_t)}), \\ \text{from } \pi_1 = t \text{ to } \pi_t = n}} \prod_{k=2}^{t} \frac{\partial u^{(\pi_k)}}{\partial u^{(\pi_{k-1})}}$$

- Since the number of paths from node j to node n can grow exponentially in the length of these paths, the number of terms in the above sum can grow exponentially with the depth of the forward propagation graph.

- This large cost would be incurred because the same computation for $\partial^{(i)}/\partial u^{(j)}$ would be redone many times.
- To avoid such recomputation, we can think of back-propagation as a table-filling algorithm that takes advantage of storing intermediate results $\partial u^{(n)}/\partial u^{(i)}$. Each node in the graph has a corresponding slot in a table to store the gradient for that node. By filling in these table entries in order, back-propagation avoids repeating many common subexpressions.
- This table-filling strategy is sometimes called **dynamic programming**.

6.5.7 Example: Backprop for MLP

- Here we develop a multilayer perception with a single hidden layer. To train this model, we will use minibatch stochastic gradient descent.
- The backpropagation algorithm is used to compute the gradient of the cost on a single minibatch. We use a minibatch of examples from the training set formatted as design matrix $\boldsymbol{X}$ and a vector of associated class labels $\boldsymbol{y}$.
- The network computes a layer of hidden features $\boldsymbol{H}$ = max{0, $\boldsymbol{X}\boldsymbol{W}$(1)}. To simplify the presentation we do not use biases in this model.
- We assume that our graph language includes a ReLU operation that can compute max{0, $\boldsymbol{Z}$} elementwise.

Example: Backprop for MLP

- The predictions of the unnormalized log probabilities over classes are then given by $\boldsymbol{HW}^{(2)}$.
- We assume that our graph language includes a cross_entropy operation that computes the cross-entropy between the targets $\boldsymbol{y}$ and $\boldsymbol{HW}^{(2)}$. The resulting crossentropy defines the cost J_{MLE}. Minimizing this cross-entropy performs max. likelihood estim. of the classifier.
- We also include a regularization term. The total cost

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

consists of the cross-entropy and a weight decay term with coefficient λ. The comp. graph is shown in fig. 6.11.

Example: Backprop for MLP

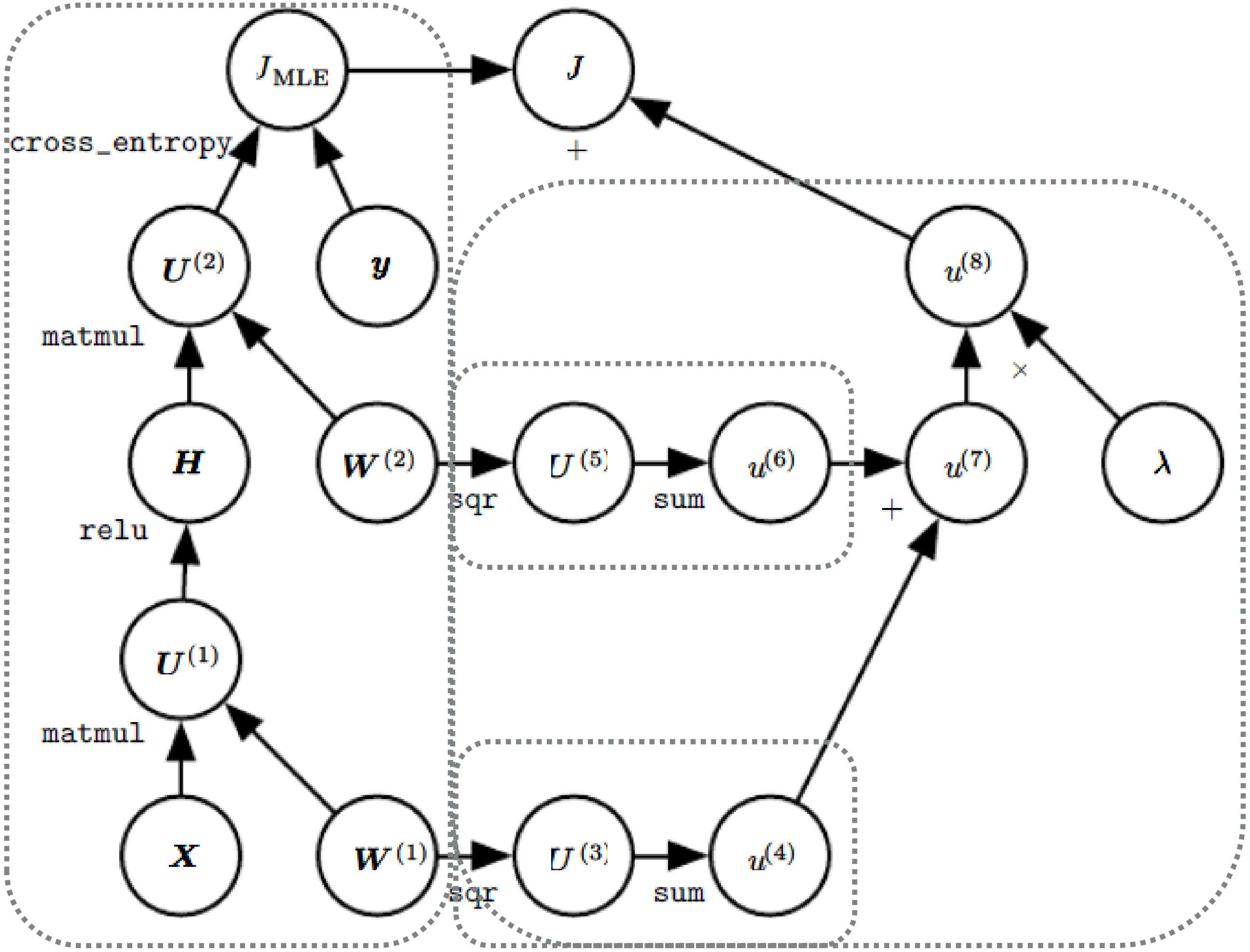


Fig. 6.11: The computational graph used to compute the cost used to train our example of a single-layer MLP using the cross-entropy loss and weight decay.

- The comp. graph for the gradient of this example is large and tedious to draw or to read.
- We can roughly trace out the behavior of the backprop algorithm by looking at the forward propagation graph in figure 6.11.
- To train, we wish to compute both $\nabla_{W^{(1)}} J$ and $\nabla_{W^{(2)}} J$.
- There are two different paths leading backward from J to the weights: one through the cross-entropy cost, and one through the weight decay cost.
- The weight decay cost is relatively simple; it will always contribute $2\lambda \boldsymbol{W}^{(i)}$ to the gradient on $\boldsymbol{W}^{(i)}$.
- The other path through the cross-entropy cost is slightly more complicated.

- Let $\boldsymbol{G}$ be the gradient on the unnormalized log probab. $\boldsymbol{U}^{(2)}$ provided by the cross_entropy operation.
- The backpropagation algorithm now needs to explore two different branches:
- On the shorter branch, it adds $\boldsymbol{H}^{T}\boldsymbol{G}$ to the gradient on $\boldsymbol{W}^{(2)}$, using the backpropagation rule for the second argument to the matrix multiplication operation.
- The other branch corresponds to the longer chain descending further along the network. First, the back-propagation algorithm computes $\nabla_{\boldsymbol{H}}J = \boldsymbol{G}\,\boldsymbol{W}^{(2)}$ using the back-propagation rule for the first argument to the matrix multiplication operation.

- Next, the ReLU operation uses its backpropagation rule to zero out components of the gradient corresponding to entries of $\boldsymbol{U}^{(1)}$ that were < 0. Let the result be called $\boldsymbol{G'}$.
- The last step of the back-propagation algorithm is to use the backpropagation rule for the second argument of the matmul operation to add $\boldsymbol{X}^T \boldsymbol{G'}$ to the gradient on $\boldsymbol{W}$(1).
- After these gradients have been computed, it is the responsibility of the gradient descent algorithm to use these gradients to update the parameters.
- For the MLP, the computational cost is dominated by the cost of matrix multiplication. During the forward stage, we multiply by each weight matrix, resulting in $O(\boldsymbol{w})$ multiply-adds, where $\boldsymbol{w}$ is the number of weights.

- During the backward propagation stage, we multiply by the transpose of each weight matrix, which has the same computational cost.
- The main memory cost of the algorithm is that we need to store the input to the nonlinearity of the hidden layer. This value is stored from the time it is computed until the backward pass has returned to the same point. The memory cost is thus O($m\ n_h$), where m is the number of examples in the minibatch and n_h is the number of hidden units.

6.5.8 Complications

- Operations might need to return more values than only a single tensor.
- Memory consumption may need to be controlled: Back-propagation often involves summation of many tensors together. If each tensor is computed separately, and all of them are added in a second step, this has a high memory bottleneck. This can be avoided by maintaining a single buffer and adding each value to that buffer as it is computed.
- Real-world implementations of back-propagation also need to handle various data types, such as 16-bit or 32-bit or 64-bit floating point, and integer values.