Deep Feedforward Networks

Lecture slides for Chapter 6 of *Deep Learning*
www.deeplearningbook.org
Ian Goodfellow
Last updated 2016-10-04
Adapted by m.n. for CMPS 392

Multilayer perceptrons (MLP)

- Approximate some function f^*
- A feedforward network defines a mapping

$$\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$$

- No feedback connections
- Functions are composed in a chain $f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$
  - $f^{(1)}$ is the first layer, $f^{(2)}$is the second layer, and so on …
- The training examples specify what the output layer must do
  - The other layers are called hidden layers

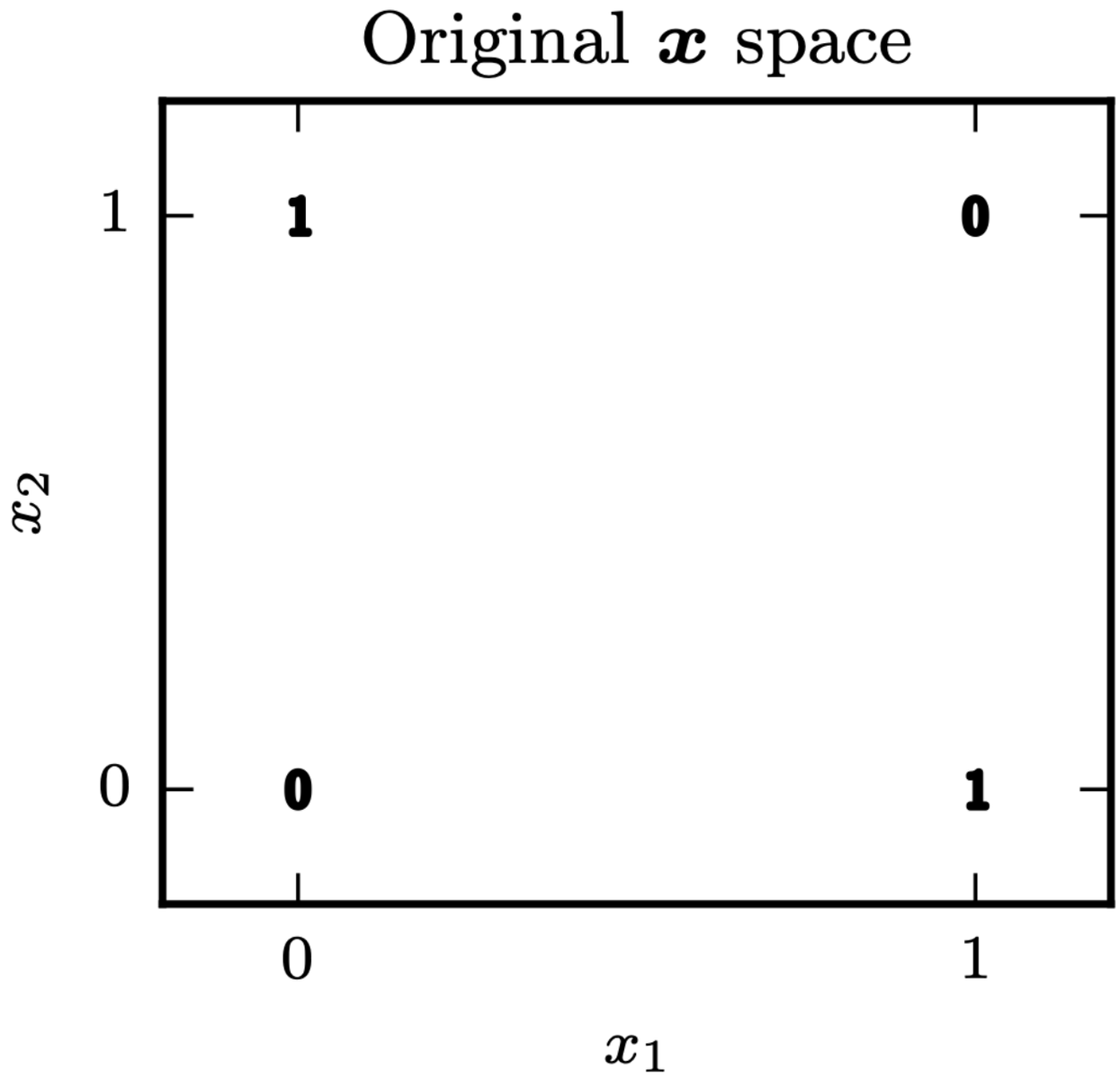
What's the idea?

- The strategy of deep learning is to learn a new represenation of the data $\phi(\boldsymbol{x}, \boldsymbol{\theta})$. Think of this as the output of a hidden layer.
- Parameters w maps the output of hidden layers to the desired output: $\phi(\boldsymbol{x}, \boldsymbol{\theta})^T \boldsymbol{w}$
- We give up on the convexity of the training problem
- $\phi(\boldsymbol{x}, \boldsymbol{\theta})$ can be very generic, in the same time its design can be guided by human experts

Roadmap

- Example: Learning XOR
- Gradient-Based Learning
- Hidden Units
- Architecture Design
- Back-Propagation

XOR is not linearly separable



Linear model?

- Cost function:

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

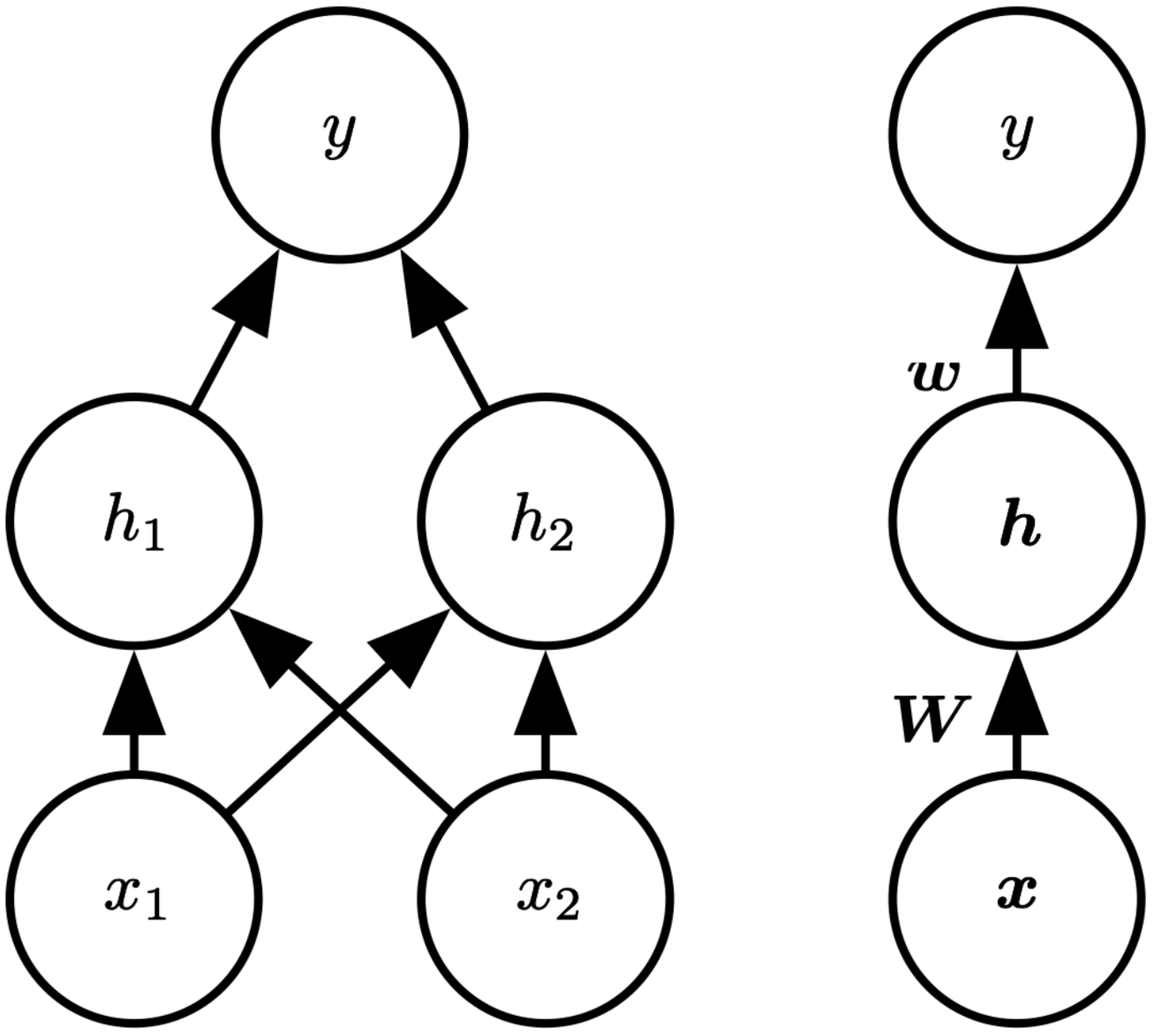
- Linear model

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

- Normal equations ?
  - W = [0,0], b=0.5

Network Diagrams

$$f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{W}^\top \boldsymbol{x}$$



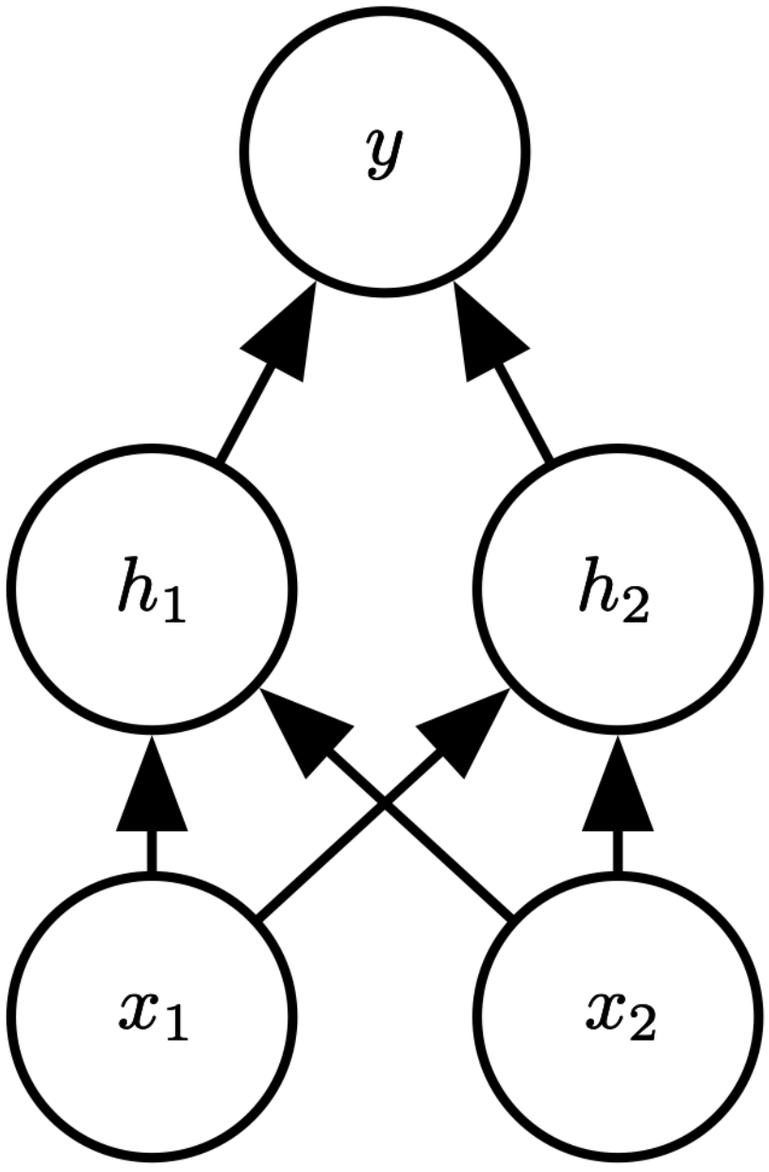
$$f^{(2)}(\boldsymbol{h}) = \boldsymbol{h}^\top \boldsymbol{w}$$

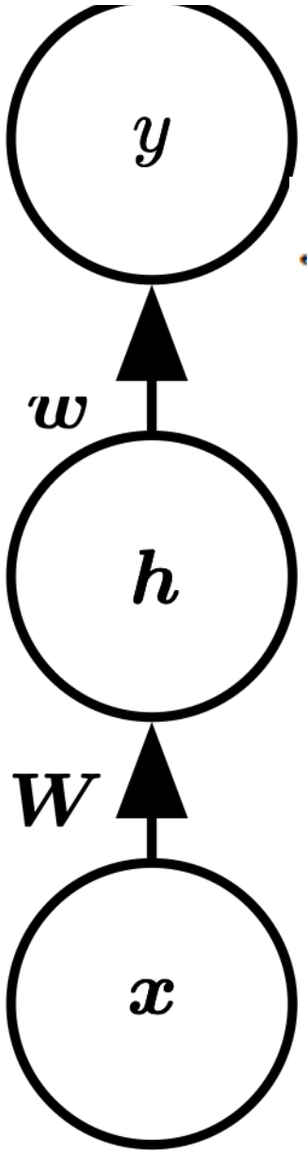
$$f^{(1)}(\boldsymbol{x}) = \boldsymbol{W}^\top \boldsymbol{x}$$

Figure 6.2

Network Diagrams

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





$$f^{(2)}(\boldsymbol{h}) = \boldsymbol{h}^\top \boldsymbol{w}$$

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

$$g(z) = \max\{0, z\}$$

Figure 6.2

Rectified Linear Activation

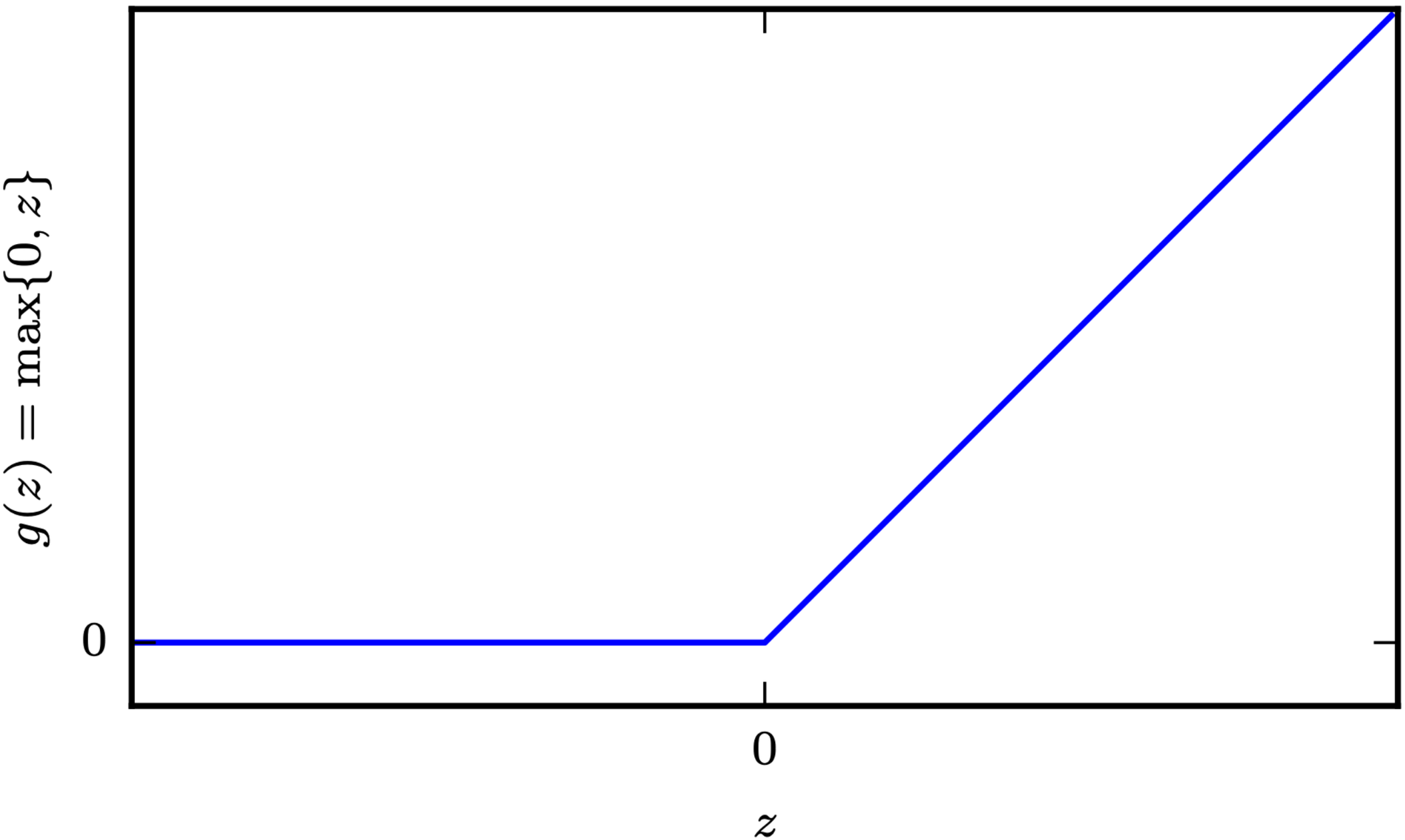


Figure 6.3

Solving XOR

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

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

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

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

Solving XOR

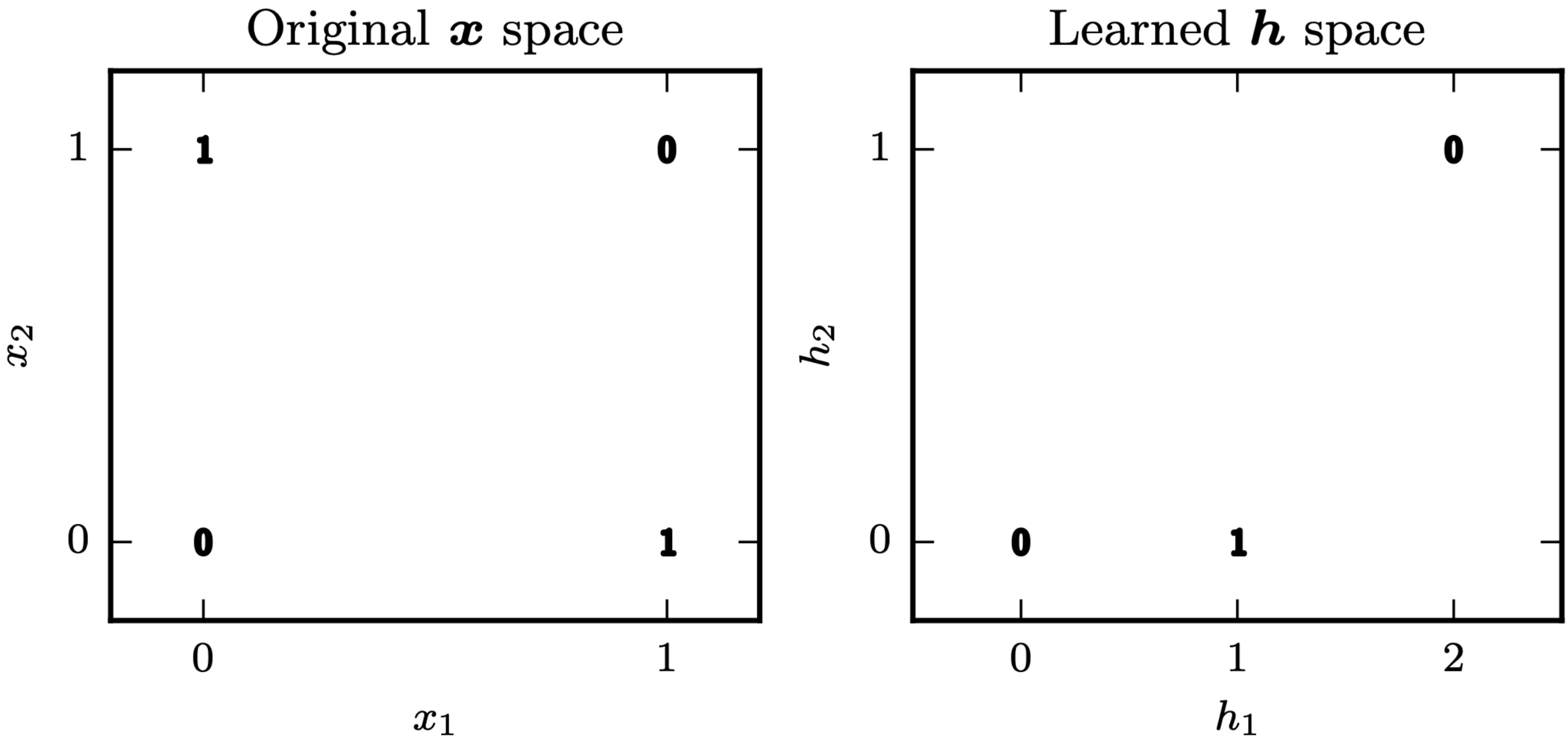


Figure 6.1

Roadmap

- Example: Learning XOR
- Gradient-Based Learning
- Hidden Units
- Architecture Design
- Back-Propagation

Gradient-Based Learning

- Specify
  - Model
  - Cost
- Design model and cost so cost is smooth
- Minimize cost using **gradient descent** or related techniques
  - Rather than linear equations solvers used in *linear regression*
  - Or convex optimization algorithms used in *SVM and logistic regression*

Conditional Distributions and Cross-Entropy

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} L(\boldsymbol{x}, y, \boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta})$$

$$L(\boldsymbol{x}, y, \boldsymbol{\theta}) = -\log p(y \mid \boldsymbol{x}; \boldsymbol{\theta}).$$

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

Stochastic gradient descent

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} L(\boldsymbol{x}, y, \boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta})$$

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta})$$

Computing the gradient is O(m)

Sample a mini-batch $\mathbb{B} = \{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m')}\}$

$$\boldsymbol{g} = \frac{1}{m'}\nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m'} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta})$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \boldsymbol{g}$$

Stochastic gradient descent

- No convergence guarantees
- Sensitive to the values of the initial parameters
- recipe
  - Intialize all weights to small random numbers
  - Biases must be initialized to zero or small positive values

Linear output units

- Suppose the network provides a set of hidden features $\boldsymbol{h} = f(\boldsymbol{x}, \boldsymbol{\theta})$
- Predict a vector of continuous variables $\boldsymbol{y}$
- Linear output unit: $\hat{\boldsymbol{y}} = \boldsymbol{W}^{\boldsymbol{T}}\boldsymbol{h} + \boldsymbol{b}$
- It corresponds to produce the mean of a conditional Gaussian distribution:
  - $p(\boldsymbol{y}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; \hat{\boldsymbol{y}}; \boldsymbol{I})$
  - Minimizing cross entropy is equivalent to minimizing the mean squared error
- Linear units do not saturate

Sigmoid output units

- Predict a binary variable y (binary classification problem)
- $\hat{y} = P(y = 1|\boldsymbol{x})$ based on: $\boldsymbol{h} = f(\boldsymbol{x}, \boldsymbol{\theta})$
- Bad idea: $P(y = 1 \mid \boldsymbol{x}) = \max\left\{0, \min\left\{1, \boldsymbol{w}^\top \boldsymbol{h} + b\right\}\right\}$
- Sigmoid output unit $\hat{y} = \sigma\left(\boldsymbol{w}^\top \boldsymbol{h} + b\right)$
  - $z = \boldsymbol{w}^T \boldsymbol{h} + b$ is called logit
  - $\hat{y} = \sigma(z)$
- The log in the cost function undoes the exp of the sigmoid
  - No gradient saturation

Saturation

- $J(\theta) = -\log \sigma(z)$ if $y = 1$
- $J(\theta) = -\log\big(1 - \sigma(z)\big) = -\log(\sigma(-z))$ if $y = 0$
- One equation: $J(\theta) = -\log \sigma\big((2y - 1)z\big)$

$$= \zeta\big((1 - 2y)z\big)$$

- Saturates only when $(1 - 2y)z$ is very negative
  - When we have converged to the solution
- In the limit of extremely incorrect z , the softplus function does not shrink the gradient at all

Softmax output units

- A generalization of the sigmoid to represent a distribution over n values
  - muli-label classification
  - In fact it is **soft arg-max**, winner-take-all
    - one of the outputs is nearly one and the others are nearly 0.

- $\hat{y}_i = P(y = i|x) = \text{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$
- $\boldsymbol{z} = \boldsymbol{W}^T \boldsymbol{h} + \boldsymbol{b}$ (we can think that z are unnormalized log probabilities given by the network)
- Cost function: $-\log(\text{softmax}(\boldsymbol{z})_i) = -z_i + \log \sum_j \exp(z_j)$

this term cannot saturate, learning can proceed

- We can impose a requirement that one element of z be fixed.
  - But it is simpler to implement the overparametrized version.

Saturation

- If z_i is extremely positive => saturates to 1
- If z_i is extremely negative => saturates to 0

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

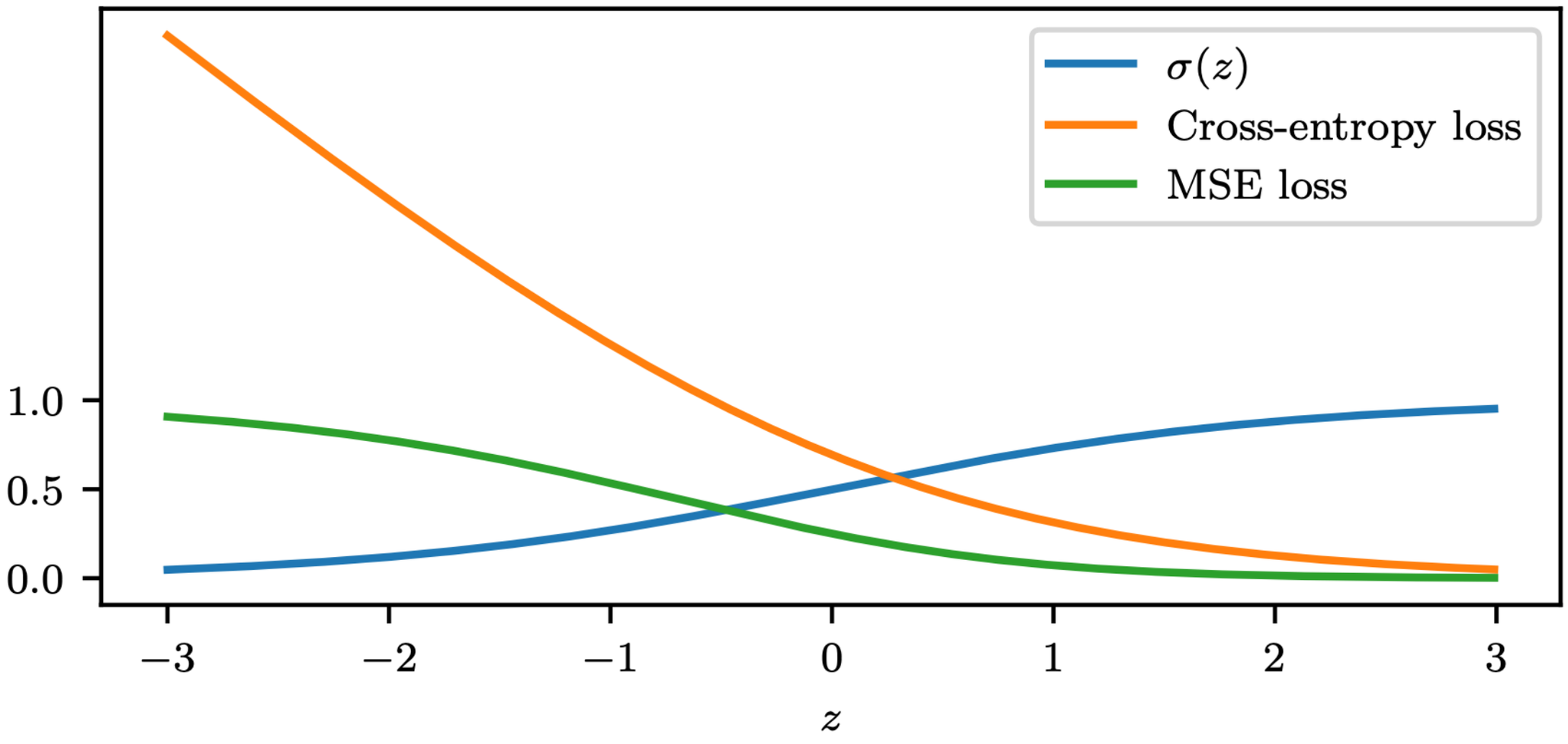
  - An output $\text{softmax}(\boldsymbol{z})_i$ saturates to 1 when the corresponding input is maximal ($z_i = \max_i z_i$) and z_i is much greater than all the other inputs.
  - The output $\text{softmax}(\boldsymbol{z})_i$ can also saturate to 0 when z_i is not maximal and the maximum is much greater.

- MSE loss function will perform poorly because of these gradient vanishing problems

Output Types

Output Type	Output Distribution	Output Layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary cross-entropy
Discrete	Multinoulli	Softmax	Discrete cross-entropy
Continuous		Linear	(MSE)

Don’t mix and match



Other output units

$$\mathcal{N}(x; \mu, \beta^{-1}) = \sqrt{\frac{\beta}{2\pi}} \exp\left(-\frac{1}{2}\beta(x-\mu)^2\right). \tag{3.22}$$

- *Design a neural network to estimate the variance of a gaussian distribution representing* $P(y|x)$

- Loss function:
  - $-\log \mathcal{N}(x; \mu; \beta^{-1}) \propto -\frac{1}{2}\log\beta + \frac{1}{2}\beta(x-\mu)^2$

- Check notebook

Roadmap

- Example: Learning XOR
- Gradient-Based Learning
- Hidden Units
- Architecture Design
- Back-Propagation

Hidden units

- Most hidden units can be described as:
  - accepting a vector of inputs $\boldsymbol{x}$,
  - computing an affine transformation $\boldsymbol{z} = \boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}$,
  - And applying an element-wise nonlinear function $g(\boldsymbol{z})$.

- Use ReLUs, 90% of the time
- For some research projects, get creative
- Many hidden units perform comparably to ReLUs. New hidden units that perform comparably are rarely interesting.

ReLu is not differentiable at 0!

- 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.
  - 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.

When ReLu is active

- When the input is positive, we say that the rectifier is active:
  - the gradient is 1
  - The second derivative is 0, no second-order effects.
- When initializing the parameters of $\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}$,
  - set all elements of $\boldsymbol{b}$ to a small, positive value, such as 0.1
  - This 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.

ReLu generalizations

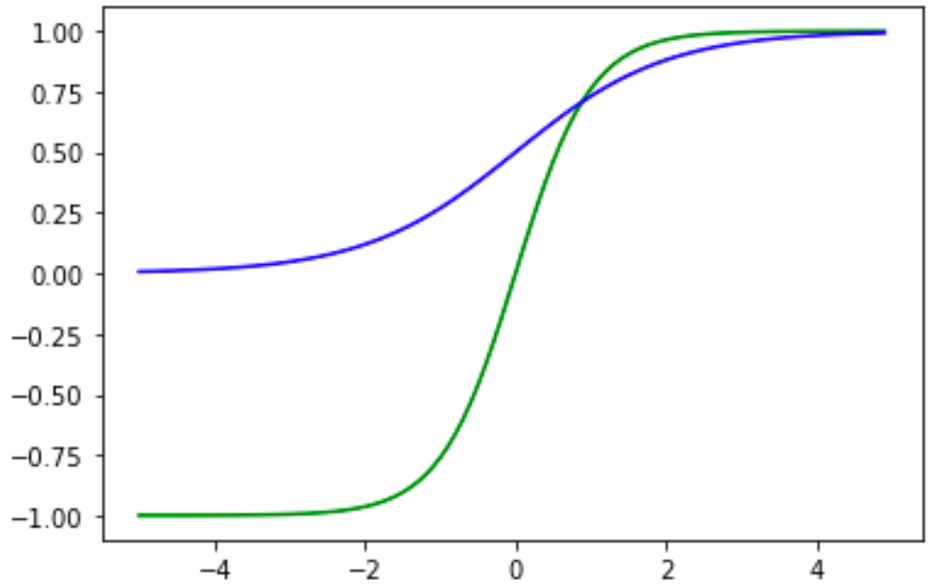
$$h_i = g(\boldsymbol{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i)$$

- Absolute value rectification: $\alpha_i = -1$
- Leaky ReLu: $\alpha_i = 0.01$
- Parametric ReLu: α_i is learnable

- Rectified linear units and its generalizations are based on the principle that models are easier to optimize if their behavior is closer to linear.

Other hidden units

- Sigmoid
  - $g(z) = \sigma(z)$
  - They can saturate
    - which makes learning difficult
- Hyperbolic tangent
  - $g(z) = \tanh(z)$.
  - Related to sigmoid: $\tanh(z) = 2\sigma(2z) - 1$.
  - It typically performs better
    - since it resembles the identity function
    - So long as the activations of the network can be kept small.



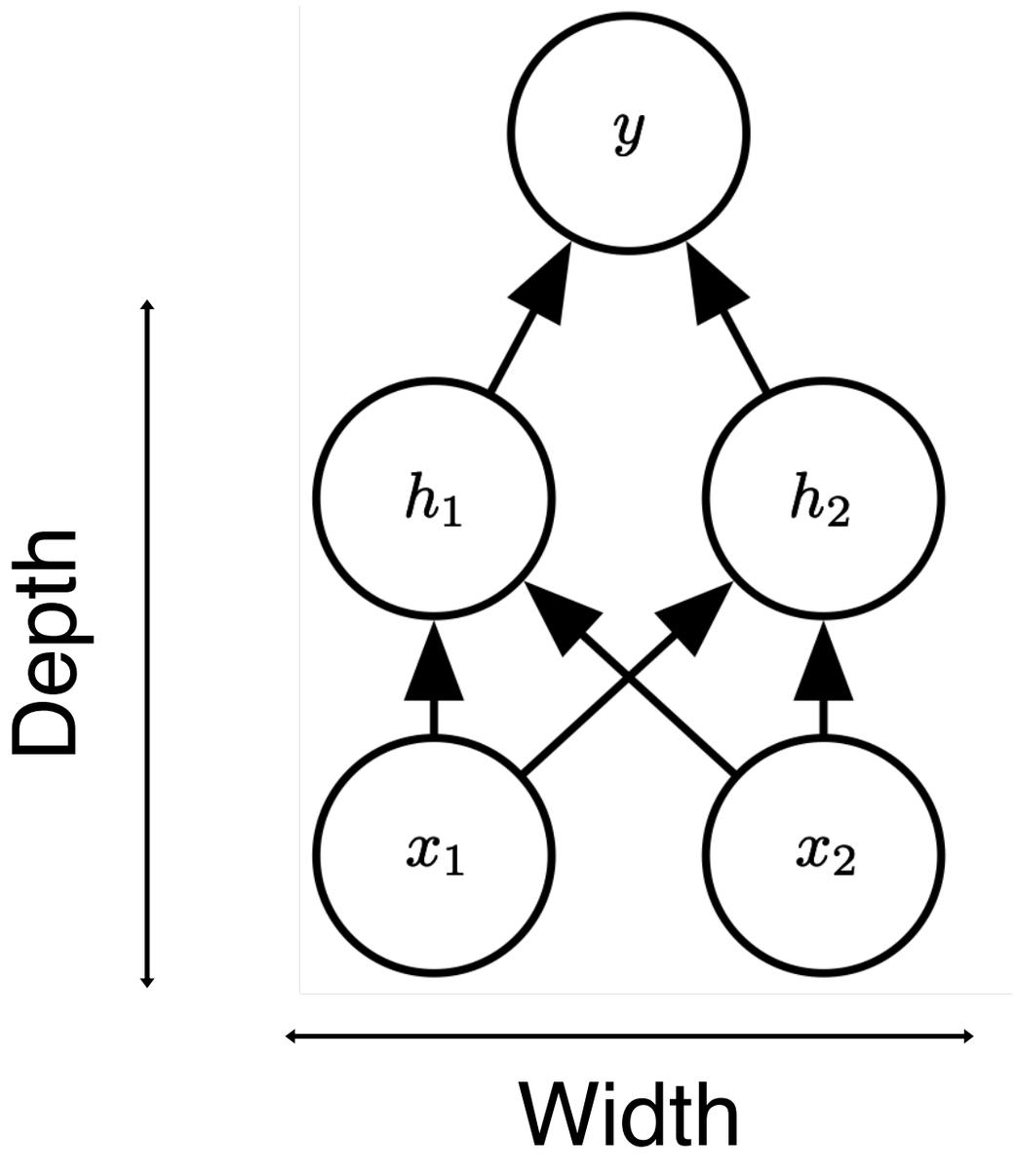
Other hidden units

- The authors tested a feedforward network using $\boldsymbol{h} = \cos(\boldsymbol{W}^{\boldsymbol{T}} \boldsymbol{x} + \boldsymbol{b})$ on the MNIST dataset and obtained an error rate of less than 1%
- Linear hidden units offer an effective way of reducing the number of parameters in a network
  - Assume $\boldsymbol{W}^{\boldsymbol{T}}$ is p x n
  - $\boldsymbol{W}^{\boldsymbol{T}} = \mathbf{V}^{\mathbf{T}}\mathbf{U}^{\mathbf{T}}$ where $\mathbf{V}^{\mathbf{T}}$ is pxq and $\mathbf{U}^{\mathbf{T}}$ is qxn
  - In total we have qx(n+p) trainable params
    - Much less than << pxn for small q

Roadmap

- Example: Learning XOR
- Gradient-Based Learning
- Hidden Units
- Architecture Design
- Back-Propagation

Architecture Basics



- $h^{(1)} = g^{(1)}(W^{(1)T}x + b^{(1)})$
- $h^{(2)} = g^{(2)}(W^{(2)T}h^{(1)} + b^{(2)})$
- …

Universal Approximator Theorem

- One hidden layer is enough to *represent* (not *learn*) an approximation of any function to an arbitrary degree of accuracy
- So why deeper?
  - Shallow nets may need (exponentially) more width
  - Shallow nets may overfit more

Bad idea: One hidden layer

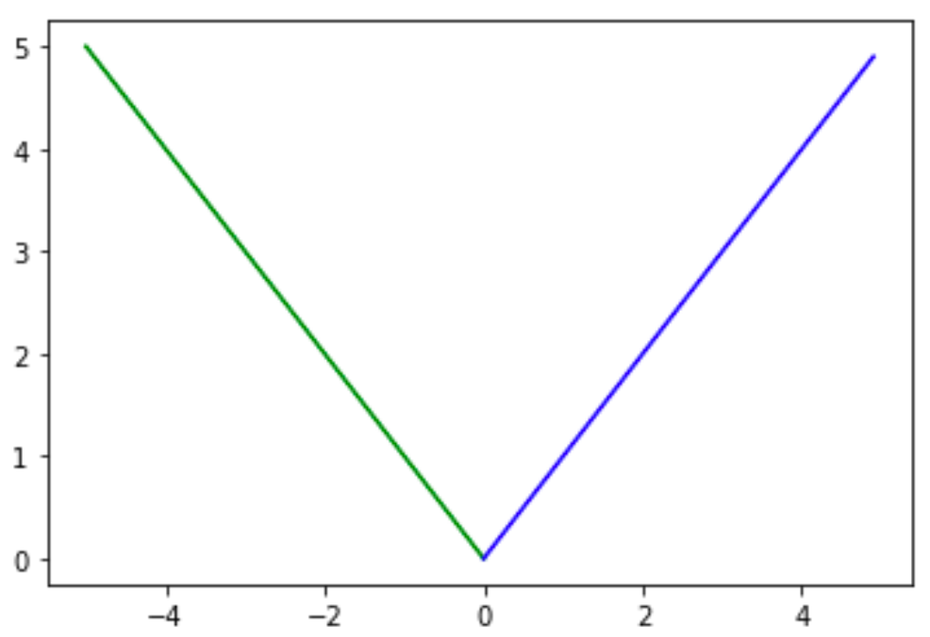
- Consider binary functions over $v \in \{0,1\}^n$
  - The image of any member of the possible 2^n inputs can be 0 or 1
    - In total, we have 2^{2^n} possible functions
  - Selecting one such function requires $O(2^n)$ degrees of freedom
- Working with a single layer is sufficient to represent 'any' function
  - Yet the width of the layer can be exponential, and prone to overfitting

Montufar et. al. (2014)

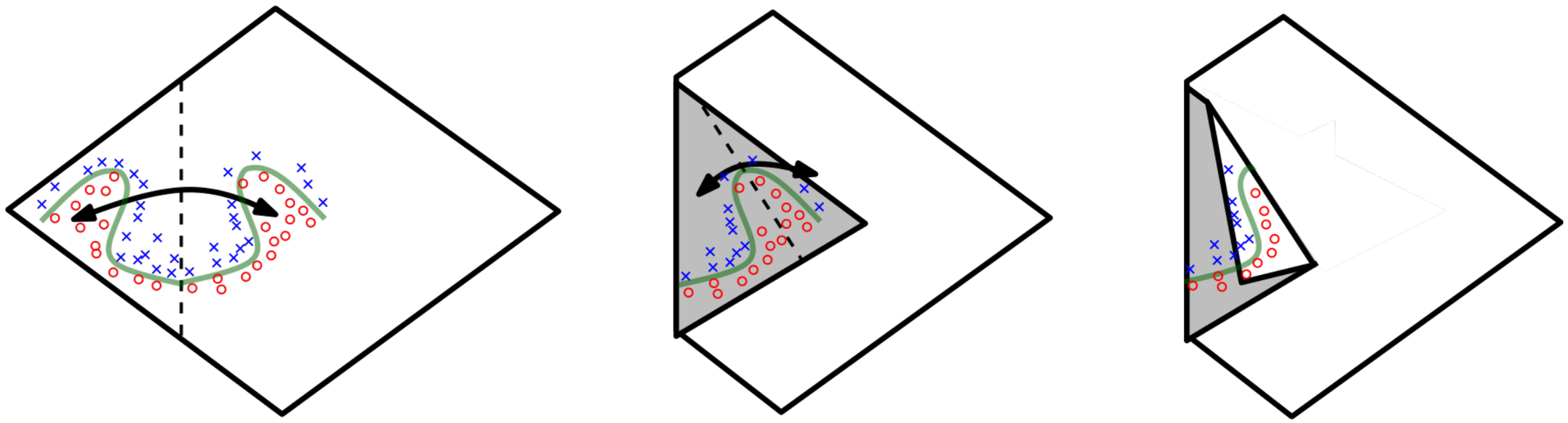
- A deep rectifier network can represent functions with a number of regions that is exponential in the depth of the network.

Example: Absolute value rectification

- Has the same output for every pair of mirror points
- The mirror axis of symmetry is given by $Wh + b$
- A function computed on top of that unit is simpler
- If we fold again, we get an even simpler function, and so on ..



Exponential Representation Advantage of Depth

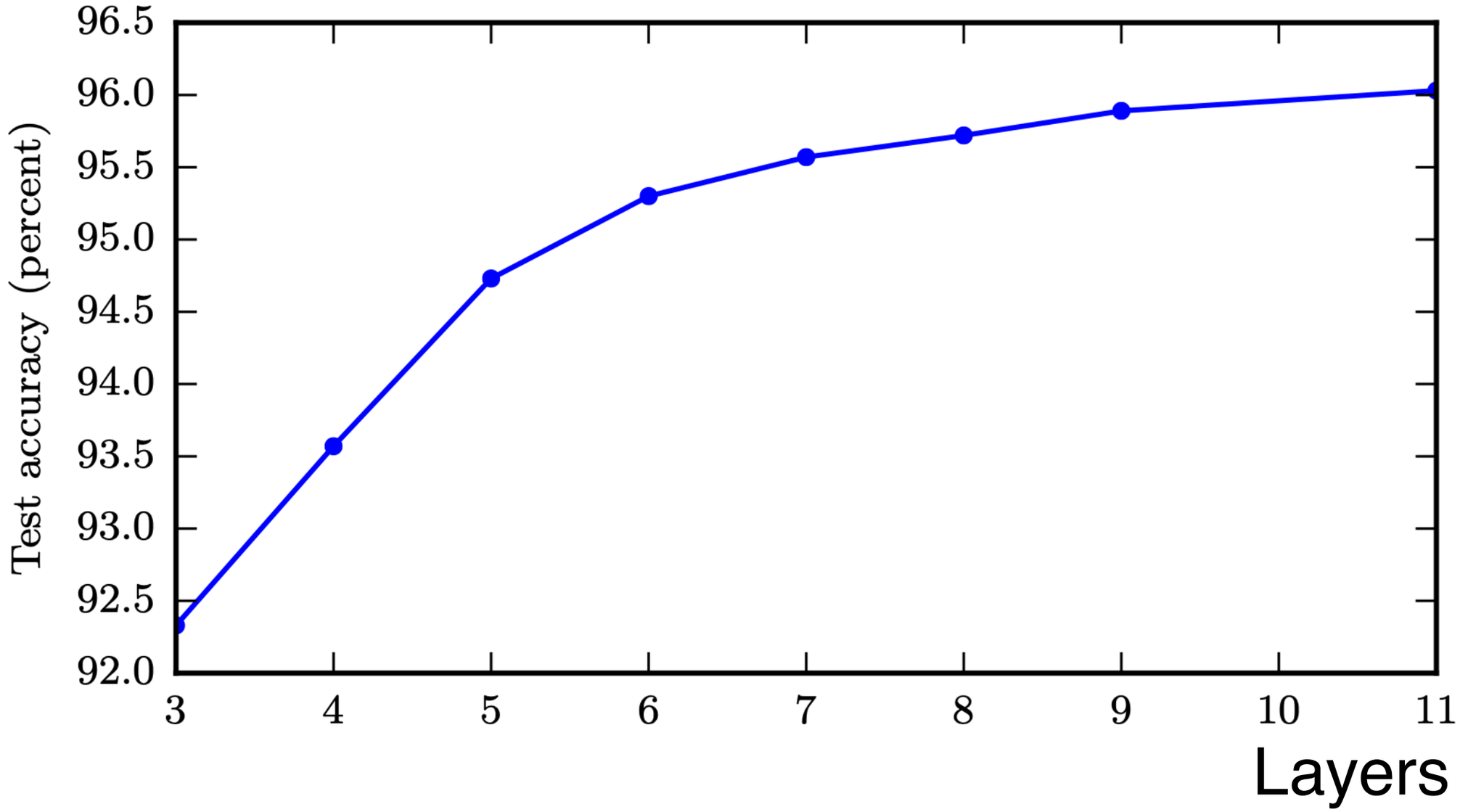


We obtain an exponentially large number of piecewise linear regions which can capture all kinds of repeating patterns.

Why deep?

- Choosing a deep model encodes a very general belief (or prior) 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, where each step makes use of the previous step's output
- Empirically, greater depth does seem to result in better generalization for a wide variety of tasks

Better Generalization with Greater Depth



Task: transcribe multi-digit numbers from photographs of addresses

Large, Shallow Models Overfit More

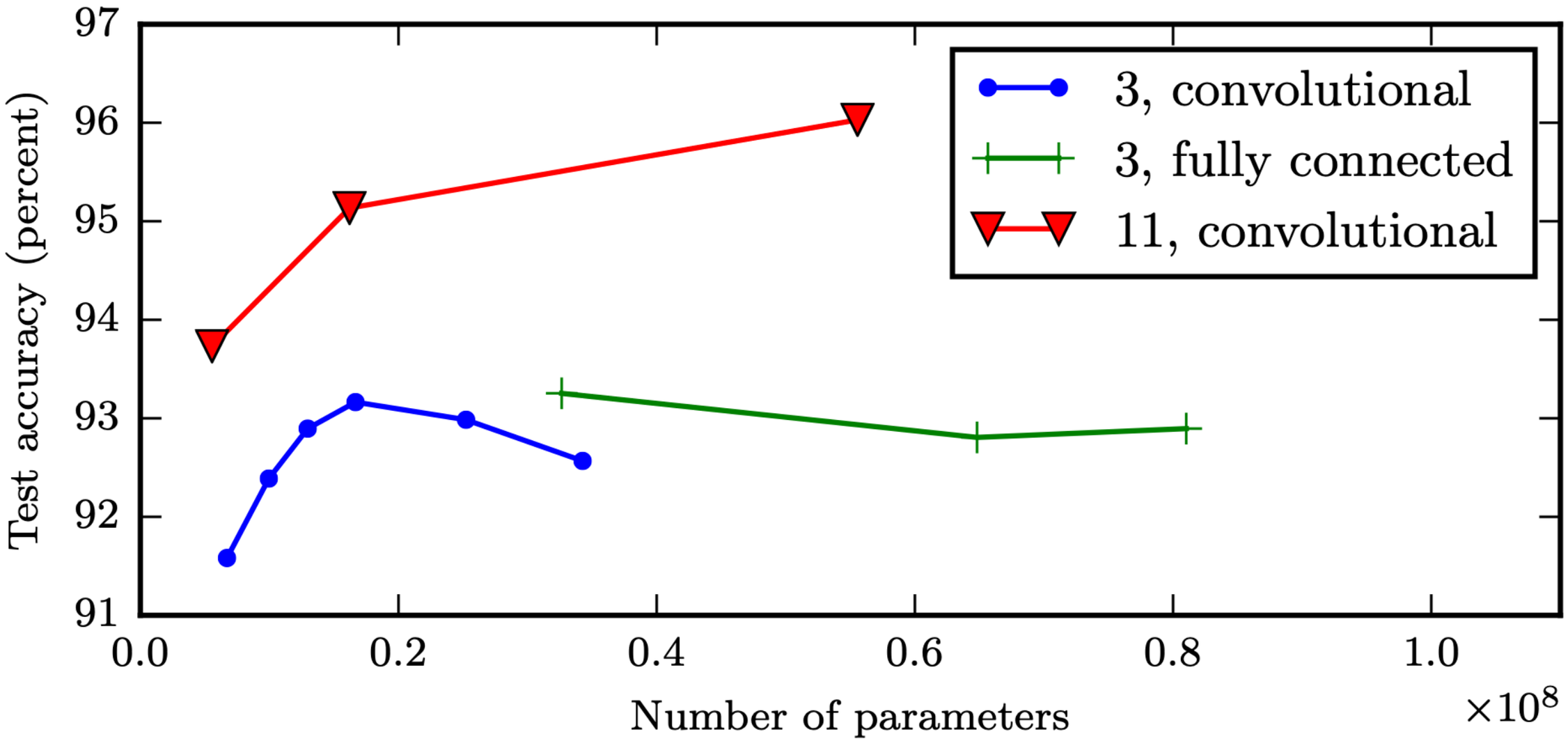


Figure 6.7

Other architectural considerations

- Other architectures: CNN, RNN
- Skip connections: going from layer i to layer i + 2 or higher.
  - make it easier for the gradient to flow from output layers to layers nearer the input.

Design some neural networks for fun:
https://playground.tensorflow.org/

Roadmap

- Example: Learning XOR
- Gradient-Based Learning
- Hidden Units
- Architecture Design
- Back-Propagation

Compute the gradient

- Forward propagation: $x \rightarrow \hat{y} \rightarrow \mathrm{J}(\theta)$
- Backprop: $\frac{\partial J(\theta)}{\partial \theta} =?$
  - Computing derivatives by propagating information backward through the network
  - Simple and inexpensive
  - Based on the chain rule:
    - $\mathbf{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{c}$
    - $\boldsymbol{h} = g(\boldsymbol{z})$
    - $\hat{y} = \boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{h} + b$
    - $J(\hat{y}) = (y - \hat{y})^2$

$$\theta = \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b$$

$$\nabla_\theta J =?$$

$$\frac{\partial J}{\partial c_1} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h_1} \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial c_1}$$

$$\frac{\partial J}{\partial W_{11}} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h_1} \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial W_{11}}$$

Chain Rule

- More generally
  - $\boldsymbol{x} \in \mathbb{R}^n, \boldsymbol{y} \in \mathbb{R}^m, z \in \mathbb{R}$
  - $z = f(\boldsymbol{y}) = f\big(g(\boldsymbol{x})\big)$

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

  - Vector notation:
    - $\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^T \nabla_{\boldsymbol{y}} z$
    - Or: $\nabla_{\boldsymbol{x}} z = \sum_j \frac{\partial z}{\partial y_j} \nabla_{\boldsymbol{x}} y_j$

$n \times m$ Jacobian matrix

- Backprop consists of performing Jacobian-gradient product for each operation in the graph.

Back-Propagation

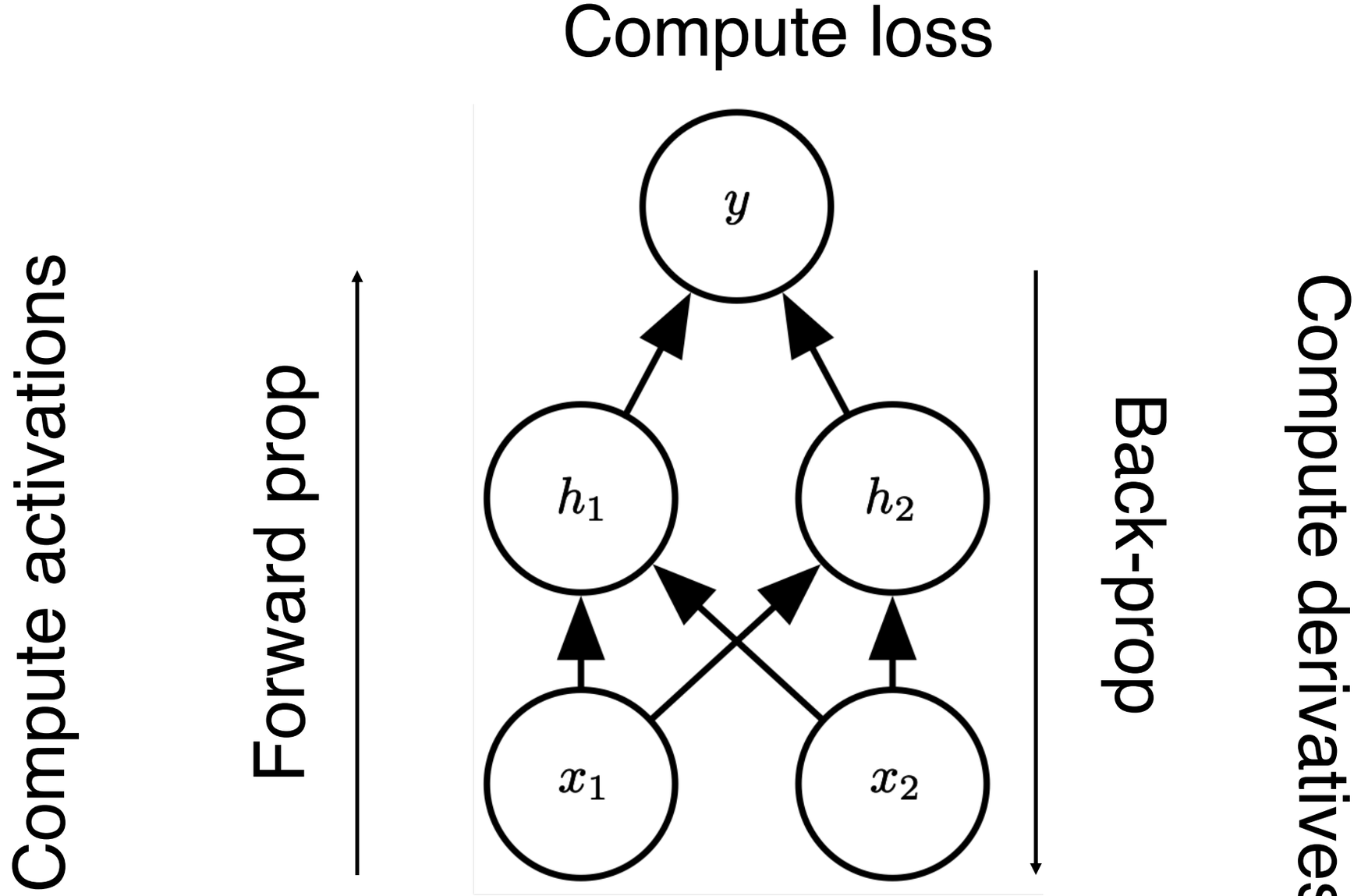
- Back-propagation is "just the chain rule" of calculus

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}. \tag{6.44}$$

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{\top} \nabla_{\boldsymbol{y}} z, \tag{6.46}$$

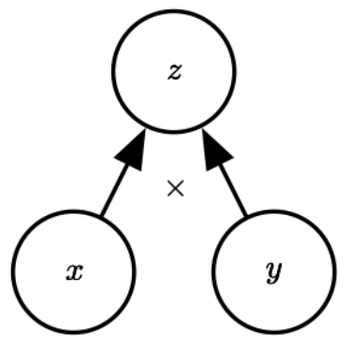
- But it's a particular implementation of the chain rule
  - Uses dynamic programming (table filling)
  - Avoids recomputing repeated subexpressions
  - Speed vs memory tradeoff

Simple Back-Prop Example

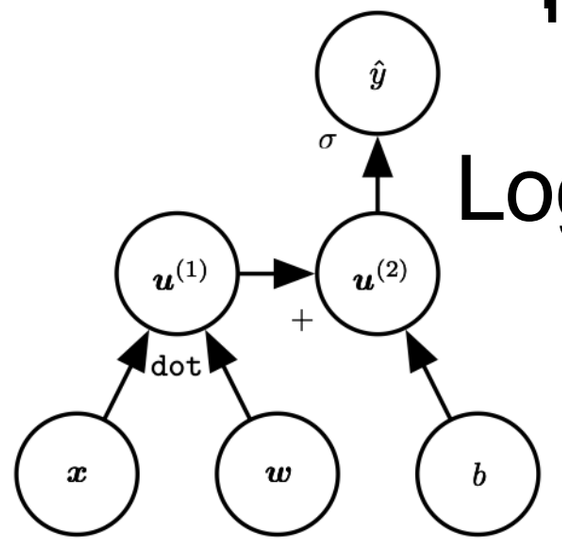


General implementation
Computation Graphs

Multiplication



(a)



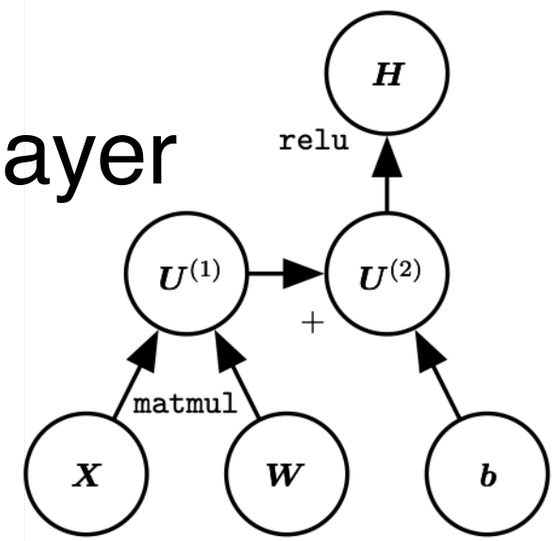
(b)

Logistic regression

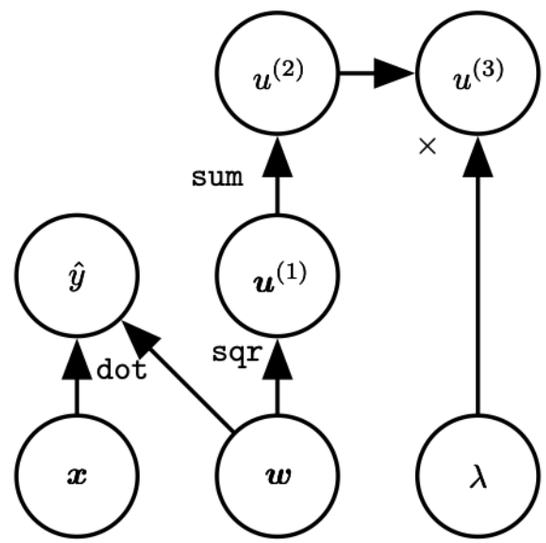
$$\hat{y} = x^T w$$

$$u^3 = \lambda \sum w_i^2$$

ReLU layer



(c)



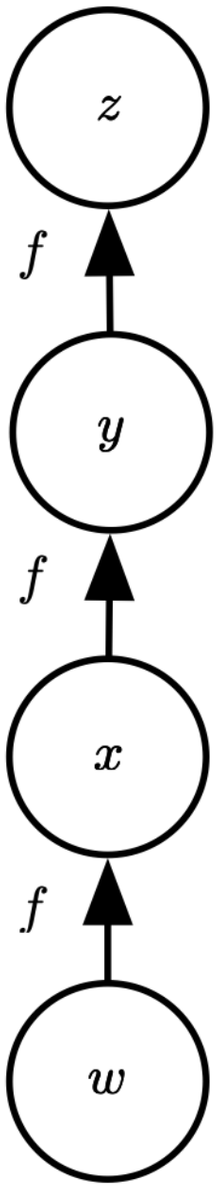
(d)

Linear regression and weight decay

Mini-batch

Figure 6.8

Repeated Subexpressions



$$\frac{\partial z}{\partial w} \tag{6.50}$$

$$= \frac{\partial z}{\partial y} \frac{\partial y}{\partial x} \frac{\partial x}{\partial w} \tag{6.51}$$

$$= f'(y) f'(x) f'(w) \tag{6.52}$$

$$= f'(f(f(w))) f'(f(w)) f'(w) \tag{6.53}$$

Back-prop avoids computing this twice

Figure 6.9

Symbol-to-Symbol Differentiation

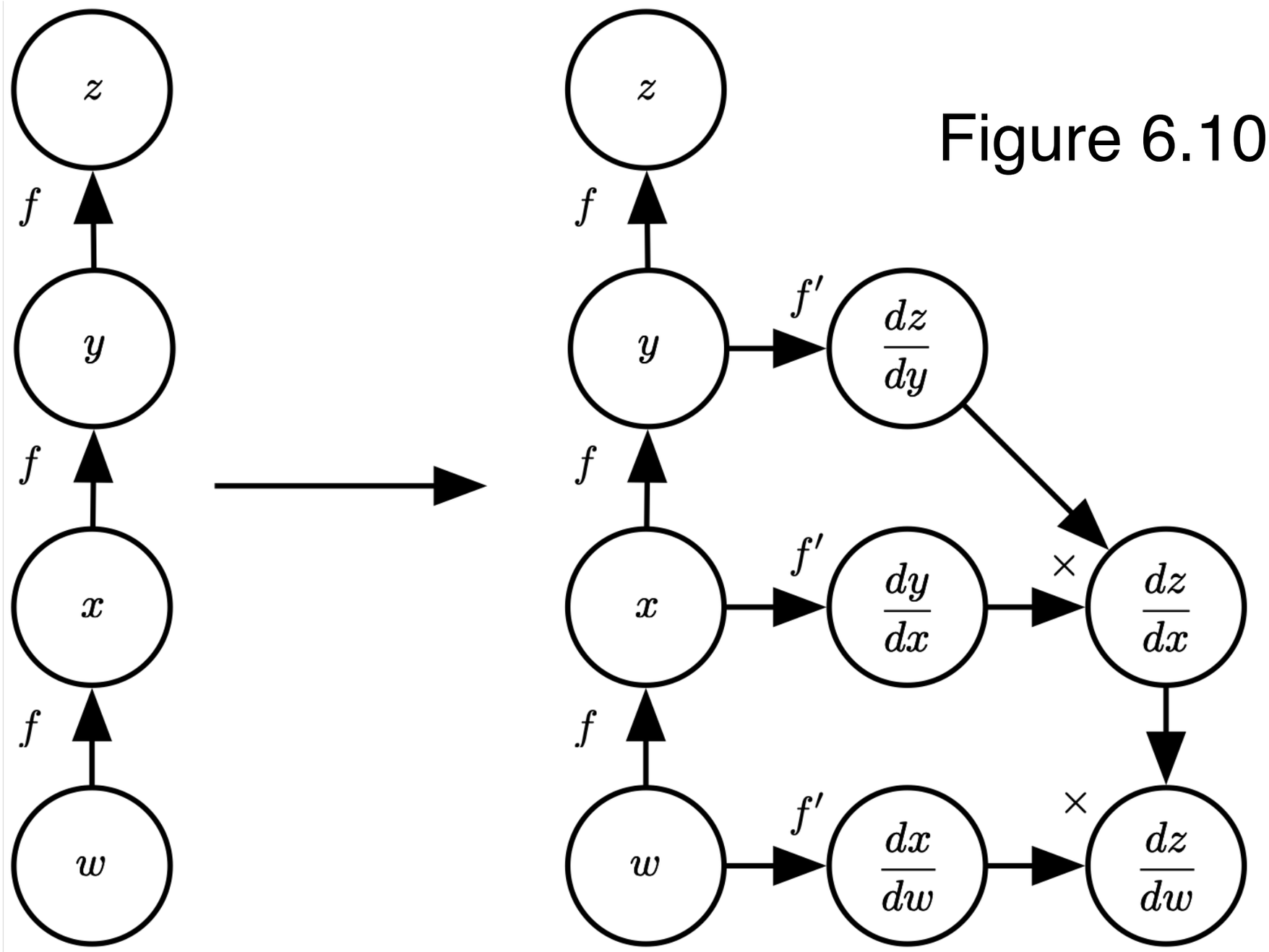


Figure 6.10

Implementing the chain rule

- Given a computation graph, we wish to compute $\frac{\partial u^{(n)}}{\partial u^{(i)}}$, for $i = 1, \ldots, n_i$
  - $u^{(n)}$ can be the cost function
  - $u^{(i)}$ are the trainable parameters of the network
  - $u^{(i)}$ is associated with an operation f
    - $u^{(i)} = f(\mathbb{A}_i)$ where $\mathbb{A}_i$ is the set of parents of $u^{(i)}$

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

Forward propagation

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

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

Backward propagation

Algorithm 6.2 Simplified version of the back-propagation algorithm for computing the derivatives of $u^{(n)}$ with respect to the variables in the graph. This example is intended to further understanding by showing a simplified case where all variables are scalars, and we wish to compute the derivatives with respect to $u^{(1)}, \ldots, u^{(n_i)}$. This simplified version computes the derivatives of all nodes in the graph. The computational cost of this algorithm is proportional to the number of edges in the graph, assuming that the partial derivative associated with each edge requires a constant time. This is of the same order as the number of computations for the forward propagation. Each $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ is a function of the parents $u^{(j)}$ of $u^{(i)}$, thus linking the nodes of the forward graph to those added for the back-propagation graph.

Run forward propagation (algorithm 6.1 for this example) to obtain the activations of the network

Initialize `grad_table`, a data structure that will store the derivatives that have been computed. The entry `grad_table`$[u^{(i)}]$ will store the computed value of $\frac{\partial u^{(n)}}{\partial u^{(i)}}$.

`grad_table`$[u^{(n)}] \leftarrow 1$

for $j = n - 1$ down to 1 **do**

  The next line computes $\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$ using stored values:

  `grad_table`$[u^{(j)}] \leftarrow \sum_{i:j \in Pa(u^{(i)})}$ `grad_table`$[u^{(i)}] \frac{\partial u^{(i)}}{\partial u^{(j)}}$

end for

return {`grad_table`$[u^{(i)}] \mid i = 1, \ldots, n_i$}

$$\frac{\delta u^n}{\delta u^n} = 1$$

Efficiency

- The amount of computation required for performing the back-propagation scales linearly with the number of edges in G,
  - the computation for each edge corresponds to computing a partial derivative (of one node with respect to one of its parents)
  - as well as performing one multiplication and one addition.
- Naïve approach:

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

Two implementaion approaches

- Take a computational graph and a set of numerical values for the inputs to the graph, then return a set of numerical values describing the gradient at those input values. We call this approach “symbol-to-number” differentiation. This is the approach used by **PyTorch.**

- Another approach is to take a computational graph and add additional nodes to the graph that provide a symbolic description of the desired derivatives. This is the approach taken **TensorFlow.**
  - Because the derivatives are just another computational graph, it is possible to run back-propagation again, derivatives.

Forward propagation for a neural network

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

Require: Network depth, l
Require: $\boldsymbol{W}^{(i)}, i \in \{1, \ldots, l\}$, the weight matrices of the model
Require: $\boldsymbol{b}^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
Require: $\boldsymbol{x}$, the input to process
Require: $\boldsymbol{y}$, the target output

$\boldsymbol{h}^{(0)} = \boldsymbol{x}$
for $k = 1, \ldots, l$ **do**
  $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)}$
  $\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
end for
$\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
$J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \Omega(\theta)$

Gradients of biases and weights

- We have $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)}\boldsymbol{h}^{(k-1)}$
  - $\frac{\partial J}{\partial b_i^{(k)}} = \frac{\partial L}{\partial a_i^{(k)}} . 1 + \lambda \frac{\partial \Omega}{\partial b_i^{(k)}}$
  - Vectorized: $\nabla_{\boldsymbol{b}^{(k)}} J = \nabla_{\boldsymbol{a}^{(k)}} L + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega$
  - $\frac{\partial J}{\partial W_{i,j}^{(k)}} = \frac{\partial L}{\partial a_i^{(k)}} . h_j^{(k-1)} + \lambda \frac{\partial \Omega}{\partial W_{i,j}^{(k)}}$
  - Vectorized: $\nabla_{\boldsymbol{W}^{(k)}} J = (\nabla_{\boldsymbol{a}^{(k)}} L)\ h^{(k-1)^T} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega$

Outer Product

Gradient of representations

- We have $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)}\boldsymbol{h}^{(k-1)}$
  - ❑ $\nabla_{\boldsymbol{h}^{(k-1)}} L = \left(\frac{\partial a^{(k)}}{\partial h^{(k-1)}}\right)^{T} \nabla_{\boldsymbol{a}^{(k)}} L = \boldsymbol{W}^{(k)^{T}} \nabla_{\boldsymbol{a}^{(k)}} L$
- $\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
  - ❑ $\nabla_{\boldsymbol{a}^{(k)}} L = \nabla_{\boldsymbol{h}^{(k)}} L \odot f'\left(\boldsymbol{a}^{(k)}\right)$

Backward propagation for a neural network

Algorithm 6.4 Backward computation for the deep neural network of algorithm 6.3, which uses in addition to the input $\boldsymbol{x}$ a target $\boldsymbol{y}$. This computation yields the gradients on the activations $\boldsymbol{a}^{(k)}$ for each layer k, starting from the output layer and going backwards to the first hidden layer. From these gradients, which can be interpreted as an indication of how each layer's output should change to reduce error, one can obtain the gradient on the parameters of each layer. The gradients on weights and biases can be immediately used as part of a stochastic gradient update (performing the update right after the gradients have been computed) or used with other gradient-based optimization methods.

After the forward computation, compute the gradient on the output layer:

$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$

for $k = l, l-1, \ldots, 1$ **do**

Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation (element-wise multiplication if f is element-wise):

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$

Compute gradients on weights and biases (including the regularization term, where needed):

$\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\theta)$

$\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\, \boldsymbol{h}^{(k-1)\top} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\theta)$

Propagate the gradients w.r.t. the next lower-level hidden layer's activations:

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top} \boldsymbol{g}$

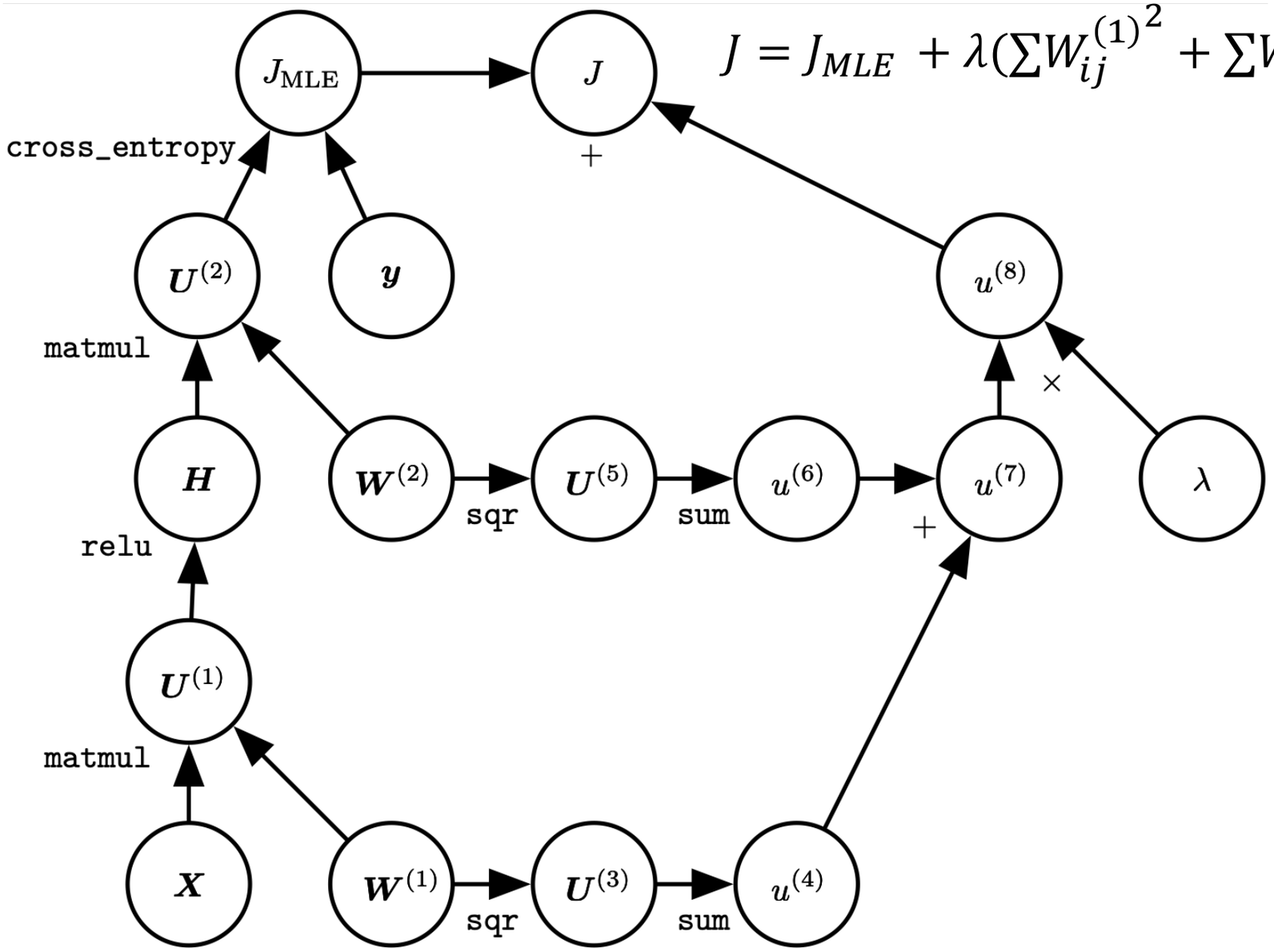
end for

(Goodfellow 2016)

Gradients for matrix multiplication

- $HW = U$
  - $\nabla_{\boldsymbol{H}} f = \nabla_{\boldsymbol{U}} f \; W^T$
  - $\nabla_{\boldsymbol{W}} f = H^T \nabla_{\boldsymbol{U}} f$
- Useful to backprop over a batch of samples
- A layer receives G:
  - GW^T is the update for W
  - $H^T G$ is back propagated

Backprop for a batch of samples



$$J = J_{MLE} + \lambda(\sum W_{ij}^{(1)^2} + \sum W_{ij}^{(2)^2})$$

$$G = \nabla_{U^{(2)}} J_{MLE}$$

$$HW^{(2)} = U^{(2)}$$

$$\nabla_{W^{(2)}} J = H^T G + 2\lambda W^{(2)}$$

$$\nabla_H J_{MLE} = GW^T$$

$$\nabla_{U^{(1)}} J_{MLE} = G'$$

$$XW^{(1)} = U^{(1)}$$

$$\nabla_{W^{(1)}} J = X^T G' + 2\lambda W^{(1)}$$

Cost forward propagation: $O(w)$ multiply-adds
Cost backward propagation: $O(w)$ multiply-adds
Memory-cost: $O(mn_h)$

Backprop in deep learning vs. the general field of automatic differentiation

- Reverse mode accumulation
- Forward mode accumulation
- Simplifying algebraic expressions (e.g. Theano)
  - $q_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$
  - $J = -\sum p_j \log(q_j)$
  - $\frac{\partial J}{\partial z_i} = q_i - p_i$
- Specialized libraries vs. automatic generation
  - Data structures with *bprop()* method
- Backprop is not the only way or the optimal way of computing the gradient, but it is practical

High-order derivatives

- Properties of the Hessian may guide the learning
- $J(\theta)$ is a function $\mathbb{R}^n \rightarrow \mathbb{R}$
  - The hession of $J(\theta)$ is in $\mathbb{R}^{n \times n}$
- In typical deep learning, n can be in the billions
  - Computing the Hessian is not practical
- **Krylov methods** are a set of iterative techniques for performing various operations like:
  - approximately inverting a matrix
  - or finding eigenvectors or eigenvalues,
  - without using any operation other than matrix-vector products.

Hessian-vector Products

$$\boldsymbol{H}\boldsymbol{v} = \nabla_{\boldsymbol{x}} \left[\left(\nabla_{\boldsymbol{x}} f(x) \right)^{\top} \boldsymbol{v} \right]. \tag{6.59}$$

- Compute $\boldsymbol{H}\boldsymbol{e}^{(i)}$ for all $i = 1, \ldots, n$
- Where $\boldsymbol{e}^{(i)}$ is the one hot vector:
  - $e_{\mathrm{i}}^{(i)} = 1$
  - $e_{\mathrm{j}}^{(i)} = 0$

Questions

- Watch Ian's lecture:
  - https://drive.google.com/file/d/0B64011x02sIkRExCY0FDVXFCOHM/view