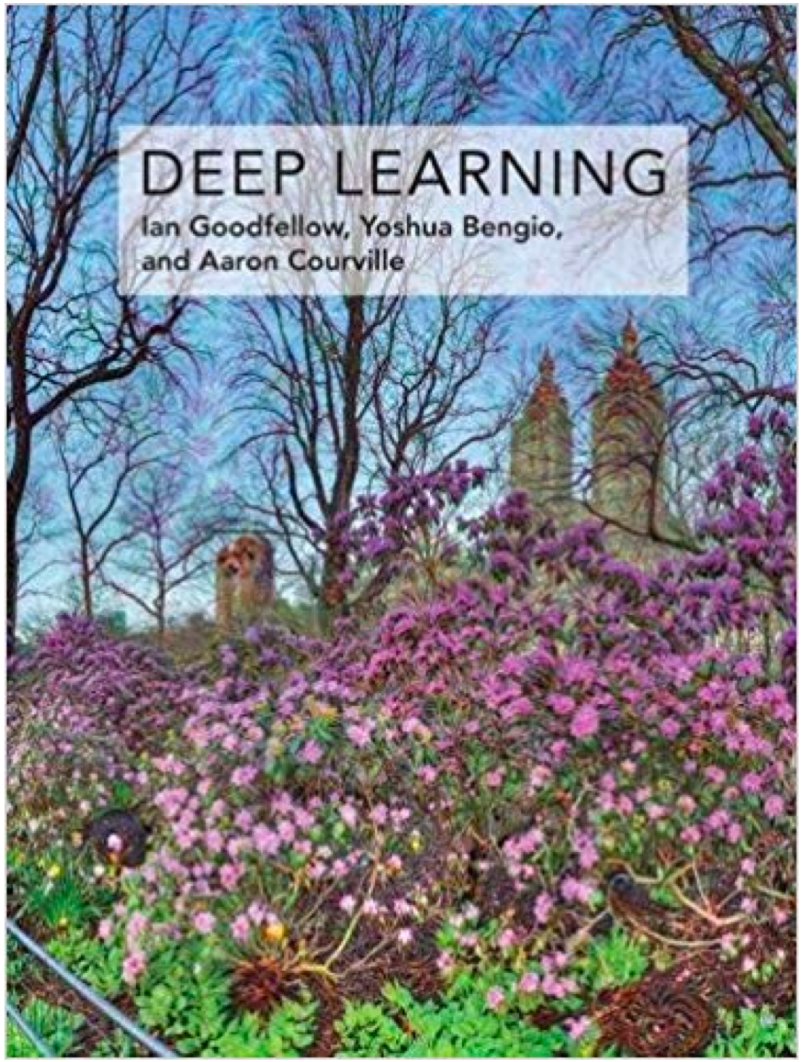
國立交通大學
National Chiao Tung University

Deep Learning
深度學習
Fall 2018



Deep Feedforward Networks
(Chapter 6.1-6.4)

Prof. Chia-Han Lee
李佳翰 副教授

Deep feedforward networks

- Deep feedforward networks, also called feedforward neural networks, or multilayer perceptrons (MLPs), are the quintessential deep learning models.
- The goal of a feedforward network is to approximate some function f^*. For example, for a classifier, $y = f^*(x)$ maps an input $\boldsymbol{x}$ to a category y. 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.
- These models are called feedforward because information flows through the function being evaluated from $\boldsymbol{x}$, through the intermediate computations used to define f, and finally to the output $\boldsymbol{y}$. There are no feedback connections in which outputs of the model are fed back into itself.

Deep feedforward networks

- Feedforward neural networks are called networks because they are typically represented by composing together many different functions. The model is associated with a directed acyclic graph describing how the functions are composed together.

- For example, we might have three functions $f^{(1)}$,$f^{(2)}$, and $f^{(3)}$ connected in a chain, to form $f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$. These chain structures are the most commonly used structures of neural networks. In this case, $f^{(1)}$ is called the first layer of the network, $f^{(2)}$ is called the second layer, and so on. The overall length of the chain gives the depth of the model. The name “deep learning” arose from this terminology. The final layer of a feedforward network is called the output layer.

Deep feedforward networks

- During neural network training, we drive $f(\boldsymbol{x})$ to match $f^*(\boldsymbol{x})$. The training data provides us with noisy, approximate examples of $f^*(\boldsymbol{x})$ evaluated at different training points.

- Each example x is accompanied by a label $y \approx f^*(\boldsymbol{x})$.

- The training examples specify directly what the output layer must do at each point $\boldsymbol{x}$; it must produce a value that is close to y.

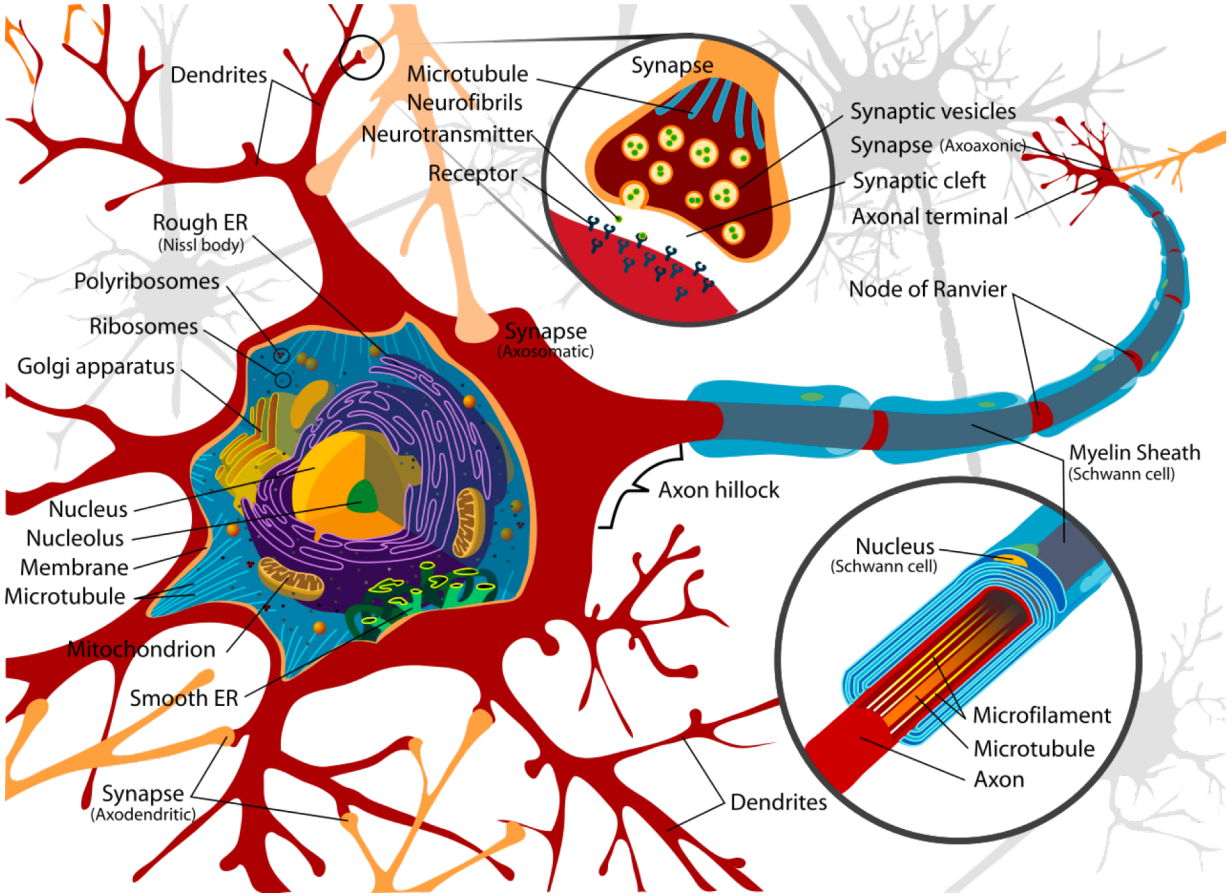
Deep feedforward networks

- The behavior of the other layers is not directly specified by the training data. The learning algorithm must decide how to use those layers to produce the desired output, but the training data do not say what each individual layer should do. Instead, 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, they are called hidden layers.

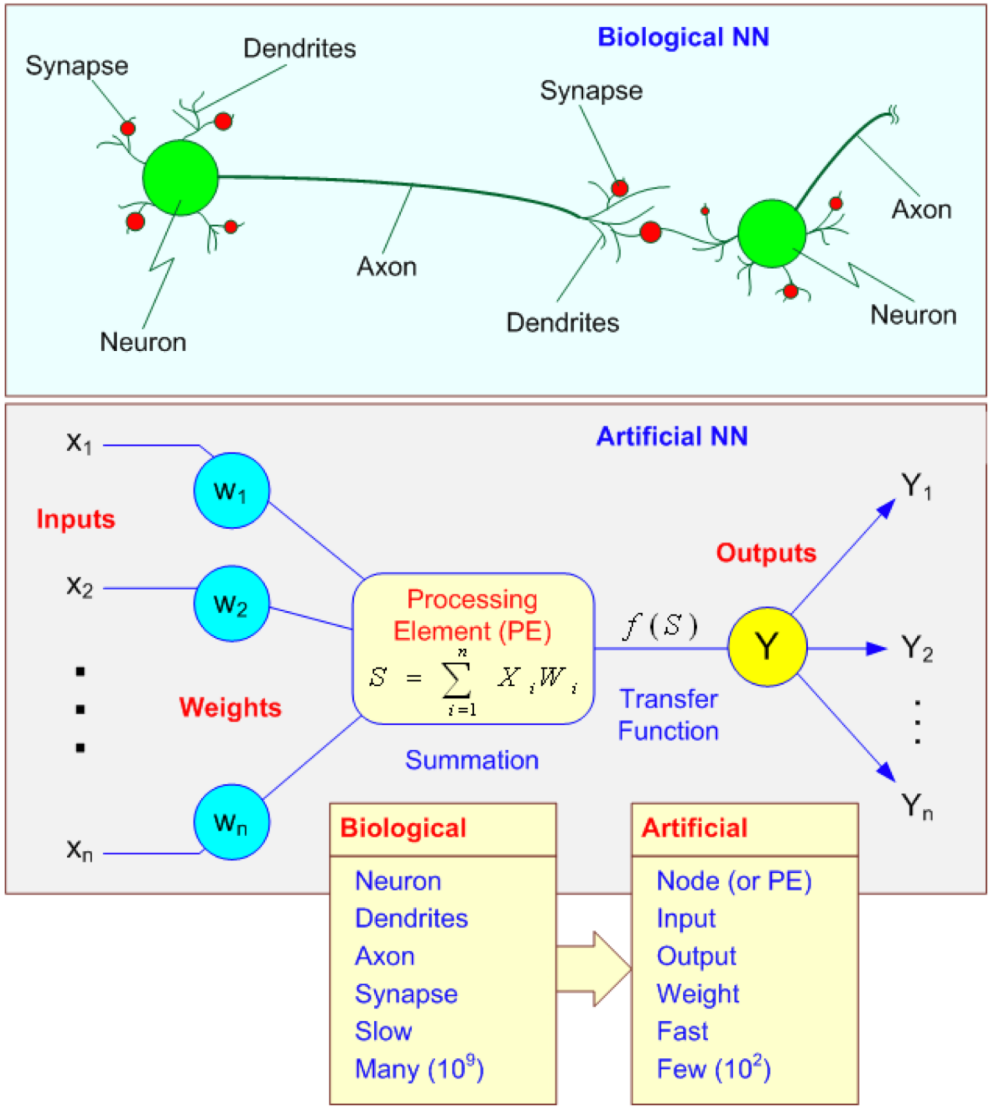
Deep feedforward networks

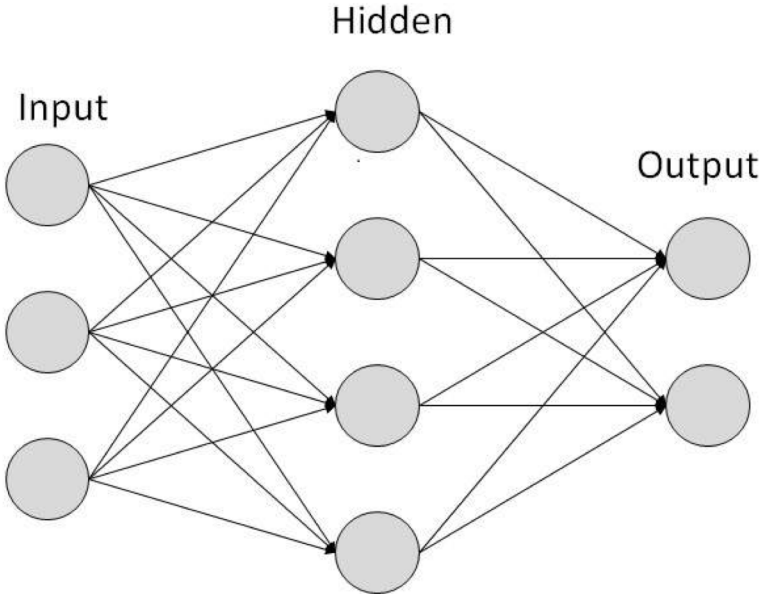
- These networks are called neural because they are loosely inspired by neuroscience.



- https://upload.wikimedia.org/wikipedia/commons/thumb/a/a9/Complete_neuron_cell_diagram_en.svg/1280px-Complete_neuron_cell_diagram_en.svg.png

Deep feedforward networks





- http://3.bp.blogspot.com/-X81SfTlAm5Y/T4y7VyIo7cI/AAAAAAAAAIA/sLAzr5qYNYs/s1600/biological_neuron_vs_ANN.png

Deep feedforward networks

- Each hidden layer of the network is typically vector valued. The dimensionality of these hidden layers determines the width of the model. Each element of the vector may be interpreted as playing a role analogous to a neuron.

- Rather than thinking of the layer as representing a single vector-to-vector function, we can also think of the layer as consisting of many units that act in parallel, each representing a vector-to-scalar function. Each unit resembles a neuron in the sense that it receives input from many other units and computes its own activation value.

Deep feedforward networks

- The idea of using many layers of vector-valued representations is drawn from neuroscience. The choice of the functions $f^{(i)}(\boldsymbol{x})$ used to compute these representations is also loosely guided by neuroscientific observations about the functions that biological neurons compute.

- Modern neural network research, however, is guided by many mathematical and engineering disciplines, and the goal of neural networks is not to perfectly model the brain. It is best to think of feedforward networks as function approximation machines that are designed to achieve statistical generalization, occasionally drawing some insights from what we know about the brain, rather than as models of brain function.

Deep feedforward networks

- One way to understand feedforward networks is to begin with linear models and consider how to overcome their limitations. Linear models, such as logistic regression and linear regression, are appealing because they can be fit efficiently and reliably, either in closed form or with convex optimization.

- Linear models also have the obvious defect that the model capacity is limited to linear functions, so the model cannot understand the interaction between any two input variables.

Deep feedforward networks

- To extend linear models to represent nonlinear functions of $\boldsymbol{x}$, we can apply the linear model not to $\boldsymbol{x}$ itself but to a transformed input $\phi(\boldsymbol{x})$, where ϕ is a nonlinear transformation.

- Equivalently, we can apply the kernel trick to obtain a nonlinear learning algorithm based on implicitly applying the ϕ mapping. We can think of ϕ as providing a set of features describing $\boldsymbol{x}$, or as providing a new representation for $\boldsymbol{x}$.

Deep feedforward networks

The question is then how to choose the mapping ϕ.

1. One option is to use a very generic ϕ, such as the infinite-dimensional ϕ that is implicitly used by kernel machines based on the radial basis function (RBF) kernel.
   - If $\phi(\boldsymbol{x})$ is of high enough dimension, we can always have enough capacity to fit the training set, but generalization to the test set often remains poor.
   - Very generic feature mappings are usually based only on the principle of local smoothness and do not encode enough prior information to solve advanced problems.

Deep feedforward networks

2. Another option is to manually engineer ϕ.
   - Until the advent of deep learning, this was the dominant approach.
   - It requires decades of human effort for each separate task, with practitioners specializing in different domains, such as speech recognition or computer vision, and with little transfer between domains.

Deep feedforward networks

3. The strategy of deep learning is to learn ϕ.
   - In this approach, we have a model $y = f(\boldsymbol{x}; \boldsymbol{\theta}, \boldsymbol{\omega}) = \phi(\boldsymbol{x}; \boldsymbol{\theta})^{\mathrm{T}} \boldsymbol{\omega}$. We now have parameters $\boldsymbol{\theta}$ that we use to learn ϕ from a broad class of functions, and parameters $\boldsymbol{\omega}$ 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.
   - This approach is the only one of the three that gives up on the convexity of the training problem, but the benefits outweigh the harms.
   - In this approach, 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.

Deep feedforward networks

3. The strategy of deep learning is to learn ϕ. (cont'd)
   - If we wish, this approach can capture the benefit of the first approach by being highly generic—we do so by using a very broad family $\phi(\boldsymbol{x};\boldsymbol{\theta})$.
   - Deep learning can also capture the benefit of the second approach. Human practitioners can encode their knowledge to help generalization by designing families $\phi(\boldsymbol{x};\boldsymbol{\theta})$ that they expect will perform well. The advantage is that the human designer only needs to find the right general function family rather than finding precisely the right function.

Example: Learning XOR

- To make the idea of a feedforward network more concrete, we begin with an example of a fully functioning feedforward network on a very simple task: learning the XOR function.

- The XOR function ("exclusive or") is an operation on two binary values, x_1 and x_2. When exactly one of these binary values is equal to 1, the XOR function returns 1. Otherwise, it returns 0.

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

Example: Learning XOR

- In this simple example, we will not be concerned with statistical generalization. We want our network to perform correctly on the four points $\mathbb{X} = \{[0,0]^{\mathrm{T}}, [0,1]^{\mathrm{T}}, [1,0]^{\mathrm{T}}, \text{and } [1,1]^{\mathrm{T}}\}$. We will train the network on all four of these points. The only challenge is to fit the training set.

- We can treat this problem as a regression problem and use a mean squared error loss function. We have chosen this loss function to simplify the math for this example as much as possible. In practical applications, MSE is usually not an appropriate cost function for modeling binary data.

Example: Learning XOR

- Evaluated on our whole training set, the MSE loss function is

$$J(\boldsymbol{\theta}) = \frac{1}{4} \sum_{\boldsymbol{x} \in \mathbb{X}} (f^*(\boldsymbol{x}) - f(\boldsymbol{x}; \boldsymbol{\theta}))^2 . \tag{6.1}$$

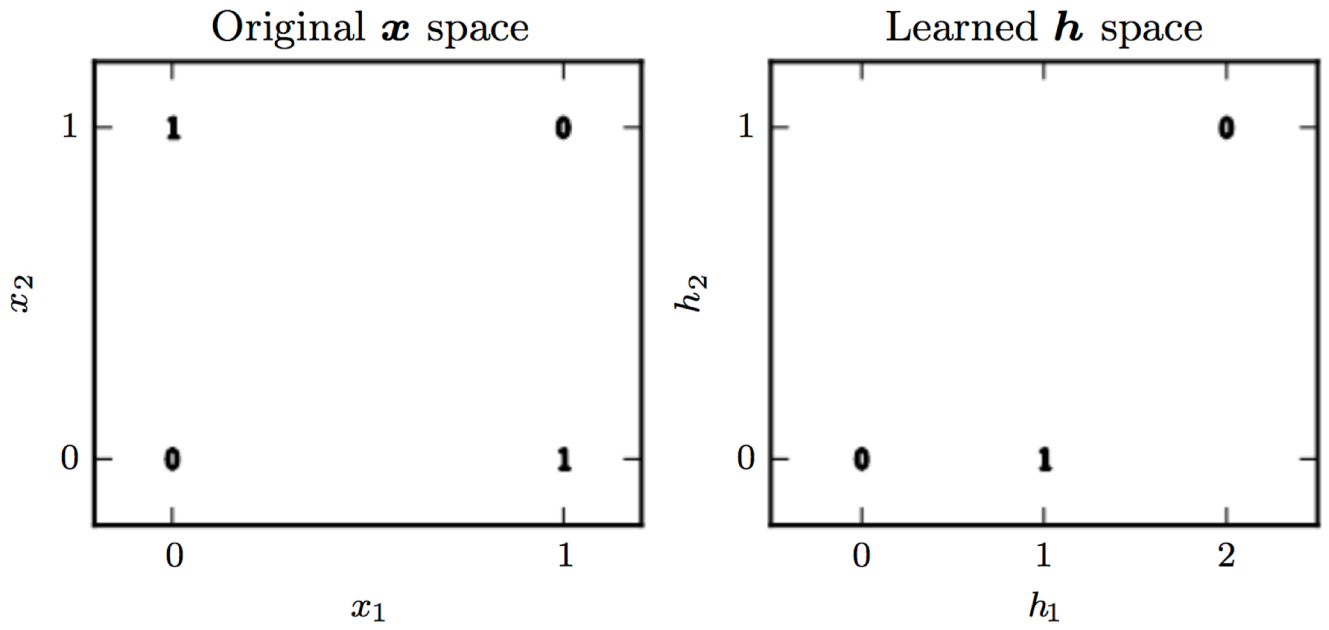
- Now we must choose the form of our model, $f(x; \boldsymbol{\theta})$. Suppose that we choose a linear model, with $\boldsymbol{\theta}$ consisting of $\boldsymbol{\omega}$ and b. Our model is defined to be

$$f(\boldsymbol{x}; \boldsymbol{w}, b) = \boldsymbol{x}^\top \boldsymbol{w} + b. \tag{6.2}$$

- We can minimize $J(\boldsymbol{\theta})$ in closed form with respect to $\boldsymbol{\omega}$ and b using the normal equations.

Example: Learning XOR

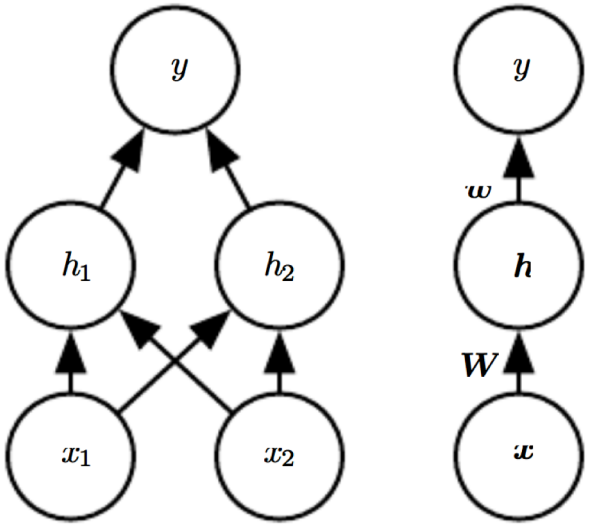
- After solving the normal equations, we obtain $\boldsymbol{\omega} = 0$ and $b = \frac{1}{2}$. The linear model simply outputs 0.5 everywhere.



- One way to solve this problem is to use a model that learns a different feature space in which a linear model is able to represent the solution.

Example: Learning XOR

- Specifically, we will introduce a simple feedforward network with one hidden layer containing two hidden units.



- This feedforward network has a vector of hidden units h that are computed by a function $f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})$. The values of these hidden units are then used as the input for a second layer. The second layer is the output layer of the network. The output layer is still just a linear regression model, but now it is applied to $\boldsymbol{h}$ rather than to $\boldsymbol{x}$.
- The network now contains two functions chained together, $\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})$ and $y = f^{(2)}(\boldsymbol{h}; \boldsymbol{\omega}, b)$, with the complete model being $f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{\omega}, b) = f^{(2)}(f^{(1)}(\boldsymbol{x}))$.

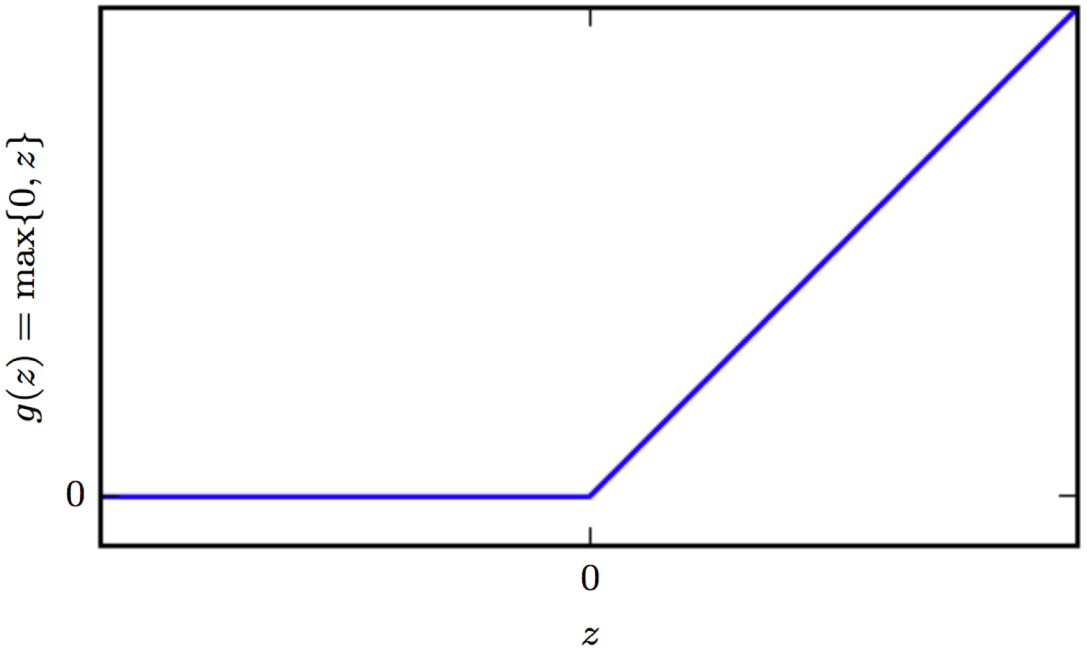
Example: Learning XOR

- What function should $f^{(1)}$ compute? If $f^{(1)}$ were linear, then the feedforward network as a whole would remain a linear function of its input. Ignoring the intercept terms for the moment, suppose $f^{(1)}(\boldsymbol{x}) = \boldsymbol{W}^{\mathrm{T}}\boldsymbol{x}$ and $f^{(2)}(\boldsymbol{h}) = \boldsymbol{h}^{T}\boldsymbol{\omega}$. Then $f(\boldsymbol{x}) = \boldsymbol{\omega}^{T}\boldsymbol{W}^{\mathrm{T}}\boldsymbol{x}$. We could represent this function as $f(\boldsymbol{x}) = \boldsymbol{x}^{\mathrm{T}}\boldsymbol{\omega}'$ where $\boldsymbol{\omega}' = \boldsymbol{W}\boldsymbol{\omega}$.

- Clearly, we must use a nonlinear function to describe the features. Most neural networks do so using an affine transformation controlled by learned parameters, followed by a fixed nonlinear function called an activation function. We use that strategy here, by defining $\boldsymbol{h} = g(\boldsymbol{W}^{\mathrm{T}}\boldsymbol{x} + \boldsymbol{c})$, where $\boldsymbol{W}$ provides the weights of a linear transformation and $\boldsymbol{c}$ the biases.

Example: Learning XOR

- The activation function g is typically chosen to be a function that is applied element-wise, with $h_i = g(\boldsymbol{x}^{\mathrm{T}} W_{:,i} + c_i)$. In modern neural networks, the default recommendation is to use the rectified linear unit, or ReLU, defined by the activation function $g(z) = \max\{0, z\}$.



Example: Learning XOR

- We can now specify our complete network as

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^\top \max\{0, \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{c}\} + b. \tag{6.3}$$

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

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \tag{6.4}$$

$$\boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \tag{6.5}$$

$$\boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, \tag{6.6}$$

and $b = 0$.

Example: Learning XOR

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

$$\boldsymbol{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}. \tag{6.7}$$

- The first step in the neural network is to multiply the input matrix by the first layer's weight matrix:

$$\boldsymbol{XW} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}. \tag{6.8}$$

Example: Learning XOR

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

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}. \tag{6.9}$$

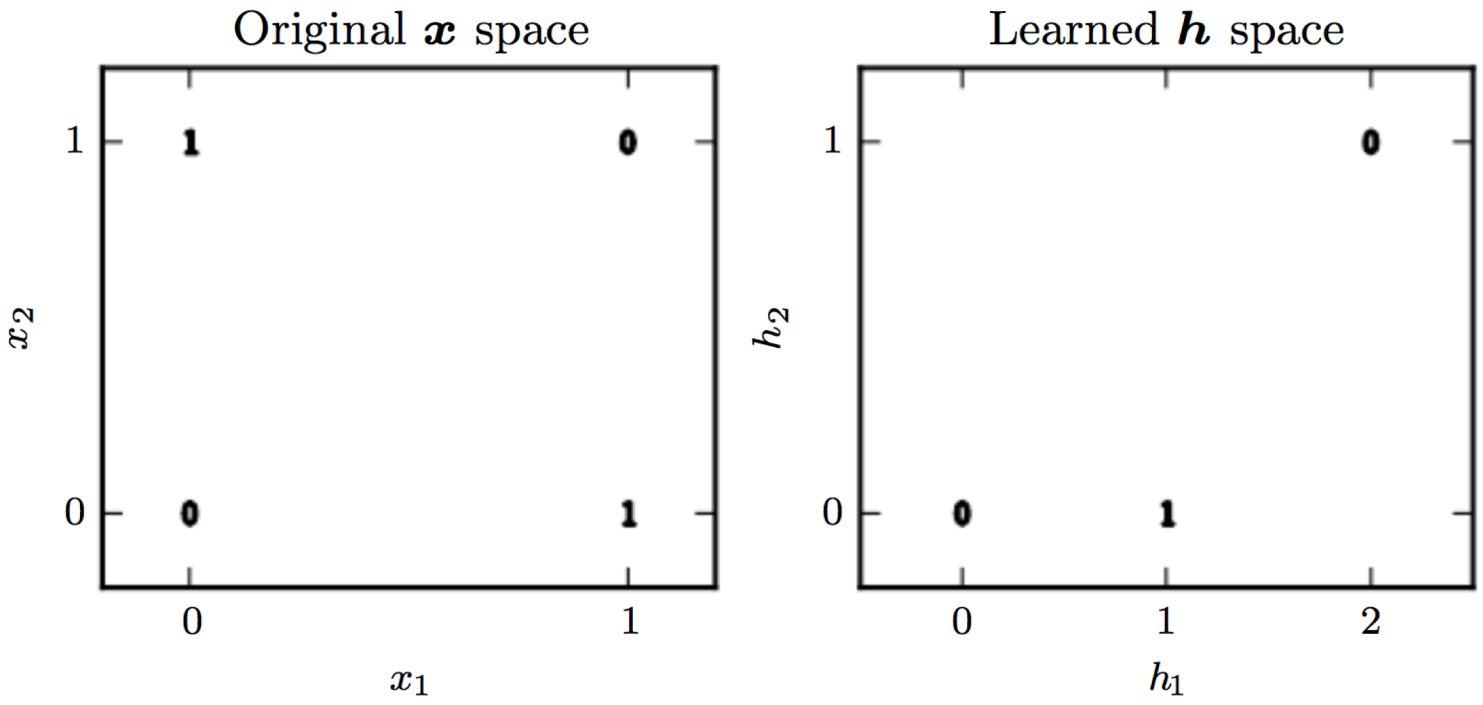
- In this space, all the examples lie along a line with slope 1. As we move along this line, the output needs to begin at 0, then rise to 1, then drop back down to 0. A linear model cannot implement such a function.

- To finish computing the value of h for each example, we apply the rectified linear transformation:

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}. \tag{6.10}$$

Example: Learning XOR

- This transformation has changed the relationship between the examples. They no longer lie on a single line. As shown in figure 6.1, they now lie in a space where a linear model can solve the problem.



Example: Learning XOR

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

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}. \tag{6.11}$$

- The neural network has obtained the correct answer for every example in the batch.

- In this example, we simply specified the solution, then showed that it obtained zero error. In a real situation, there might be billions of model parameters and billions of training examples, so one cannot simply guess the solution as we did here.

Example: Learning XOR

- Instead, a gradient-based optimization algorithm can find parameters that produce very little error. The solution we described to the XOR problem is at a global minimum of the loss function, so gradient descent could converge to this point.

- There are other equivalent solutions to the XOR problem that gradient descent could also find. The convergence point of gradient descent depends on the initial values of the parameters. In practice, gradient descent would usually not find clean, easily understood, integer-valued solutions like the one here.

Gradient-based learning

- The largest difference between the linear models we have seen so far and neural networks is that the nonlinearity of a neural network causes most interesting loss functions to become nonconvex.

- This means that neural networks are usually trained by using iterative, gradient-based optimizers that merely drive the cost function to a very low value, rather than the linear equation solvers used to train linear regression models or the convex optimization algorithms with global convergence guarantees used to train logistic regression or SVMs.

Gradient-based learning

- Stochastic gradient descent applied to nonconvex loss functions 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 of course train models such as linear regression and SVMs with gradient descent too, and in fact this is common when the training set is extremely large. From this point of view, training a neural network is not much different from training any other model. Computing the gradient is slightly more complicated for a neural network but can still be done efficiently and exactly.

Cost functions

- The cost functions for neural networks are more or less the same as those for other parametric models, such as linear models.

- 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. Rather than predicting a complete probability distribution over $\boldsymbol{y}$, we merely predict some statistic of $\boldsymbol{y}$ conditioned on $\boldsymbol{x}$. Specialized loss functions enable us to train a predictor of these estimates.

Cost functions

- The total cost function used to train a neural network will often combine one of the primary cost functions described here with a regularization term.

- The weight decay approach used for linear models is also directly applicable to deep neural networks and is among the most popular regularization strategies.

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. This cost function is given by

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x}). \tag{6.12}$$

- The specific form of the cost function changes from model to model, depending on the specific form of $\log p_{\text{model}}$.

- The expansion of the above equation typically yields some terms that do not depend on the model parameters and may be discarded.

Maximum likelihood

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

$$J(\theta) = \frac{1}{2}\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}}||\boldsymbol{y} - f(\boldsymbol{x}; \boldsymbol{\theta})||^2 + \text{const}, \tag{6.13}$$

  up to a scaling factor of $\frac{1}{2}$ and a term that does not depend on $\boldsymbol{\theta}$. The discarded constant is based on the variance of the Gaussian distribution, which in this case we chose not to parametrize.

- Previously, we saw that the equivalence between maximum likelihood estimation with an output distribution and minimization of mean squared error holds for a linear model, but in fact, the equivalence holds regardless of the $f(\boldsymbol{x}; \boldsymbol{\theta})$ used to predict the mean of the Gaussian.

Maximum likelihood

- One recurring theme throughout neural network design is that the gradient of the cost function must be large and predictable enough to serve as a good guide for the learning algorithm.

- Functions that saturate (become very flat) undermine this objective because they make the gradient become very small. In many cases this happens because the activation functions used to produce the output of the hidden units or the output units saturate.

- The negative log-likelihood helps to avoid this problem for many models. Several output units involve an exp function that can saturate when its argument is very negative. The log function in the negative log-likelihood cost function undoes the exp of some output units.

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.

- Any kind of neural network unit that may be used as an output can also be used as a hidden unit. 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.

Linear units for Gaussian output distributions

- One simple kind of output unit is based on an affine transformation with no nonlinearity. These are often just called linear units.

- Given features $\boldsymbol{h}$, a layer of linear output units produces a vector $\hat{\boldsymbol{y}} = \boldsymbol{W}^{\mathrm{T}}\boldsymbol{h} + \boldsymbol{b}$.

- Linear output layers are often used to produce the mean of a conditional Gaussian distribution:

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; \hat{\boldsymbol{y}}, \boldsymbol{I}). \tag{6.17}$$

  Maximizing the log-likelihood is then equivalent to minimizing the mean squared error.

Linear units for Gaussian output distributions

- The maximum likelihood framework makes it straightforward to learn the covariance of the Gaussian too, or to make the covariance of the Gaussian be a function of the input. However, the covariance must be constrained to be a positive definite matrix for all inputs. It is difficult to satisfy such constraints with a linear output layer, so typically other output units are used to parametrize the covariance.

- Because linear units do not saturate, they pose little difficulty for gradient-based optimization algorithms and may be used with a wide variety of optimization algorithms.

Sigmoid units for Bernoulli output distributions

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

Sigmoid units for Bernoulli output distributions

- Suppose we were to use a linear unit and threshold its value to obtain a valid probability:

$$P(y = 1 \mid \boldsymbol{x}) = \max\left\{0, \min\left\{1, \boldsymbol{w}^\top \boldsymbol{h} + b\right\}\right\}. \tag{6.18}$$

- This would indeed define a valid conditional distribution, but we would not be able to train it very effectively with gradient descent.

- Any time that $\boldsymbol{\omega}^{\mathrm{T}}\boldsymbol{h} + b$ strayed outside the unit interval, the gradient of the output of the model with respect to its parameters would be $\mathbf{0}$. A gradient of $\mathbf{0}$ is typically problematic because the learning algorithm no longer has a guide for how to improve the corresponding parameters.

Sigmoid units for Bernoulli output distributions

- It is better to use a different approach that ensures there is always a strong gradient whenever the model has the wrong answer.

- This approach is based on using sigmoid output units combined with maximum likelihood. A sigmoid output unit is defined by

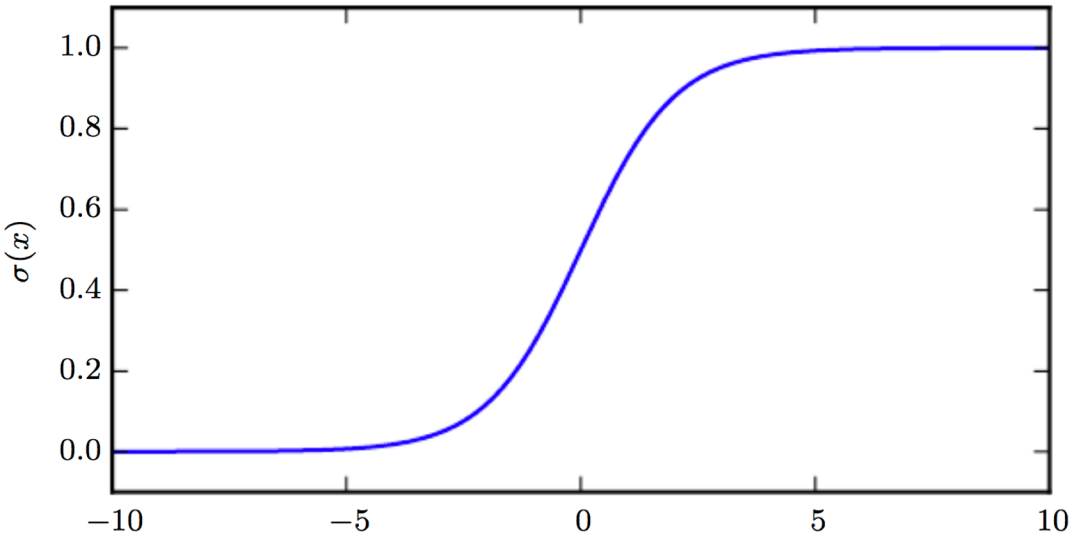
$$\hat{y} = \sigma\left(\boldsymbol{w}^\top \boldsymbol{h} + b\right), \tag{6.19}$$

where σ is the logistic sigmoid function

$$\sigma(x) = \frac{1}{1 + \exp(-x)}. \tag{3.30}$$

Sigmoid units for Bernoulli output distributions

- The sigmoid function saturates when its argument is very positive or very negative, meaning that the function becomes very flat and insensitive to small changes in its input.



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

Sigmoid units for Bernoulli output distributions

- We omit the dependence on $\boldsymbol{x}$ for the moment to discuss how to define a probability distribution over y using the value z.

- The $\mathrm{sigmoid}$ can be motivated by constructing an unnormalized probability distribution $\tilde{P}(y)$, which does not sum to 1. We can then divide by an appropriate constant to obtain a valid probability distribution.

- If we begin with the assumption that the unnormalized log probabilities are linear in y and z, we can exponentiate to obtain the unnormalized probabilities.

Sigmoid units for Bernoulli output distributions

- We then normalize to see that this yields a Bernoulli distribution controlled by a sigmoidal transformation of z:

$$\log \tilde{P}(y) = yz, \tag{6.20}$$

$$\tilde{P}(y) = \exp(yz), \tag{6.21}$$

$$P(y) = \frac{\exp(yz)}{\sum_{y'=0}^{1} \exp(y'z),} \tag{6.22}$$

$$P(y) = \sigma\left((2y-1)z\right). \tag{6.23}$$

- 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 for Bernoulli output distributions

- Because the cost function used with maximum likelihood is $-\log P(y|\boldsymbol{x})$, the log in the cost function undoes the $\exp$ of the $\mathrm{sigmoid}$. Without this effect, the saturation of the $\mathrm{sigmoid}$ could prevent gradient-based learning from making good progress.

- The loss function for maximum likelihood learning of a Bernoulli parametrized by a $\mathrm{sigmoid}$ is

$$J(\boldsymbol{\theta}) = -\log P(y \mid \boldsymbol{x}) \tag{6.24}$$

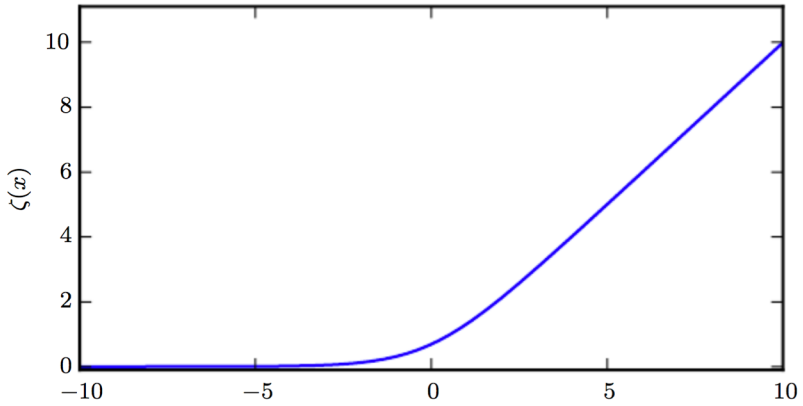
$$= -\log \sigma\left((2y-1)z\right) \tag{6.25}$$

$$= \zeta\left((1-2y)z\right). \tag{6.26}$$

Sigmoid units for Bernoulli output distributions

- The softplus function:

$$\zeta(x) = \log\left(1 + \exp(x)\right). \tag{3.31}$$



- The softplus function can be useful for producing the β or σ parameter of a normal distribution because its range is $(0, \infty)$. It also arises commonly when manipulating expressions involving sigmoids.

- The name of the softplus function comes from the fact that it is a smoothed, or "softened," version of

$$x^+ = \max(0, x). \tag{3.32}$$

Sigmoid units for Bernoulli output distributions

- By rewriting the loss $J(\boldsymbol{\theta})$ in terms of the $\mathrm{softplus}$ function, we can 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 is very positive, or $y=0$ and z is very negative.

- When z has the wrong sign, the argument to the $\mathrm{softplus}$ function, $(1-2y)z$, may be simplified to $|z|$. As $|z|$ becomes large while z has the wrong sign, $\mathrm{softplus}$ asymptotes toward simply returning its argument $|z|$.

- The derivative with respect to z asymptotes to $\mathrm{sign}(z)$, so, in the limit of extremely incorrect z, the $\mathrm{softplus}$ function does not shrink the gradient at all. This property is useful because it means that gradient-based learning can act to quickly correct a mistaken z.

Sigmoid units for Bernoulli output distributions

- When we use other loss functions, such as mean squared error, the loss can saturate anytime $\sigma(z)$ saturates. The $\mathrm{sigmoid}$ activation function saturates to 0 when z becomes very negative and saturates to 1 when z becomes very positive. The gradient can shrink too small to be useful for learning when this happens, whether the model has the correct answer or the incorrect answer.

- For this reason, maximum likelihood is almost always the preferred approach to training $\mathrm{sigmoid}$ output units.

Softmax units for multinoulli 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.

- Softmax functions are most often used as the output of a classifier, to represent the probability distribution over n different classes.

Softmax units for multinoulli output distributions

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

- First, a linear layer predicts unnormalized log probabilities:

$$\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}, \tag{6.28}$$

where $z_i = \log \tilde{P}(y = i|x)$. The $\mathrm{softmax}$ function can then exponentiate and normalize z to obtain the desired $\hat{y}$.

- Formally, the softmax function is given by

$$\mathrm{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}. \tag{6.29}$$

Softmax units for multinoulli output distributions

- Using maximum log-likelihood, we wish to maximize $\log P(y = i; \boldsymbol{z}) = \log \text{softmax}(\boldsymbol{z})_i$. The log in the log-likelihood can undo the exp of the softmax:

$$\log \text{softmax}(\boldsymbol{z})_i = z_i - \log \sum_j \exp(z_j). \tag{6.30}$$

- The first term shows that the input z_i always has a direct contribution to the cost function. Because this term cannot saturate, we know that learning can proceed, even if the contribution of z_i to the second term of becomes very small.

- 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.

Softmax units for multinoulli output distributions

- The second term can be roughly approximated by $\max_j z_j$. This approximation is based on the idea that $\exp(z_k)$ is insignificant for any z_k that is noticeably less than $\max_j z_j$.

- The negative log-likelihood cost function always strongly penalizes the most active incorrect prediction. If the correct answer already has the largest input to the softmax, then the $-z_i$ term and the $\log \sum_j \exp(z_j) \approx \max_j z_j = z_i$ terms will roughly cancel. This example will then contribute little to the overall training cost, which will be dominated by other examples that are not yet correctly classified.

Softmax units for multinoulli output distributions

- Many objective functions other than the log-likelihood do not work as well with the softmax function. Specifically, objective functions that do not use a log to undo the exp of the softmax fail to learn when the argument to the exp becomes very negative, causing the gradient to vanish.

- In particular, squared error is a poor loss function for softmax units and can fail to train the model to change its output, even when the model makes highly confident incorrect predictions.

Softmax units for multinoulli output distributions

- To understand why these other loss functions can fail, we need to examine the softmax function itself.
- The softmax has multiple output values. These output values can saturate when the differences between input values become extreme.
- When the softmax saturates, many cost functions based on the softmax also saturate, unless they are able to invert the saturating activating function.

Softmax units for multinoulli output distributions

- To see the $\mathrm{softmax}$ function responds to the difference between its inputs, observe that the $\mathrm{softmax}$ output is invariant to adding the same scalar to all its inputs:

$$\mathrm{softmax}(\boldsymbol{z}) = \mathrm{softmax}(\boldsymbol{z} + c). \tag{6.32}$$

- A numerically stable variant of the $\mathrm{softmax}$:

$$\mathrm{softmax}(\boldsymbol{z}) = \mathrm{softmax}(\boldsymbol{z} - \max_i z_i). \tag{6.33}$$

- The $\mathrm{softmax}$ function is driven by the amount that its arguments deviate from $\max_i z_i$.

- An output $\mathrm{softmax}(\boldsymbol{z})_i$ saturates to 1 when the corresponding input is maximal ($z_i = \max_i z_i$) and z_i is much greater than all the other inputs. The output $\mathrm{softmax}(\boldsymbol{z})_i$ can also saturate to 0 when z_i is not maximal and the maximum is much greater.

Softmax units for multinoulli output distributions

- The argument $\boldsymbol{z}$ to the $\mathrm{softmax}$ function can be produced in two different ways. The most common is simply to have an earlier layer of the neural network output every element of $\boldsymbol{z}$, as described above using the linear layer $\boldsymbol{z} = \boldsymbol{W}^{\mathrm{T}}\boldsymbol{h} + \boldsymbol{b}$. While straightforward, this approach actually overparametrizes the distribution.

- The constraint that the n outputs must sum to 1 means that only $n - 1$ parameters are necessary; the probability of the n-th value may be obtained by subtracting the first $n - 1$ probabilities from 1.

- In practice, there is rarely much difference (except different learning dynamics) between using the overparametrized version or the restricted version, and it is simpler to implement the overparametrized one.

Softmax units for multinoulli output distributions

- From a neuroscientific point of view, it is interesting to think of the $\mathrm{softmax}$ as a way to create a form of competition between the units that participate in it: the $\mathrm{softmax}$ outputs always sum to 1 so an increase in the value of one unit necessarily corresponds to a decrease in the value of others. This is analogous to the lateral inhibition that is believed to exist between nearby neurons in the cortex.

- At the extreme (when the difference between the maximal a_i and the others is large in magnitude) it becomes a form of winner-take-all (one of the outputs is nearly 1, and the others are nearly 0).

Softmax units for multinoulli output distributions

- The name "softmax" can be somewhat confusing. The function is more closely related to the argmax function than the max function.

- The term "soft" derives from the fact that the softmax function is continuous and differentiable. The argmax function, with its result represented as a one-hot vector, is not continuous or differentiable. The softmax function thus provides a "softened" version of the argmax.

- The corresponding soft version of the maximum function is $\text{softmax}(\boldsymbol{z})^{\mathrm{T}}\boldsymbol{z}$. It would perhaps be better to call the softmax function "softargmax", but the current name is an entrenched convention.

Hidden units

- Now we turn to an issue that is unique to feedforward neural networks: how to choose the type of hidden unit to use in the hidden layers of the model.

- Rectified linear units are an excellent default choice of hidden unit. Many other types of hidden units are available.

- It can be difficult to determine when to use which kind (though rectified linear units are usually an acceptable choice). Predicting in advance which will work best is usually impossible. The design process consists of trial and error, intuiting that a kind of hidden unit may work well, and then training a network with that kind of hidden unit and evaluating its performance on a validation set.

Hidden units

- Some of the hidden units included in this list are not differentiable at all input points. For example, the rectified linear function $g(z) = \max\{0, z\}$ is not differentiable at $z = 0$. This may seem like it invalidates g for use with a gradient-based learning algorithm.

- In practice, gradient descent still performs well enough for these models to be used for machine learning tasks. This is in part because neural network training algorithms do not usually arrive at a local minimum of the cost function, but instead merely reduce its value significantly. Because we do not expect training to actually reach a point where the gradient is 0, it is acceptable for the minima of the cost function to correspond to points with undefined gradient.

Hidden units

- Hidden units that are not differentiable are usually nondifferentiable at only a small number of points.

- In general, a function $g(z)$ has a left derivative defined by the slope of the function immediately to the left of z and a right derivative defined by the slope of the function immediately to the right of z. A function is differentiable at z only if both the left derivative and the right derivative are defined and equal to each other.

- The functions used in the context of neural networks usually have defined left derivatives and defined right derivatives. In the case of $g(z) = \max\{0, z\}$, the left derivative at $z = 0$ is 0, and the right derivative is 1.

Hidden units

- Software implementations of neural network training usually return one of the one-sided derivatives rather than reporting that the derivative is undefined or raising an error.

- This may be heuristically justified by observing that gradient-based optimization on a digital computer is subject to numerical error anyway. When a function is asked to evaluate $g(0)$, it is very unlikely that the underlying value truly was 0. Instead, it was likely to be some small value ϵ that was rounded to 0.

- The important point is that in practice one can safely disregard the nondifferentiability of the hidden unit activation functions.

Hidden units

- Unless indicated otherwise, most hidden units can be described as accepting a vector of inputs $\boldsymbol{x}$, computing an affine transformation $\boldsymbol{z} = \boldsymbol{W}^{\mathrm{T}}\boldsymbol{x} + \boldsymbol{b}$, and then applying an element-wise nonlinear function $g(\boldsymbol{z})$.

- Most hidden units are distinguished from each other only by the choice of the form of the activation function $g(\boldsymbol{z})$.

Rectified linear units and their generalizations

- Rectified linear units use the activation function $g(z) = \max\{0, z\}$.

- These units are easy to optimize because they are so similar to linear units. This makes the derivatives through a rectified linear unit remain large whenever the unit is active.

- The gradients are not only large but also consistent. The second derivative of the rectifying operation is 0 almost everywhere, and the derivative of the rectifying operation is everywhere that the unit is active. The gradient direction is far more useful for learning than activation functions that introduce second-order effects.

Rectified linear units and their generalizations

- Rectified linear units are typically used on top of an affine transformation:

$$\boldsymbol{h} = g(\boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b}). \tag{6.36}$$

- When initializing the parameters of the affine transformation, it can be a good practice to set all elements of $\boldsymbol{b}$ to a small positive value, such as 0.1. Doing so makes it very likely that the rectified linear units will be initially active for most inputs in the training set and allow the derivatives to pass through.

Rectified linear units 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 nonzero 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 fixes $\alpha_i = -1$ to obtain $g(z) = |z|$. It is used for object recognition from images, where it makes sense to seek features that are invariant under a polarity reversal of the input illumination.
  - A leaky ReLU fixes α_i to a small value like 0.01.
  - A parametric ReLU, or PReLU, treats α_i as a learnable parameter.

Rectified linear units and their generalizations

- Maxout units generalize rectified linear units further. Instead of applying an element-wise function $g(z)$, maxout units divide $\boldsymbol{z}$ into groups of k values. Each maxout unit then outputs the maximum element of one of these groups:

$$g(\boldsymbol{z})_i = \max_{j \in \mathbb{G}^{(i)}} z_j, \tag{6.37}$$

  where $\mathbb{G}^{(i)}$ is the set of indices into the inputs for group i, $\{(i-1)k+1, \ldots, ik\}$.

- This provides a way of learning a piecewise linear function that responds to multiple directions in the input $\boldsymbol{x}$ space.

Rectified linear units and their generalizations

- A maxout unit can learn a piecewise linear, convex function with up to k pieces. Maxout units can thus be seen as learning the activation function itself rather than just the relationship between units.

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

- In particular, a maxout layer with two pieces can learn to implement the same function of the input $\boldsymbol{x}$ as a traditional layer using the rectified linear activation function, the absolute value rectification function, or the leaky or parametric ReLU, or it can learn to implement a totally different function altogether.

Rectified linear units and their generalizations

- The $\max\text{out}$ layer will be parametrized differently from any of these other layer types, so the learning dynamics will be different even in the cases where $\max\text{out}$ learns to implement the same function of x as one of the other layer types.

- Each $\max\text{out}$ unit is now parametrized by k weight vectors instead of just one, so $\max\text{out}$ units typically need more regularization than rectified linear units.

Logistic sigmoid and hyperbolic tangent

- Prior to the introduction of rectified linear units, most neural networks used the logistic sigmoid function

$$g(z) = \sigma(z) \tag{6.38}$$

or the hyperbolic tangent activation function

$$g(z) = \tanh(z). \tag{6.39}$$

These activation functions are closely related because $\tanh(z) = 2\sigma(2z) - 1$.

- Unlike piecewise linear units, sigmoidal units saturate across most of their domain—they saturate to a high value when z is very positive, saturate to a low value when z is very negative, and are only strongly sensitive to their input when z is near 0.

Logistic sigmoid and hyperbolic tangent

- The widespread saturation of sigmoidal units can make gradient-based learning very difficult. For this reason, their use as hidden units in feedforward networks is now discouraged.

- Their use as output units is compatible with the use of gradient-based learning when an appropriate cost function can undo the saturation of the sigmoid in the output layer.

Logistic sigmoid and hyperbolic tangent

- When a sigmoidal activation function must be used, the hyperbolic tangent activation function typically performs better than the logistic $\mathrm{sigmoid}$. It resembles the identity function more closely, in the sense that $\tanh(0) = 0$ while $\sigma(0) = \frac{1}{2}$.

- Because $\tanh$ is similar to the identity function near 0, training a deep neural network $\hat{y} = \boldsymbol{\omega}^{\mathrm{T}} \tanh(\boldsymbol{U}^{\mathrm{T}} \tanh(\boldsymbol{V}^{\mathrm{T}}\boldsymbol{x}))$ resembles training a linear model $\hat{y} = \boldsymbol{\omega}^{\mathrm{T}}\boldsymbol{U}^{\mathrm{T}}\boldsymbol{V}^{\mathrm{T}}\boldsymbol{x}$ as long as the activations of the network can be kept small. This makes training the $\tanh$ network easier.

Logistic sigmoid and hyperbolic tangent

- Sigmoidal activation functions are more common in settings other than feed-forward networks.

- Recurrent networks, many probabilistic models, and some autoencoders have additional requirements that rule out the use of piecewise linear activation functions and make sigmoidal units more appealing despite the drawbacks of saturation.

Other hidden units

- Softmax units are sometimes used as a hidden unit. Softmax units naturally represent a probability distribution over a discrete variable with k possible values, so they may be used as a kind of switch.

- These kinds of hidden units are usually only used in more advanced architectures that explicitly learn to manipulate memory.

Other hidden units

- A few other reasonably common hidden unit types include
  - Radial basis function (RBF), unit: $h_i = \exp\left(-\frac{1}{\sigma_i^2}\left\|\boldsymbol{W}_{:,i} - \boldsymbol{x}\right\|^2\right)$. This function becomes more active as $\boldsymbol{x}$ approaches a template $\boldsymbol{W}_{:,i}$. Because it saturates to 0 for most $\boldsymbol{x}$, it can be difficult to optimize.
  - Hard $\tanh$. This is shaped similarly to the $\tanh$ and the rectifier, but unlike the latter, it is bounded, $g(a) = \max(-1, \min(1, a))$.

Other hidden units

- Softplus: $g(a) = \zeta(a) = \log(1 + e^a)$. This is a smooth version of the rectifier, introduced for function approximation and for the conditional distributions of undirected probabilistic models.

- The use of the $\mathrm{softplus}$ is generally discouraged. The $\mathrm{softplus}$ demonstrates that the performance of hidden unit types can be very counterintuitive—one might expect it to have an advantage over the rectifier due to being differentiable everywhere or due to saturating less completely, but empirically it does not.

Other hidden units

- Many other types of hidden units are possible but are used less frequently. In general, a wide variety of differentiable functions perform perfectly well. Many unpublished activation functions perform just as well as the popular ones.

- For example, a feedforward network using $\boldsymbol{h} = \cos(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$ on the MNIST dataset obtained an error rate of less than 1 percent, which is competitive with results obtained using more conventional functions.

Other hidden units

- During research and development of new techniques, it is common to test many different activation functions and find that several variations on standard practice perform comparably.

- This means that usually new hidden unit types are published only if they are clearly demonstrated to provide a significant improvement. New hidden unit types that perform roughly comparably to known types are so common as to be uninteresting.

Architecture design

- A key design consideration for neural networks is determining the architecture, 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. Most neural network architectures arrange these layers in a chain structure, with each layer being a function of the layer that preceded it. In this structure, the first layer is given by

$$\boldsymbol{h}^{(1)} = g^{(1)}\left(\boldsymbol{W}^{(1)\top}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right); \tag{6.40}$$

the second layer is given by

$$\boldsymbol{h}^{(2)} = g^{(2)}\left(\boldsymbol{W}^{(2)\top}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}\right); \tag{6.41}$$

and so on.

Architecture design

- In these chain-based architectures, the main architectural considerations are choosing the depth of the network and the width of each layer.

- A network with even one hidden layer is sufficient to fit the training set. Deeper networks are often able to use far fewer units per layer and far fewer parameters, as well as frequently generalizing to the test set, but they also tend to be harder to optimize.

- The ideal network architecture for a task must be found via experimentation guided by monitoring the validation set error.

Universal approximation properties and depth

- The universal approximation theorem 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 sigmoid activation function) can approximate any Borel measurable function from one finite-dimensional space to another with any desired nonzero 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.

Universal approximation properties and depth

- For our purposes it suffices to say that any continuous function on a closed and bounded subset of $\mathbb{R}^n$ is Borel measurable and therefore may be approximated by a neural network. A neural network may also approximate any function mapping from any finite dimensional discrete space to another.

- While the original theorems were first stated in terms of units with activation functions that saturate for both very negative and very positive arguments, universal approximation theorems have also been proved for a wider class of activation functions, which includes the now commonly used rectified linear unit.

Universal approximation properties and depth

- 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.

- We are not guaranteed, however, that the training algorithm will be able to learn that function. Learning can fail for two different reasons. First, the optimization algorithm used for training may not be able to find the value of the parameters that corresponds to the desired function. Second, the training algorithm might choose the wrong function as a result of overfitting.

Universal approximation properties and depth

- The universal approximation theorem does not say how large this network will be. In the worst case, an exponential number of hidden units (possibly with one hidden unit corresponding to each input configuration that needs to be distinguished) may be required.

- In summary, 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. Using deeper models can reduce the number of units required to represent the desired function.

Universal approximation properties and depth

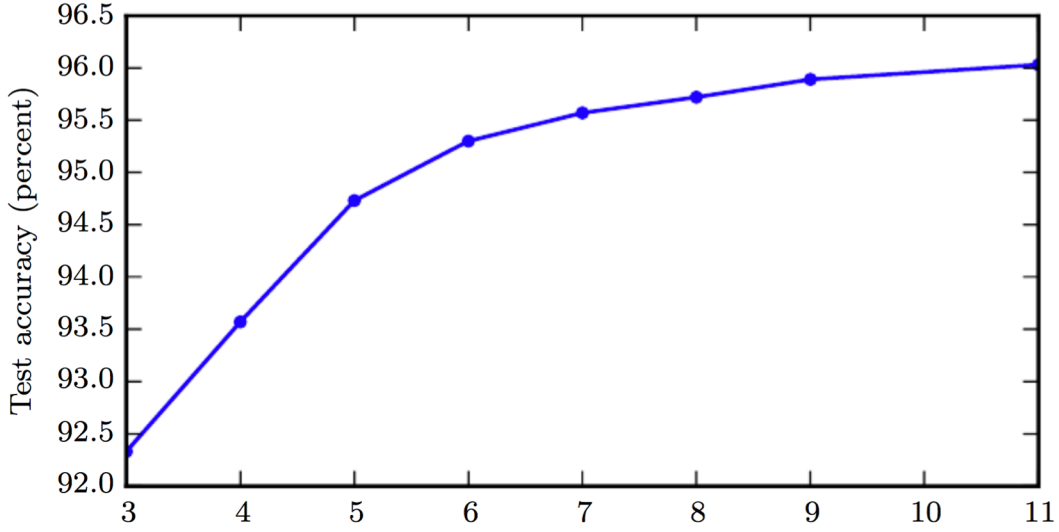
- Any time we choose a specific machine learning algorithm, we are implicitly stating some set of prior beliefs we have about what kind of function the algorithm should learn.

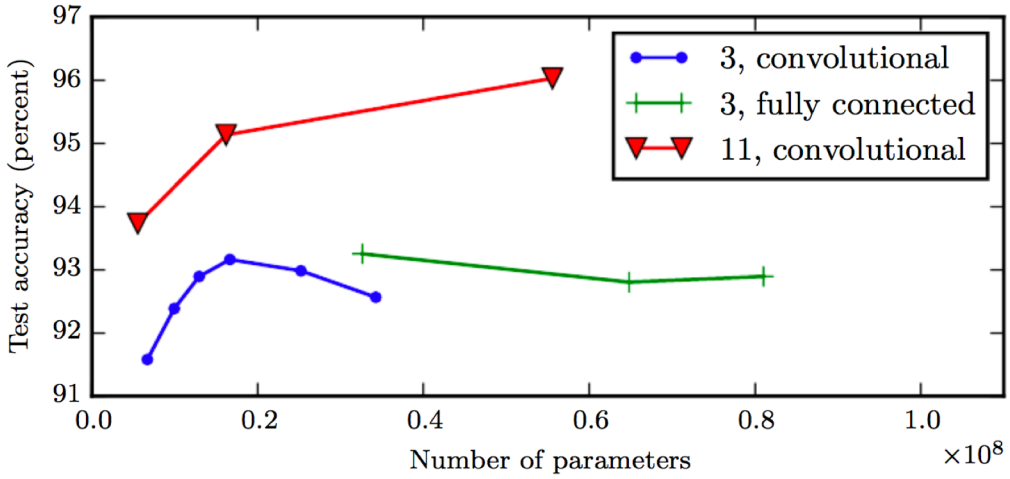
- Choosing a deep model encodes a very general belief that the function we want to learn should involve composition of several simpler functions.

- Alternately, we can interpret the use of a deep architecture as expressing a belief that the function we want to learn is a computer program consisting of multiple steps. These intermediate outputs are not necessarily factors of variation but can instead be analogous to counters or pointers that the network uses to organize its internal processing.

Universal approximation properties and depth

- Empirically, greater depth does seem to result in better generalization for a wide variety of tasks.





Other architectural considerations

- In general, the layers need not be connected in a chain, even though this is the most common practice.

- Many architectures build a main chain but then add extra architectural features to it, such as skip connections going from layer i to layer $i + 2$ or higher. These skip connections make it easier for the gradient to flow from output layers to layers nearer the input.

Other architectural considerations

- Another key consideration of architecture design is exactly how to connect a pair of layers to each other.
- In the default neural network layer described by a linear transformation via a matrix $\boldsymbol{W}$, every input unit is connected to every output unit (i.e., fully connected).
- Many specialized networks have fewer connections, so that each unit in the input layer is connected to only a small subset of units in the output layer.
- These strategies for decreasing the number of connections reduce the number of parameters and the amount of computation required to evaluate the network but are often highly problem dependent.