CAP 5625: Computational Foundations for Artificial Intelligence

Neural networks

Linear classifier limitations

We have spent several lectures discussing a variety of linear classifiers including LDA, logistic regression, and SVMs as well linear predictors through (penalized) least squares regression.

However, in particular for classification problems, we have seen that each of these linear classifiers can have poor accuracy if decision boundaries are (highly) non-linear.

Example solutions have been quadratic decision boundaries through QDA, expanding input feature dimension through basis expansions for logistic regression, or through kernels in SVMs.

However, all the non-linear examples we encountered so far have been fairly complex.

A simple example motivating non-linear classifiers

Consider the example $X^T = [X_1, X_2]$ in which observations contain $p = 2$ binary features $X_1 \in \{0,1\}$ and $X_2 \in \{0,1\}$, where the goal is to decide whether the two features have the same value or not.

That is, $X_1 \neq \mathrm{X}_2$ is one class, and $X_1 = X_2$ is another class.

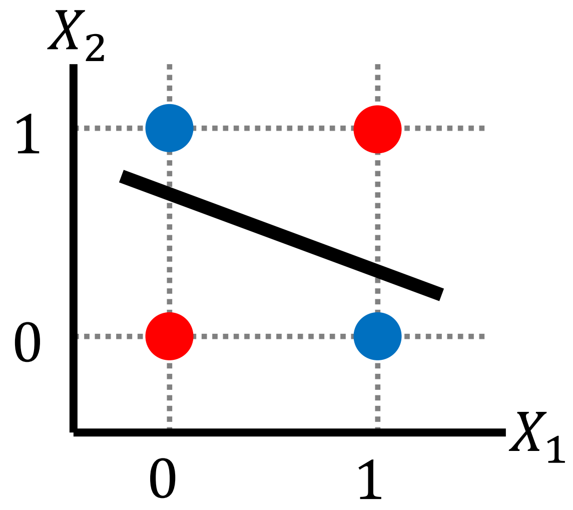
This is an example of the exclusive or (xor) function, where

$$\mathrm{xor}(X_1, X_2) = \begin{cases} 1 & \text{if } X_1 \neq X_2 \\ 0 & \text{if } X_1 = X_2 \end{cases}$$

This may seem simple enough, as the input can take on only four possible combinations of values $[0,0]$, $[0,1]$, $[1,0]$, and $[1,1]$.

The two classes cannot be separated by a plane

$$\mathrm{xor}(X_1, X_2) = \begin{cases} 1 & \text{if } X_1 \neq X_2 \\ 0 & \text{if } X_1 = X_2 \end{cases}$$

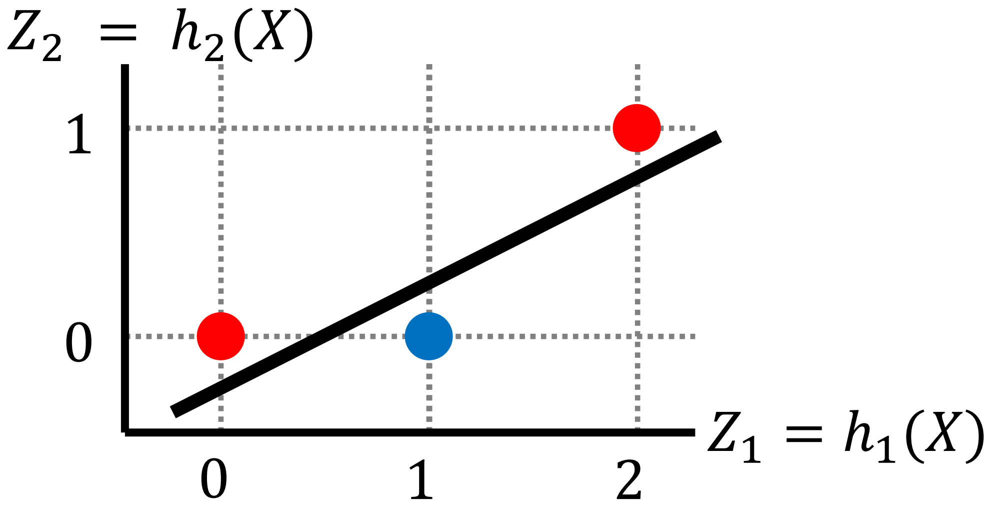


In this feature space, it is impossible to find a hyperplane that would separate the positive (blue) and negative (red) classes.

We can solve this by transforming the input space.

Transforming the input space

$$\text{xor}(X_1, X_2) = \begin{cases} 1 & \text{if } X_1 \neq X_2 \\ 0 & \text{if } X_1 = X_2 \end{cases}$$

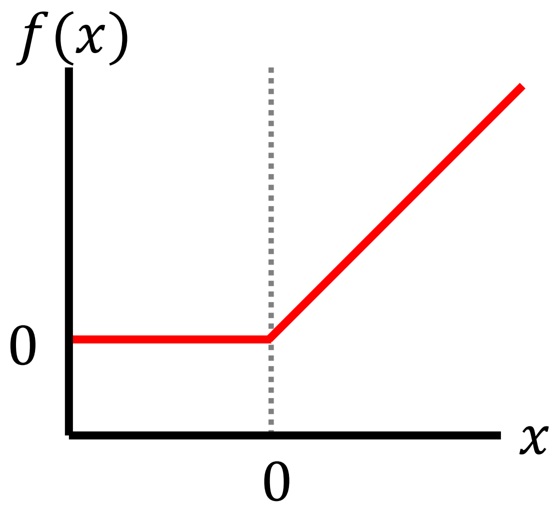


Here, both positive examples $[0,1]$ and $[1,0]$ are mapped to the same transformed feature $Z = [1,0]$.

Here we discuss a technique that naturally solves such problems.

How do we perform this non-linear transformation?

Define the function $f(x) = \max\{0, x\}$, which is non-linear.



We have seen a similar function when discussing coordinate descent for elastic net regression, where this function was the called the positive part of input x and we wrote it as x_+.

Defining the transformation

We define the transformation $(m = 1 \text{ or } 2)$

$$h_m(x) = f(\alpha_{m0} + x^T\alpha_m)$$

where $\alpha_m^T = [1,1]$ and $\alpha_{10} = 0$ and $\alpha_{20} = -1$.

The transformation is then

$$\begin{aligned} h_m(X) &= f\left(\alpha_{m0} + \begin{bmatrix} X_1 & X_2 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \end{bmatrix}\right) \\ &= \max\{0, \alpha_{m0} + X_1 + X_2\} \\ &= \begin{cases} X_1 + X_2 & \text{if } m = 1 \\ \max\{0, X_1 + X_2 - 1\} & \text{if } m = 2 \end{cases} \end{aligned}$$

Computing the transformation

$$h_m(X) = \begin{cases} X_1 + X_2 & \text{if } m = 1 \\ \max\{0, X_1 + X_2 - 1\} & \text{if } m = 2 \end{cases}$$

The transformed variables are

$$Z_1 = h_1(X) = \begin{cases} 0 & \text{if } X^T = [0,0] \\ 1 & \text{if } X^T = [0,1] \\ 1 & \text{if } X^T = [1,0] \\ 2 & \text{if } X^T = [1,1] \end{cases} \qquad Z_2 = h_2(X) = \begin{cases} 0 & \text{if } X^T = [0,0] \\ 0 & \text{if } X^T = [0,1] \\ 0 & \text{if } X^T = [1,0] \\ 1 & \text{if } X^T = [1,1] \end{cases}$$

Using $\beta^T = [1, -2]$ and $\beta_0 = 0$, we classify Z as a positive (different values) if $g(Z) = 1$ and as a negative (same values) if $g(Z) = 0$ using the linear function

$$g(Z) = \beta_0 + Z^T\beta = Z_1 - 2Z_2$$

Performing the classification

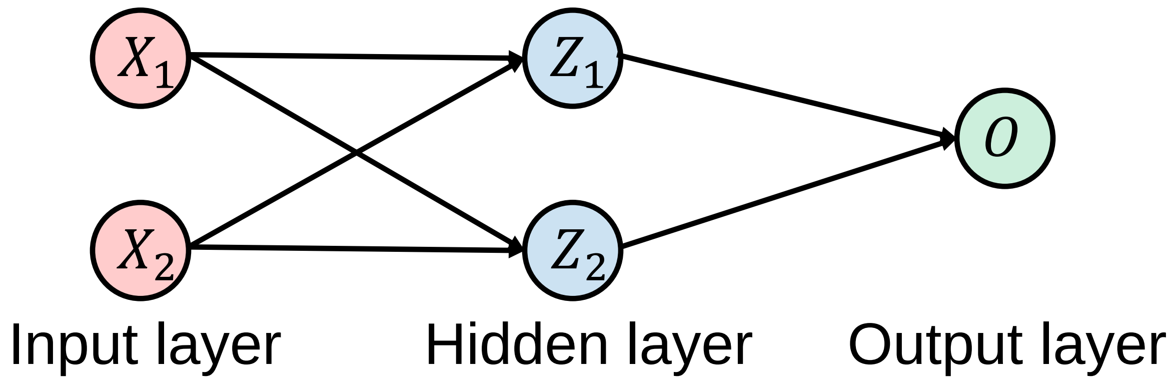
$$Z_1 = h_1(X) = \begin{cases} 0 & \text{if } X^T = [0,0] \\ 1 & \text{if } X^T = [0,1] \\ 1 & \text{if } X^T = [1,0] \\ 2 & \text{if } X^T = [1,1] \end{cases} \qquad Z_2 = h_2(X) = \begin{cases} 0 & \text{if } X^T = [0,0] \\ 0 & \text{if } X^T = [0,1] \\ 0 & \text{if } X^T = [1,0] \\ 1 & \text{if } X^T = [1,1] \end{cases}$$

We therefore have the output label O

$$O = g(Z) = Z_1 - 2Z_2 = \begin{cases} 0 & \text{if } X^T = [0,0] \\ 1 & \text{if } X^T = [0,1] \\ 1 & \text{if } X^T = [1,0] \\ 0 & \text{if } X^T = [1,1] \end{cases}$$

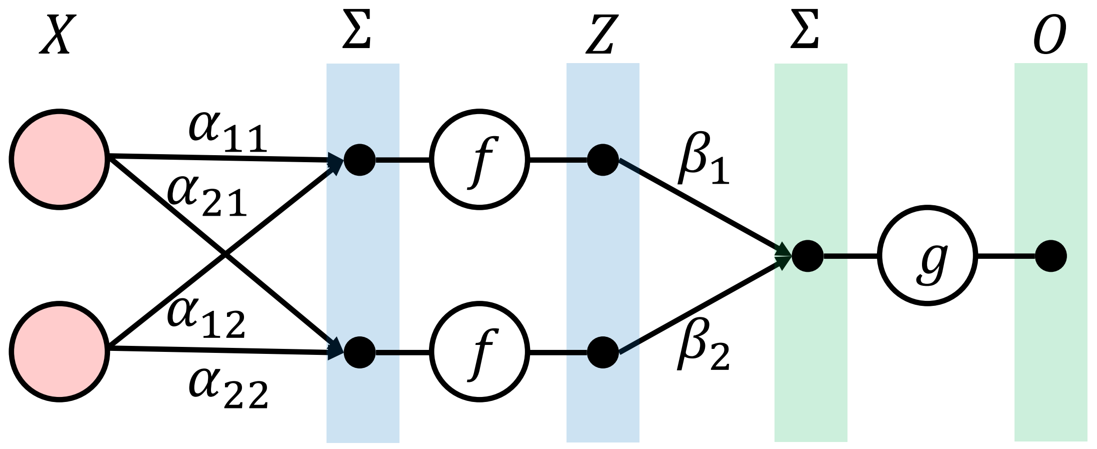
Thereby providing a linear classifier based on a linear function in the transformed space $[h_1(X), h_2(X)]$, and so we just need an approach to learn the transformations (*i.e.*, the α parameters) and to learn the linear model (*i.e.*, β parameters).

This can be solved via neural networks



This network has an input layer, a hidden layer of transformed variables, and an output layer.

We can expand a bit to see what layers are doing



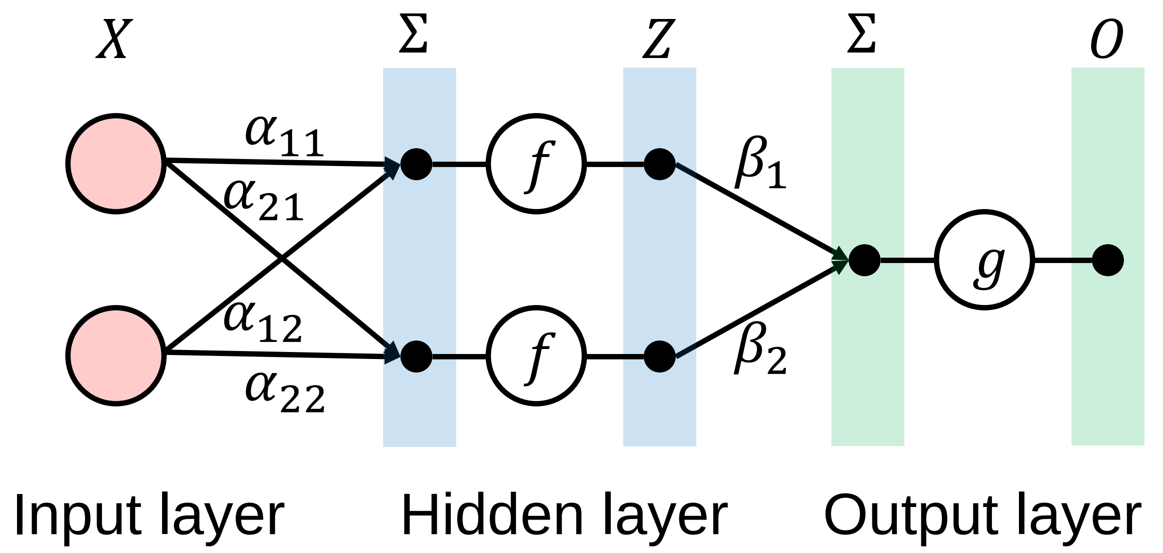
Input layer Hidden layer Output layer

Taking input $X^T = [X_1, X_2]$, we non-linearly transform to new input $Z^T = [Z_1, Z_2]$ by first creating a linear combination of the inputs and then applying a non-linear function f to this linear combination as

$$Z_1 = h_1(X) = f(\alpha_{10} + \alpha_{11}X_1 + \alpha_{12}X_2) = \max\{0, \alpha_{10} + \alpha_{11}X_1 + \alpha_{12}X_2\}$$

$$Z_2 = h_2(X) = f(\alpha_{20} + \alpha_{21}X_1 + \alpha_{22}X_2) = \max\{0, \alpha_{20} + \alpha_{21}X_1 + \alpha_{22}X_2\}$$

We can expand a bit to see what layers are doing



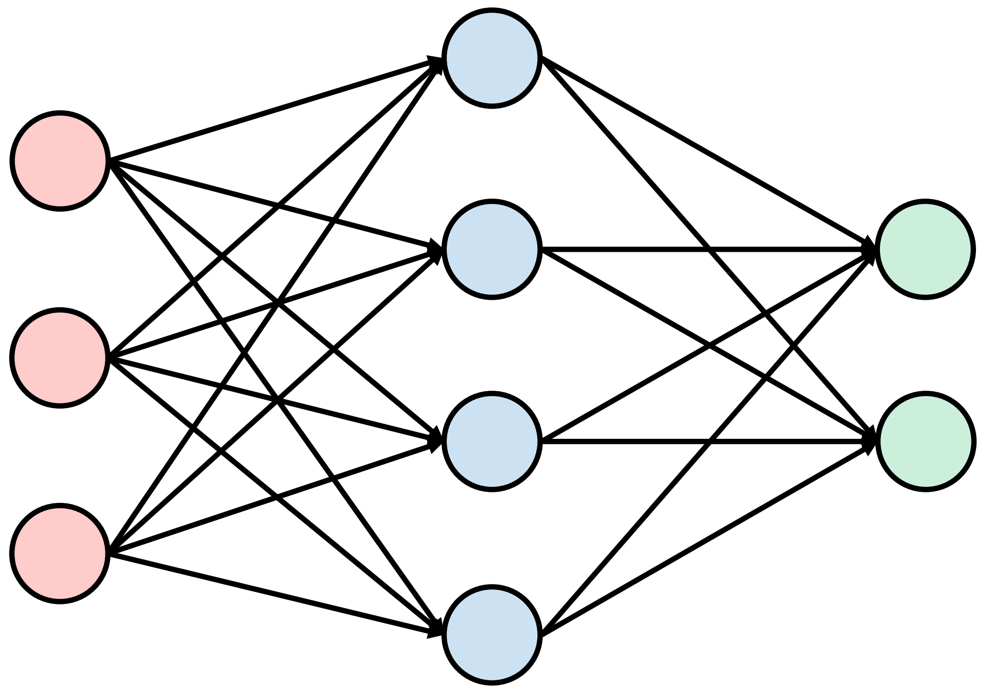
Taking transformed inputs $Z^T = [Z_1, Z_2]$, we then classify each input as a positive or a negative based on the linear function

$$O = g(Z) = \beta_0 + \beta_1 Z_1 + \beta_2 Z_2$$

We would just need to learn the 6 α parameters leading to the hidden layer and the 3 β parameters leading to the output layer.

Architecture of a feedforward neural network

Network with one layer of four hidden units and one layer of two output units.

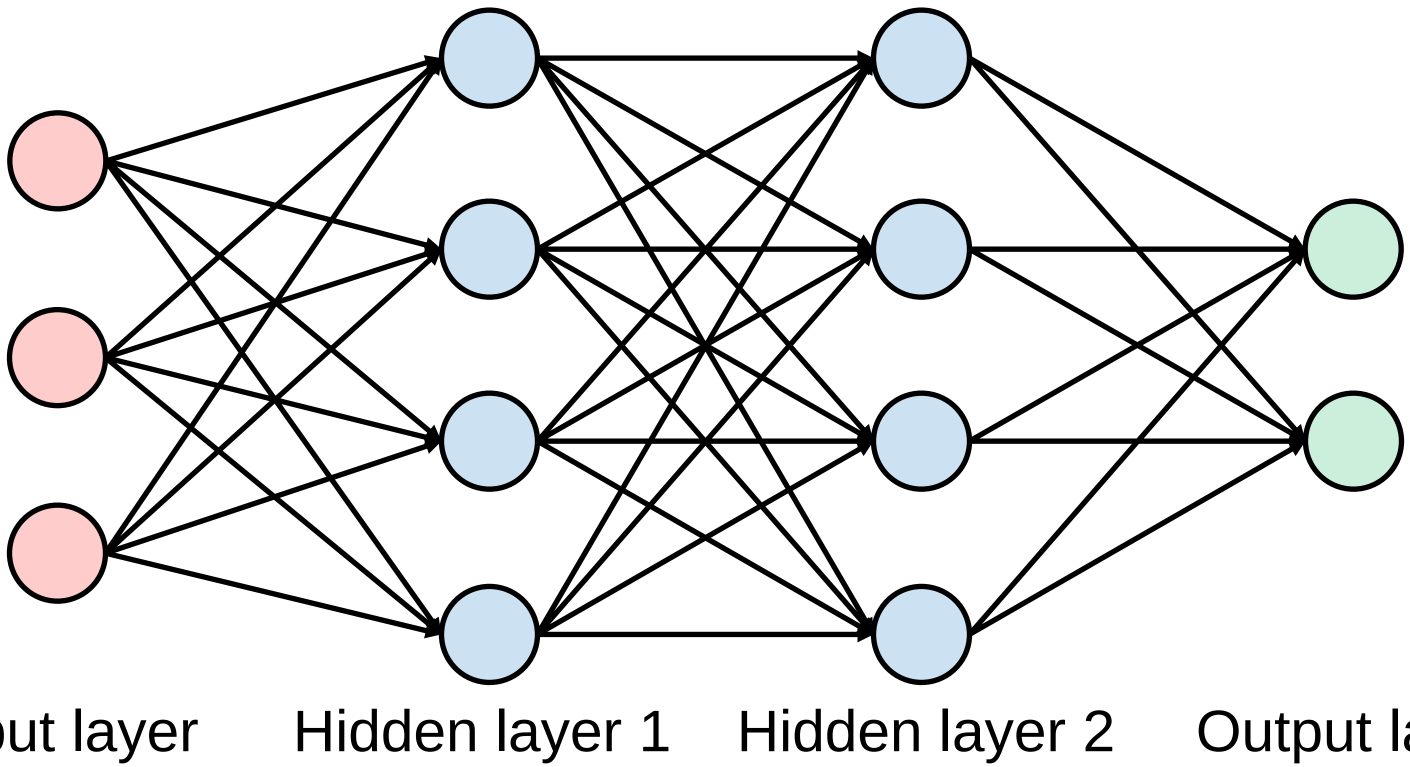


Input layer   Hidden layer   Output layer

A neural network with 1 layer of hidden units and 1 output layer is termed a 2-layer neural network.

Architecture of a feedforward neural network

Network with two layers of four hidden units each and one layer of two output units.

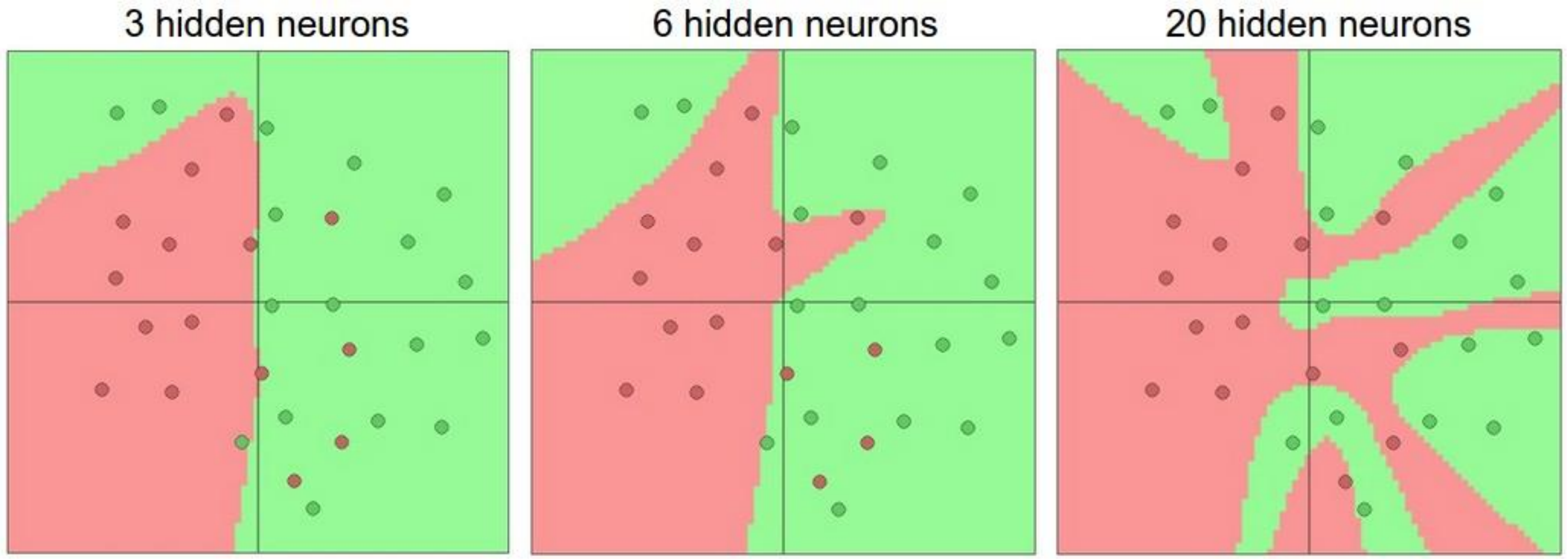


A neural network with $D - 1$ layers of hidden units and 1 output layer is termed a D-layer neural network.

Effect of the number of hidden units

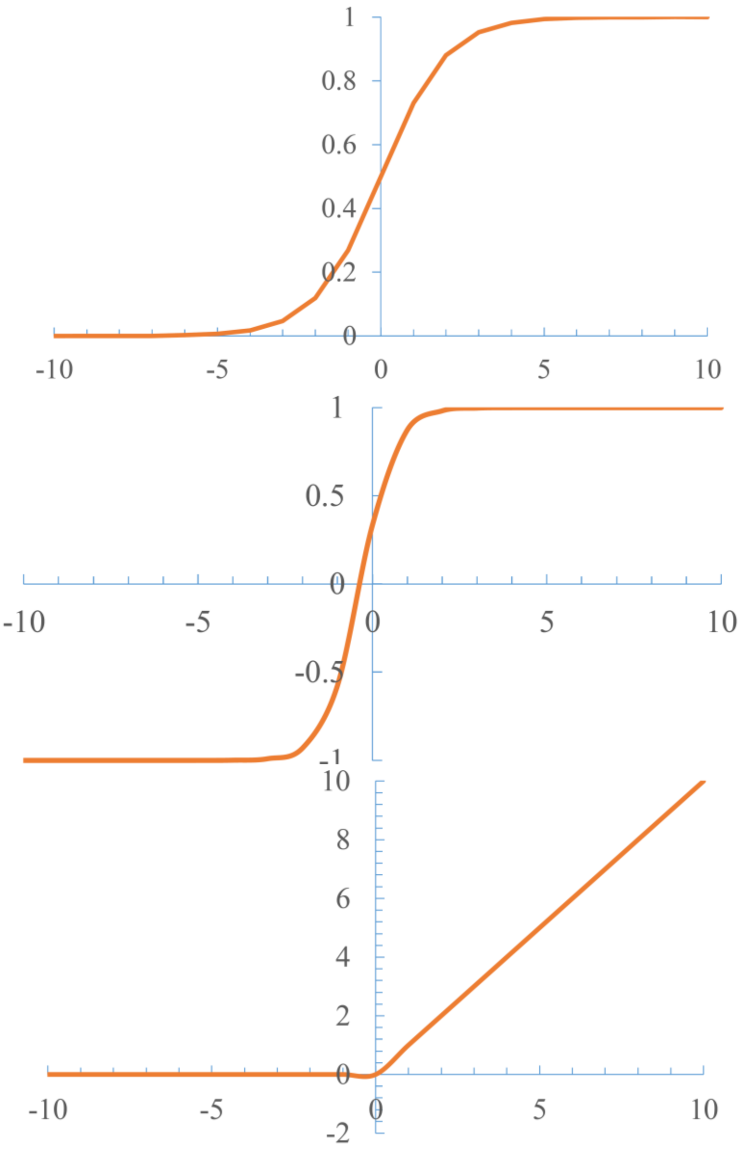
Cybenko (1989) proved that a 2 layer (1 hidden layer) neural network can approximate any function.

Therefore neural networks can be flexible enough to model any function underlying the training data.



Cybenko (1989) *Math. Control Signals Systems* 2:303-314.
http://cs231n.github.io/neural-networks-1/

Typical non-linear transformations used for f and g



Sigmoid (logistic) function

$$\sigma(x) = \frac{\exp(x)}{1 + \exp(x)}$$

Hyperbolic tangent

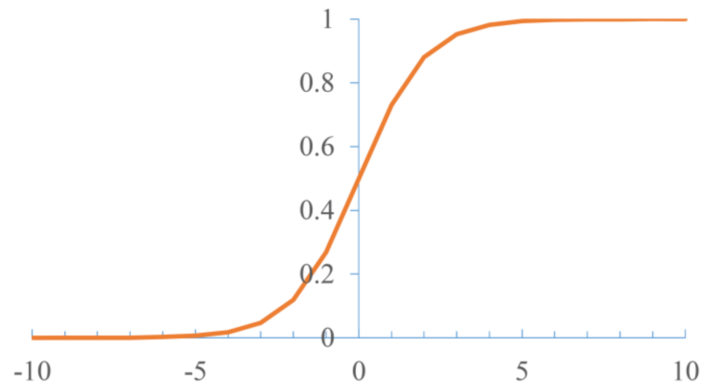
$$\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)} = 2\sigma(2x) - 1$$

Rectified Linear Unit (ReLU)

$$\mathrm{ReLU}(x) = \max\{0, x\}$$

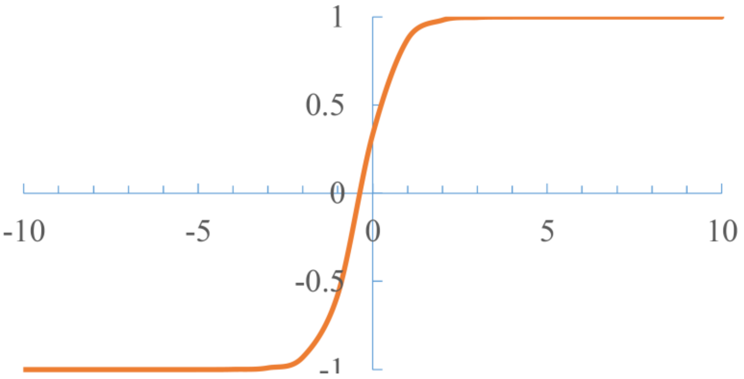
Adapted from Yang, Yang (2018) *Algorithms* 11:28.

Some thoughts about these functions



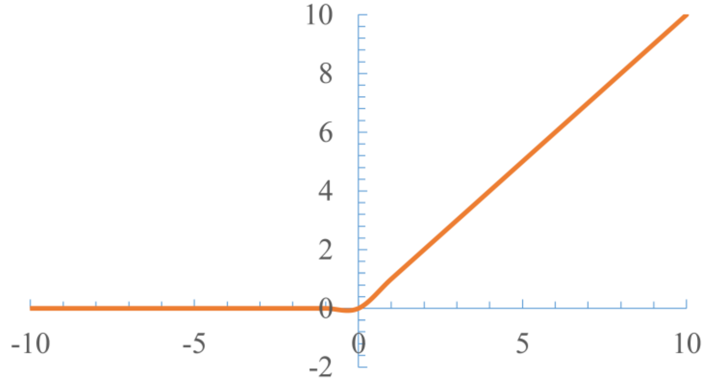
Values of $\sigma(x)$ saturate for large magnitude x, and can decrease learning speed.

Values passed are always positive.



Values of $\tanh(x)$ saturate for large magnitude x.

Values passed can be positive or negative, and are zero-centered



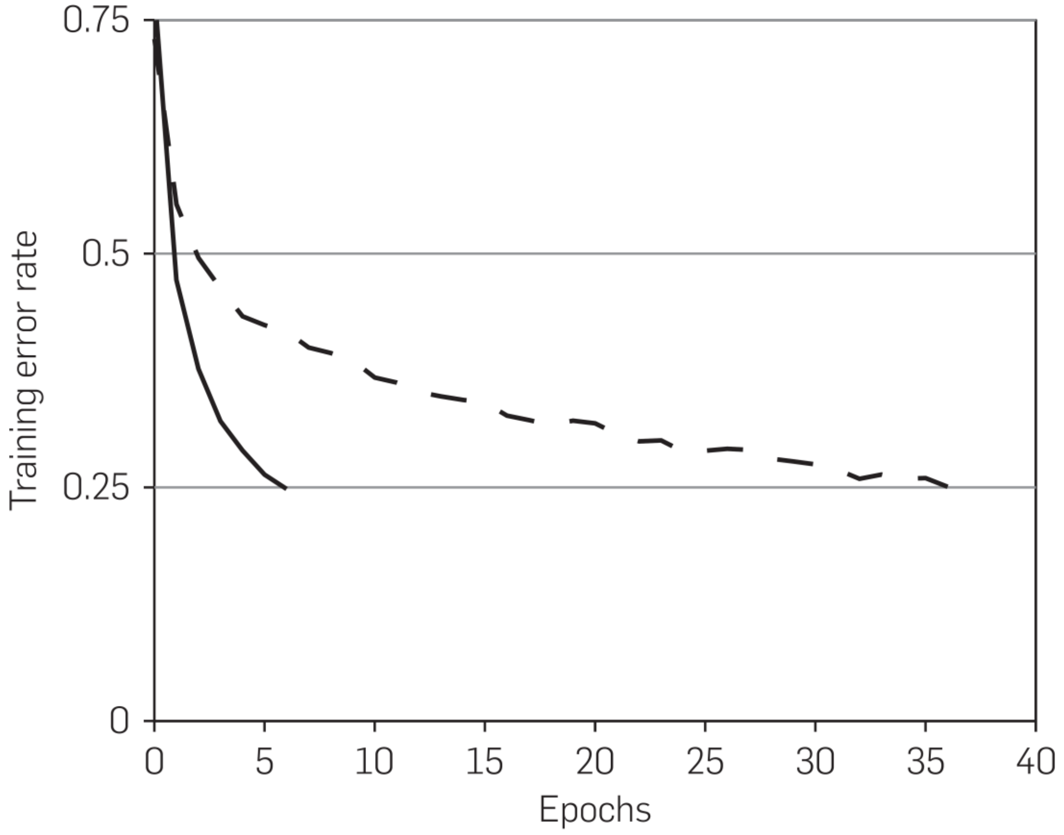
Computation of $\mathrm{ReLU}(x)$ is less expensive (simple threshold rather than computation of exponentials).

Can speedup learning relative to other two.

Adapted from Yang, Yang (2018) *Algorithms* 11:28.

Training speedup of $\mathrm{ReLU}(x)$ relative to $\tanh(x)$

Using $\mathrm{ReLU}(x)$ (solid curve) relative to $\tanh(x)$ (dashed curve) can provide a substantial speedup in model training, here showing that the number of iterations (epochs) for stochastic gradient descent is 6 times smaller for $\mathrm{ReLU}(x)$ compared to $\tanh(x)$ to achieve a training error of 0.25.



Krizhevsky *et al.* (2017) *Communications of the ACM* 60:84-90..

Predicting outputs and learning parameters

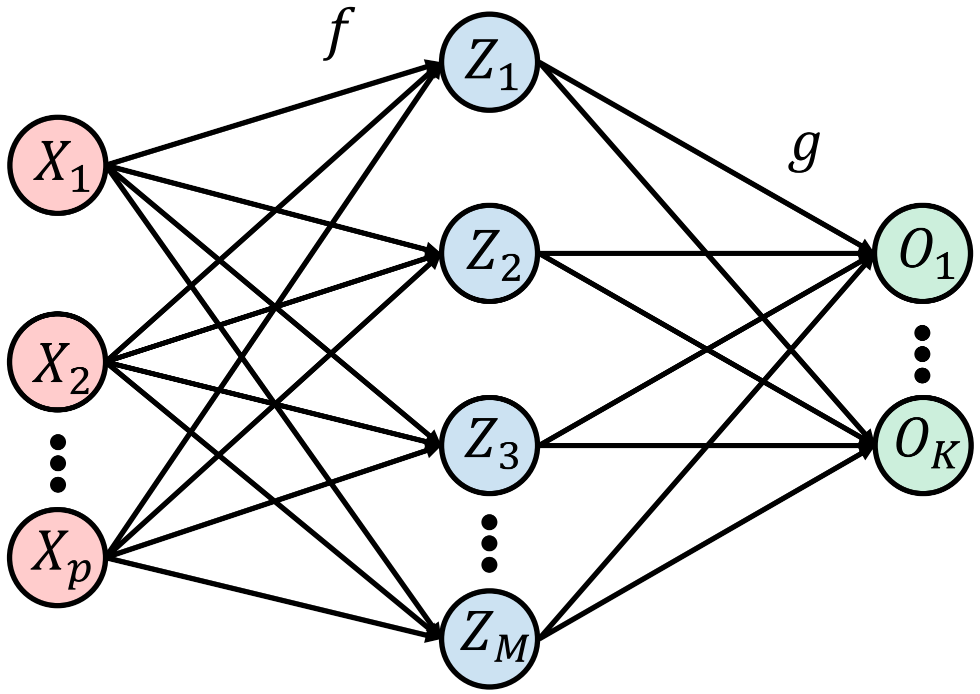
We can train and make predictions from neural networks using two algorithms: **forward** and **backward propagation**.

With forward propagation we can compute the output from a set of inputs.

With backward propagation, we can evaluate how changes in parameters changes the objective (cost) function.

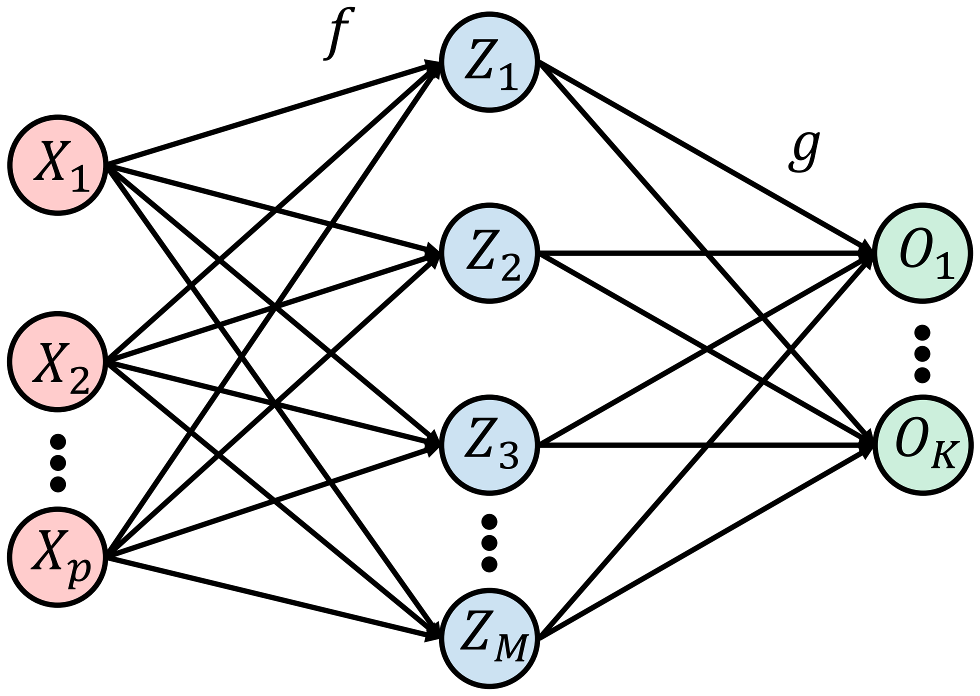
Using both forward and backgward propagation, we can learn the model parameters.

Computing the output from forward propagation



Consider the above network that takes in input vector $X \in \mathbb{R}^p$, transforms these inputs into feature vector $Z \in \mathbb{R}^M$, and then outputs the vector $O \in \mathbb{R}^K$.

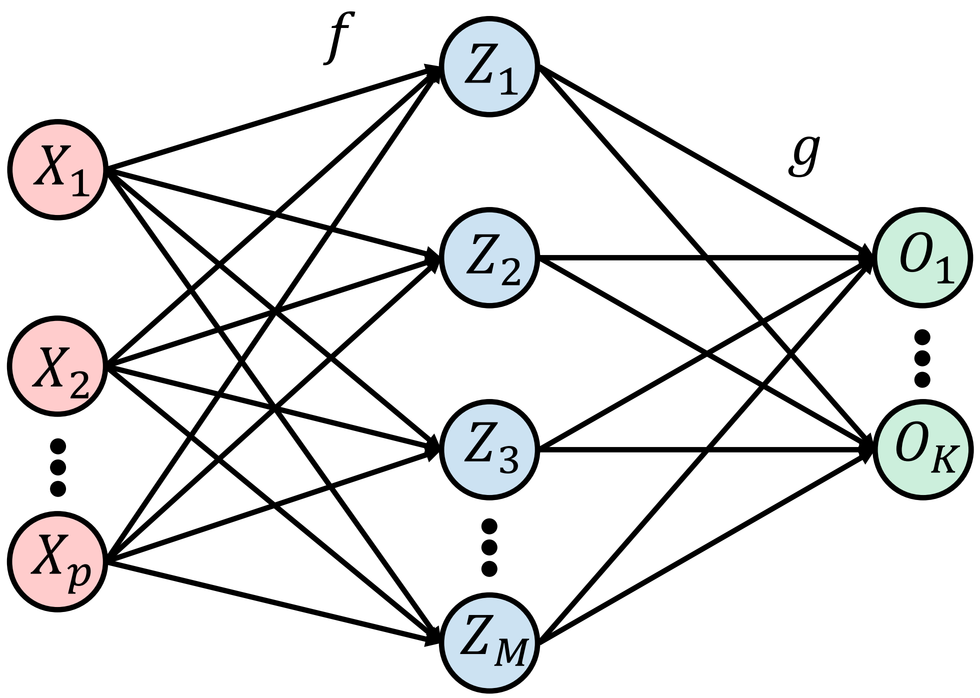
Computing the output from forward propagation



For $m = 1,2,\ldots,M$, we have

$$Z_m = h_m(X) = f\left(\alpha_{m0} + \sum_{j=1}^{p} \alpha_{mj} X_j\right)$$

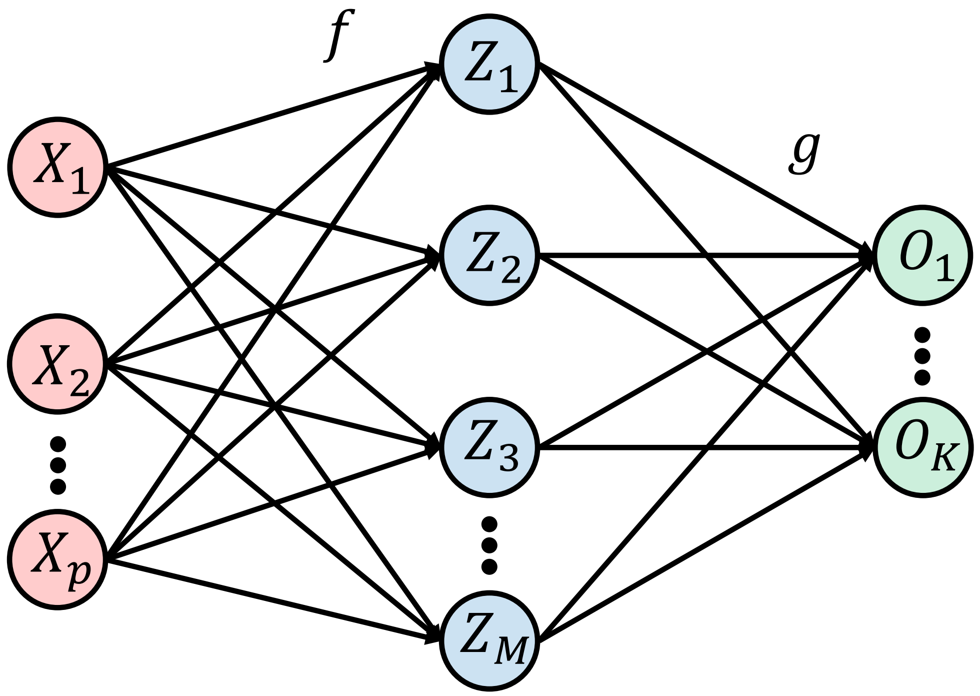
Computing the output from forward propagation



Given transformed input $Z^T = [Z_1, Z_2, \dots, Z_M]$, for $k = 1,2,\dots,K$, we have the output

$$O_k = g\left(\beta_{k0} + \sum_{m=1}^{M} \beta_{km} h_m(X)\right) = g\left(\beta_{k0} + \sum_{m=1}^{M} \beta_{km} Z_m\right)$$

Computing the output from forward propagation



The choices of f and g are often the non-linear functions

$$\sigma(x) = \frac{\exp(x)}{1 + \exp(x)}$$

$$\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)} = 2\sigma(2x) - 1$$

$$\mathrm{ReLU}(x) = \max\{0, x\}$$

How do we learn the model parameters?

So far we have discussed forward propagation for making predictions of outputs $O \in \mathbb{R}^K$ from an input set of features $X \in \mathbb{R}^p$.

But how do we learn the underlying model parameters to be able to make this predictions?

That is, given a set of observations (x_i, y_i), $i = 1,2, \dots, N$, where $x_i \in \mathbb{R}^p$ and $y_i \in \mathbb{R}^K$, how do we train a neural network model for making quantitative predictions (regression) or for performing classification?

To train a neural network we need to define a cost function $J(\theta)$ of the set of model parameters θ, and then identify the set of parameters θ that minimize a cost function $J(\theta)$.

Defining the cost function

For regression type problems with K-dimensional response, we might choose the squared loss cost function (*e.g.*, least squares regression)

$$J(\theta) = \sum_{i=1}^{N} \sum_{k=1}^{K} (y_{ik} - o_{ik})^2$$

where y_{ik} is the value of the kth response for observation i and o_{ik} is the value of the kth predicted response for observation i.

For classification problems with K classes, we might use the categorical cross-entropy cost function (*e.g.*, logistic regression)

$$J(\theta) = -\sum_{i=1}^{N} \sum_{k=1}^{K} y_{ik} \log o_{ik}$$

where $y_{ik} = 1$ if observation i is from class k, and $y_{ik} = 0$ otherwise, and where o_{ik} is the probability that observation i is from class k.

Training the neural network

Once we have defined the cost function, then we can perform gradient descent with updates of the form

$$\theta := \theta - \alpha \frac{\partial}{\partial \theta} J(\theta)$$

where α is the learning rate and $\frac{\partial}{\partial \theta} J(\theta)$ is the gradient vector of the cost function.

Some useful derivatives are

$$\frac{d}{dx}\sigma(x) = \sigma(x)[1 - \sigma(x)]$$

$$\frac{d}{dx}\tanh(x) = \frac{1}{\cosh^2 x}$$

$$\frac{d}{dx}\mathrm{ReLU}(x) = I(x > 0) = \begin{cases} 0 & \text{if } x \le 0 \\ 1 & \text{if } x > 0 \end{cases}$$

Training the network with backward propagation

Backward propagation is an approach for computing gradients across the parameters in a neural-network with multiple layers in an efficient manner, in order to be able to apply gradient descent.

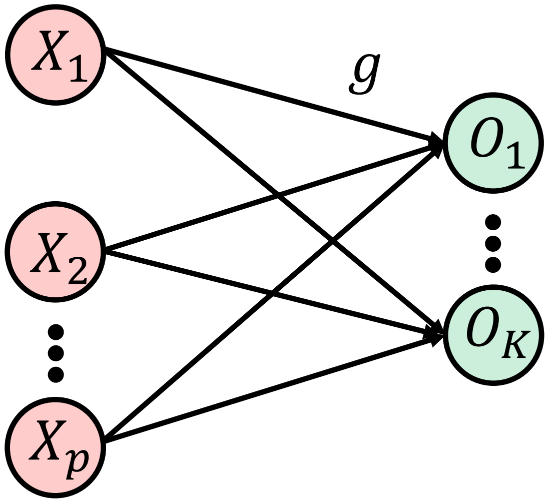
One hurdle is that we do not have observed values for the hidden units, and so we cannot compare the predicted values at hidden units to observed.

Instead, we can compute the rate at which the cost (error) function changes at these units using derivatives.

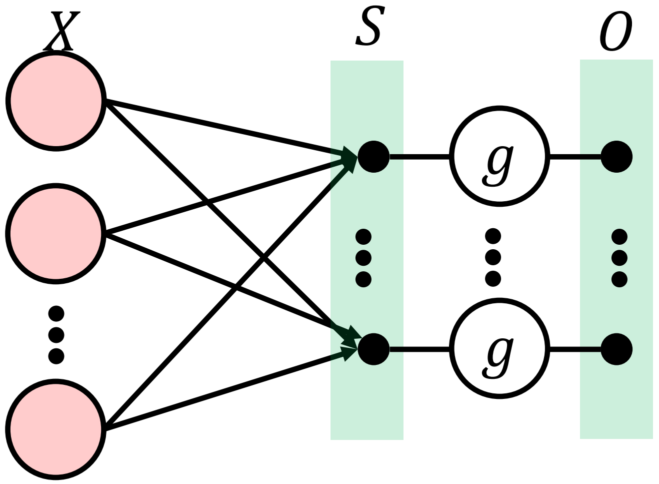
If we have the cost function derivatives for hidden unit in a layer, then it is easy to obtain the cost function derivatives of the parameters leading to these hidden units.

Example backward propagation in a single-layer network

Consider the single-layer network below

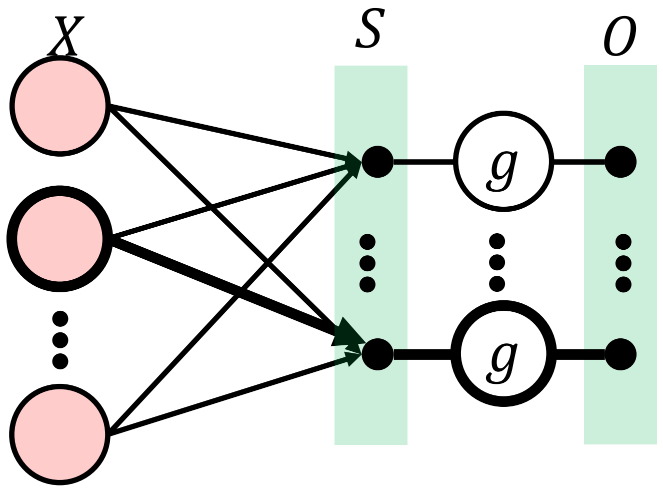


As in earlier slides, we will expand the output state of this network



Example backward propagation in a single-layer network

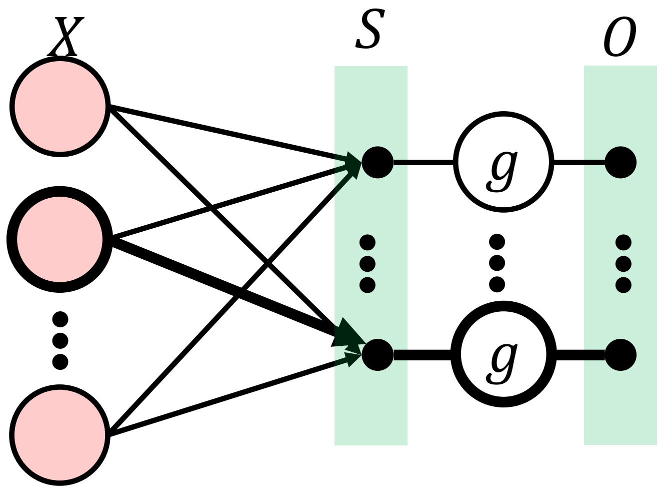
Assume $X_0 = 1$ is a new input unit, such that there are $p + 1$ units.



Consider input unit $j \in \{0,1, \ldots, p\}$ and output unit $k \in \{1,2, \ldots, K\}$.

We wish to know the rate of change of the cost function with respect to changes in the parameter β_{kj} for input j leading to output k.

Example backward propagation in a single-layer network



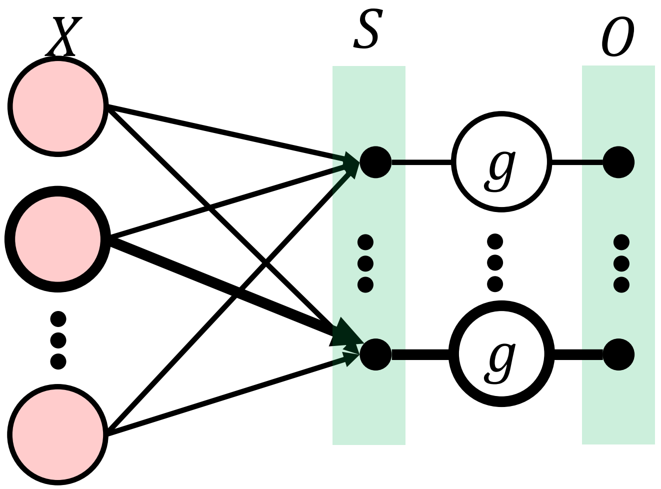
Assume that

$$J(\theta) = \sum_{i=1}^{N} \sum_{k=1}^{K} (y_{ik} - o_{ik})^2 = \sum_{i=1}^{N} J_i(\theta)$$

$$s_{ik} = \beta_{k0} + \sum_{j=1}^{p} \beta_{kj} x_{ij}$$

$$o_{ik} = g(s_k) = \sigma(s_{ik})$$

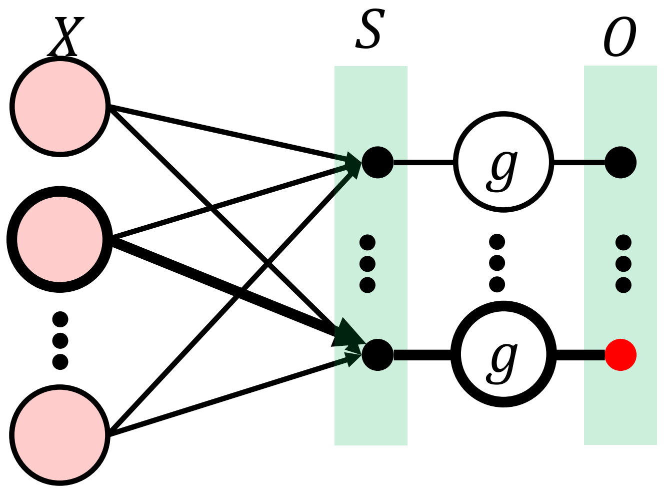
Example backward propagation in a single-layer network



Applying the chain rule we have

$$\frac{\partial J}{\partial \beta_{kj}} = \sum_{i=1}^{N} \frac{\partial J_i}{\partial o_{ik}} \frac{\partial o_{ik}}{\partial s_{ik}} \frac{\partial s_{ik}}{\partial \beta_{kj}}$$

Example backward propagation in a single-layer network



Because

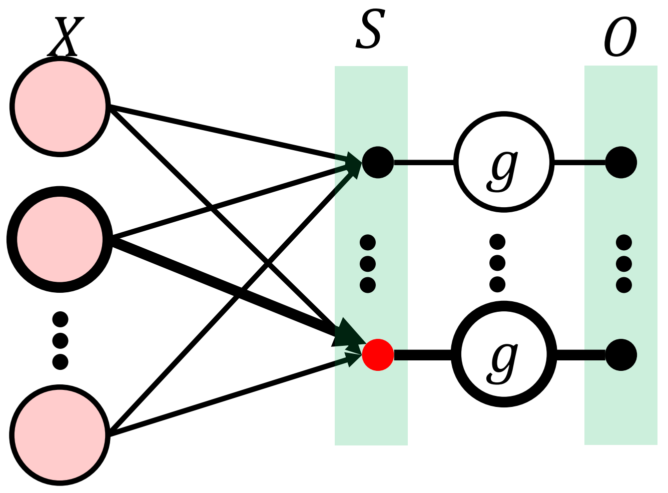
$$\frac{\partial}{\partial o_{ik}} J_i(\theta) = \frac{\partial}{\partial o_{ik}} \sum_{k=1}^{K} (y_{ik} - o_{ik})^2 = -2(y_{ik} - o_{ik})$$

we have

$$\frac{\partial J}{\partial \beta_{kj}} = \sum_{i=1}^{N} \frac{\partial J_i}{\partial o_{ik}} \frac{\partial o_{ik}}{\partial s_{ik}} \frac{\partial s_{ik}}{\partial \beta_{kj}}$$

$$= -2 \sum_{i=1}^{N} (y_{ik} - o_{ik}) \frac{\partial o_{ik}}{\partial s_{ik}} \frac{\partial s_{ik}}{\partial \beta_{kj}}$$

Example backward propagation in a single-layer network



Because

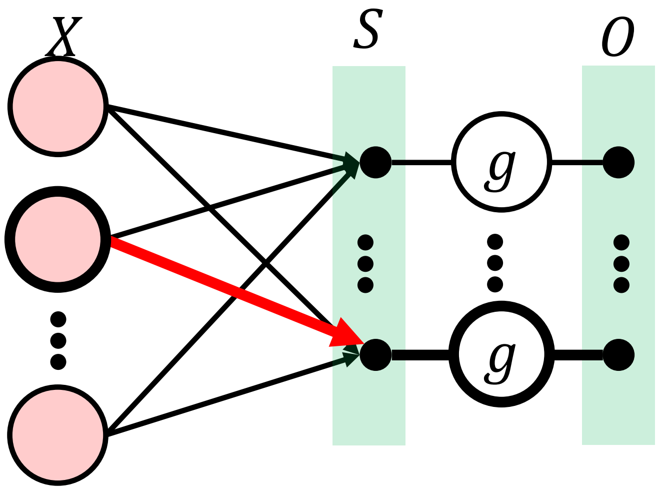
$$\frac{\partial}{\partial s_{ik}} o_{ik} = \frac{\partial}{\partial s_{ik}} \sigma(s_{ik}) = \sigma(s_{ik})[1 - \sigma(s_{ik})] = o_{ik}(1 - o_{ik})$$

we have

$$\frac{\partial J}{\partial \beta_{kj}} = \sum_{i=1}^{N} \frac{\partial J_i}{\partial o_{ik}} \frac{\partial o_{ik}}{\partial s_{ik}} \frac{\partial s_{ik}}{\partial \beta_{kj}}$$

$$= -2 \sum_{i=1}^{N} (y_{ik} - o_{ik}) o_{ik}(1 - o_{ik}) \frac{\partial s_{ik}}{\partial \beta_{kj}}$$

Example backward propagation in a single-layer network



Because

$$\frac{\partial}{\partial \beta_{kj}} s_{ik} = \frac{\partial}{\partial \beta_{kj}} \left(\beta_{k0} + \sum_{j=1}^{p} \beta_{kj} x_{ij} \right) = x_{ij}$$

we have

$$\frac{\partial J}{\partial \beta_{kj}} = \sum_{i=1}^{N} \frac{\partial J_i}{\partial o_{ik}} \frac{\partial o_{ik}}{\partial s_{ik}} \frac{\partial s_{ik}}{\partial \beta_{kj}}$$

$$= -2 \sum_{i=1}^{N} (y_{ik} - o_{ik}) o_{ik} (1 - o_{ik}) x_{ij}$$

Finding θ with batch gradient descent

Fix learning rate α

1. Randomly initialize $\theta = \{\beta_1, \beta_2, \ldots, \beta_K\}$ where $\beta_k \in \mathbb{R}^{p+1}$.
2. For each observation $i \in \{1,2,\ldots,N\}$ propagate forward to obtain s_{ik} and o_{ik} for $k \in \{1,2,\ldots,K\}$.
3. For each $j \in \{0,1,\ldots,p\}$ and $k \in \{1,2,\ldots,K\}$ propagate backward to compute

$$\frac{\partial}{\partial \beta_{kj}} J(\theta) = -2 \sum_{i=1}^{N} (y_{ik} - o_{ik}) o_{ik} (1 - o_{ik}) x_{ij}$$

4. For $j \in \{0,1,\ldots,p\}$ and $k \in \{1,2,\ldots,K\}$ update parameters

$$\beta_{kj} := \beta_{kj} - \alpha \frac{\partial}{\partial \beta_{kj}} J(\theta)$$

5. Repeat steps 2 through 4 until convergence.

Finding θ with stochastic gradient descent

Fix learning rate α

1. Randomly initialize $\theta = \{\beta_1, \beta_2, \ldots, \beta_K\}$ where $\beta_k \in \mathbb{R}^{p+1}$.
2. Randomly permute (shuffle) the order of the N observations and update their indices.
3. For observation i, propagate forward to obtain s_{ik} and o_{ik} for $k \in \{1,2,\ldots,K\}$, propagate backward to compute

$$\frac{\partial}{\partial \beta_{kj}} J_i(\theta) = -2(y_{ik} - o_{ik})o_{ik}(1 - o_{ik})x_{ij}$$

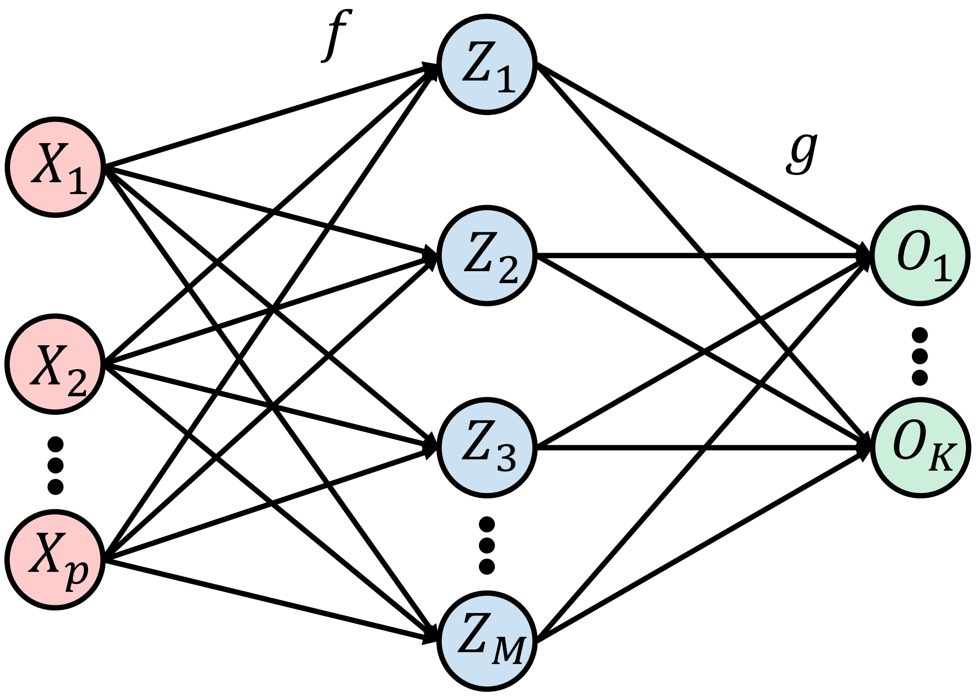
   for $j \in \{0,1,\ldots,p\}$, and then update the update parameter

$$\beta_{kj} := \beta_{kj} - \alpha \frac{\partial}{\partial \beta_{kj}} J_i(\theta)$$

4. Repeat step 3 for each observation $i \in \{1,2,\ldots,N\}$ in shuffled list.
5. Repeat steps 2 through 4 until convergence.

Example backward propagation in a multi-layer network

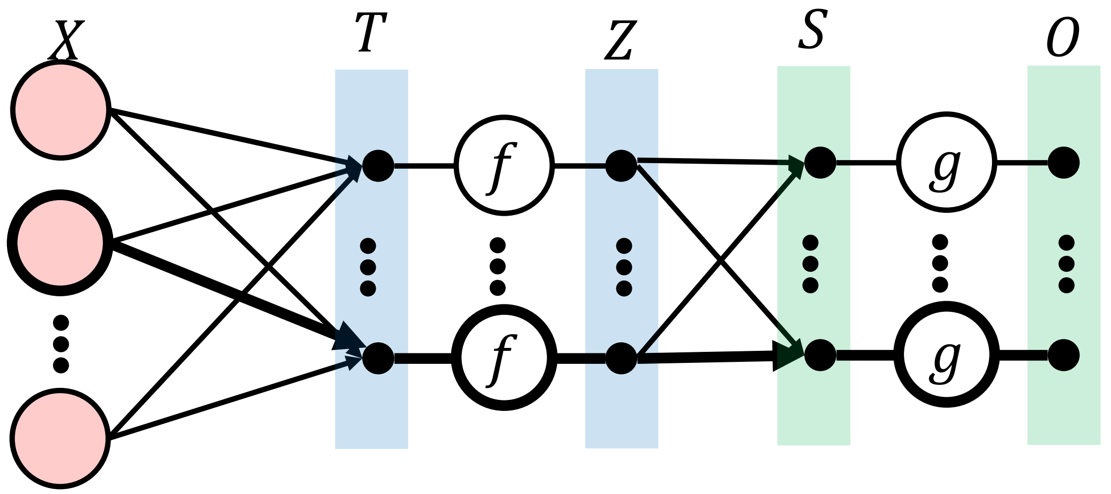
Consider the 2-layer network below



To begin discussing backpropagation, we need to expand the hidden and output layers of this network.

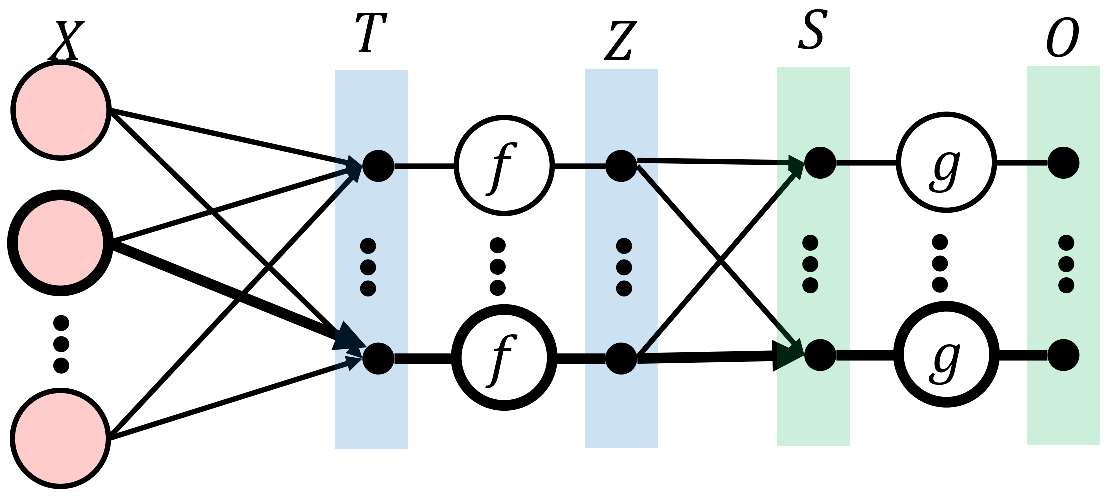
Example backward propagation in a multi-layer network

Expanded hidden and output states.



Example backward propagation in a multi-layer network

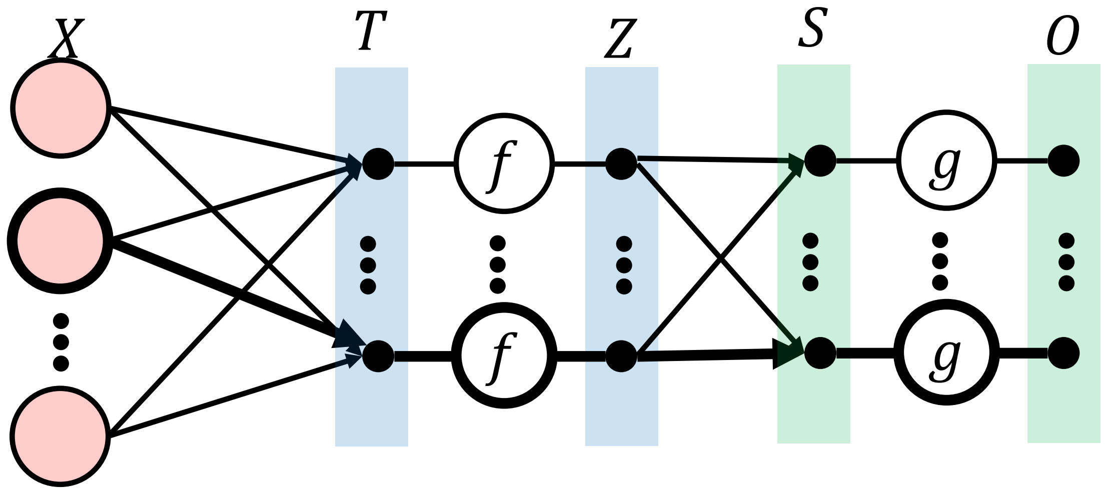
Assume $X_0 = 1$ is a new input unit, such that there are $p + 1$ units in the input layer and $Z_0 = 1$ is a new hidden unit, such that there are $M + 1$ units in the hidden layer.



Consider hidden unit $m \in \{0,1,\ldots,M\}$ of the hidden layer (layer 1) and output unit $k \in \{1,2,\ldots,K\}$ of the output layer (layer 2).

We wish to know the rate of change of the cost function with respect to changes in the parameter β_{km} for hidden unit m of layer 1 leading to output unit k of layer 2.

Example backward propagation in a multi-layer network



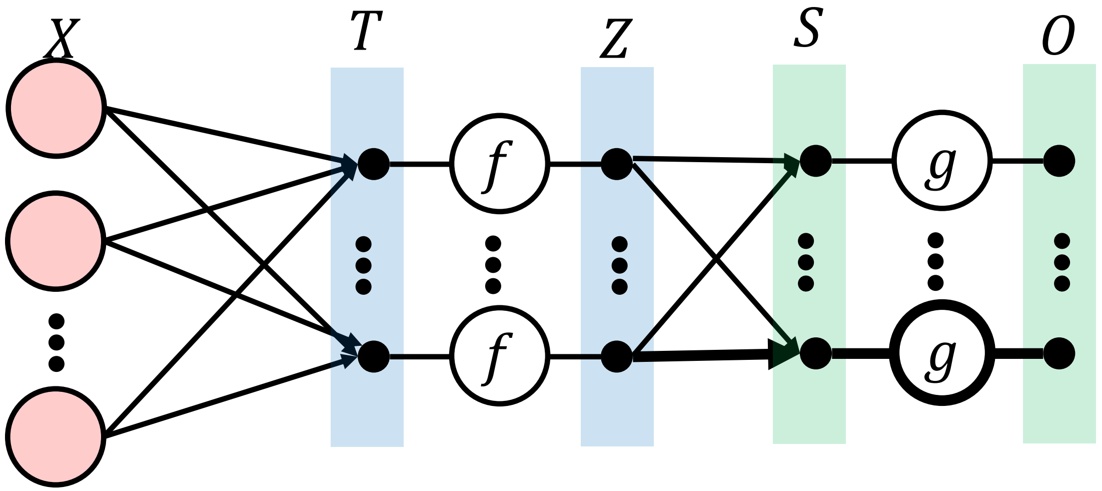
Assume that

$$J(\theta) = \sum_{i=1}^{N} \sum_{k=1}^{K} (y_{ik} - o_{ik})^2 = \sum_{i=1}^{N} J_i(\theta) = \sum_{i=1}^{N} \sum_{k=1}^{K} J_{ik}(\theta)$$

$$t_{im} = \alpha_{m0} + \sum_{j=1}^{p} \alpha_{mj} x_{ij} \qquad s_{ik} = \beta_{k0} + \sum_{m=1}^{M} \beta_{km} z_{im}$$

$$z_{im} = f(t_{im}) = \mathrm{ReLU}(t_{im}) \qquad o_{ik} = g(s_{ik}) = \sigma(s_{ik})$$

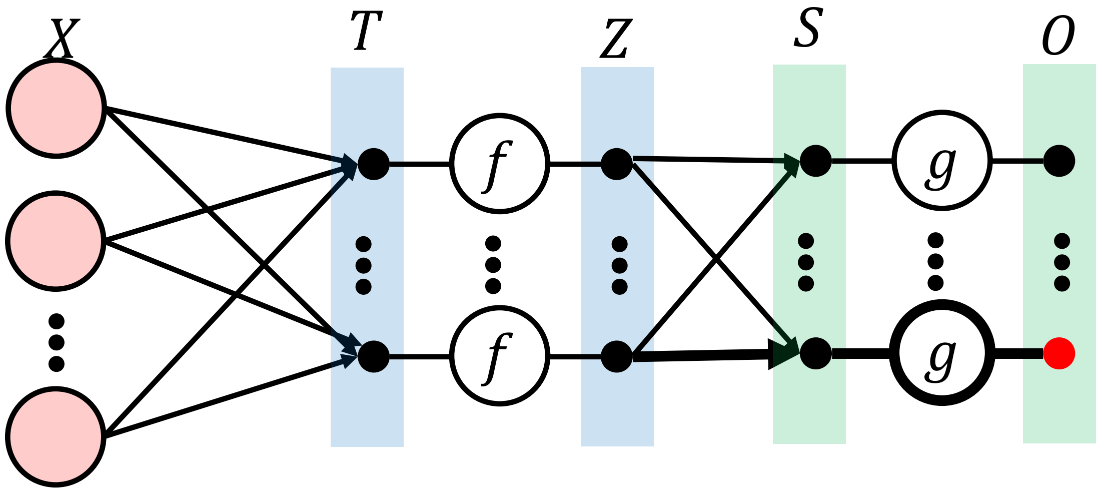
Example backward propagation in a multi-layer network



Applying the chain rule we have

$$\frac{\partial J}{\partial \beta_{km}} = \sum_{i=1}^{N} \frac{\partial J_i}{\partial o_{ik}} \frac{\partial o_{ik}}{\partial s_{ik}} \frac{\partial s_{ik}}{\partial \beta_{km}}$$

Example backward propagation in a multi-layer network



Because

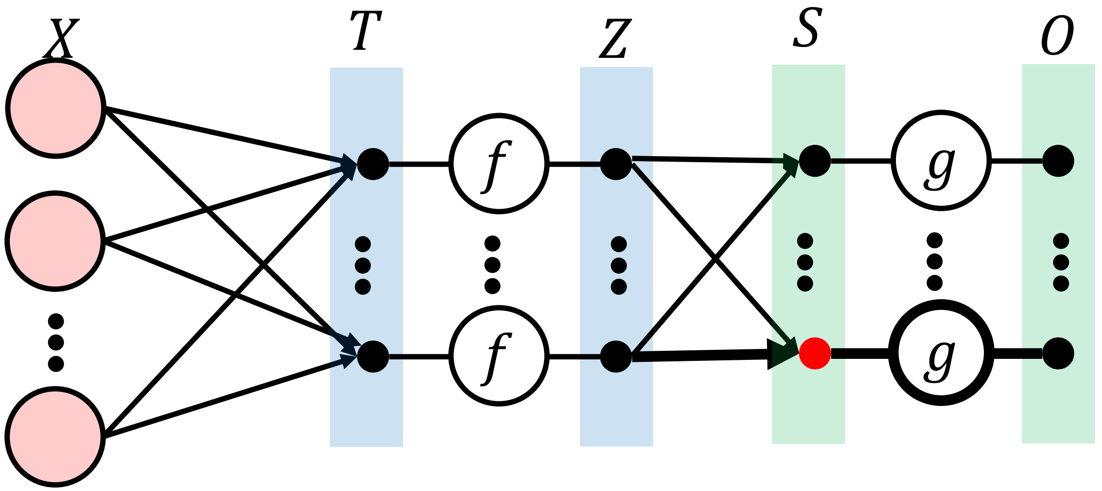
$$\frac{\partial}{\partial o_{ik}} J_i(\theta) = \frac{\partial}{\partial o_{ik}} \sum_{k=1}^{K} (y_{ik} - o_{ik})^2 = -2(y_{ik} - o_{ik})$$

we have

$$\frac{\partial J}{\partial \beta_{km}} = \sum_{i=1}^{N} \frac{\partial J_i}{\partial o_{ik}} \frac{\partial o_{ik}}{\partial s_{ik}} \frac{\partial s_{ik}}{\partial \beta_{km}}$$

$$= -2 \sum_{i=1}^{N} (y_{ik} - o_{ik}) \frac{\partial o_{ik}}{\partial s_{ik}} \frac{\partial s_{ik}}{\partial \beta_{km}}$$

Example backward propagation in a multi-layer network



Because

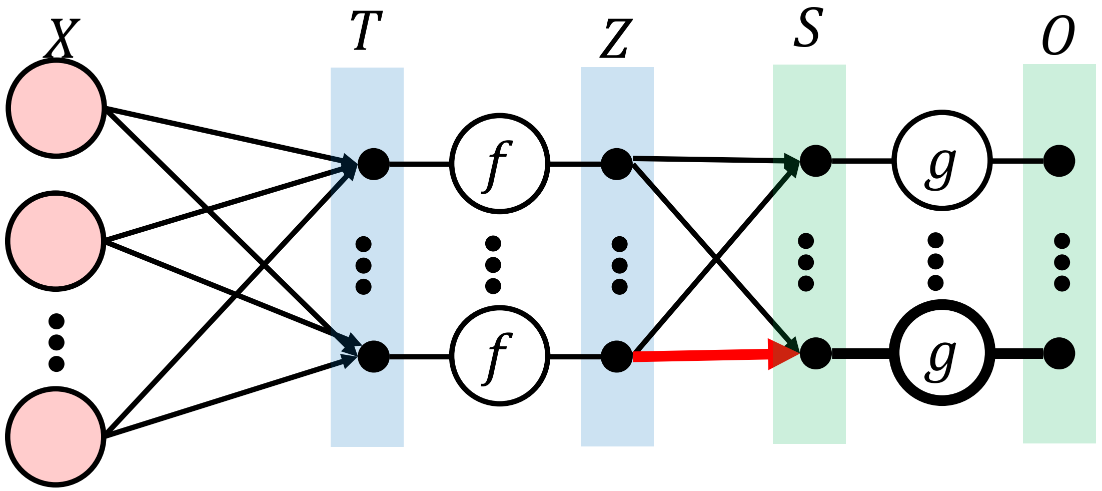
$$\frac{\partial}{\partial s_{ik}} o_{ik} = \frac{\partial}{\partial s_{ik}} \sigma(s_{ik}) = \sigma(s_{ik})[1 - \sigma(s_{ik})] = o_{ik}(1 - o_{ik})$$

we have

$$\frac{\partial J}{\partial \beta_{km}} = \sum_{i=1}^{N} \frac{\partial J_i}{\partial o_{ik}} \frac{\partial o_{ik}}{\partial s_{ik}} \frac{\partial s_{ik}}{\partial \beta_{km}}$$

$$= -2 \sum_{i=1}^{N} (y_{ik} - o_{ik}) o_{ik}(1 - o_{ik}) \frac{\partial s_{ik}}{\partial \beta_{km}}$$

Example backward propagation in a multi-layer network



Because

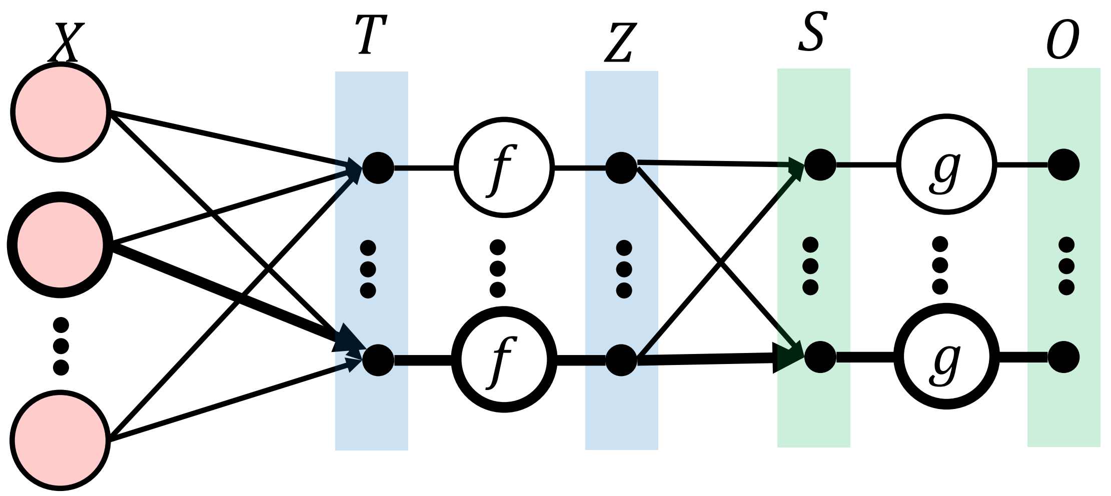
$$\frac{\partial}{\partial \beta_{km}} s_{ik} = \frac{\partial}{\partial \beta_{km}} \left(\beta_{k0} + \sum_{m=1}^{M} \beta_{km} z_{im} \right) = z_{im}$$

we have

$$\frac{\partial J}{\partial \beta_{km}} = \sum_{i=1}^{N} \frac{\partial J_i}{\partial o_{ik}} \frac{\partial o_{ik}}{\partial s_{ik}} \frac{\partial s_{ik}}{\partial \beta_{km}}$$

$$= -2 \sum_{i=1}^{N} (y_{ik} - o_{ik}) o_{ik} (1 - o_{ik}) z_{im}$$

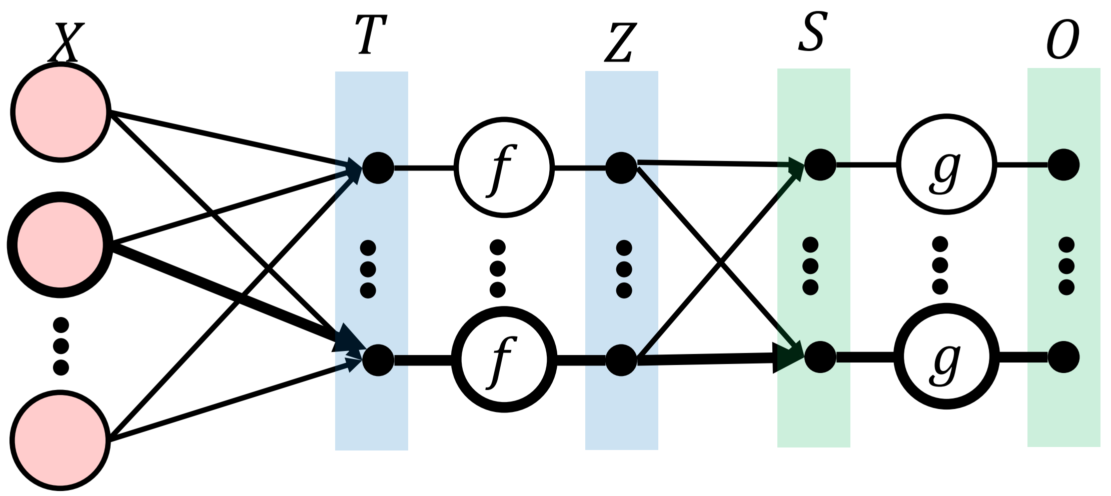
Example backward propagation in a multi-layer network



Consider hidden unit $j \in \{0,1,\ldots,p\}$ of the input layer and hidden unit $m \in \{0,1,\ldots,M\}$ of the hidden layer (layer 1).

We wish to know the rate of change of the cost function with respect to changes in the parameter α_{mj} for input unit j of the input layer leading to hidden unit m of layer 1.

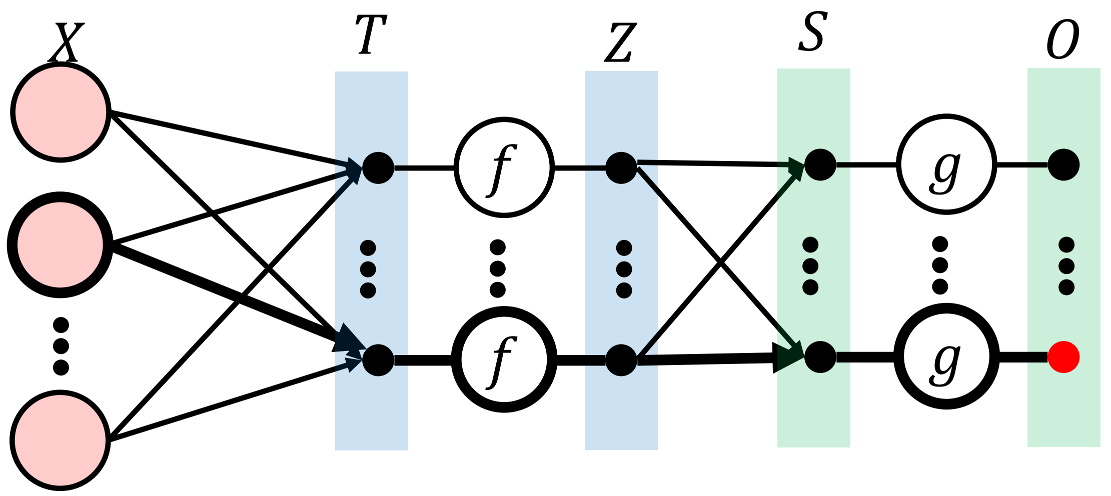
Example backward propagation in a multi-layer network



Applying the chain rule we have

$$\frac{\partial J}{\partial \alpha_{mj}} = \sum_{i=1}^{N} \sum_{k=1}^{K} \frac{\partial J_{ik}}{\partial o_{ik}} \frac{\partial o_{ik}}{\partial s_{ik}} \frac{\partial s_{ik}}{\partial z_{im}} \frac{\partial z_{im}}{\partial t_{im}} \frac{\partial t_{im}}{\partial \alpha_{mj}}$$

Example backward propagation in a multi-layer network



Because

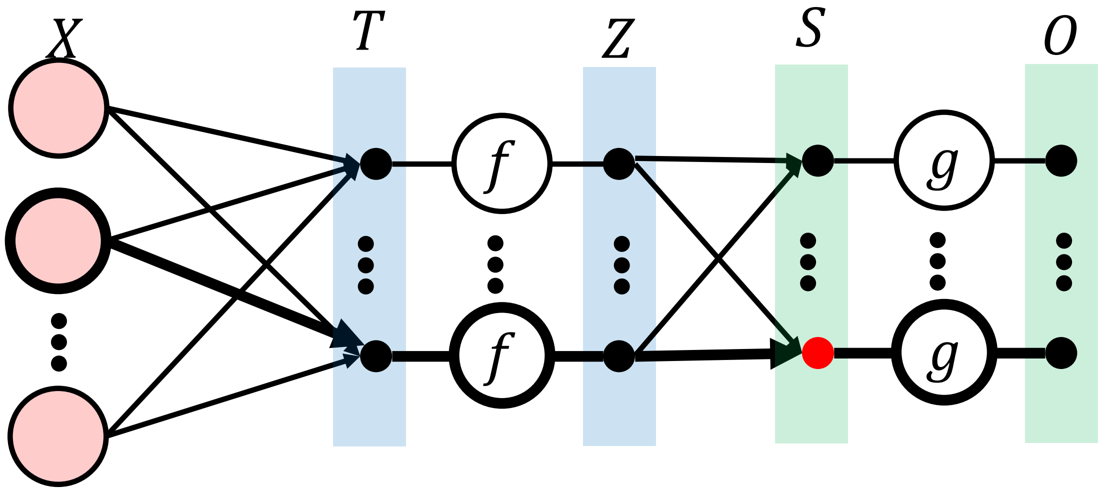
$$\frac{\partial}{\partial o_{ik}} J_{ik}(\theta) = \frac{\partial}{\partial o_{ik}} (y_{ik} - o_{ik})^2 = -2(y_{ik} - o_{ik})$$

we have

$$\frac{\partial J}{\partial \alpha_{mj}} = \sum_{i=1}^{N} \sum_{k=1}^{K} \frac{\partial J_{ik}}{\partial o_{ik}} \frac{\partial o_{ik}}{\partial s_{ik}} \frac{\partial s_{ik}}{\partial z_{im}} \frac{\partial z_{im}}{\partial t_{im}} \frac{\partial t_{im}}{\partial \alpha_{mj}}$$

$$= -2 \sum_{i=1}^{N} \sum_{k=1}^{K} (y_{ik} - o_{ik}) \frac{\partial o_{ik}}{\partial s_{ik}} \frac{\partial s_{ik}}{\partial z_{im}} \frac{\partial z_{im}}{\partial t_{im}} \frac{\partial t_{im}}{\partial \alpha_{mj}}$$

Example backward propagation in a multi-layer network



Because

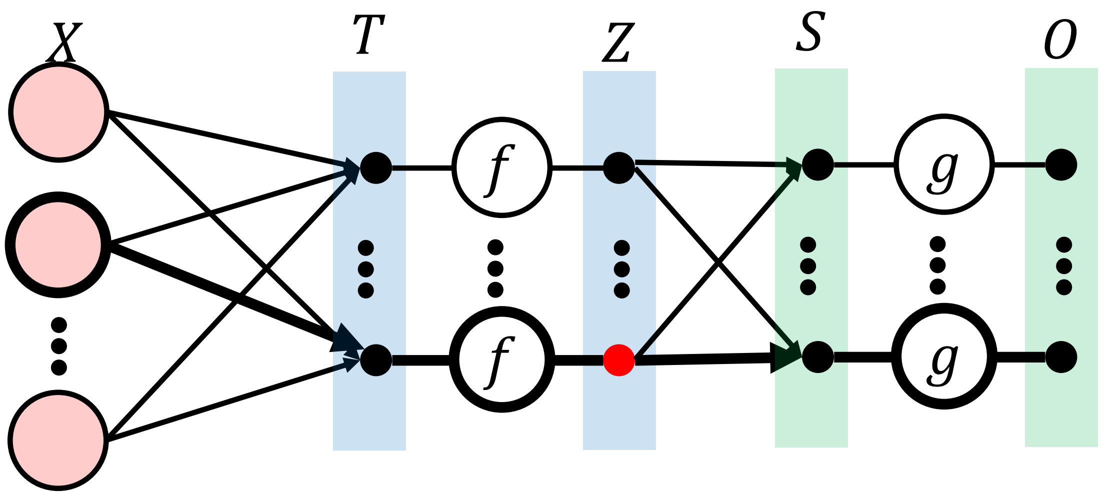
$$\frac{\partial}{\partial s_{ik}} o_{ik} = \frac{\partial}{\partial s_{ik}} \sigma(s_{ik}) = \sigma(s_{ik})[1 - \sigma(s_{ik})] = o_{ik}(1 - o_{ik})$$

we have

$$\frac{\partial J}{\partial \alpha_{mj}} = \sum_{i=1}^{N} \sum_{k=1}^{K} \frac{\partial J_{ik}}{\partial o_{ik}} \frac{\partial o_{ik}}{\partial s_{ik}} \frac{\partial s_{ik}}{\partial z_{im}} \frac{\partial z_{im}}{\partial t_{im}} \frac{\partial t_{im}}{\partial \alpha_{mj}}$$

$$= -2 \sum_{i=1}^{N} \sum_{k=1}^{K} (y_{ik} - o_{ik}) o_{ik}(1 - o_{ik}) \frac{\partial s_{ik}}{\partial z_{im}} \frac{\partial z_{im}}{\partial t_{im}} \frac{\partial t_{im}}{\partial \alpha_{mj}}$$

Example backward propagation in a multi-layer network



Because

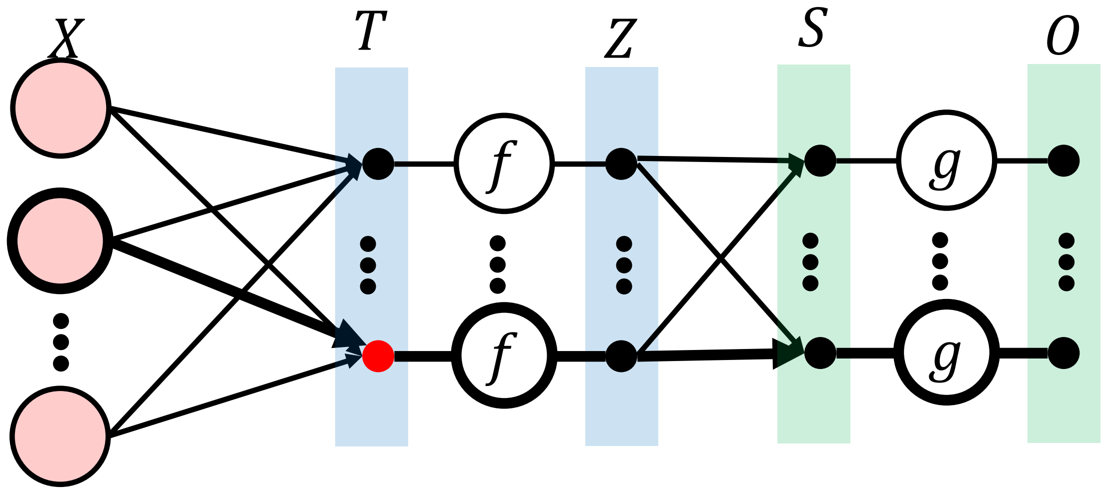
$$\frac{\partial}{\partial z_{im}} s_{ik} = \frac{\partial}{\partial z_{im}}\left(\beta_{k0} + \sum_{m=1}^{M} \beta_{km} z_{im}\right) = \beta_{km}$$

we have

$$\frac{\partial J}{\partial \alpha_{mj}} = \sum_{i=1}^{N}\sum_{k=1}^{K} \frac{\partial J_{ik}}{\partial o_{ik}} \frac{\partial o_{ik}}{\partial s_{ik}} \frac{\partial s_{ik}}{\partial z_{im}} \frac{\partial z_{im}}{\partial t_{im}} \frac{\partial t_{im}}{\partial \alpha_{mj}}$$

$$= -2\sum_{i=1}^{N}\sum_{k=1}^{K} (y_{ik} - o_{ik}) o_{ik} (1 - o_{ik}) \beta_{km} \frac{\partial z_{im}}{\partial t_{im}} \frac{\partial t_{im}}{\partial \alpha_{mj}}$$

Example backward propagation in a multi-layer network



Because

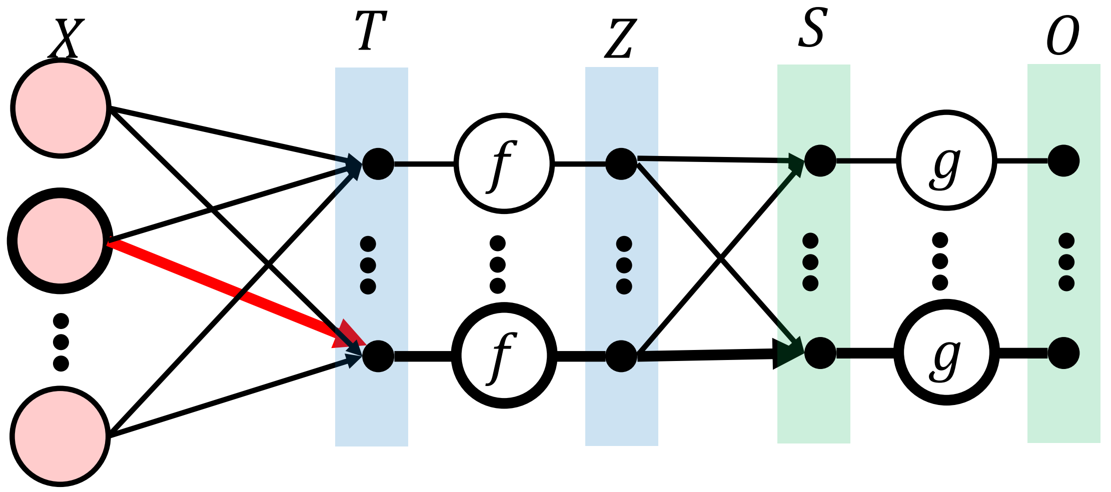
$$\frac{\partial}{\partial t_{im}} z_{im} = \frac{\partial}{\partial t_{im}} \mathrm{ReLU}(t_{im}) \ = I(t_{im} > 0)$$

we have

$$\frac{\partial J}{\partial \alpha_{mj}} = \sum_{i=1}^{N} \sum_{k=1}^{K} \frac{\partial J_{ik}}{\partial o_{ik}} \frac{\partial o_{ik}}{\partial s_{ik}} \frac{\partial s_{ik}}{\partial z_{im}} \frac{\partial z_{im}}{\partial t_{im}} \frac{\partial t_{im}}{\partial \alpha_{mj}}$$

$$= -2 \sum_{i=1}^{N} \sum_{k=1}^{K} (y_{ik} - o_{ik}) o_{ik} (1 - o_{ik}) \beta_{km} I(t_{im} > 0) \frac{\partial t_{im}}{\partial \alpha_{mj}}$$

Example backward propagation in a multi-layer network



Because

$$\frac{\partial}{\partial \alpha_{mj}} t_{im} = \frac{\partial}{\partial \alpha_{mj}} \left(\alpha_{m0} + \sum_{j=1}^{p} \alpha_{mj} x_{ij} \right) = x_{ij}$$

we have

$$\frac{\partial J}{\partial \alpha_{mj}} = \sum_{i=1}^{N} \sum_{k=1}^{K} \frac{\partial J_{ik}}{\partial o_{ik}} \frac{\partial o_{ik}}{\partial s_{ik}} \frac{\partial s_{ik}}{\partial z_{im}} \frac{\partial z_{im}}{\partial t_{im}} \frac{\partial t_{im}}{\partial \alpha_{mj}}$$

$$= -2 \sum_{i=1}^{N} \sum_{k=1}^{K} (y_{ik} - o_{ik}) o_{ik} (1 - o_{ik}) \beta_{km} I(t_{im} > 0) x_{ij}$$

Overfitting can occur

Because a neural network with even 1 hidden layer is a universal approximator, it can fit any function.

However, training observations are not made with complete fidelity, and therefore have sampling error.

Because of this error, a model that can fit any function will not only fit the true signal, but will also conform to the error.

We therefore need a way to prevent a neural network from overfitting the training data.

Strategies for avoiding overfitting

A number of strategies exist for avoiding overfitting for neural networks.

Three commons ones are:

1. Reducing the capacity of the network by ensuring that the total number of hidden units is controlled.
2. Performing regularization in which the lengths of parameter vectors (often called weight vectors in the neural network literature) are controlled (**weight decay**).
3. Stopping training early (**early stopping**).

Controlling number of units to decrease capacity

The capacity of a neural network is the set or space of functions that it can represent.

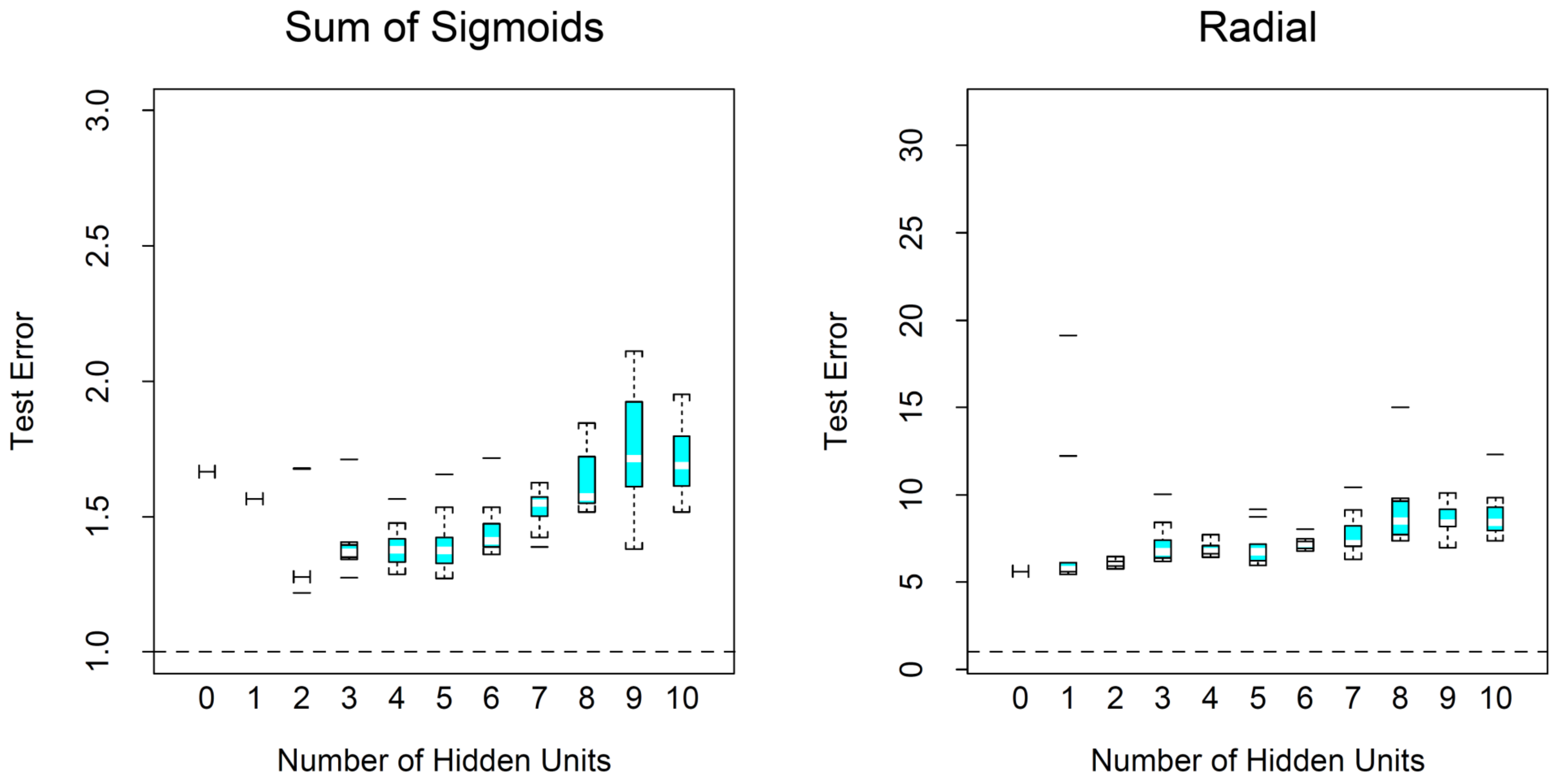
As the number of hidden units in a layer increases or as the number of hidden layers increases, the capacity of the network increases.

Therefore, a way to prevent overfitting is to limit the capacity of the network by reducing the number of hidden units (or layers), and therefore the overall number of parameters describing the network.

This should be clear, as permitting no hidden layers reduces the network to a linear model that we have discussed for the majority of this class, and so we consider other approaches for reducing network capacity.

Increasing number of hidden units overfits the data

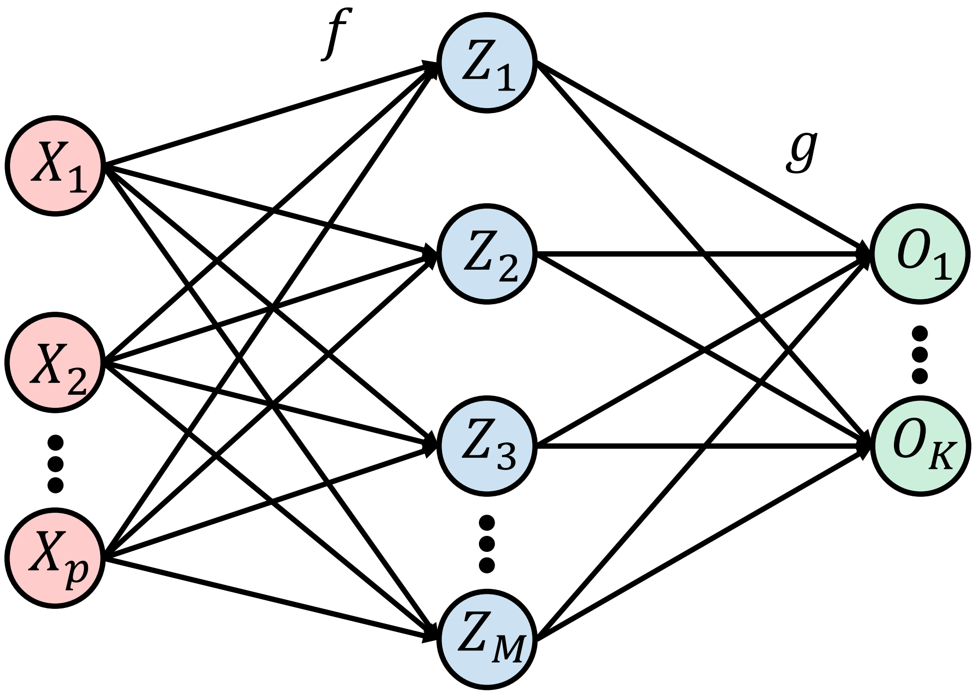
As expected, due to bias-variance tradeoff, increasing the number of units in a single hidden layer can increase test error (box plots based on random start positions).



Hastie *et al* (2009) *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2nd Ed.

Regularization through weight decay

Consider the following network



Regularization through **weight decay** is akin to ridge regression penalization, in which the weights (parameters) are reduced to 0 as a tuning parameter λ increases, and can reduce network capacity.

Suppose the network above is the network we wish to estimate.

Regularization through weight decay

Consider a regression-style problem using neural networks in which we have a loss function that is based on the residual sum of squares

$$R(\theta) = \sum_{i=1}^{N} \sum_{k=1}^{K} (y_{ik} - o_{ik})^2$$

We can penalize the magnitude of the set of parameter vectors by incorporating a tuning parameter λ and minimizing the cost function

$$J(\theta, \lambda) = R(\theta) + \lambda P(\theta)$$

where

$$P(\theta) = \sum_{m=1}^{M} \sum_{j=1}^{p} \alpha_{mj}^2 + \sum_{k=1}^{K} \sum_{m=1}^{M} \beta_{km}^2$$

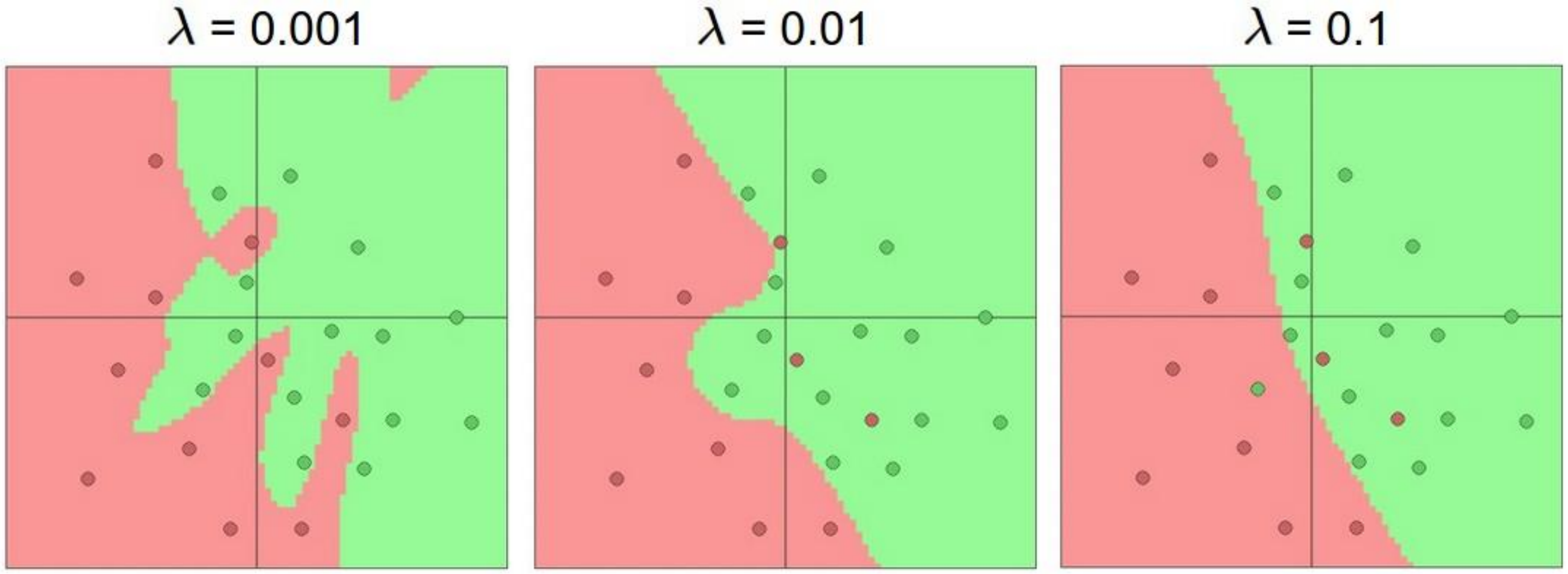
is a penalty term that shrink to 0 as λ goes to ∞.

Tuning parameter performs smoothing

As the tuning parameter λ in

$$J(\theta, \lambda) = R(\theta) + \lambda P(\theta)$$

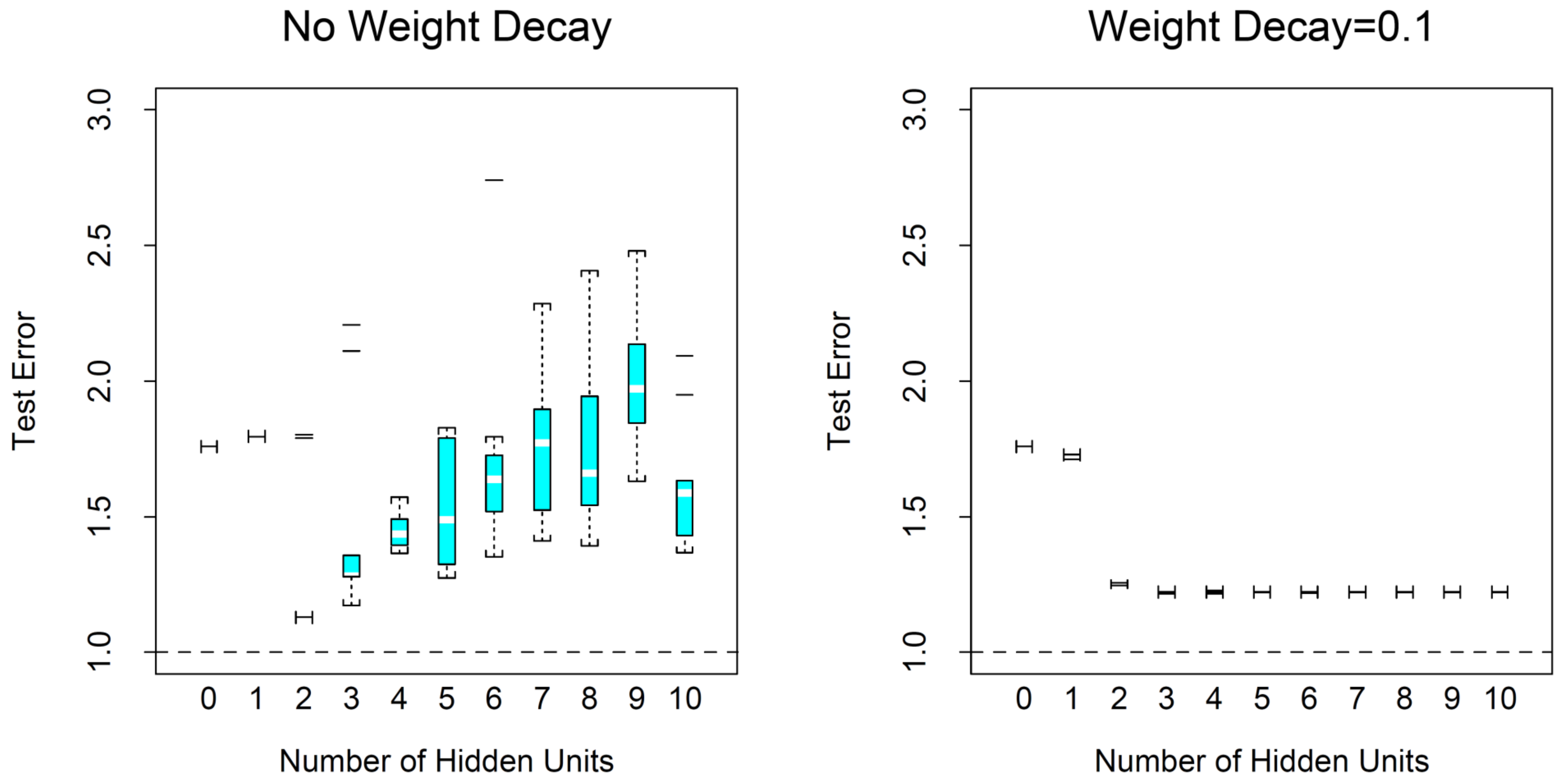
increases, the fit model becomes less flexible, and therefore trades increased bias for decreased variance.



http://cs231n.github.io/neural-networks-1/.

Weight decay controls overfitting

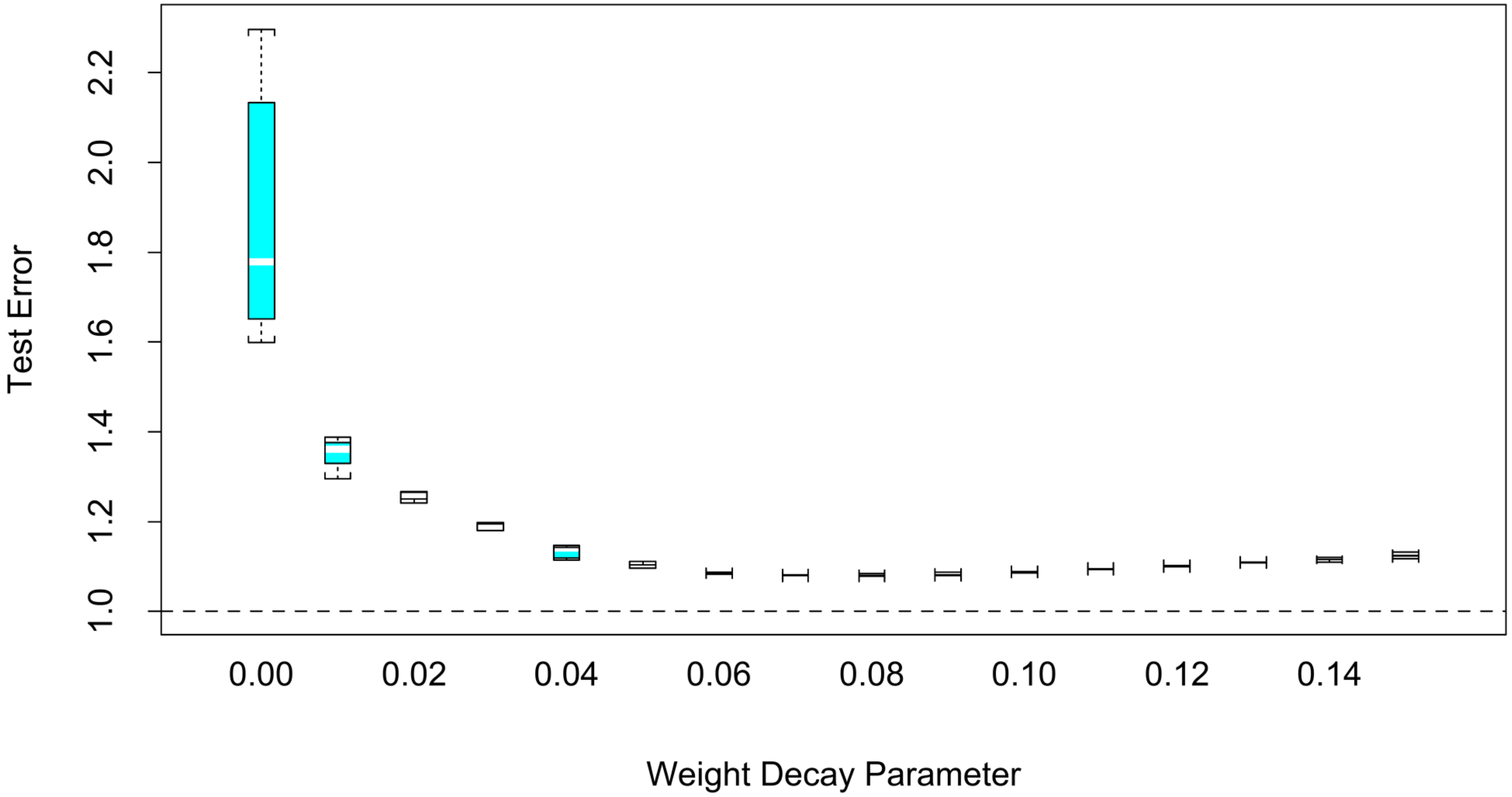
Penalizing the magnitude of the vectors based using a tuning parameter of $\lambda = 0.1$ controls overfitting, even as the number of units in a hidden layer increases.



Hastie *et al* (2009) *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2nd Ed.

Weight decay controls overfitting

For a given number of hidden units (10), there is a sweet spot for test accuracy with an intermediate value of the tuning parameter λ .



Hastie *et al* (2009) *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2nd Ed.

Choice of tuning parameter

The choice of tuning parameter can be highly computationally-burdensome, as it requires (cross) validation approaches.

In particular, for each (cross) validation step, each value of the tuning parameter λ would need to be evaluated, and with a parameter-rich model, this could take a significant amount of time.

Moreover, the cost function is generally not convex, which requires many random starts—only increases the computational burden.

Hence, an approach that does not require choice of a tuning parameter could be helpful.

Note: Because cost function generally not convex, the variability offered by stochastic gradient descent is preferred over batch.

Early stopping

A third approach is to stop model fitting prior to overfitting.

That is, keep training the model as long as its validation error is decreasing, but stop training as soon as validation error starts increasing.

Because a neural network with at least 1 hidden layer is universal approximator, with a large number of hidden units it is inevitable that the model will begin to overfit the data.

Therefore, it would be beneficial to stop the training process at a certain stage, such that validation (proxy for test) error is minimized.

An iteration is known as an epoch, and want to stop at optimal epoch.

Early stopping

Typically the initial model parameters are set to be extremely small in magnitude.

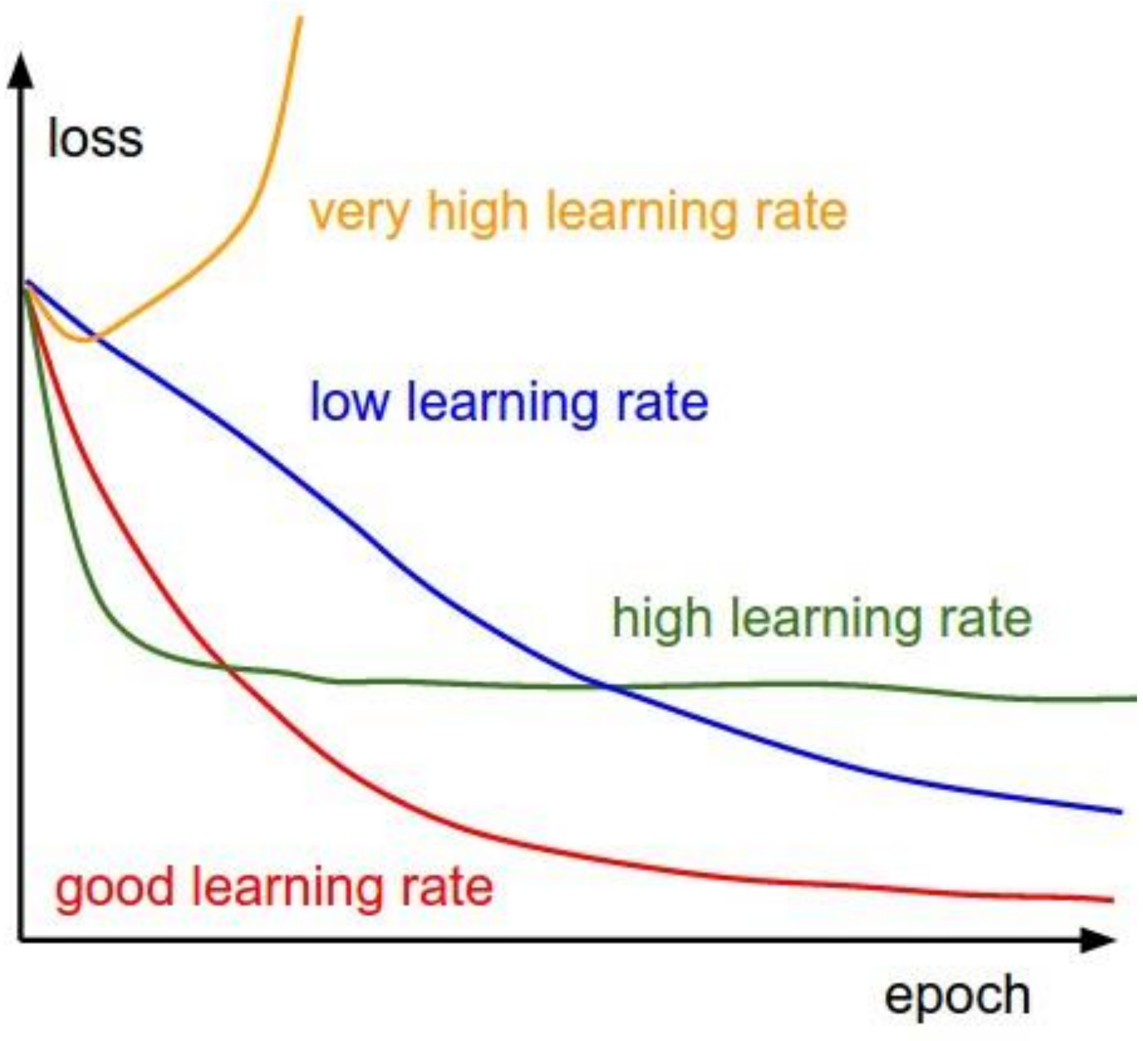
By setting the parameters small, the linear combination of features with these parameters will also be small.

These linear combinations will be used as input to non-linear functions f and g, but for very small magnitude inputs, the functions are roughly linear.

Therefore, by starting the parameter values as small in magnitude, the initial epochs (iterations) will represent roughly linear models, and as the number of epochs (iterations) proceeds, the learned model may become increasing non-linear.

Impact of epoch number on cost function

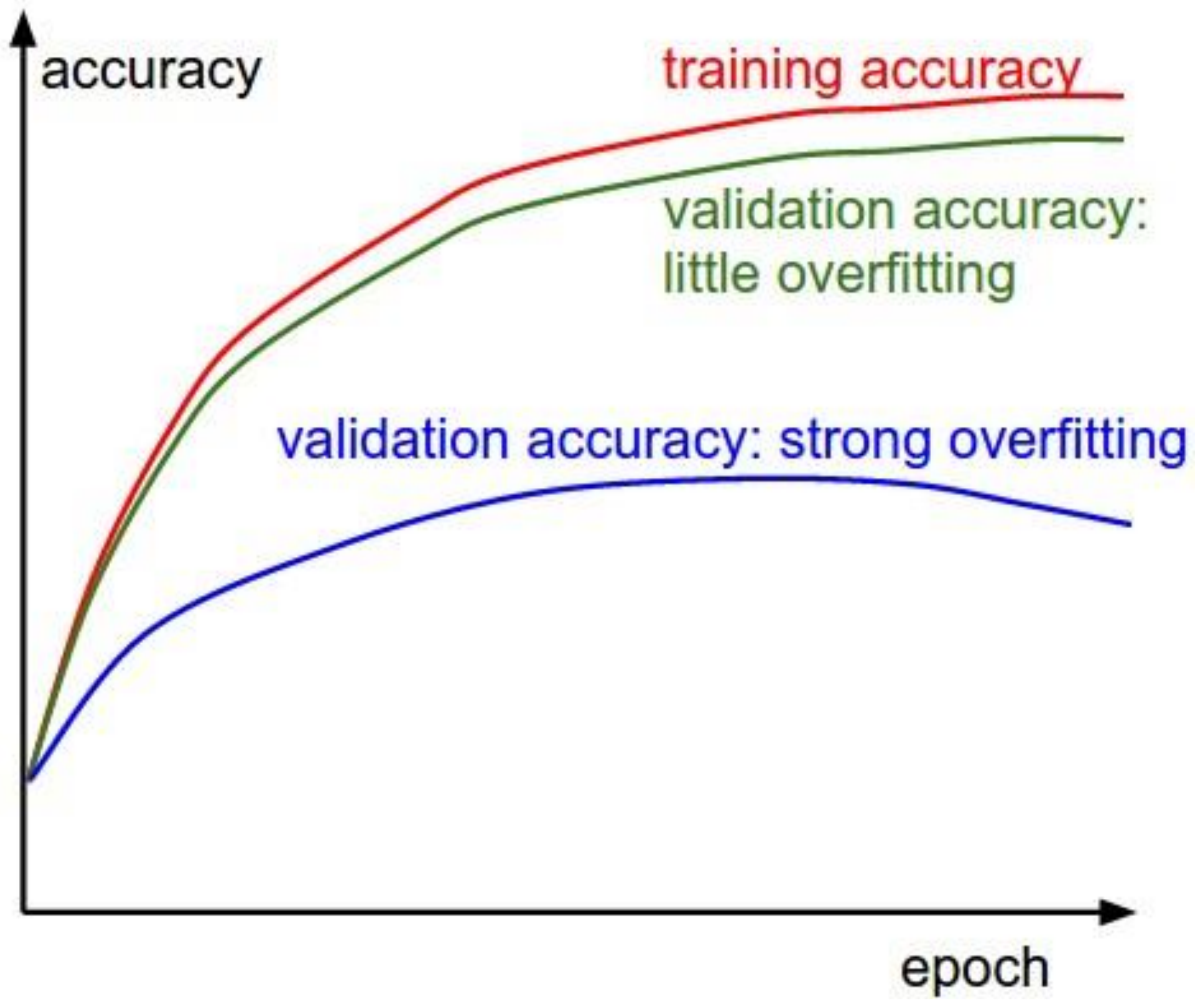
Impact of epoch, as a function of learning rate α, on the cost function $J(\theta)$.



http://cs231n.github.io/neural-networks-3/.

Impact of epoch number of accuracy

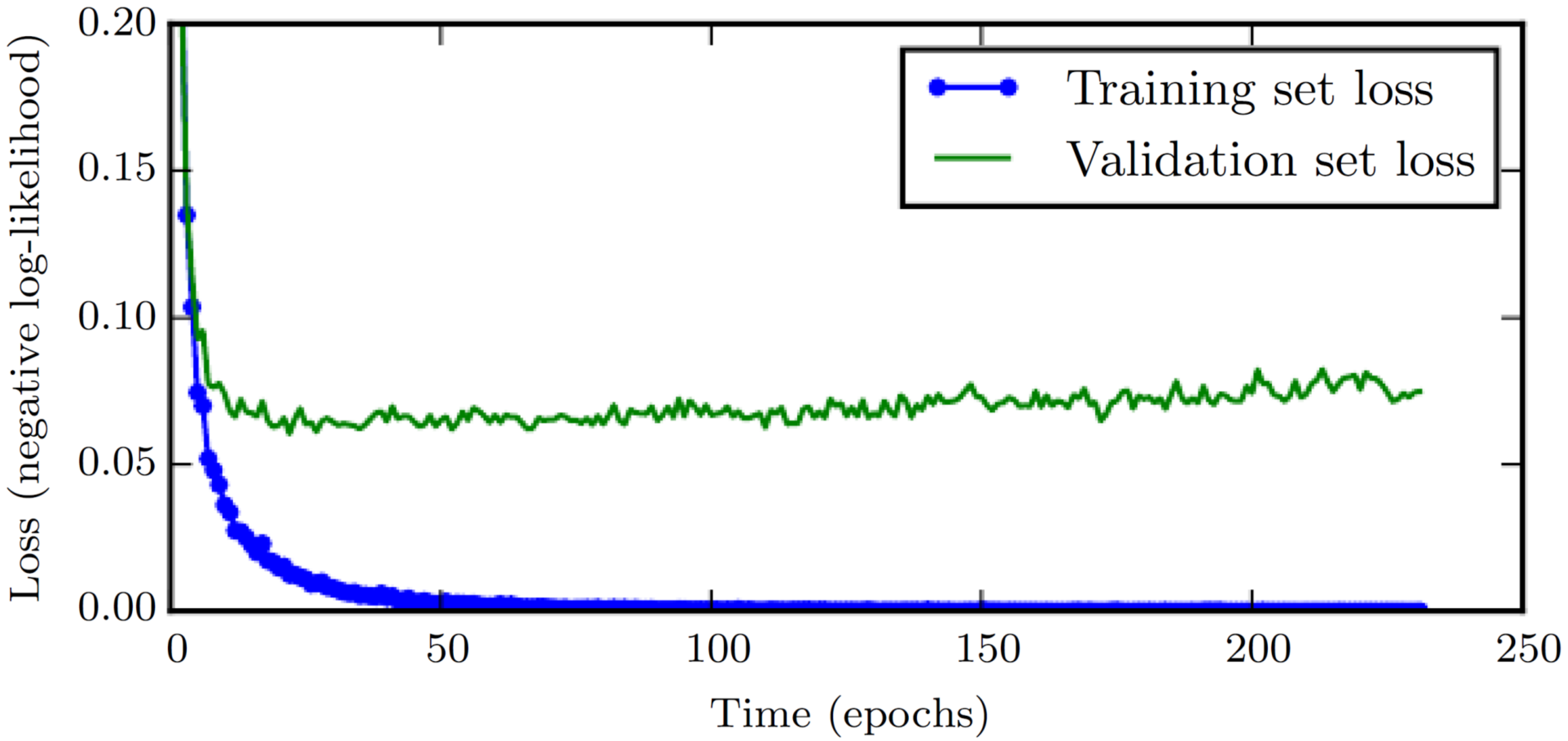
Impact of epoch on training and validation accuracy.



http://cs231n.github.io/neural-networks-3/.

Impact of epoch number of accuracy

Increasing number of epochs decreases training cost function, while increasing validation set cost function after a certain number of optimal epochs, yielding a U-shaped curve.



Goodfellow *et al* (2016) *Deep Learning*.MIT Press.