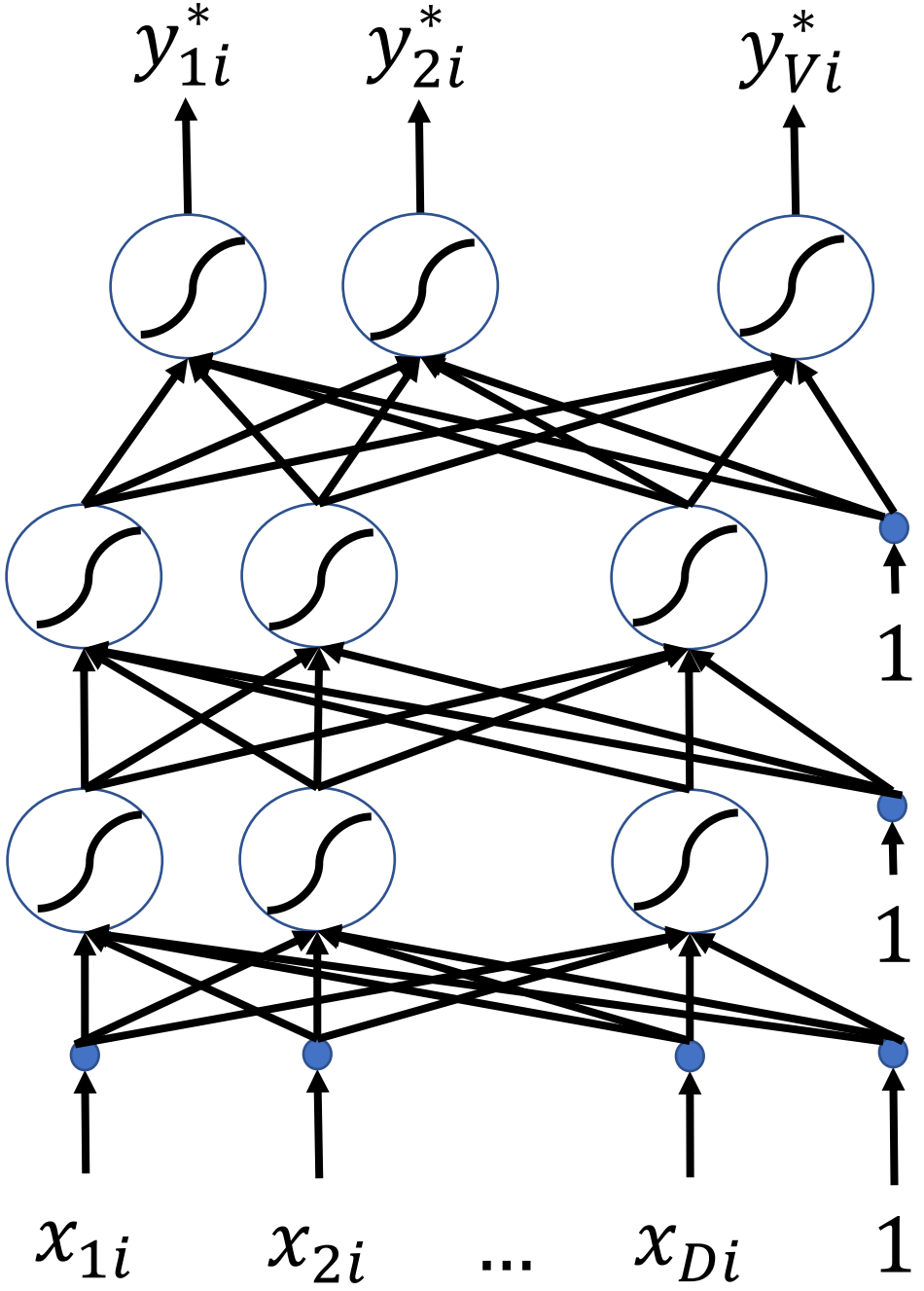
Lecture 28: Back-Propagation

Mark Hasegawa-Johnson

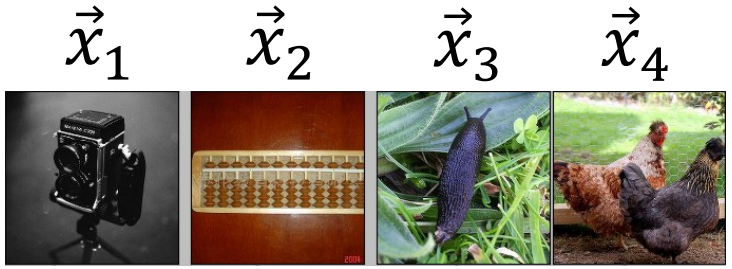
April 6, 2020



How to make a neural network

- Training Data
- Forward propagation
- Loss Function
- Back propagation

Training Data



$\vec{y}_1 = [0,1,0,0]$

$\vec{y}_2 = [1,0,0,0]$

$\vec{y}_3 = [0,0,0,1]$

$\vec{y}_4 = [0,0,1,0]$

Class Index	Class Name
1	abacus
2	camera
3	chickens
4	slug

A training database is a set of n feature vectors $\vec{x}_i$ $(1 \leq i \leq n)$, each with its reference label $\vec{y}_i$:

$$\mathcal{D} = \{(\vec{x}_1, \vec{y}_1), (\vec{x}_2, \vec{y}_2), \ldots, (\vec{x}_n, \vec{y}_n)\}$$

Thus far, we've seen two general types of reference labels:

- Scalars: y_i
- Vectors: $\vec{y}_i$

Scalar Labels

$y_1 = -1$

I'm warning you, it's pretty pathetic.

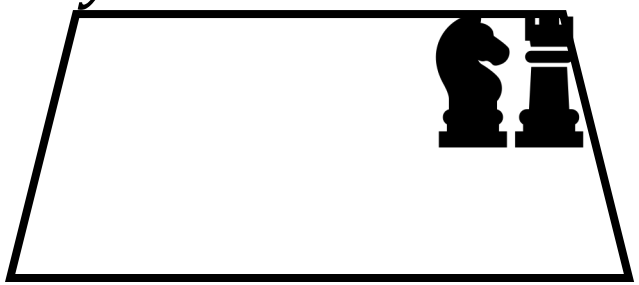
$y_2 = +1$

This is a beautiful movie, you'll love it.

$y_1 = 1$ $y_2 = 0$ $y_3 = 1$ $y_4 = 0$

Learning Multiple Layers of Features from Tiny Images, Alex Krizhevsky, 2009

$y = 97$

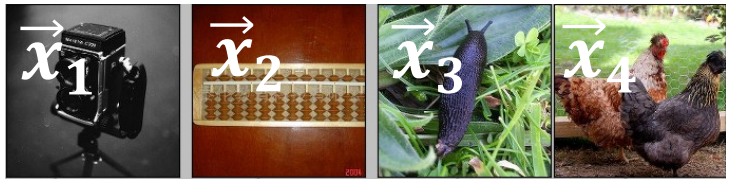


$$\mathcal{D} = \{(\vec{x}_1, y_1), (\vec{x}_2, y_2), \ldots, (\vec{x}_n, y_n)\}$$

Scalar labels:

- Signed binary (+1 or -1)
  - Example, -1=negative sentiment, +1=positive sentiment
- Unsigned binary (0 or 1)
  - Example: 1=animal, 0=non-animal.
- Any real number, $-\infty < y_i < \infty$
  - For example, the estimated value (for the black player) of a particular board position in chess

Vector Labels

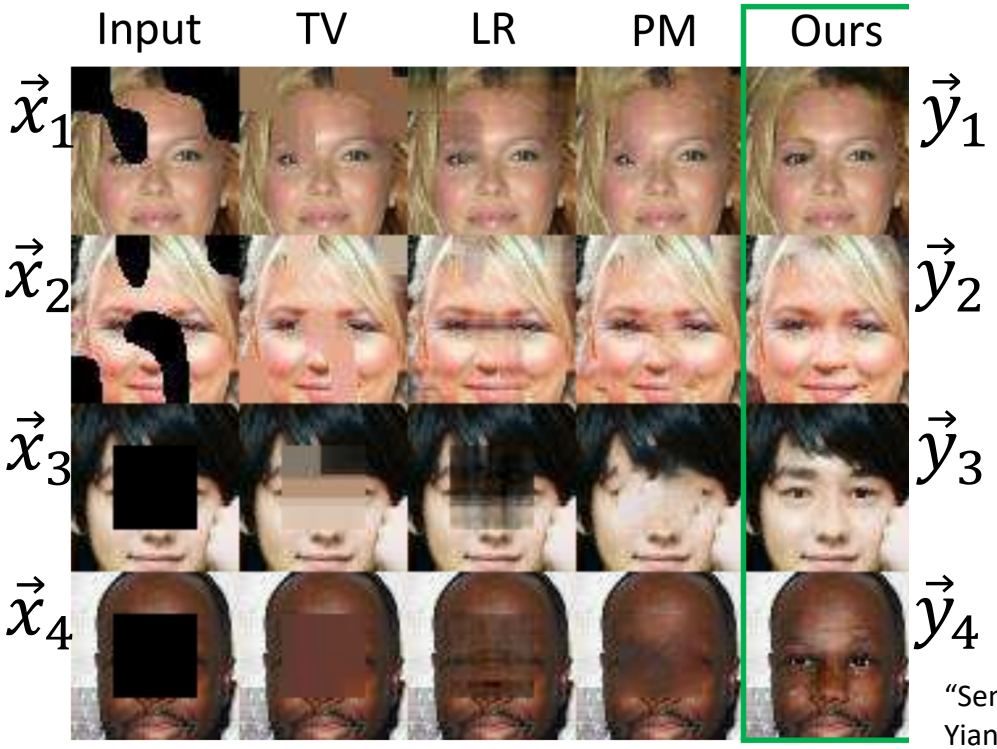


$\vec{y}_1 = [0,1,0,0]$ $\vec{y}_2 = [1,0,0,0]$ $\vec{y}_3 = [0,0,0,1]$ $\vec{y}_4 = [0,0,1,0]$

$$\mathcal{D} = \{(\vec{x}_1, \vec{y}_1), (\vec{x}_2, \vec{y}_2), \ldots, (\vec{x}_n, \vec{y}_n)\}$$

Vector labels:

- A one-hot vector, in which the correct class has a “1”, and all other classes have a “0”.
  - Example: object recognition
  - Example: part of speech tagging
- A real vector
  - Example: input is a noisy image, desired output is the clean image.



“Semantic Image Inpainting with Deep Generative Models,” Raymond Yeh, Chen Chen, Teck Yian Lim, Alexander G. Schwing, Mark Hasegawa-Johnson and Minh Do

How to make a neural network

- Training Data
- Forward propagation
- Loss Function
- Back propagation

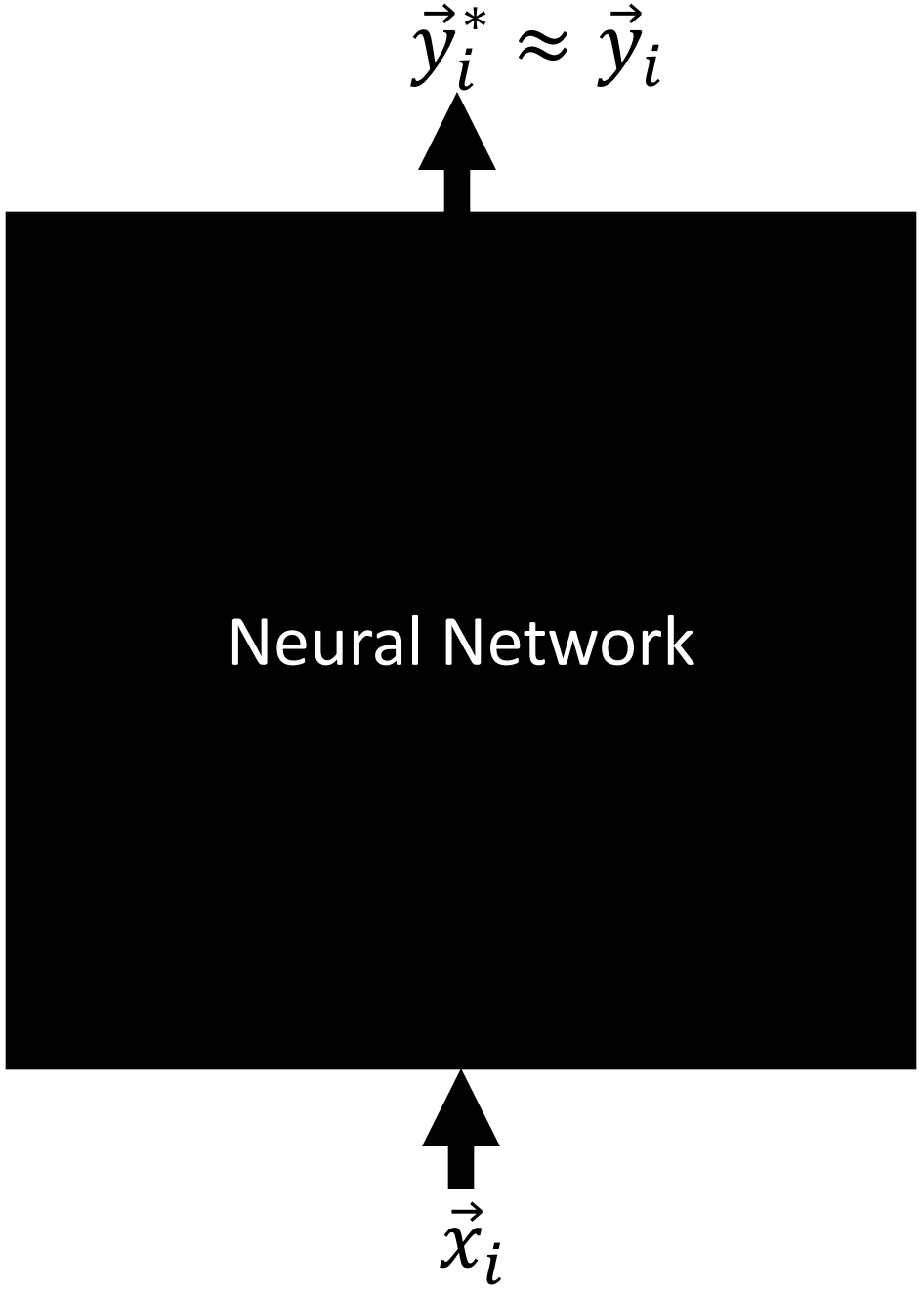
Forward propagation

Remember that a training database is a set of feature vectors, each with a reference label: $\mathcal{D} = \{(\vec{x}_1, \vec{y}_1), \ldots, (\vec{x}_n, \vec{y}_n)\}$

"Forward propagation" means computing the neural network output, $\vec{y}_i^*$, from its input, $\vec{x}_i$. Our goal is going to be to train the neural network so that

$$\vec{y}_i^* \approx \vec{y}_i$$

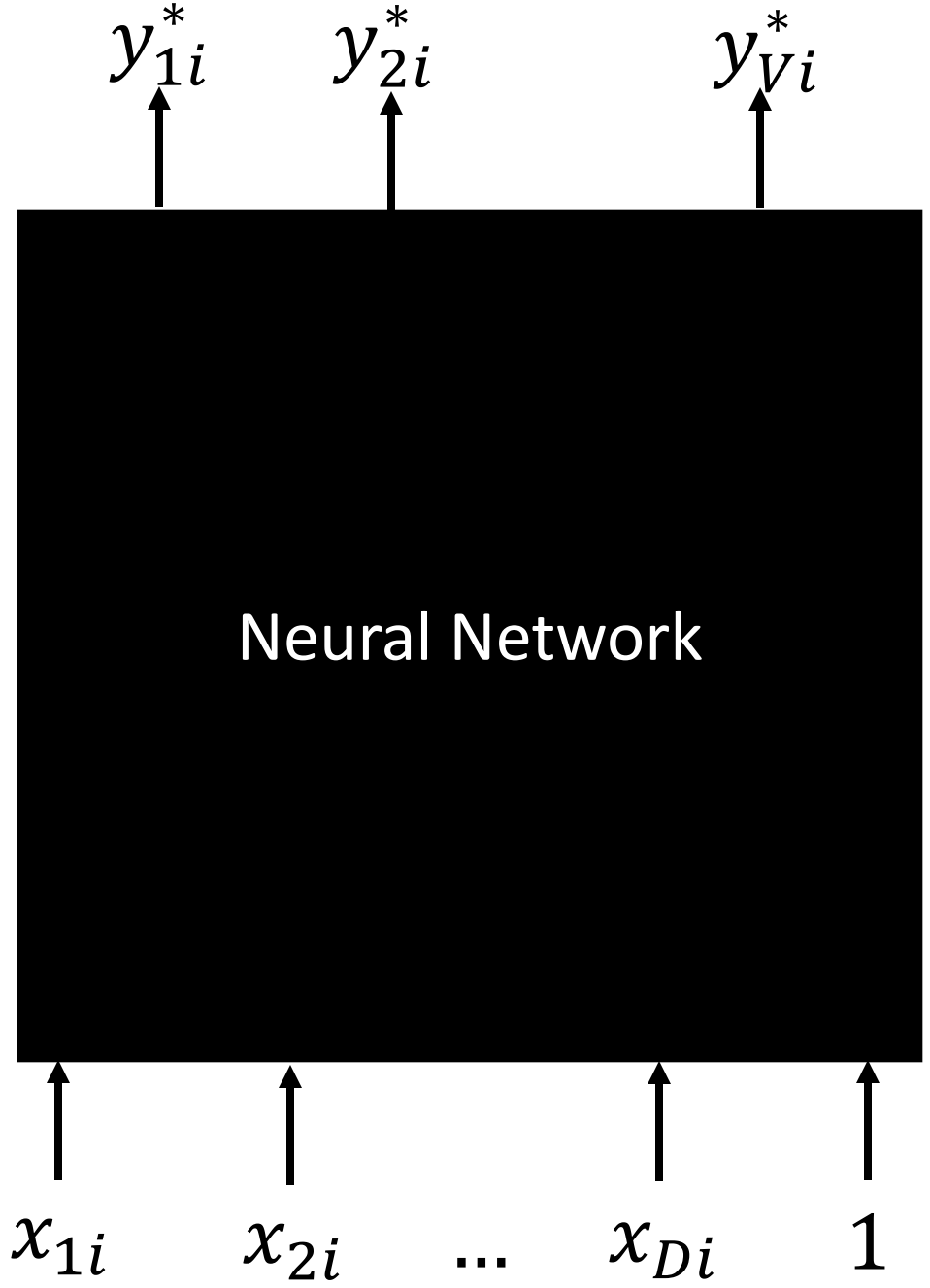
For each of the training tokens (for $1 \leq i \leq n$).



Forward propagation

"Forward propagation" means:

- Computing the neural network output, $\vec{y}_i^* = [y_{1i}^*, \ldots, y_{Vi}^*]$,
- from its input, $\vec{x}_i = [x_{1i}, \ldots, x_{Di}]$.



Forward propagation

Tokens $(\vec{x}_i, \vec{y}_i)$:

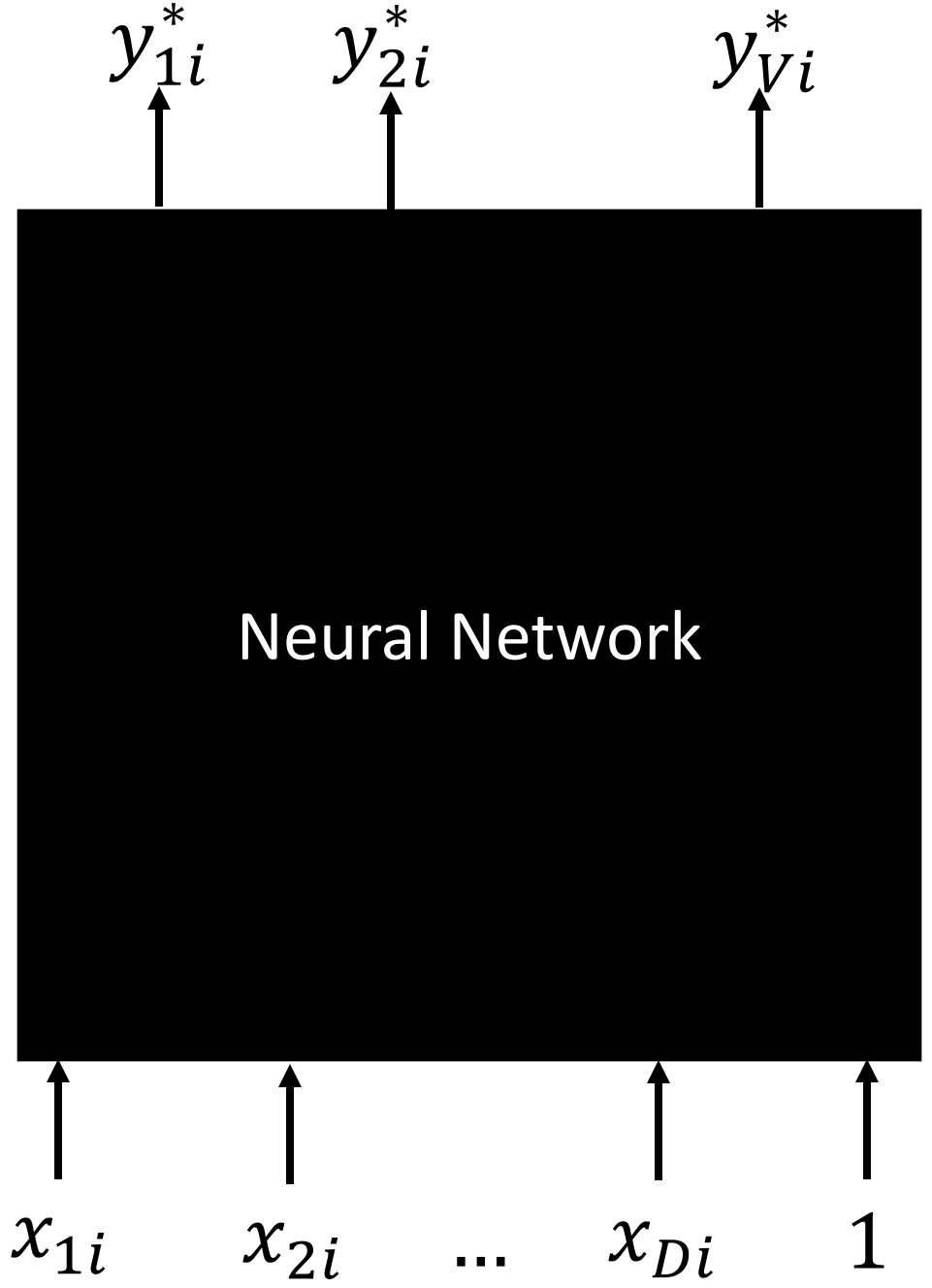
- $i =$ training token index, $1 \leq i \leq n.$
- n is the number of training tokens.

Classes y^*_{ci}:

- c is the class index, $1 \leq c \leq V$ (this terminology makes sense if $\vec{y}_i$ is a one-hot vector, i.e., $y_{ci} = 1$ for the correct class, $y_{ci} = 0$ otherwise).
- V is the vocabulary size (the number of distinct classes).

Feature dimensions x_{di}:

- d is the input feature dimension, $1 \leq d \leq D$.
- D is the number of input features per training token.



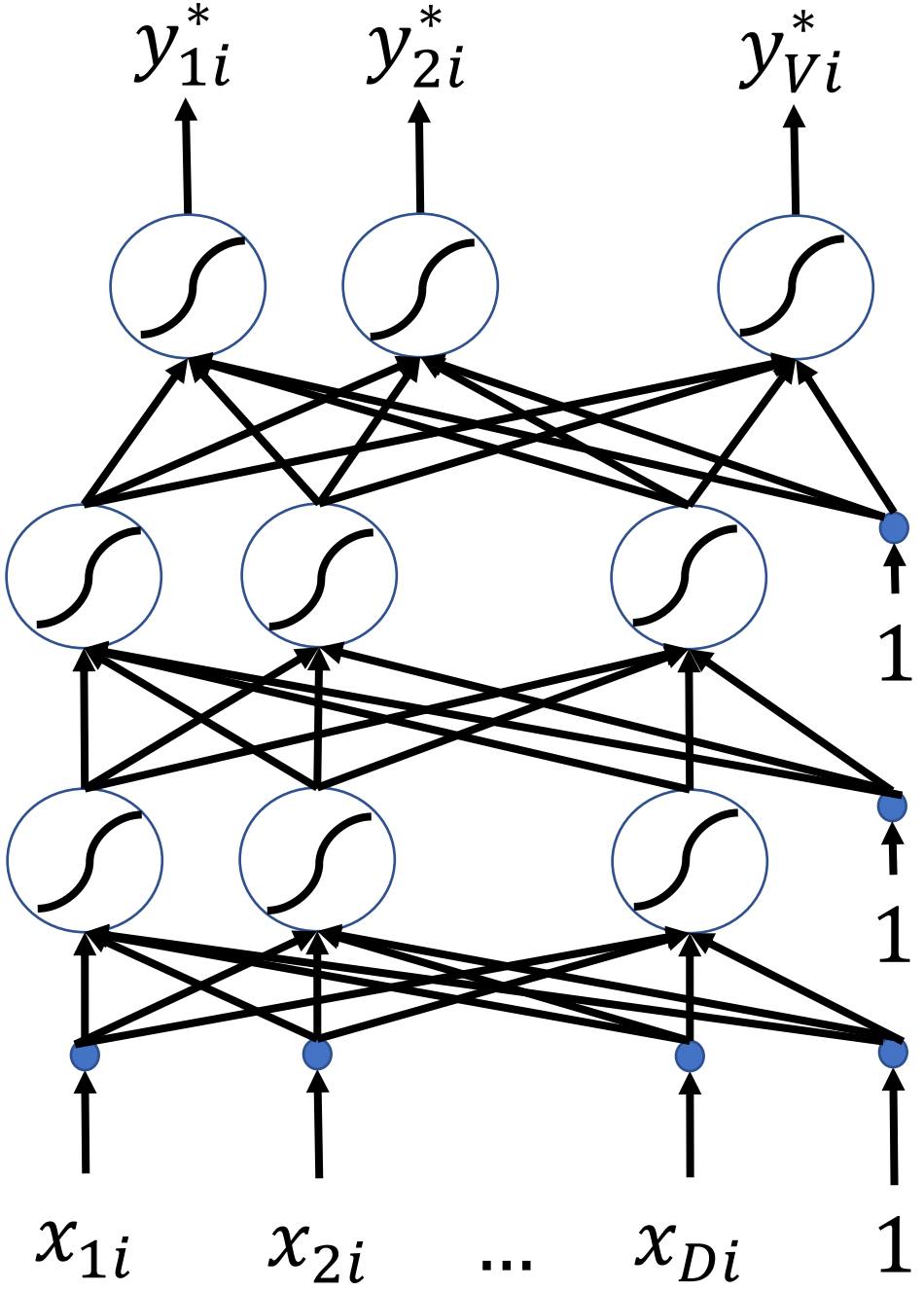
Forward propagation

Forward propagation is done by way of a series of layers.

Each layer is composed of edges (shown as arrows), that terminate on nodes (shown as circles).

There are three types of nodes:

- Input nodes = features x_{di}
- Output nodes = outputs y^*_{ci}
- Hidden nodes = the nodes in between.



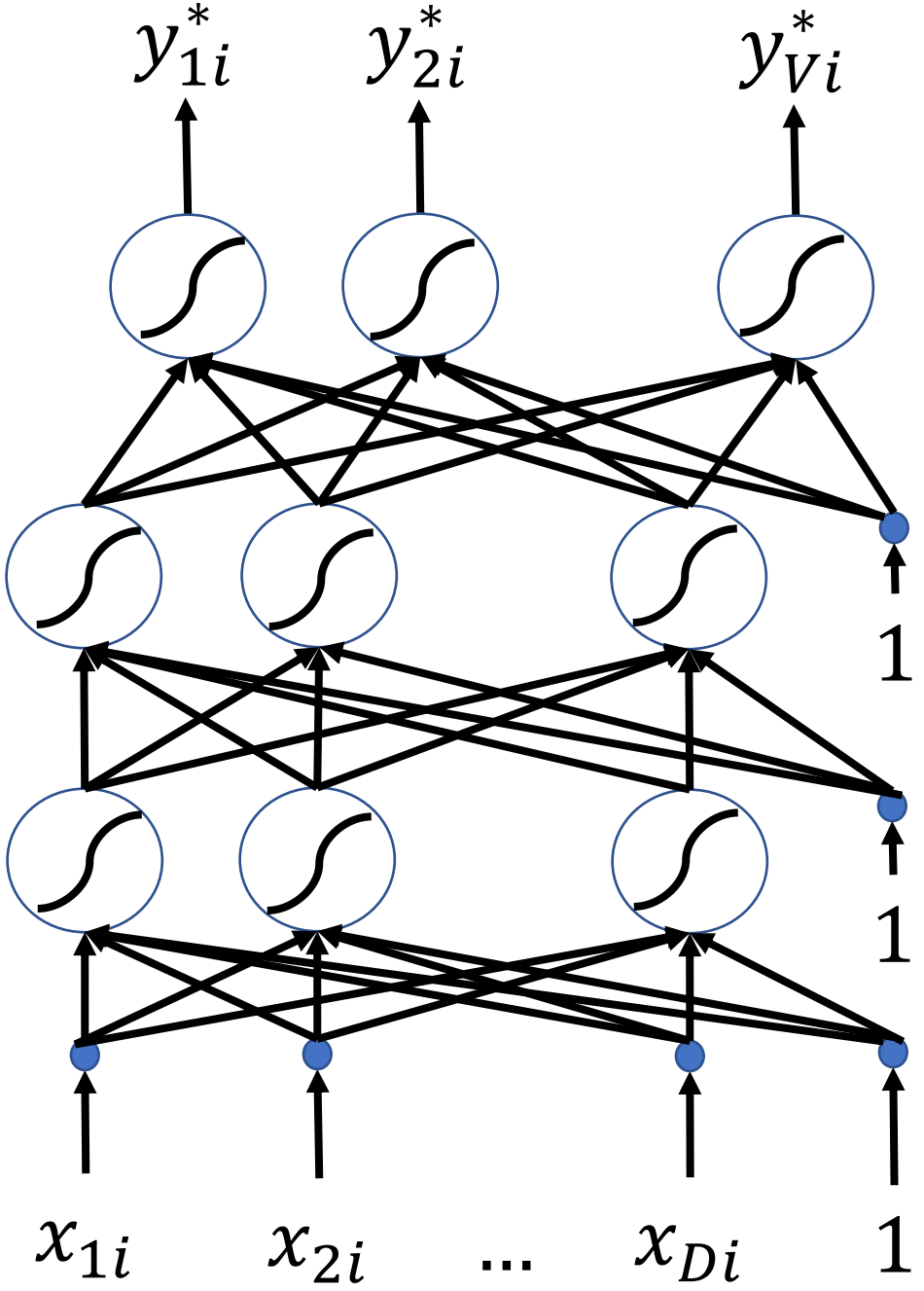
Forward propagation

Let's use this notation:

- $\beta_{ki}^{(l)}$ is the ***excitation*** of the kth node in the lth layer in response to the ith training token.
- $h_{ki}^{(l)}$ is the ***hidden node activation*** of the kth node in the lth layer in response to the ith training token.

Where:

- $1 \leq i \leq n, n =$# of training tokens
- $1 \leq k \leq N, N =$# of nodes per layer
- $1 \leq l \leq L, L =$# of layers



How forward propagation works

- Each **excitation** is a linear combination of the previous node's activations:

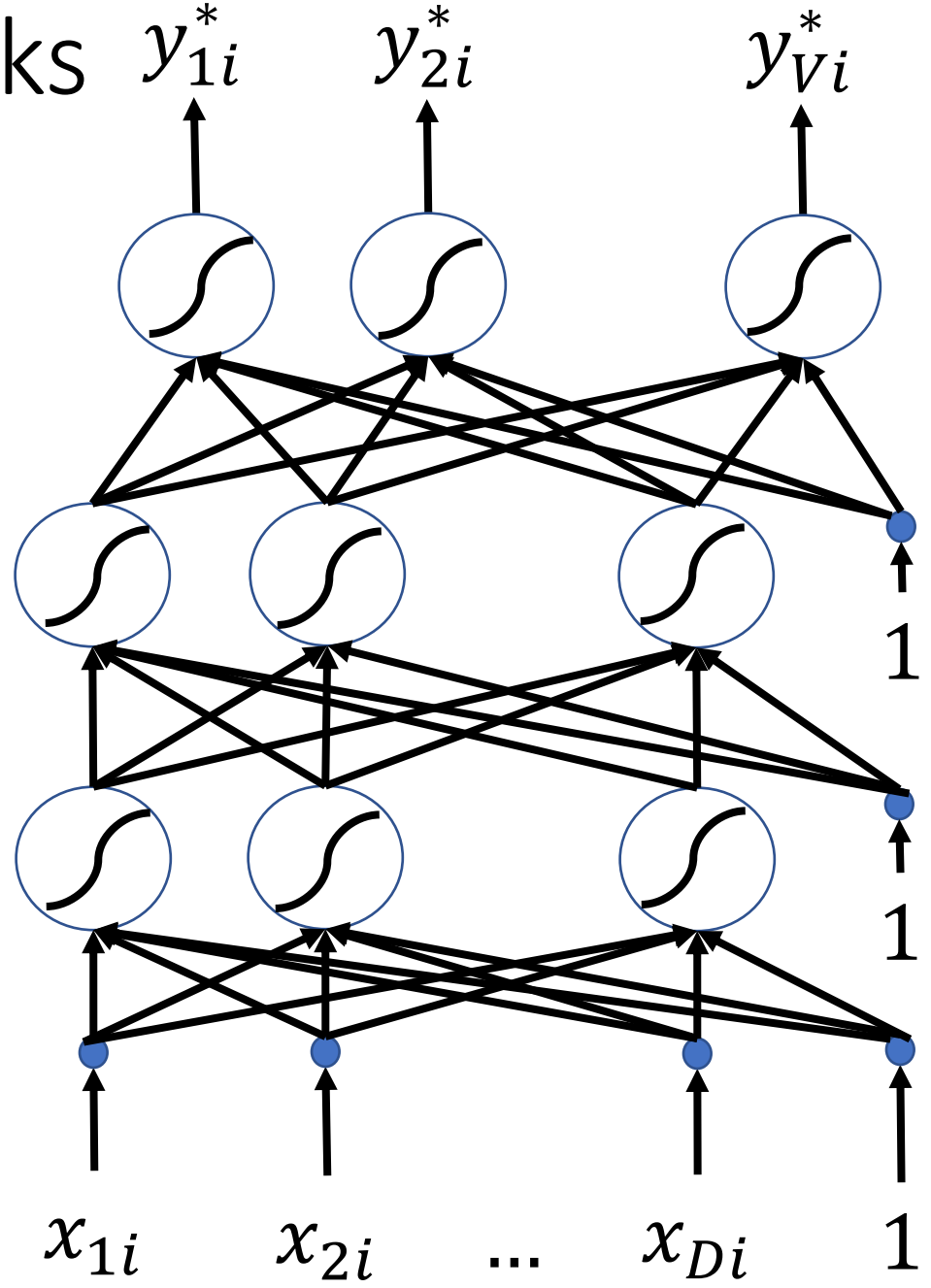
$$\beta_{ki}^{(l)} = \sum_{j=1}^{N+1} w_{kj}^{(l)} h_{ji}^{(l-1)}$$

…where $w_{kj}^{(l)}$ is called a "network weight," and will be learned using back-propagation.

- Each **activation** is a scalar nonlinearity applied to the excitation:

$$h_{ki}^{(l)} = g\left(\beta_{ki}^{(l)}\right)$$

…where $g(\cdot)$ is called the "activation function;" it needs to be chosen in advance by the network designer.



How forward propagation **starts…**

Forward propagation starts by setting the 0^{th} layer activations equal to the input features:

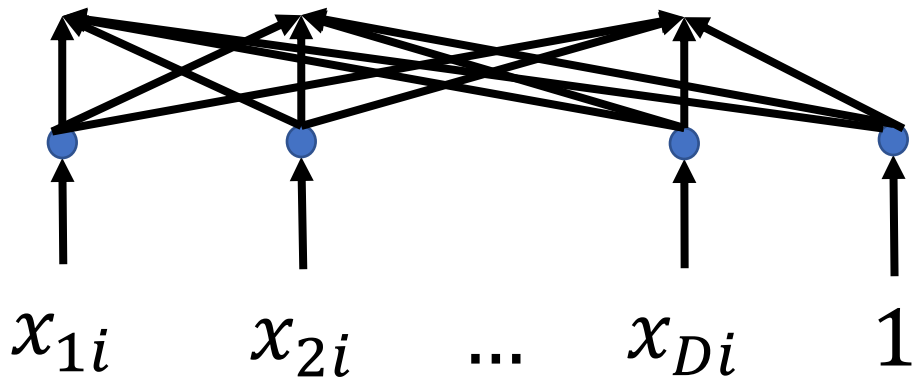
$$h_{ki}^{(0)} = x_{ki}, \qquad 1 \leq k \leq D$$

$$h_{ki}^{(0)} = 1, \qquad k = D + 1$$

How forward propagation continues…

Then we calculate the first layer excitations as:

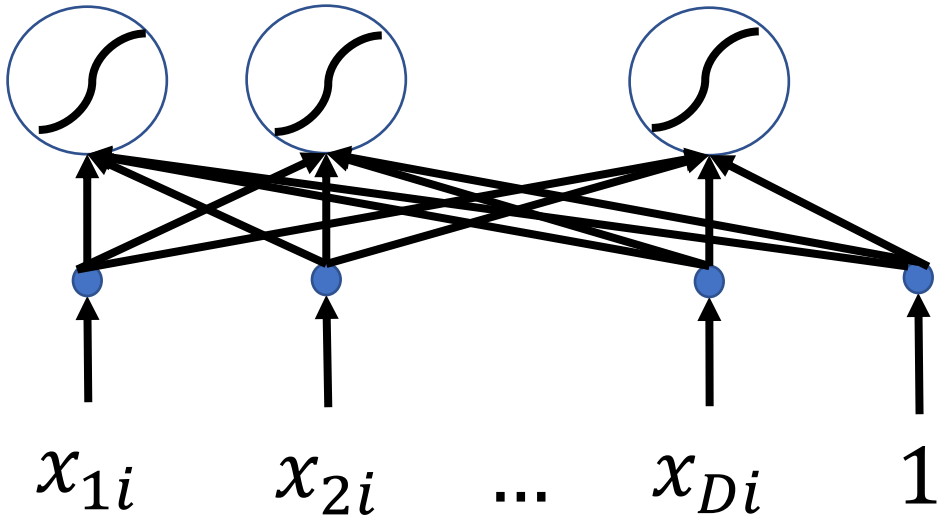
$$\beta_{ki}^{(1)} = \sum_{j=1}^{D+1} w_{kj}^{(1)} h_{ji}^{(0)}, \qquad 1 \leq k \leq N$$



How forward propagation continues…

Then we calculate the first-layer activations as:

$$h_{ki}^{(1)} = g\left(\beta_{ki}^{(1)}\right), \qquad 1 \leq k \leq N$$

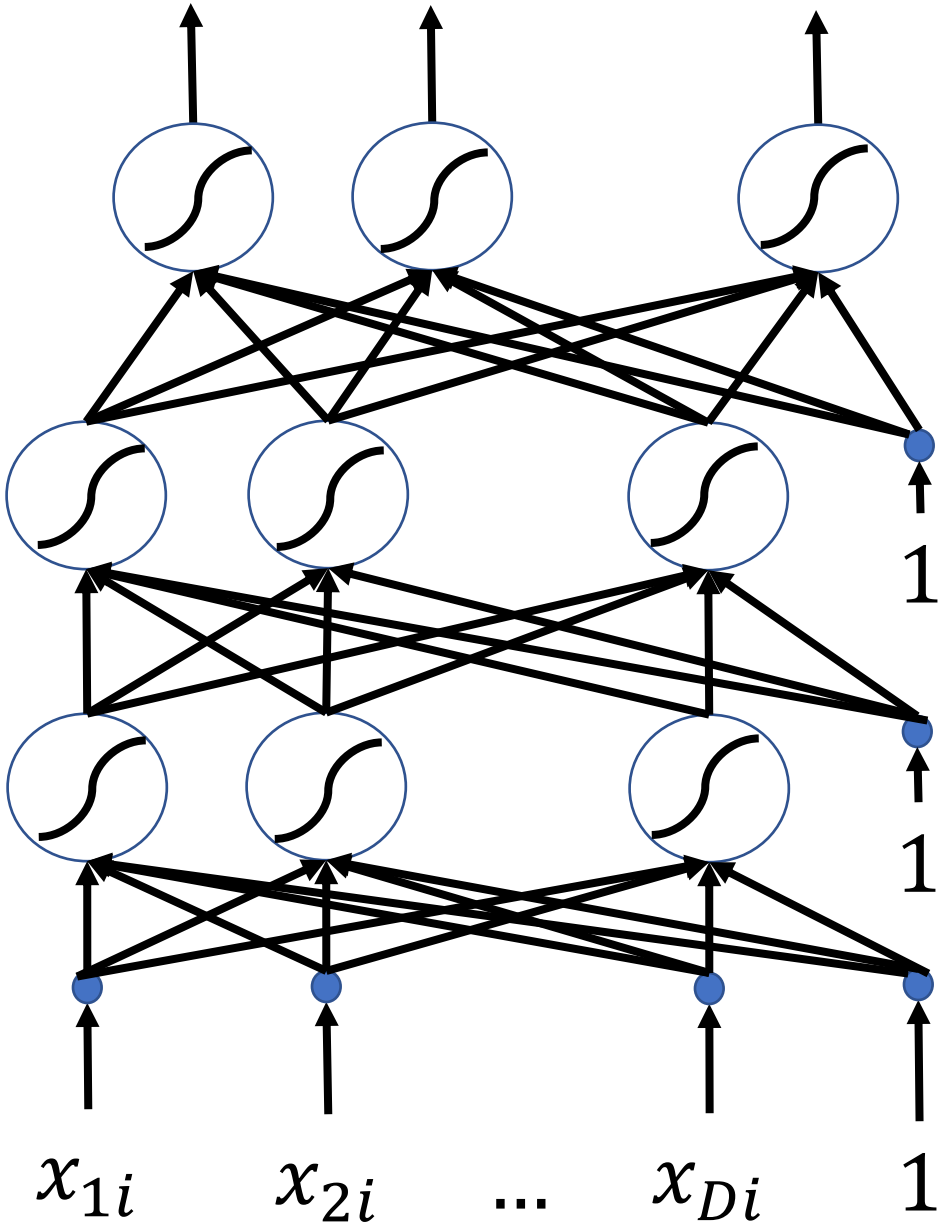


How forward propagation continues…

…and so on, repeating the following two equations, layer after layer, until we get to the L[th] layer:

$$\beta_{ki}^{(l)} = \sum_{j=1}^{N+1} w_{kj}^{(l)} h_{ji}^{(l-1)}$$

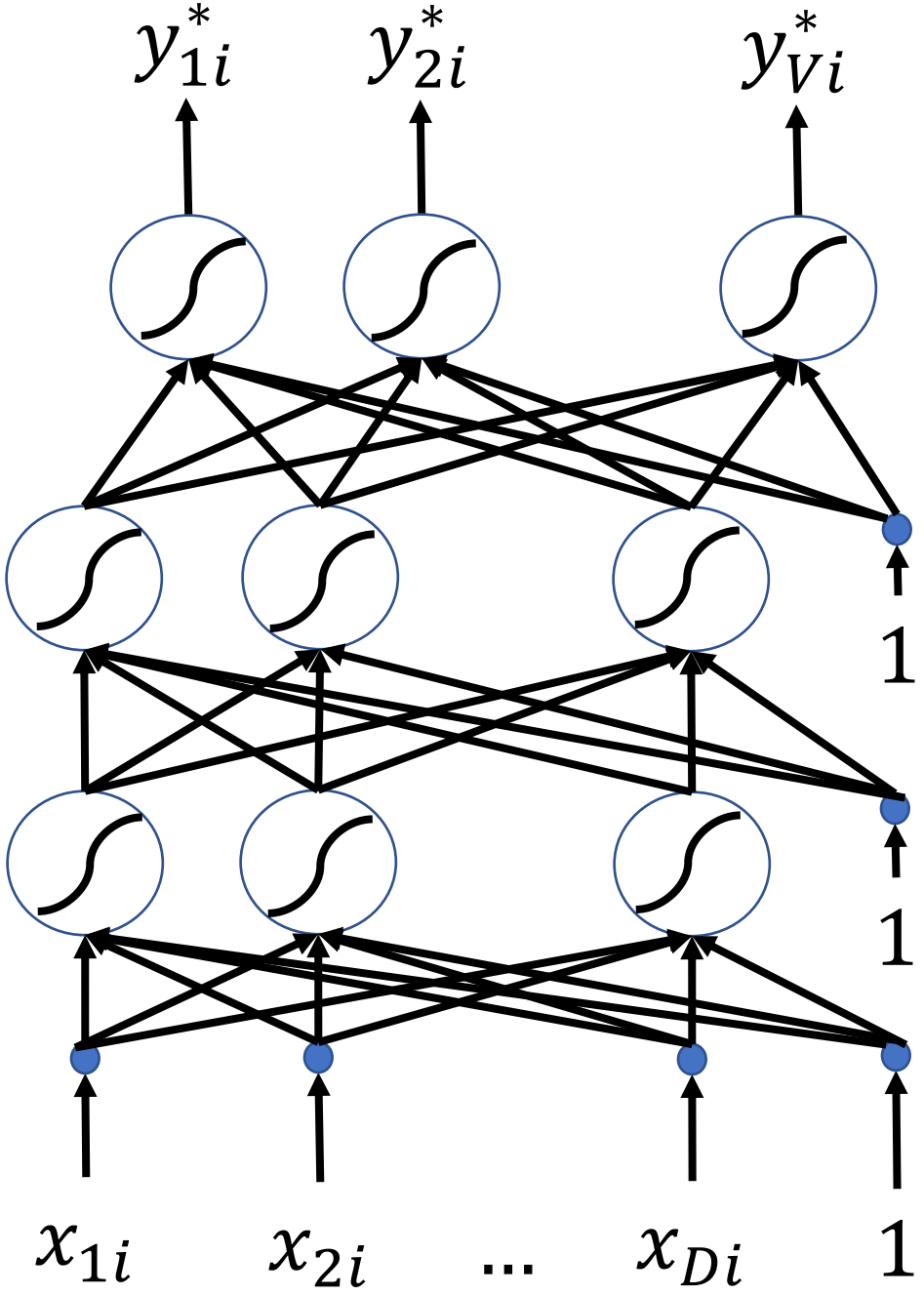
$$h_{ki}^{(l)} = g\left(\beta_{ki}^{(l)}\right)$$



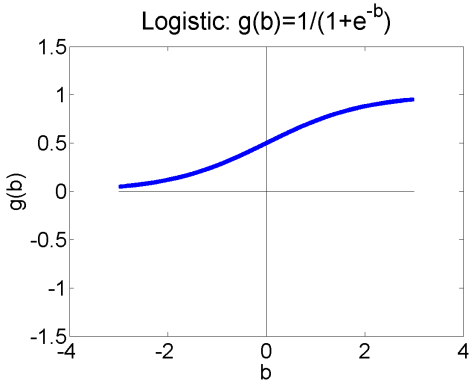
How forward propagation **ends**.

…and then we set the neural net output equal to the activation of the last layer:

$$y_{ci}^{*} = h_{ci}^{(L)}, \qquad 1 \le c \le V$$



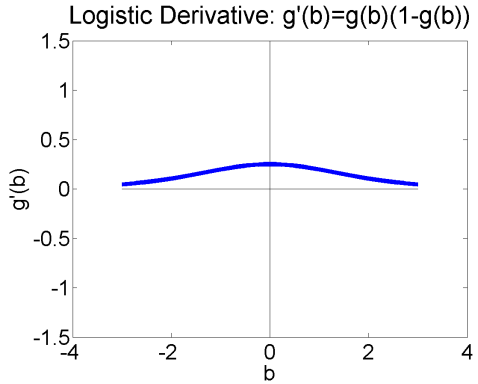
Activation functions

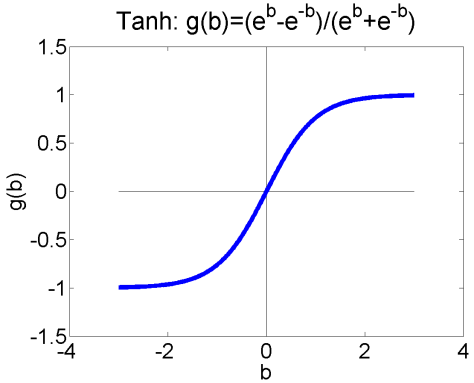


What is that "activation function?" Here are some that you should know about:

Logistic Sigmoid:

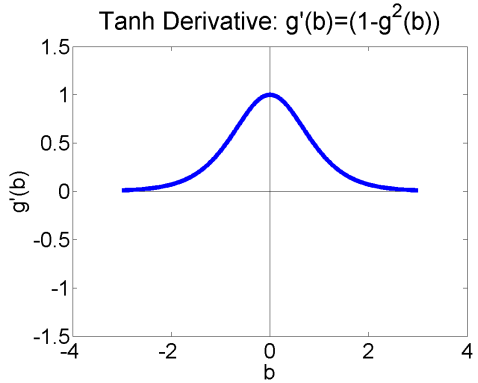
$$\sigma(\beta) = \frac{1}{1+e^{-\beta}}, \qquad \sigma'(\beta) = \sigma(\beta)\big(1-\sigma(\beta)\big)$$

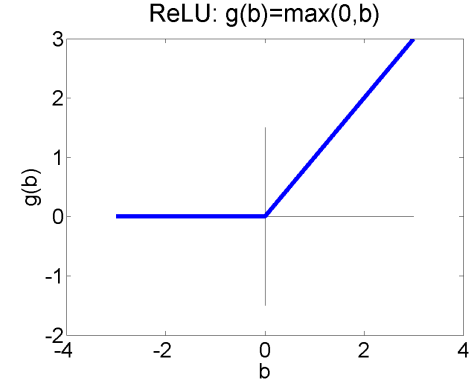




Tanh:

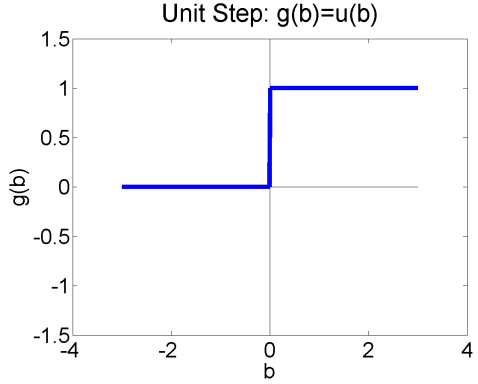
$$\tanh(\beta) = \frac{e^{\beta}-e^{-\beta}}{e^{\beta}+e^{-\beta}}, \qquad \tanh'(\beta) = 1-\tanh^2(\beta)$$





And here's one you haven't seen before, called the rectified linear unit:

$$\mathrm{ReLU}(\beta) = \max(0,\beta), \qquad \mathrm{ReLU}'(\beta) = \mathrm{u}(\beta)$$



Activation functions

The last activation function you need to know is the softmax. Softmax is almost never used anywhere except the output layer. It's basically a sigmoid, but normalized so that $\sum_{c=1}^{V} y_{ci}^{*} = 1$, so that you can interpret y_{ci}^{*} as a probability:

$$\text{softmax}(\beta_j) = \frac{e^{\beta_j}}{\sum_{k=1}^{V} e^{\beta_k}}$$

Because of the normalization, softmax is the only commonly used nonlinearity that depends on excitations other than its own:

$$\frac{\partial \,\text{softmax}(\beta_j)}{\partial \beta_k} = \begin{cases} y_j^{*}(1 - y_j^{*}) & j = k \\ -y_j^{*} y_k^{*} & j \neq k \end{cases}$$

How to make a neural network

- Training Data
- Forward propagation
- Loss Function
- Back propagation

Loss Function

- Now that the neural net has computed $\vec{y}_i^*$ for each of the training tokens $\vec{x}_i$…
- How badly did it do?

Loss Function

The “loss function” is a function that measures how badly the neural network did by comparing its hypothesis output, $\vec{y}_i^*$, to a reference label $\vec{y}_i$:

$$\ell(\vec{y}_i, \vec{y}_i^*) = \text{a “badness” score for the NN output } \vec{y}_i^*$$

The average of $\ell(\vec{y}_i, \vec{y}_i^*)$, over the entire training corpus, tells you how badly the neural net is doing on the whole training corpus.

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n} \ell(\vec{y}_i, \vec{y}_i^*) = \text{“badness” score for the whole training corpus}$$

Loss Function

How do we choose the loss function?

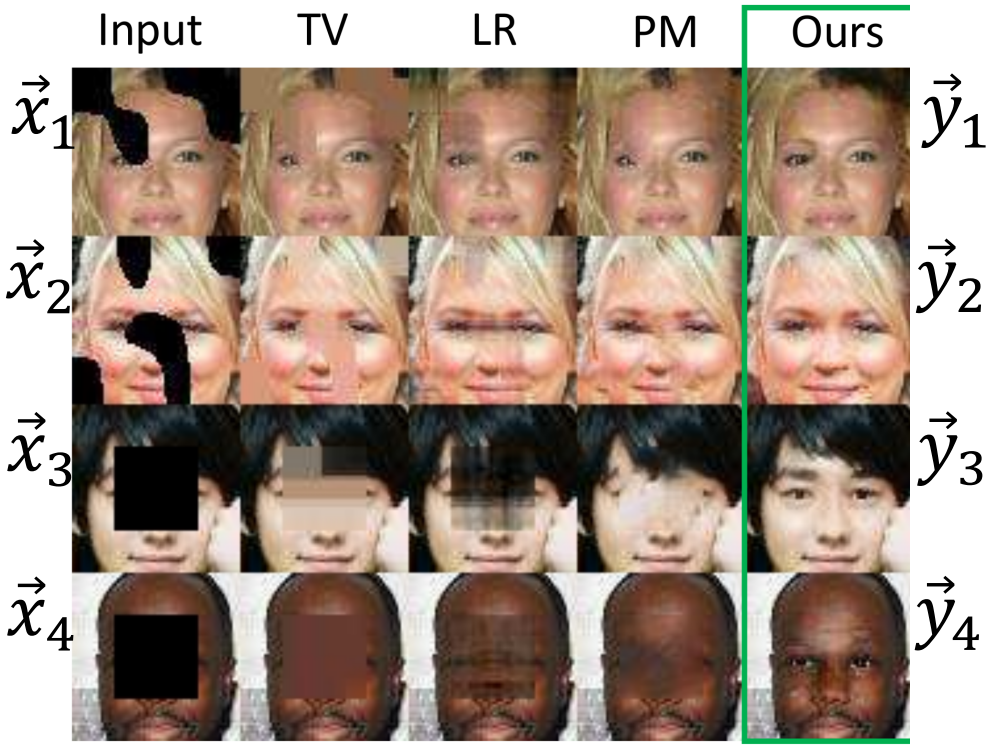
We want it to measure how badly we're doing.

- For an image de-noising task, or something like that: we want it to measure the difference between the target images, and the neural net outputs.
- For a classification task: we want it to measure something like the percentage error rate on the training corpus.

Image de-noising, and other regression tasks

A neural network that computes a real-valued output, $\vec{y}_i^* \approx \vec{y}_i$, from a real-valued input, $\vec{x}_i$, is called a "nonlinear regression."

- For example, $\vec{x}_i$ might be an image with some distortion (a bunch of pixels missing, or something).
- $\vec{y}_i$ is the corresponding clean image.
- Our goal is to compute a cleaned-up image, $\vec{y}_i^* \approx \vec{y}_i$



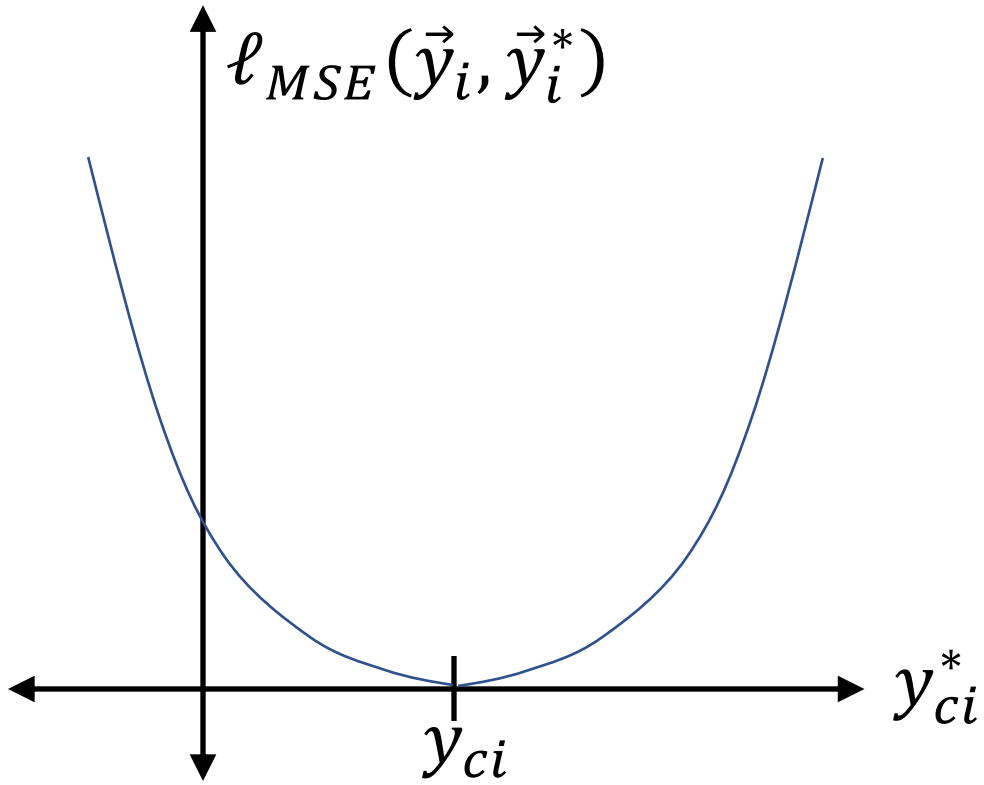
"Semantic Image Inpainting with Deep Generative Models," Raymond Yeh, Chen Chen, Teck Yian Lim, Alexander G. Schwing, Mark Hasegawa-Johnson and Minh Do

Mean-Squared Error

- $\vec{y}_i = [y_{1i}, \ldots, y_{Vi}]$ is a vector of real numbers that the neural net is supposed to produce.
- $\vec{y}_i^* = [y_{1i}^*, \ldots, y_{Vi}^*]$ is what the neural net actually produces.
- Mean Squared Error (MSE) measures how badly you did:

$$\ell_{MSE}(\vec{y}_i, \vec{y}_i^*) = \frac{1}{K}\sum_{c=1}^{V}(y_{ci} - y_{ci}^*)^2$$

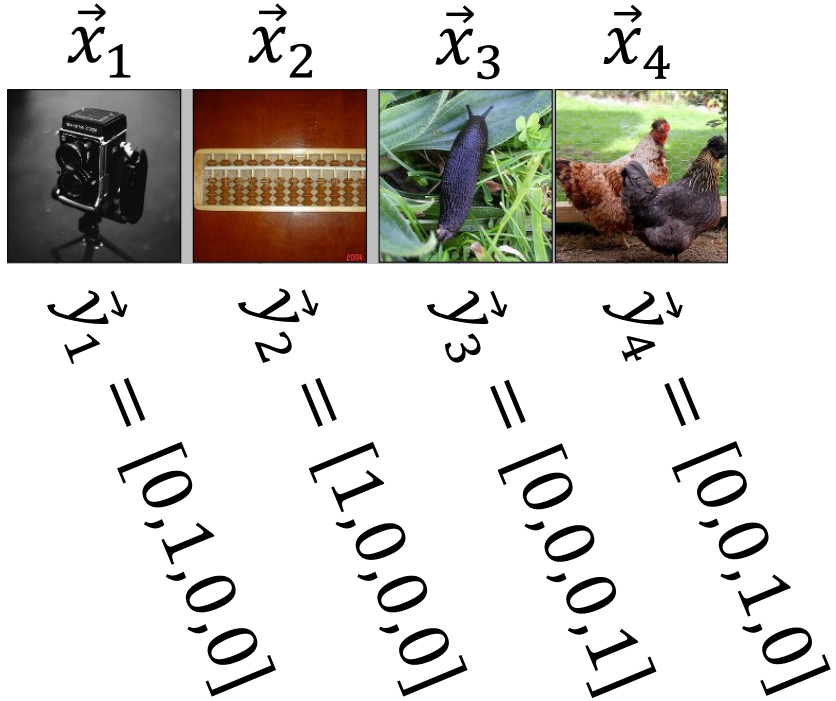
$$\mathcal{L}_{MSE} = \frac{1}{n}\sum_{i=1}^{n}\ell_{MSE}(\vec{y}_i, \vec{y}_i^*)$$



Object recognition, and other multi-class classification tasks

A neural network is called a multi-class classifier if it computes a probability distribution over a set of class labels:

$$0 \leq y^*_{ci} \leq 1, \qquad \sum_{c=1}^{V} y^*_{ci} = 1$$



Usually, we train such a network using a one-hot label vector:

$$y_{ci} = \begin{cases} 1 & c \text{ is correct class for token } i \\ 0 & \text{otherwise} \end{cases}$$

Class Index	Class Name
1	abacus
2	camera
3	chickens
4	slug

Zero-One Loss

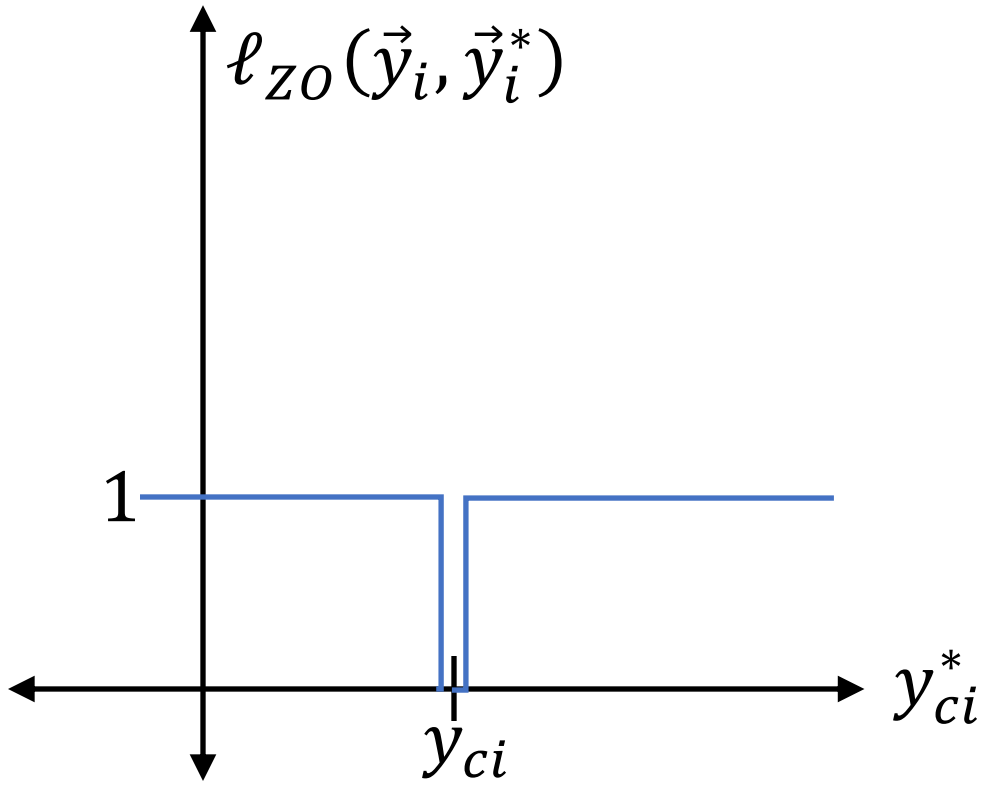
The simplest loss function is called the zero-one loss. It's equal to zero if there is no error, and it's equal to one if there is an error:

$$\ell_{ZO}(\vec{y}_i, \vec{y}_i^*) = \begin{cases} 0 & \vec{y}_i = \vec{y}_i^* \\ 1 & \text{otherwise} \end{cases}$$

The average of the zero-one loss, over the training database, is equal to the training corpus error rate:

$$\mathcal{L}_{ZO} = \frac{1}{n}\sum_{i=1}^{n} \ell_{ZO}(\vec{y}_i, \vec{y}_i^*)$$

The problem with the zero-one loss is that it's not differentiable.



Cross Entropy

$$y_{ci} = \begin{cases} 1 & c \text{ is correct class for token } i \\ 0 & \text{otherwise} \end{cases}$$

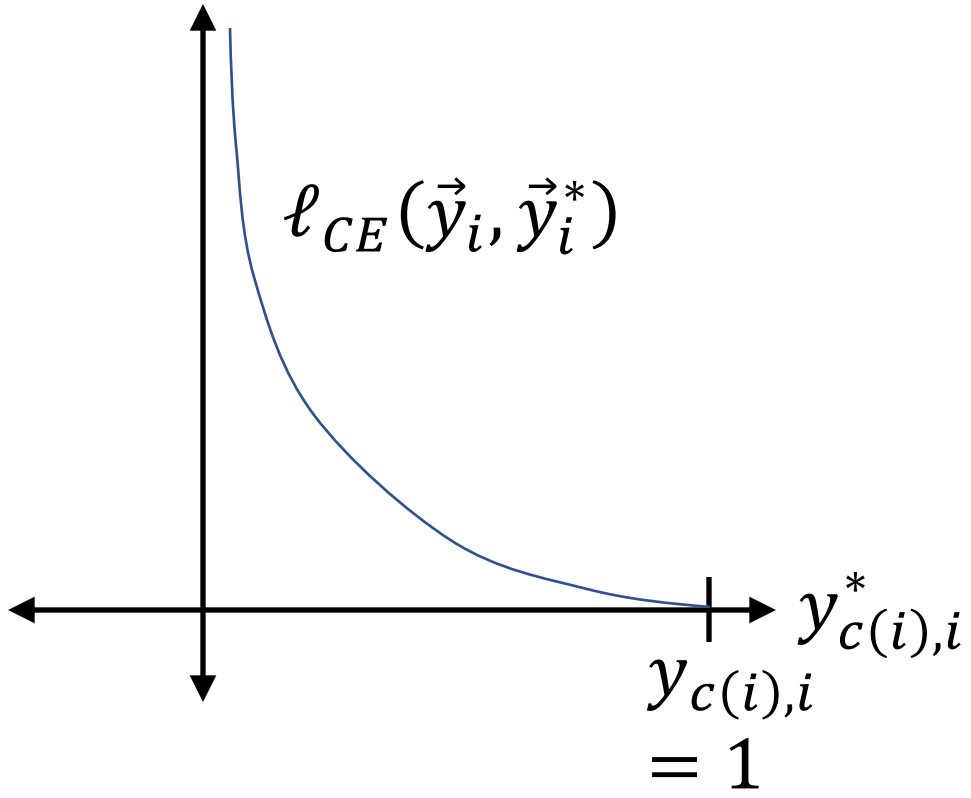
$$y^*_{ic} = \text{Estimated } P(\text{class of token } i \text{ is } c \mid \vec{x}_i)$$

Then: minimizing cross-entropy = maximizing likelihood:

$$\ell_{CE}(\vec{y}_i, \vec{y}^*_i) = -\log \text{Estimated } P(\text{correct class} \mid \vec{x}_i)$$

$$= -\log y^*_{c(i),i} = -\sum_{c=1}^{V} y_{ci} \log y^*_{ci}$$

$$\mathcal{L}_{CE} = \frac{1}{n} \sum_{i=1}^{n} \ell_{CE}(\vec{y}_i, \vec{y}^*_i)$$



Binary object recognition

Sometimes, a neural network computes a scalar output that is between 0 and 1:

$$0 \leq y_i^* \leq 1$$

$y_1 = -1$

I'm warning you, it's pretty pathetic.

$y_2 = +1$

This is a beautiful movie, you'll love it.

Usually, we train such a network using a binary target label:

$$y_i = \begin{cases} 1 & \text{token } i \text{ is of class } +1 \\ 0 & \text{otherwise} \end{cases}$$

…or…

$y_1 = 1$ $y_2 = 0$ $y_3 = 1$ $y_4 = 0$

Learning Multiple Layers of Features from Tiny Images, Alex Krizhevsky, 2009

Binary Cross Entropy

$$y_i = \begin{cases} 1 & \text{token } i \text{ is of class } +1 \\ 0 & \text{otherwise} \end{cases}$$

$$y_i^* = \text{Estimated } P(\text{class of token } i \text{ is } +1 \mid \vec{x}_i)$$

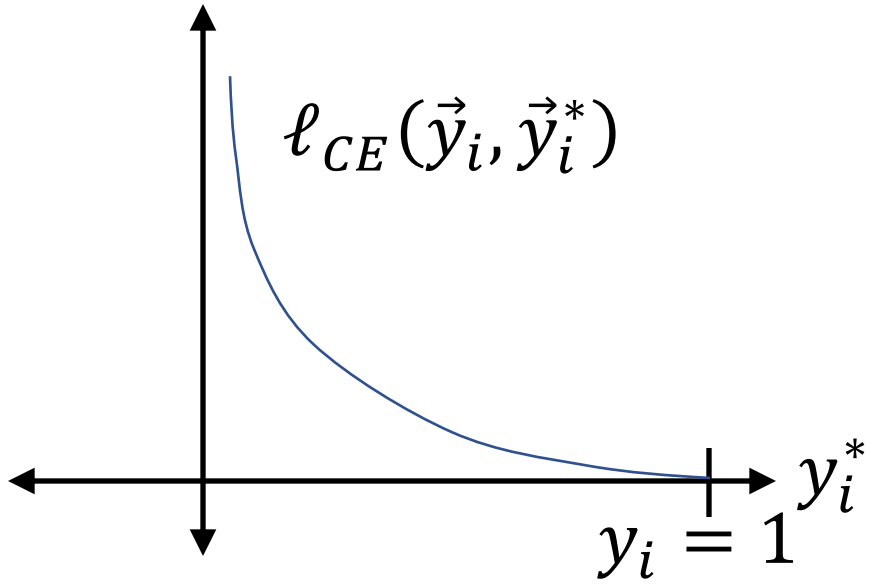
Then: minimizing binary cross-entropy = maximizing likelihood:

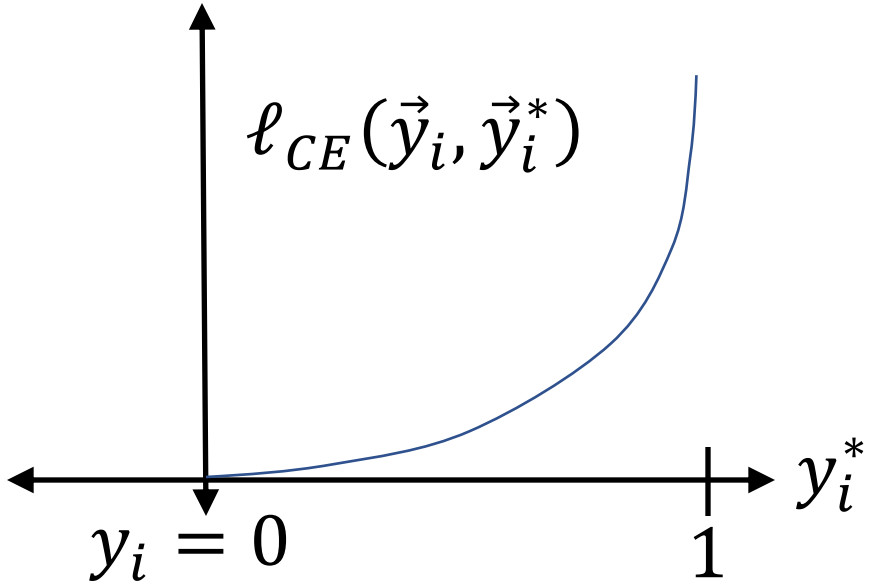
$$\ell_{BCE}(\vec{y}_i, \vec{y}_i^*) = -\log \text{Estimated } P(\text{correct class} \mid \vec{x}_i)$$

$$= -y_i \log y_i^* - (1 - y_i) \log(1 - y_i^*)$$

$$= \begin{cases} -\log y_i^* & c(i) = +1 \\ -\log(1 - y_i^*) & \text{otherwise} \end{cases}$$

$$\mathcal{L}_{BCE} = \frac{1}{n} \sum_{i=1}^{n} \ell_{BCE}(\vec{y}_i, \vec{y}_i^*)$$





Loss Function

How do we choose the loss function?

We want it to measure how badly we're doing.

- For an image de-noising task, or something like that: MSE measures the average squared difference between the target images and the neural net outputs.
- For a classification task:
  - Zero-one loss counts the errors
  - Cross entropy is the negative log probability of the correct answer

How to make a neural network

- Training Data
- Forward propagation
- Loss Function
- Back propagation

Back propagation

- OK, now we know how badly we did.
- How can we adjust the network weights, $w_{kj}^{(l)}$, in order to do better?
- General strategy: we will use some subset of the training data, $\mathcal{D} = \{(\vec{x}_1, \vec{y}_1), \ldots, (\vec{x}_n, \vec{y}_n)\}$, to compute an update step, $dw_{kj}^{(l)}$, simultaneously for all of the network weights. Then we will update them all simultaneously, as

$$w_{kj}^{(l)} \leftarrow w_{kj}^{(l)} - dw_{kj}^{(l)}, \qquad \begin{array}{c} 1 \leq l \leq L \\ 1 \leq k \leq N \\ 1 \leq j \leq N+1 \end{array}$$

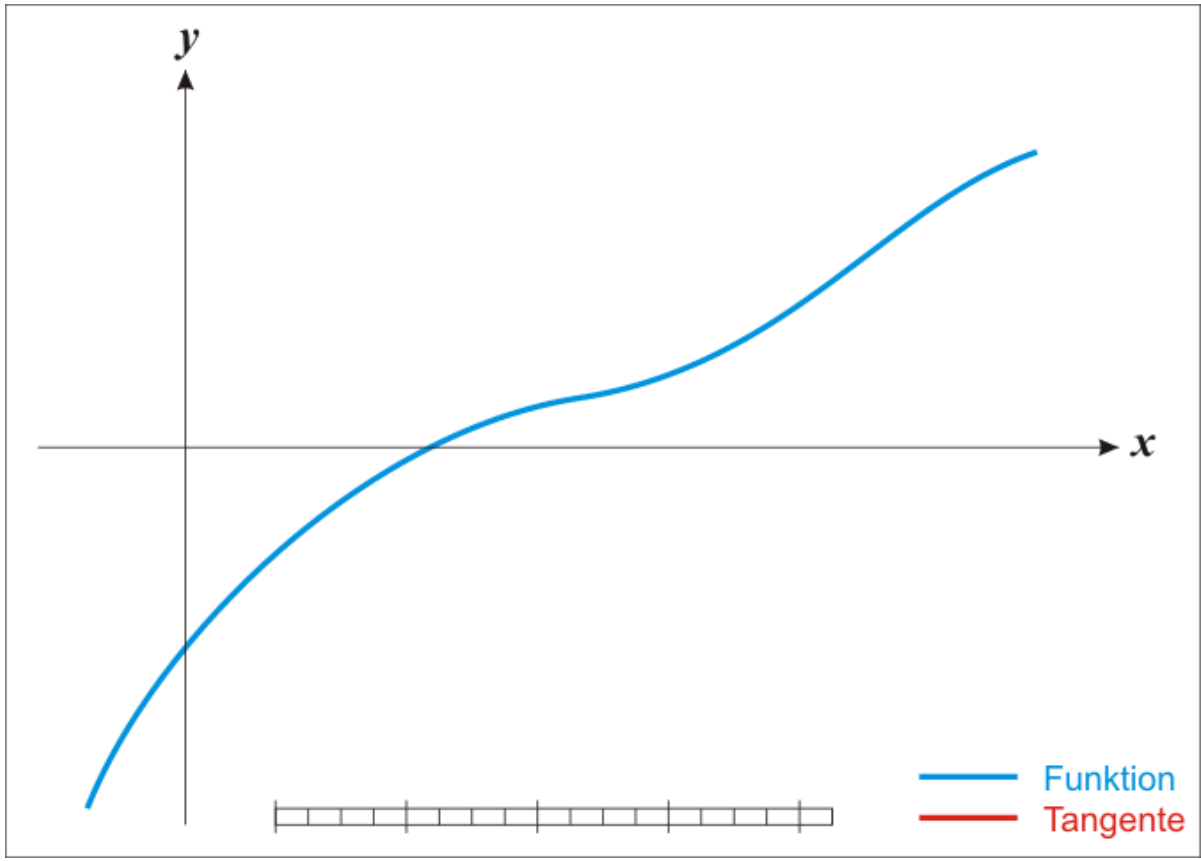
Newton's method

Basically, we use gradient descent, which is similar to Newton's method, but simpler: we only use the first derivative, not the second derivative.

For each $w_{kj}^{(l)}$, we compute $\frac{d\mathcal{L}}{dw_{kj}^{(l)}}$, and then we update all of the weights simultaneously as

$$w_{kj}^{(l)} \leftarrow w_{kj}^{(l)} - \eta \frac{d\mathcal{L}}{dw_{kj}^{(l)}}, \qquad \begin{matrix} 1 \le l \le L \\ 1 \le k \le N \\ 1 \le j \le N+1 \end{matrix}$$

where η is a learning rate.



Animated illustration of Newton's method for finding the zeros of the $f(x) = d\mathcal{L}/dx$.

Finding the gradient

- How do we find $\frac{d\mathcal{L}}{dw_{kj}^{(l)}}$?
- The technique for doing that is called “back propagation.”

Back propagation

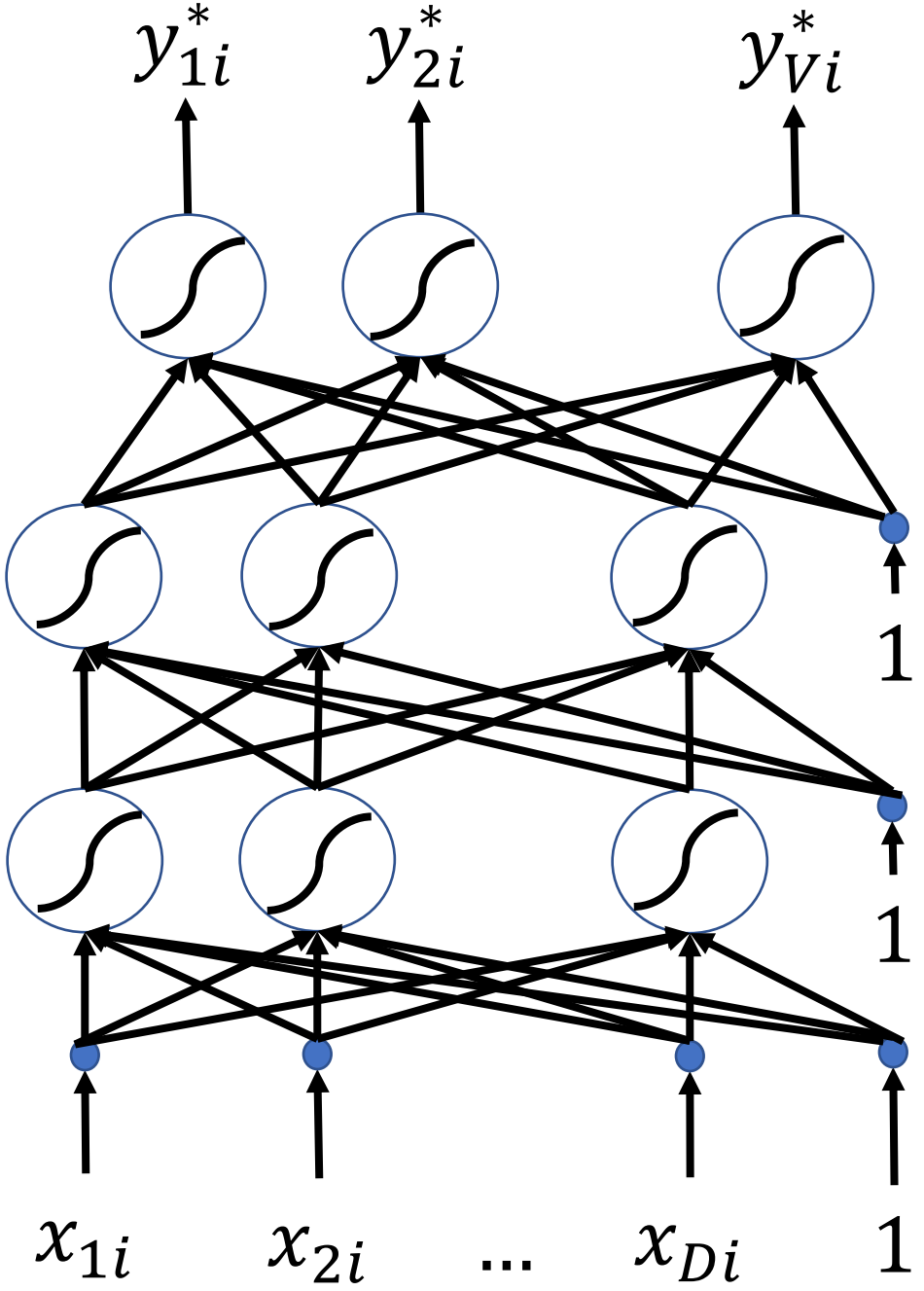
Remember that

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n} \ell(\vec{y}_i, \vec{y}_i^*)$$

so

$$\frac{d\mathcal{L}}{dw_{kj}^{(l)}} = \frac{1}{n}\sum_{i=1}^{n} \frac{d\ell(\vec{y}_i, \vec{y}_i^*)}{dw_{kj}^{(l)}}$$

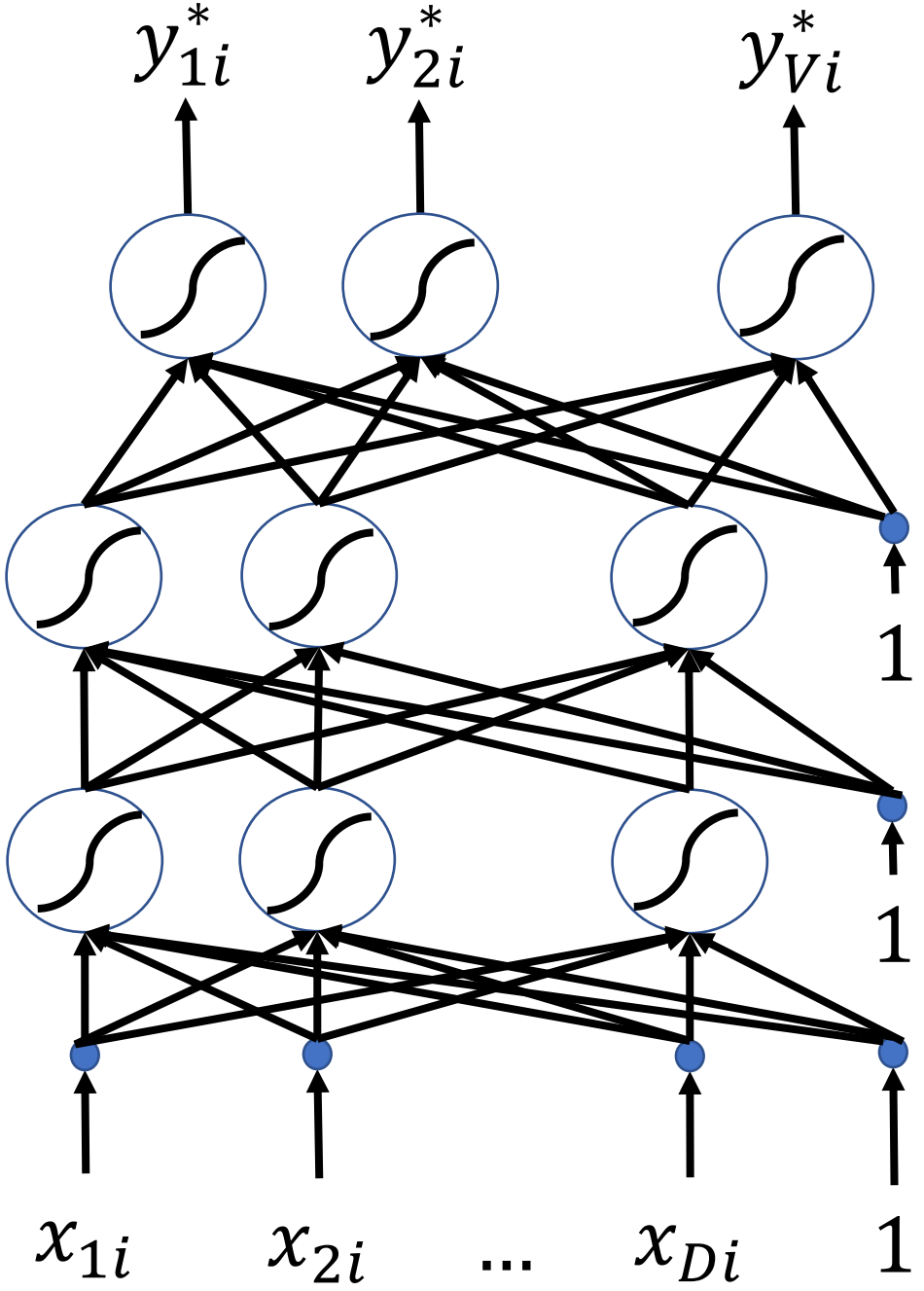
So we need to find $\frac{d\ell(\vec{y}_i, \vec{y}_i^*)}{dw_{kj}^{(l)}}$ for each of the training tokens individually, and then just average them.



Back propagation

Remember that

- $\ell(\vec{y}_i, \vec{y}_i^*)$ depends on each y_{ci}^*, for $1 \leq c \leq V$
- Each y_{ci}^* depends on its corresponding $\beta_{ci}^{(L)}$
- Each $\beta_{ci}^{(L)}$ depends on each $h_{ki}^{(L-1)}$, for $1 \leq k \leq N$
- … and so on backward through the network…



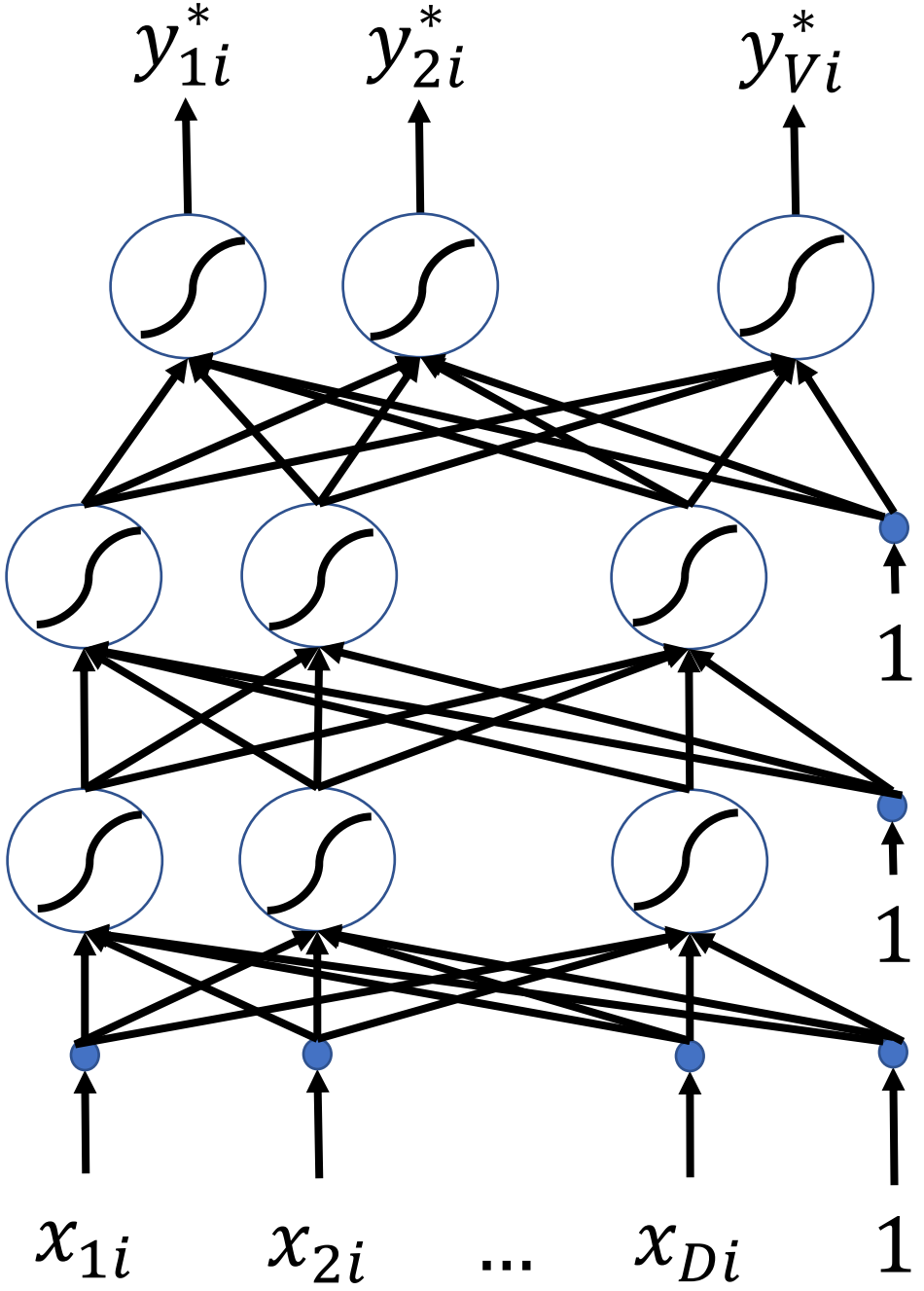
Back propagation

$$\frac{d\ell(\vec{y}_i, \vec{y}_i^*)}{dw_{kj}^{(l)}} = \sum_{c=1}^{V} \frac{\partial \ell(\vec{y}_i, \vec{y}_i^*)}{\partial y_{ci}^*} \frac{dy_{ci}^*}{dw_{kj}^{(l)}}$$

$$\frac{dy_{ci}^*}{dw_{kj}^{(l)}} = \frac{dh_{ci}^{(L)}}{dw_{kj}^{(l)}} = \frac{\partial h_{ci}^{(L)}}{\partial \beta_{ci}^{(L)}} \frac{d\beta_{ci}^{(L)}}{dw_{kj}^{(l)}}$$

$$\frac{d\beta_{ci}^{(L)}}{dw_{kj}^{(l)}} = \sum_{k=1}^{N} \frac{\partial \beta_{ci}^{(L)}}{\partial h_{ki}^{(L-1)}} \frac{dh_{ki}^{(L-1)}}{dw_{kj}^{(l)}}$$

- … and so on backward through the network…

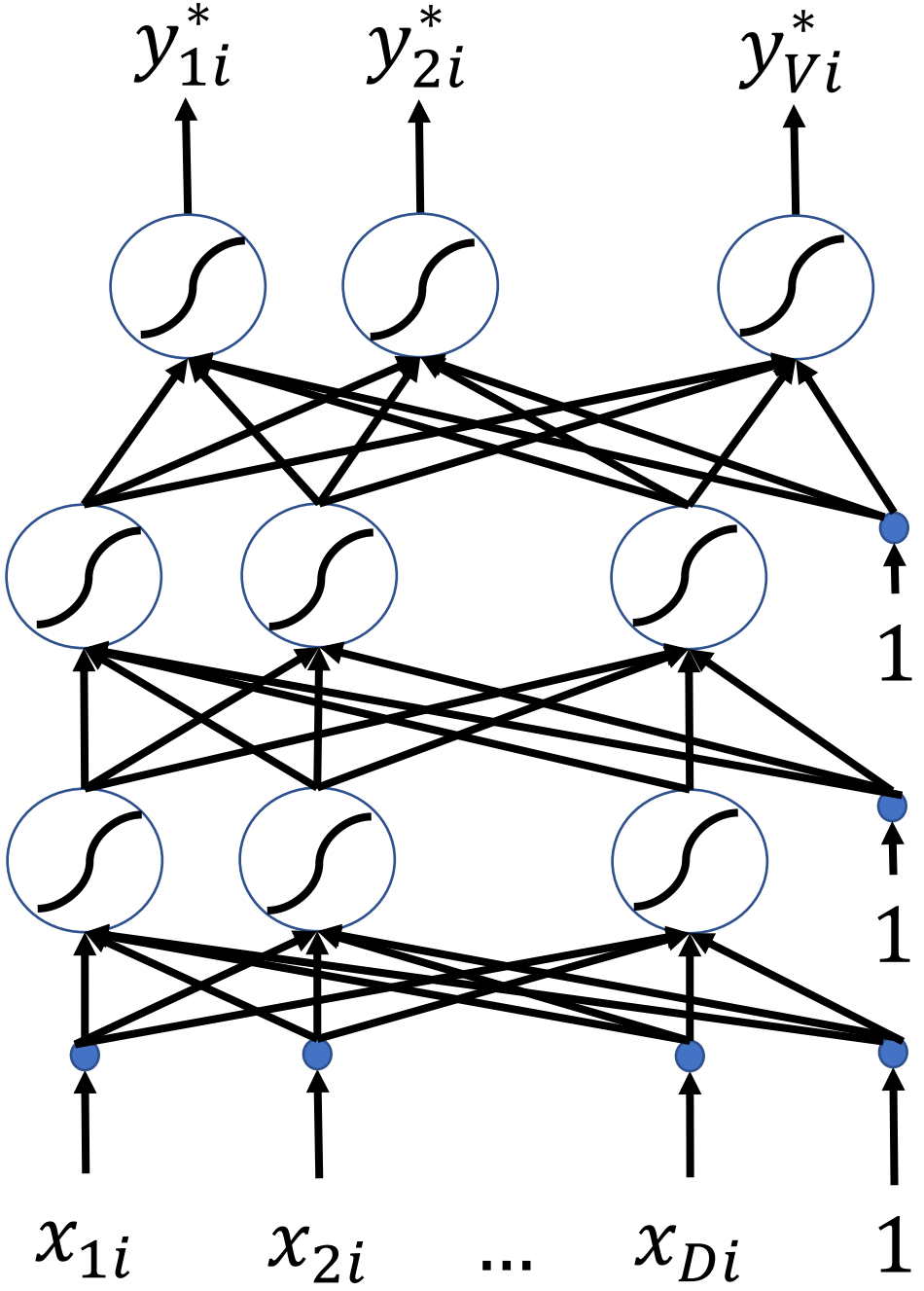


Back propagation

In other words, “back propagation” just means that you apply the chain rule of differentiation, backward through the network, for each training token, in order to compute

$$\frac{d\ell(\vec{y}_i, \vec{y}_i^*)}{dw_{kj}^{(l)}}$$

…and then you just average that quantity over the training tokens.



Back propagation

Back-propagation depends on two main types of derivatives:

- The derivative of an excitation with respect to the preceding layer's activation, $\frac{\partial \beta_{ki}^{(l)}}{\partial h_{ji}^{(l-1)}}$.

- The derivative of an activation with respect to its own excitation, $\frac{\partial h_{ki}^{(l)}}{\partial \beta_{ki}^{(l)}}$

These two quantities are both easy to compute, and interesting. But I'm not going to tell you what they are, because you don't need to know. The reason you don't need to know is that pytorch has lookup tables for each of them, and will compute these quantities for you during MP6.

How to make a neural network

- Training Data: $\mathcal{D} = \{(\vec{x}_1, \vec{y}_1), \ldots, (\vec{x}_n, \vec{y}_n)\}$
- Forward propagation: compute $\vec{y}_i^* \approx \vec{y}_i$ for each training token by alternating these two equations:

$$\beta_{ki}^{(l)} = \sum_{j=1}^{N+1} w_{kj}^{(l)} h_{ji}^{(l-1)}$$

$$h_{ki}^{(l)} = g\left(\beta_{ki}^{(l)}\right)$$

- Loss Function: $\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n} \ell(\vec{y}_i, \vec{y}_i^*)$
- Back propagation: apply the chain rule backward through the network to compute $\frac{d\ell(\vec{y}_i, \vec{y}_i^*)}{dw_{kj}^{(l)}}$, then average over training data to compute $\frac{d\mathcal{L}}{dw_{kj}^{(l)}}$, then update the weights as

$$w_{kj}^{(l)} \leftarrow w_{kj}^{(l)} - \eta \frac{d\mathcal{L}}{dw_{kj}^{(l)}}$$