Feedforward Neural Networks and Backpropagation

Vishnu Lokhande

Department of Computer Science and Engineering
University at Buffalo, SUNY
vishnulo@buffalo.edu

Deep Feedforward Neural Networks

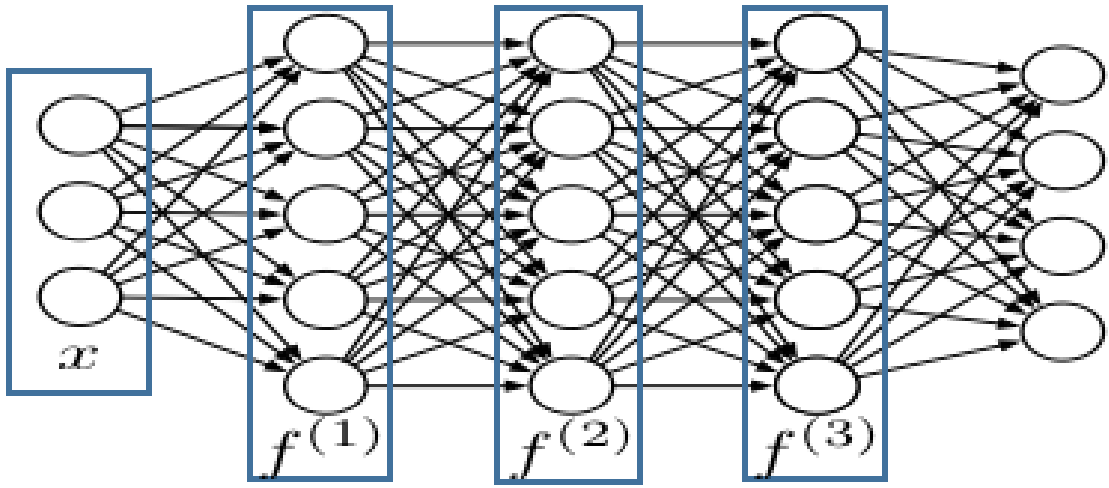
Deep Feedforward Neural Networks

1. Also called:
   - Feedforward neural networks (FNN).
   - Multilayer Perceptrons (MLP).
2. Goal: approximate some (nonlinear) function f
   - *e.g.*, a classifier $y = f(\mathbf{x})$ maps an input $\mathbf{x}$ to a category y.
3. The mapping f is parameterized by θ, thus written as $f(\mathbf{x}; \theta)$.
   - The goal is to learn θ from the data that results in the best function approximation.

How to defined f? and how to learn θ?

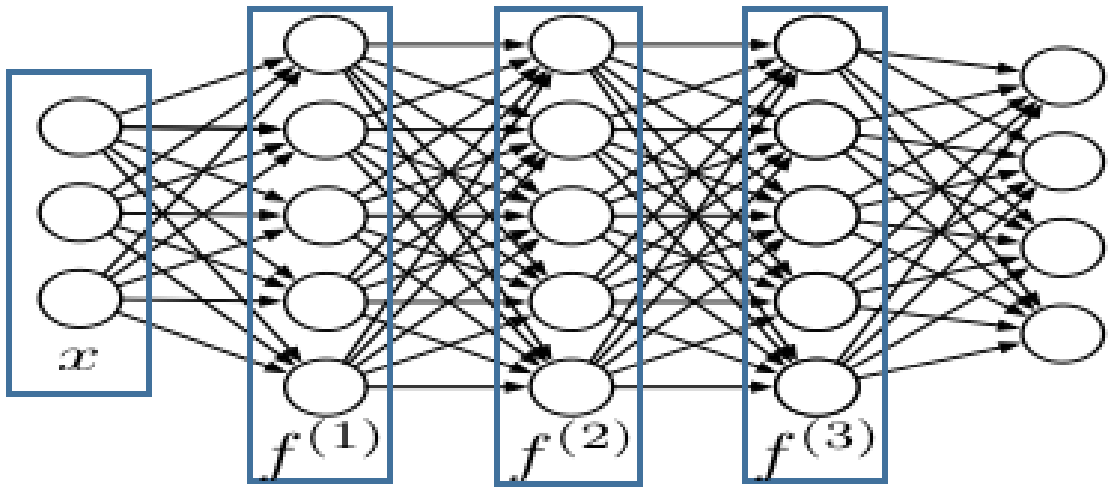
Feedforward Networks

1. It is called Feedforward because
   - information flows through function being evaluated from $\mathbf{x}$ through intermediate computations, and finally to output y.
2. Called networks because they are composed of many different functions.
3. Model is associated with a directed acyclic graph describing how functions are composed, *e.g.*, functions $f^{(1)}$, $f^{(2)}$, $f^{(3)}$, $f^{(4)}$ connected in a chain to form
   $f(\mathbf{x}) = f^{(4)}\left(f^{(3)}\left(f^{(2)}\left(f^{(1)}(\mathbf{x})\right)\right)\right) \triangleq f^{(4)} \circ f^{(3)} \circ f^{(2)} \circ f^{(1)}(\mathbf{x})$.
   - $f^{(1)}$ is called the first layer of the network, $f^{(2)}$ is the second layer, *etc*.



Terminology

1. Overall length of the chain is the depth of the model.
2. The name deep learning arises from this terminology.
3. Final layer of a feedforward network is called the output layer.
4. All layers between input and output are called hidden layers (units) because they are not from the data:
   - Hidden layers are usually deterministic.
   - Sometimes become stochastic in deep generative models for more flexible modeling ability.



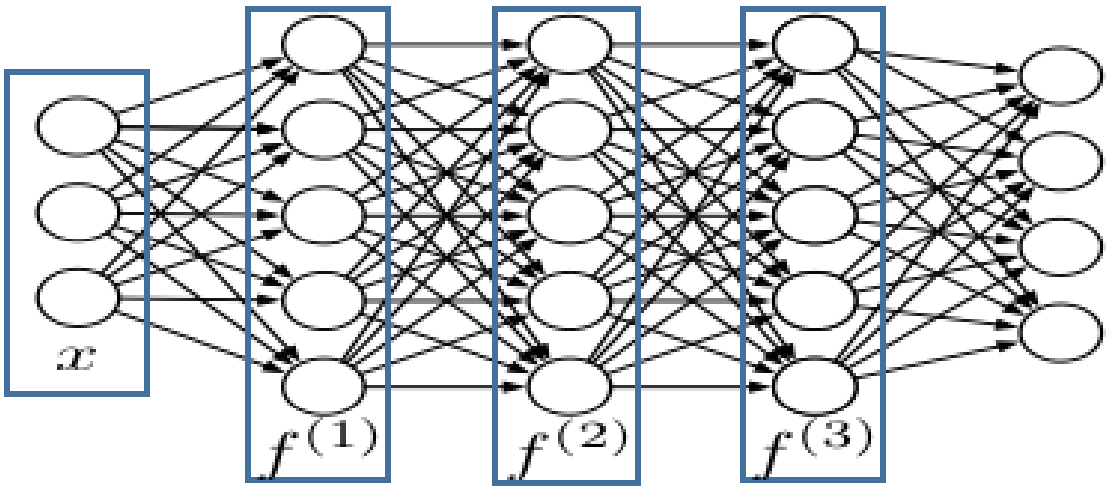
An Example of FNN

1. Define $f^{(i)}(\tilde{\mathbf{x}})$ as

$$f^{(i)}(\tilde{\mathbf{x}}) = \sigma\left({\mathbf{W}^{(i)}}^T \tilde{\mathbf{x}} + \mathbf{b}^{(i)}\right)$$

$$\sigma(\mathbf{a})_k \triangleq \frac{1}{1 + e^{-a_k}}$$

2. Nonlinearity comes in via the sigmoid function.



Using FNN to Solve the XOR Problem

1. XOR: an operation on binary-variable inputs x_1 and x_2:
   - When exactly one value equals 1 it returns 1, otherwise it returns 0:

$$x_1 \oplus x_2 \triangleq \begin{cases} 1 & \text{if } x_1 \neq x_2 \\ 0 & \text{otherwise} \end{cases}$$

   - Target function is $f^*([x_1, x_2]) \triangleq x_1 \oplus x_2$ that we want to learn.
   - Our model is $f([x_1, x_2]; \theta)$, where we learn parameters θ to make f similar to f^*.
2. Not concerned with statistical generalization:
   - Perform correctly on four training points: $X = [0,0]^T, [0,1]^T, [1,0]^T, [1,1]^T$.
   - Challenge is to fit the training set:
     - ★ we want $f([0,0]^T; \theta) = f([1,1]^T; \theta) = 0$ and $f([0,1]^T; \theta) = f([1,0]^T; \theta) = 1$, which is nonlinear.

First Try: Linear Model does not Fit

1. Treat it as regression with MSE loss function

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

   - If $f(\mathbf{x};\theta)$ perfectly matches $f^*(\mathbf{x})$, it has the minimum loss (zero loss).

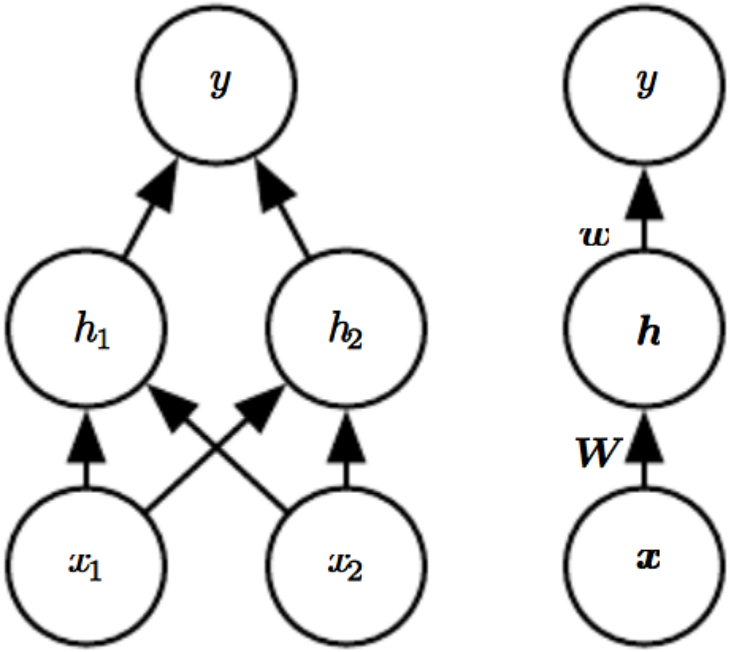
2. Consider a linear model with $\theta = \{\mathbf{w}, \mathbf{b}\}$ such that

$$f(\mathbf{x};\theta) = \mathbf{w}^T\mathbf{x} + \mathbf{b}$$

3. Minimize $J(\theta)$ to get a close-form solution:
   - Differentiate J w.r.t. $\mathbf{w}$ and $\mathbf{b}$ to obtain $\mathbf{w} = 0$ and $\mathbf{b} = 1/2$.
   - Thus $f(\mathbf{x};\theta) = 1/2$, which simply outputs 0.5 everywhere $\Rightarrow$ not equal to $f^*(\mathbf{x})$.

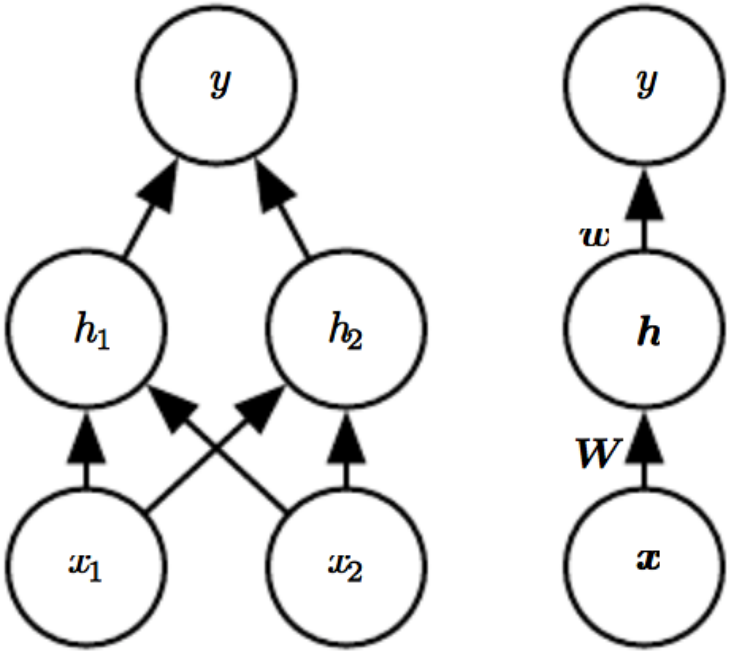
Feedforward Network for XOR

1. Use a simple feedforward network with one hidden layer **h** containing two units.
2. Matrix **W** describes mapping from **x** to **h**.
3. Vector **w** describes mapping from **h** to **y**;
4. Intercept parameters **b** are omitted for simplicity.



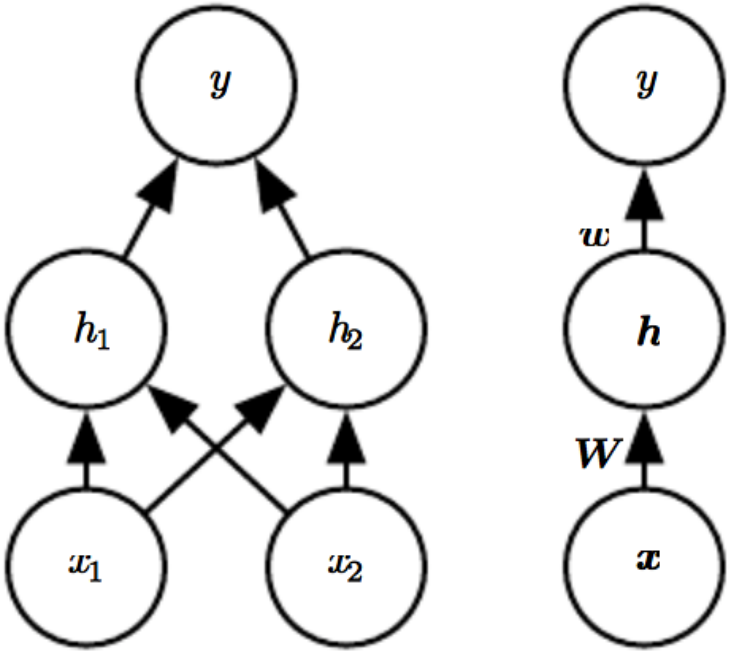
Model Specification

1. Layer 1 (hidden): $\mathbf{h} = f^{(1)}(\mathbf{x}; \mathbf{W}, \mathbf{c})$.
2. Layer 2 (output): $y = f^{(2)}(\mathbf{h}; \mathbf{w}, b)$.
3. Complete model: $f(\mathbf{x}; \mathbf{W}, \mathbf{c}, \mathbf{w}, b) = f^{(2)}(f^{(1)}(\mathbf{x}))$.



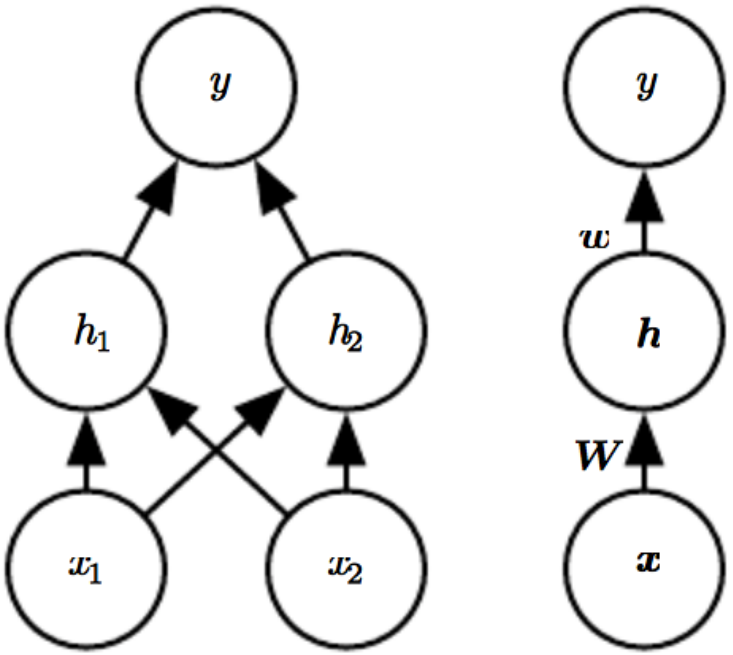
Linear vs. Nonlinear Functions

What happens if using linear functions $f^{(1)}$ and $f^{(2)}$ in a deep network architecture?



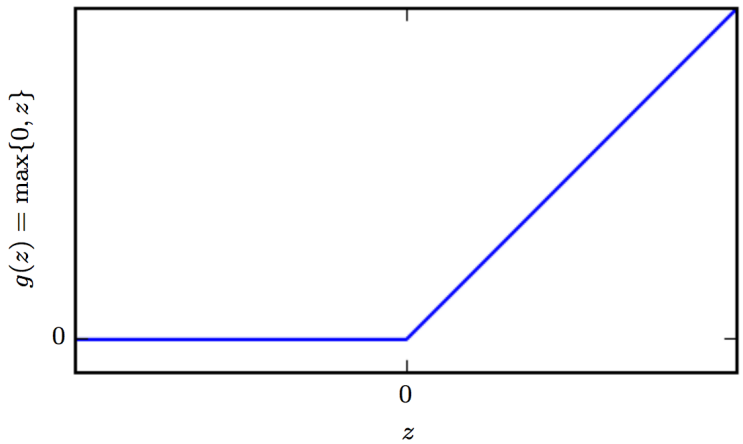
Linear vs. Nonlinear Functions

1. If we choose $f^{(1)}$ and $f^{(2)}$ to be linear functions, *e.g.*, $f^{(1)}(\mathbf{x}) = \mathbf{W}^T \mathbf{x}$ and $f^{(2)}(\mathbf{h}) = \mathbf{w}^T \mathbf{h}$, the complete model is still linear: $f(\mathbf{x}) = \mathbf{w}^T \mathbf{W}^T \mathbf{x}$.
2. Since linear is insufficient, we must use a nonlinear function to describe the features.
3. We use the strategy of neural networks by using a nonlinear activation function g: $\mathbf{h} = g(\mathbf{W}^T \mathbf{x} + \mathbf{c})$.



Activation Functions

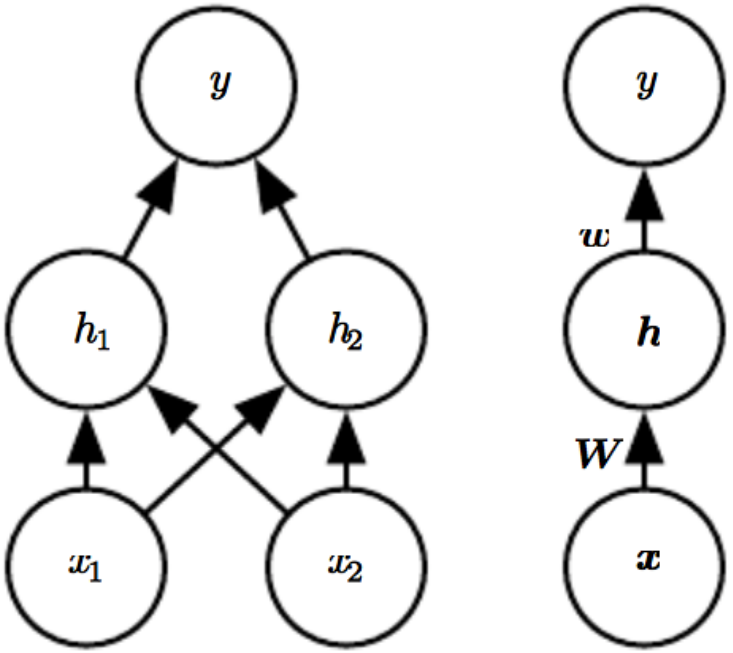
1. Activation function g is typically chosen to be applied element-wise $h_i = g(W_{:,i}^T \mathbf{x} + c_i)$.
2. Default activation function: $g(\mathbf{z}) = \max\{0, \mathbf{z}\}$.
   - Called Rectified Linear Unit (ReLU).
   - Applying ReLU to the output of a linear transformation yields a nonlinear transformation.
   - Function remains closed to linear
     - ★ preserve properties that make linear models easy to optimize with gradient-based methods
     - ★ preserve many properties that make linear models generalize well



Specifying the Network using ReLU

1. Activation: $g(\mathbf{z}) = \max\{0, \mathbf{z}\}$ for the first layer.
2. The complete model:

$$f(\mathbf{x}; \mathbf{W}, \mathbf{c}, \mathbf{w}, \mathbf{b}) = f^{(2)}(f^{(1)}(\mathbf{x})) = \mathbf{w}^T \max\{0, \mathbf{W}^T \mathbf{x} + \mathbf{c}\} + \mathbf{b}$$



XOR Solution

1. The model can be learned by gradient (introduced later). For now let's guess the following solution

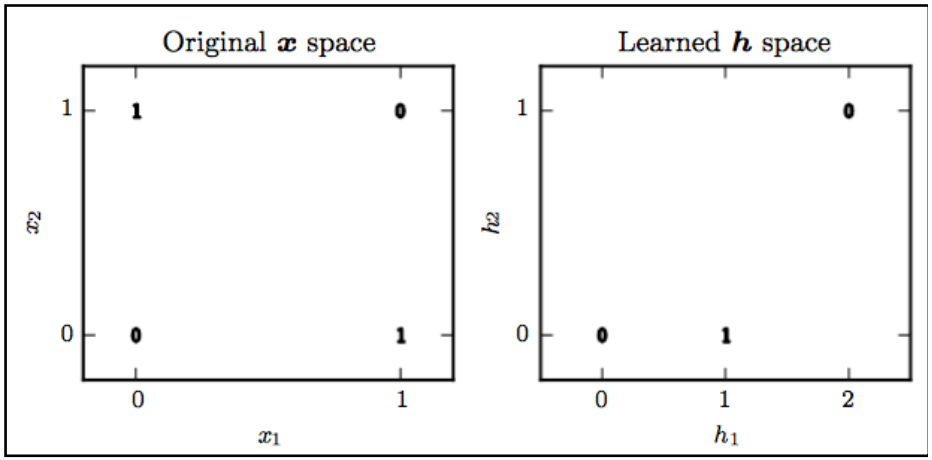
$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \mathbf{c} = \begin{bmatrix} 0 & -1 \end{bmatrix}, \mathbf{w} = \begin{bmatrix} 1 & -2 \end{bmatrix}, b = 0$$

$$f\left(\begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}; \{\mathbf{W}, \mathbf{c}, \mathbf{w}, b\} \right) = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Perfectly match!

Learned representation for XOR

1. Two points that must have output 1 have been collapsed into one in the latent space, *i.e.*, points $x = [0, 1]^T$ and $x = [1, 0]^T$ have been mapped into $h = [1, 0]^T$.
2. Data in the latent can be described in a linear model.



Summary

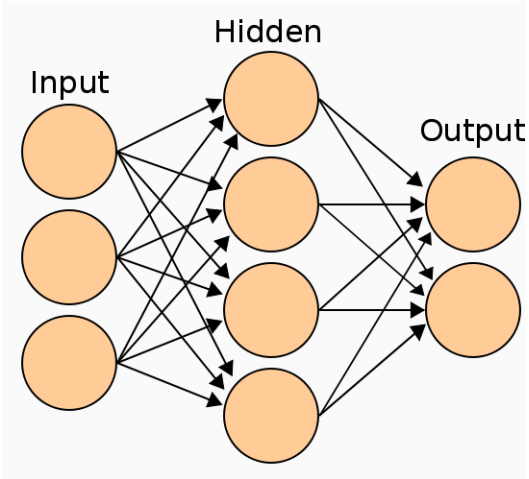
1. In the above FNN, we simply specified its solution, and showed that it achieves zero error.
2. In real situations, there might be billions of parameters and billions of training examples,
   - one cannot simply guess the solution
3. Instead gradient-descent-based optimization is used to find parameters that produce very little error:
   - detailed later.

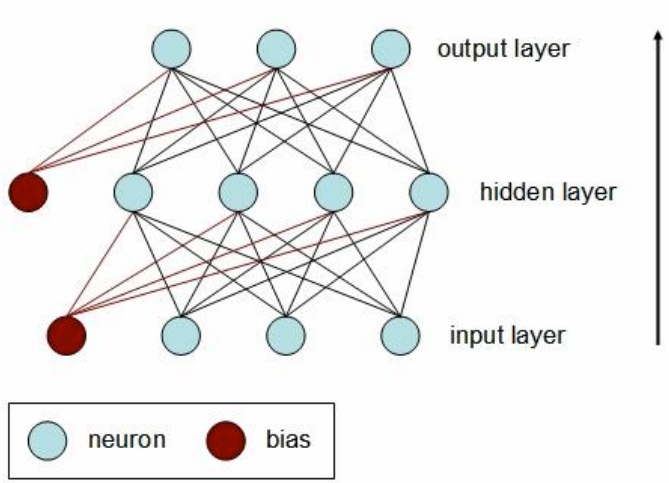
What are some popular activation functions and kinds are effective in learning?

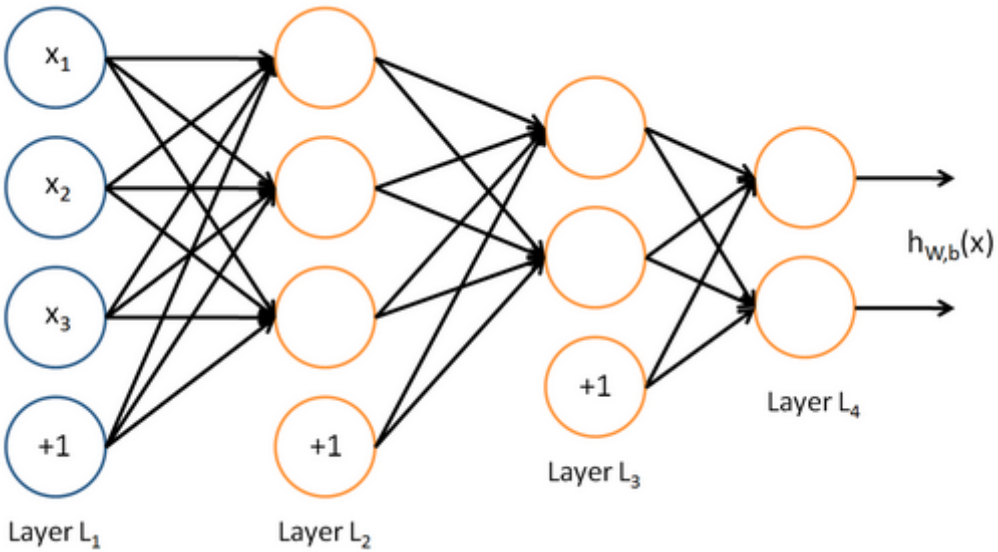
What are popular loss functions in deep learning and how to derive the gradients?

Hidden Units

Hidden Units in Neural Networks







What a Hidden unit does?

1. Accepts a vector of inputs $\mathbf{x}$ and computes an affine transformation $\mathbf{z} = \mathbf{W}^T \mathbf{x} + \mathbf{b}$.
2. Computes an element-wise non-linear function $g(\mathbf{z})$.
3. Most hidden units are distinguished from each other by the choice of activation function $g(\mathbf{z})$, *e.g.*, ReLU, sigmoid and tanh, and *etc*.

Choice of Hidden Unit

1. ReLu is a popular default choice.
2. Design of hidden units is an active research area that does not have many definitive guiding theoretical principles.
3. Design process is trial and error.

Is Differentiability Necessary?

1. Some hidden units are not differentiable at some input points:
   - Rectified Linear Unit $g(\mathbf{z}) = \max\{0, \mathbf{z}\}$ is not differentiable at $\mathbf{z} = 0$.
2. May seem like it invalidates for use in gradient-based learning, however, it performs well in practice.
3. Generally required non-differentiability at only a finite number of points:
   - otherwise there is no gradient backpropagated.

Rectified Linear Unit

1. Activation function $g(\mathbf{z}) = \max\{0, \mathbf{z}\}$
   - easy to optimize due to similarity with linear units.
2. Usually used on top of an affine transformation $\mathbf{h} = g(\mathbf{W}^T \mathbf{x} + \mathbf{b})$.
3. Good practice to set all elements of b to a small value such as 0.1
   - makes it likely that ReLU will be initially active for most training samples and allow derivatives to pass through.

Three Generalizations of ReLU

1. Three methods based on using a non-zero slope α_i when $z_i < 0$:

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

   - Absolute-value rectification:
     - ★ fix $\alpha_i = -1$ to obtain $g(\mathbf{z}) = |\mathbf{z}|$.
   - Leaky ReLU:
     - ★ fixes α_i to a small value like 0.01.
   - Parametric ReLU or PReLU:
     - ★ treats α_i as a parameter.

Maxout Units

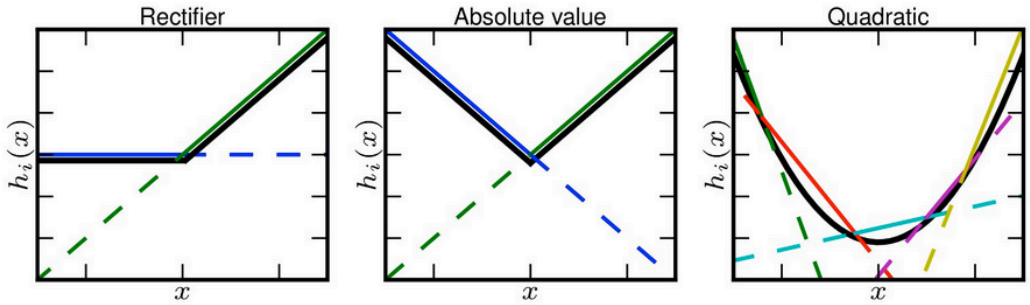
1. Instead of applying element-wise function $g(\mathbf{z})$ as in ReLU, maxout units divide $\mathbf{z}$ into k-groups.
2. Each maxout unit then outputs the maximum element of one of these groups:

$$h(\mathbf{x}) = g(\mathbf{z}) = \max\{\mathbf{z}_1, \mathbf{z}_2, \cdots, \mathbf{z}_n\}$$

   where $\mathbf{z}_i \triangleq \mathbf{x}^T \mathbf{W}_i + b_i$.
3. This provides a way of learning a piecewise linear function that responds to multiple directions in the input x space.
4. Computationally more expensive than ReLU, not as widely used.

Maxout as Learning Activation

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

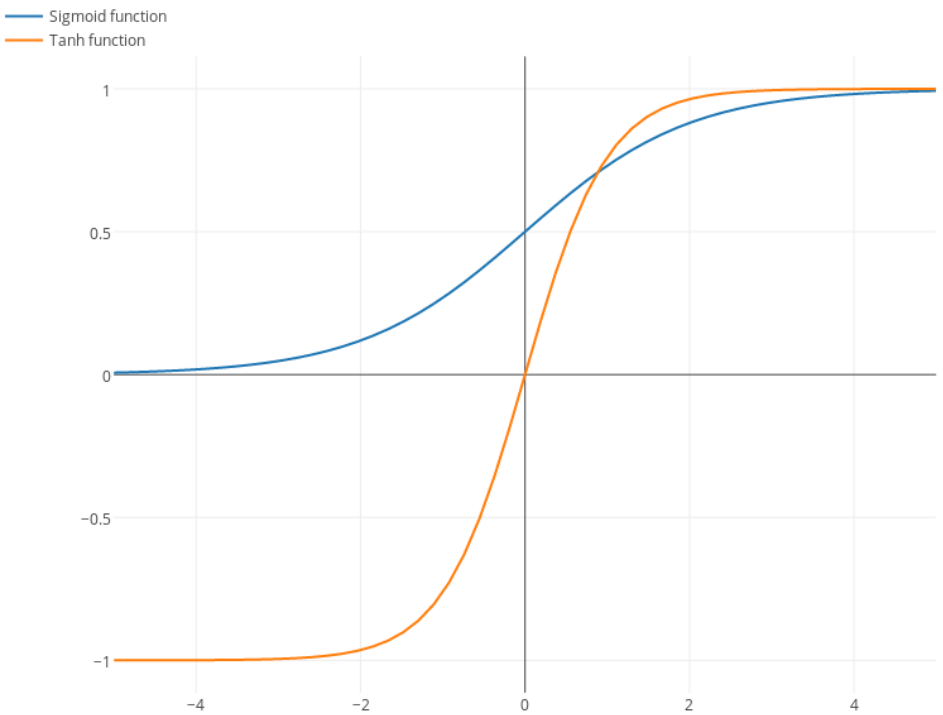


2. A maxout layer with two pieces can learn to implement the same function of the input as a traditional layer using ReLU or its generalizations.

Logistic Sigmoid

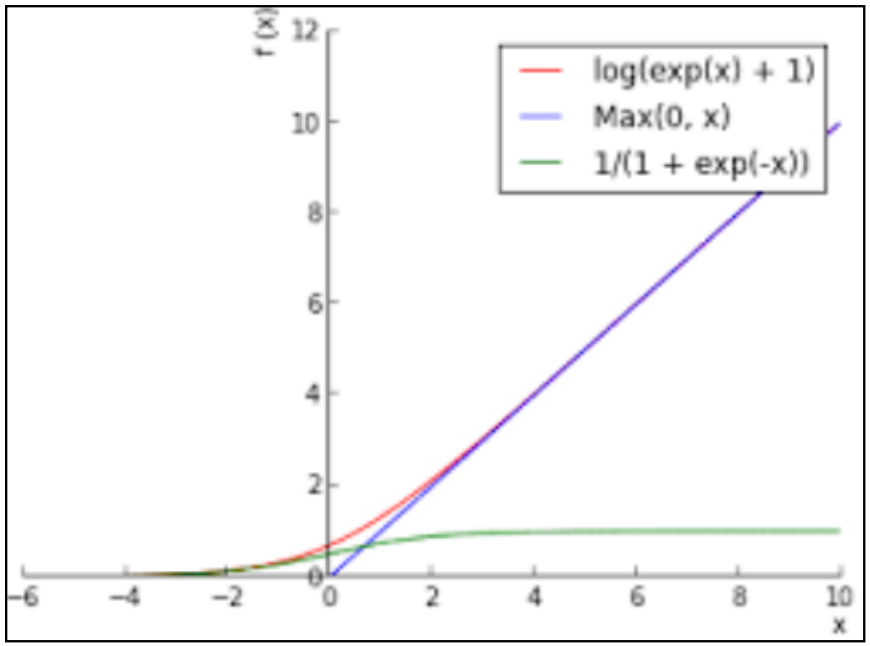
1. Prior to ReLU, most neural networks used logistic sigmoid activation: $g(\mathbf{z}) = \sigma(\mathbf{z}) \triangleq \frac{1}{1+e^{-\mathbf{z}}} \in [0, 1]$.
2. Or the hyperbolic tangent

$$g(\mathbf{z}) = tanh(\mathbf{z}) = \frac{\sinh(\mathbf{z})}{\cosh(\mathbf{z})} = \frac{e^{\mathbf{z}} - e^{-\mathbf{z}}}{e^{\mathbf{z}} + e^{-\mathbf{z}}} = 2\sigma(2\,\mathbf{z}) - 1 \in [-1, 1]$$



Sigmoid Saturation

1. Sigmoid saturates across most of domain:
   - saturate to 1 when **z** is very positive and 0 when **z** is very negative.
   - saturation makes gradient-learning difficult, *e.g.*, difficult to set the step size for the whole range.
2. ReLU and Softplus ($\log(\exp(x) + 1)$, smooth version of ReLU) increase (almost linearly) for input >0:
   - easier for gradient learning



Sigmoid vs tanh Activation

1. Hyperbolic tangent typically performs better than logistic sigmoid.
2. It resembles the identity function more closely

$$\textit{tanh}(0) = 0, \text{ while } \sigma(0) = 1/2$$

3. *tanh* still saturates when $\mathbf{z}$ is very positive or negative.
4. Since tanh is similar to identity near 0, training a deep neural network $\hat{y} = \mathbf{w}^T \tanh(\mathbf{U}^T \tanh(\mathbf{V}^T \mathbf{x}))$ resembles training a linear model $\hat{y} = \mathbf{w}^T \mathbf{U}^T \mathbf{V}^T \mathbf{x}$, as long as the activations can be kept small, thus learning is easier.
5. Sigmoid units still useful in other types of networks such as recurrent neural networks despite the saturation.

Other Hidden Units

1. Many other types of hidden units possible, but used less frequently
   - FNN with $\mathbf{h} = \sin(\mathbf{W}\mathbf{x} + \mathbf{b})$, obtained error rate similar to that using tanh*, approximately 2% error rate on MNIST for a one- or two-layer FNN.
   - Radial Basis: $h_i = \exp\left(\frac{1}{\sigma^2} \| \mathbf{W}_{:,i} - \mathbf{x} \|^2\right)$.
   - Softplus: $g(\mathbf{z}) = \log(1 + e^{\mathbf{z}})$, smooth version of the ReLU.
   - Hard tanh: shaped similar to tanh and the rectifier but it is bounded:

$$g(\mathbf{z}) = \max\left(-1, \min(1, \mathbf{z})\right)$$

* G. Parascandolo & H. Huttunen & T. Virtanen, 2017

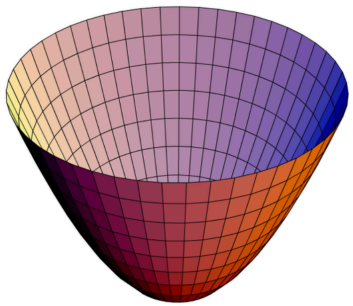
Objective Function in Gradient-Based Learning

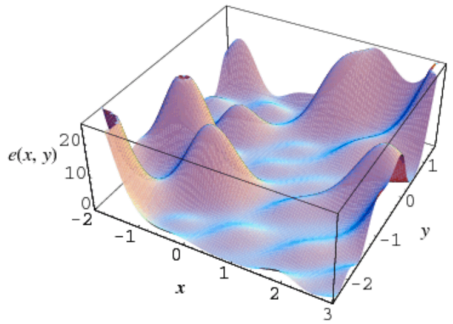
Standard ML Training vs. NN Training

1. Similar to standard ML training, in deep learning we need
   - cost function, *e.g.*, MLE.
   - models, *e.g.*, FNN.
   - optimization algorithm, *e.g.*, gradient descent.
2. Different from traditional ML: nonlinearity causes non-convex loss
   - use iterative gradient-based optimizers that merely drives cost to low value.
   - in standard ML, we have exact linear equation solvers used for linear regression, or convex optimization algorithms used for logistic regression or SVMs.

Convex vs Non-convex

1. Convex methods:
   - converge to global optima from any initial parameters.
   - Robust and theoretically sound.
2. Non-convex with SGD:
   - sensitive to initial parameters.
   - for FNN, important to initialize Weights to small values, Biases to zero or small positives.
   - SGD also adopted for training Linear Regression and SVM, especially with large training sets.
   - training neural net no different to other models, except computing gradient is more complex.
   - general convergence theory has not been established, though there are some pioneer work.

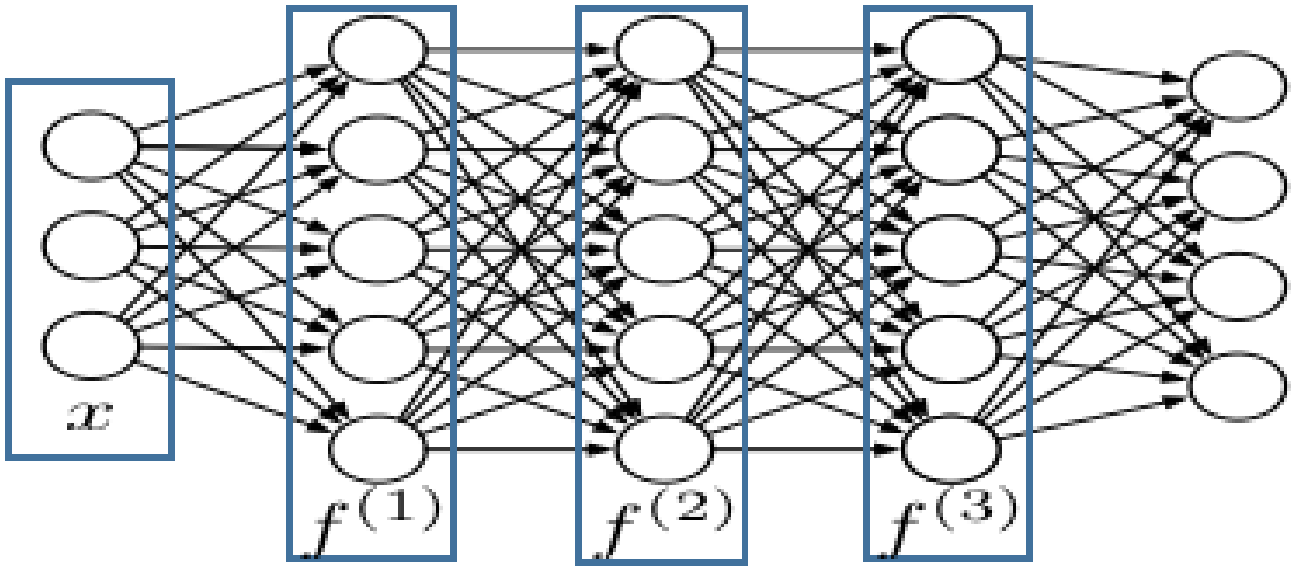




Loss Functions for Deep Learning

1. Important to make a deep neural network work:
   - similar to those for parametric models such as linear models
2. In FNN, the goal is to try to match the output to the true data (labels).
3. How to measure how well they match?

Loss function corresponds to what happens in the output layer.



Cross Entropy

Cross entropy for distributions p and q

$$H(p, q) = \mathbb{E}_p [-\log q] = -\int \log q(x) p(x) \mathrm{d}x$$

$$\xrightarrow{\text{discrete } p \text{ and } q} = -\sum_x p(x) \log q(x)$$

- To use this metric, must specify both target output data and the output from the neural network as distributions.
- Specifying the model $p(\mathbf{y} \mid \mathbf{x}; \theta)$ automatically determines a loss function:

$$J(\theta) = -\mathbb{E}_{\mathbf{x}, y \sim \hat{p}_{\text{data}}} \log p(\mathbf{y} \mid \mathbf{x}; \theta)$$

  - equivalently described as the cross-entropy between the training data and the model distribution

Output Units

1. Choice of loss function typically only depends on the choice of output unit, at least in FNN.
2. Types of output units:
   - Sigmoid units: for Bernoulli Output Distributions, *e.g.*, binary classification.
   - Softmax units: for Multinomial Output Distributions, *e.g.*, multi-class classification.
   - Linear units: no non-linearity, for Gaussian Output distributions, *e.g.*, regression.

Loss Functions for Binary Classification with Logistic Regression

1. Logistic regression: data $\{\mathbf{x}_n \in \mathbb{R}^L, y_n \in \{0,1\}\}$, likelihood

$$p(\mathbf{y} \,|\, \mathbf{x}; \boldsymbol{\theta}) = \prod_{n=1}^{N} \sigma(\boldsymbol{\theta}^T \mathbf{x}_n)^{y_n} (1 - \sigma(\boldsymbol{\theta}^T \mathbf{x}_n))^{1-y_n}$$

2. Use the principle of maximum likelihood

$$J(\boldsymbol{\theta}) = -\log p(\mathbf{y} \,|\, \mathbf{x}; \boldsymbol{\theta}) = -\sum_{n=1}^{N} \Big(y_n \log \sigma(\boldsymbol{\theta}^T \mathbf{x}_n) + (1 - y_n) \log(1 - \sigma(\boldsymbol{\theta}^T \mathbf{x}_n)) \Big)$$

3. Let $p_n \triangleq [y_n, 1 - y_n]$, $q_n = [\sigma(\boldsymbol{\theta}^T \mathbf{x}_n), 1 - \sigma(\boldsymbol{\theta}^T \mathbf{x}_n)]$:
   - they form probability vectors, can be regarded as two discrete distributions of dimension two.
4. The negative log-likelihood coincides with cross-entropy of p_n and q_n.
5. Gradient is: $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_{n=1}^{N} (\tilde{y}_n - y_n) \mathbf{x}_n$, where $\tilde{y}_n \triangleq \sigma(\boldsymbol{\theta}^T \mathbf{x}_n)$.

Binary Classification with FNN

1. Only relates to the output layer of an FNN.
2. Given features $\mathbf{h}$ prior to the output layer, a layer of sigmoid output units produces scalar

$$\hat{y} = \sigma(\mathbf{w}^T \mathbf{h} + b) = \frac{1}{1 + \exp(-\mathbf{w}^T \mathbf{h} - b)}$$

3. Use to produce a conditional Bernoulli output $y \in 0, 1$

$$p(y = 1 \mid \mathbf{x}) = \hat{y}, \text{let } z = \mathbf{w}^T \mathbf{h} + b$$

$$\xrightarrow{\text{write it in another way}} p(y \mid \mathbf{x}) = \frac{\exp(yz)}{\sum_{y'=0}^{1} \exp(y'z)} = \sigma((2y - 1)z)$$

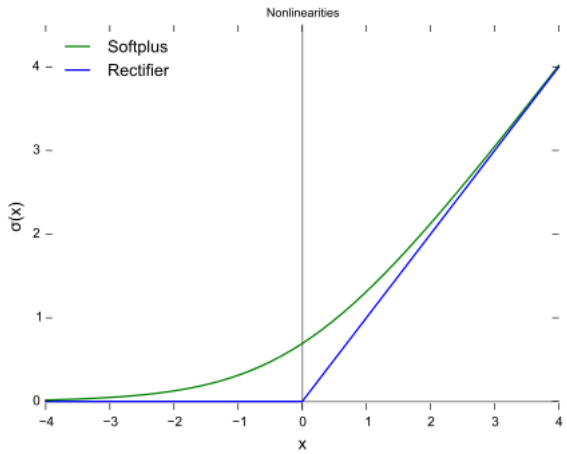
4. Loss function with negative log-likelihood:

$$J(\theta) = -\log p(y \mid \mathbf{x}) = -\log \sigma((2y - 1)z) = \varsigma((1 - 2y)z),$$

where $\varsigma(x) = \log(1 + \exp(x))$ is the softplus function.

Property of Loss Function for Bernoulli MLE

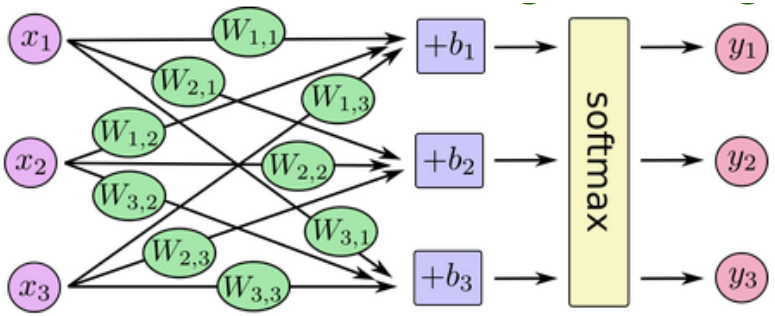
Softplus: $J(\theta) = \varsigma((1-2y)z)$



1. The loss function saturates only when $(1-2y)z << 0$.
2. Saturation occurs only when model already has the right answer:
   - *i.e.*, when $y = 1$ and $z >> 0$ or $y = 0$ and $z << 0$. This data then contributes little when applying gradient-based learning algorithms.
   - when z has the wrong sign, $(1-2y)z$ can be simplified to $|z|$, which does not shrink the gradient at all, a useful property because gradient-based learning can act quickly to correct a mistaken z.

Multi-class Classification with FNN

Generalization of Logistic Regression to multivalued output



Softmax definition

$$\boldsymbol{y} = \text{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

Network Computes

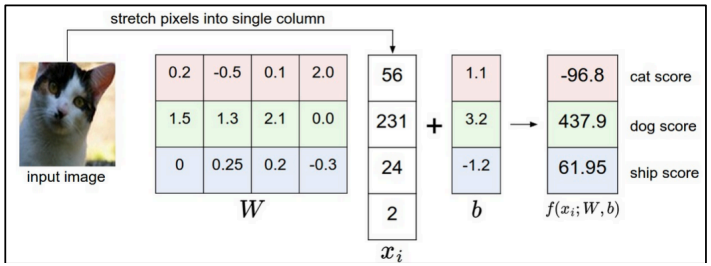
$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax}\left(\begin{bmatrix} W_{1,1}x_1 + W_{1,2}x_2 + W_{1,3}x_3 + b_1 \\ W_{2,1}x_1 + W_{2,2}x_2 + W_{2,3}x_3 + b_2 \\ W_{3,1}x_1 + W_{3,2}x_2 + W_{3,3}x_3 + b_3 \end{bmatrix} \right)$$

In matrix multiplication notation

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax}\left(\begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right)$$

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

An example



Softmax Output in FNN

1. The label is encoded as a n-dimensional probability vector y, with the y-th element being 1, while others being 0:
   - recognized as one-hot vectors.
2. The output of an FNN is Softmax, which forms a probability distribution over a discrete variable with n values.
3. Softmax produces a vector $\hat{y}$ with values $\hat{y} = P(y = i|\,\mathbf{x})$:
   - need elements of $\hat{y}$ lie in $[0, 1]$ and they sum to 1.
4. Similar to the sigmoid case, we normalize n exponentials:

$$\text{softmax}(\mathbf{z})_k = \frac{\exp(z_k)}{\sum_{k'} \exp(z_{k'})}, \quad \mathbf{z} = \mathbf{W}^T \mathbf{h} + \mathbf{b}$$

Saturation of Softmax

$$\text{softmax}(\mathbf{z})_k = \frac{\exp(z_k)}{\sum_{k'} \exp(z_{k'})}$$

1. Like sigmoid, softmax activation can saturate:
   - when the differences between input values become extreme.
2. This is a generalization of the way the sigmoid units saturate:
   - it can cause similar difficulties in learning if the loss function is not designed to compensate for it.
   - the cross-entropy loss function ease the issue, shown later.
3. In practice, we use a numerically stable variant of softmax

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

4. Reformulation allows us to evaluate softmax with high accuracy.

Likelihood with Log-Softmax

1. Using softmax as the likelihood function, we get

$$\log \text{softmax}(\mathbf{z})_i = z_i - \log \sum_j \exp(z_j) \approx z_i - max_j z_j$$

2. Easy to train with gradient descent because it behaves similarly to a linear function:
   - if the correct answer already has the largest input to softmax, then the two terms will roughly cancel. This data will then contribute little to overall training cost.

Multi-class Classification with FNN

$$\text{softmax}(\mathbf{z})_k = \frac{\exp(z_k)}{\sum_{k'} \exp(z_{k'})} \triangleq \tilde{y}_k, \quad \mathbf{z} = \mathbf{W}^T \mathbf{h} + \mathbf{b}$$

Cross entropy

$$J = -\sum_{n=1}^{N} \sum_{k=1}^{K} y_k^{(n)} \log \tilde{y}_k^{(n)}$$

Gradient (consider $N = 1$)

$$\nabla_{\mathbf{W}} J = \frac{\partial J}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{W}} = -\sum_{k=1}^{K} y_k \nabla_{z_k} \log \tilde{y}_k \frac{\partial z_k}{\partial \mathbf{W}}$$

Derivative of softmax (assume $N = 1$)

$$J = -\sum_{k=1}^{K} y_k \log \tilde{y}_k, \quad \tilde{y}_k = \frac{\exp(z_k)}{\sum_{k'} \exp(z_{k'})}$$

Derive $\frac{\partial J}{\partial z_i}$

Derivative of softmax (assume $N = 1$)

$$J = -\sum_{k=1}^{K} y_k \log \tilde{y}_k, \quad \tilde{y}_k = \frac{\exp(z_k)}{\sum_{k'} \exp(z_{k'})}$$

Derive $\frac{\partial J}{\partial z_i}$

$$\begin{aligned}
\frac{\partial J}{\partial z_i} &= -\sum_k y_k \frac{\partial \log \tilde{y}_k}{\partial z_i} = -\sum_k y_k \frac{1}{\tilde{y}_k} \frac{\partial \tilde{y}_k}{\partial z_i} \\
&= -y_i(1 - \tilde{y}_i) - \sum_{k \neq i} y_k \frac{1}{\tilde{y}_k}(-\tilde{y}_k \tilde{y}_i) \\
&= -y_i(1 - \tilde{y}_i) + \sum_{k \neq i} y_k \tilde{y}_i \\
&= \tilde{y}_i \sum_k y_k - y_i = \tilde{y}_i - y_i
\end{aligned}$$

Linear units for Gaussian regression

1. Given features $\mathbf{h}$, a layer of linear output units produces a vector

$$\tilde{y} = \mathbf{W}^T \mathbf{h} + \mathbf{b}$$

2. Use to produce mean $\tilde{y}$ of a conditional Gaussian distribution

$$p(y|\mathbf{x}) = \mathcal{N}(y; \tilde{y}, \mathbf{I}) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}\|y - \tilde{y}\|^2\right)$$

3. Cross entropy recovers the mean squared error cost ($\theta \triangleq \{\mathbf{W}, \mathbf{b}\}$)

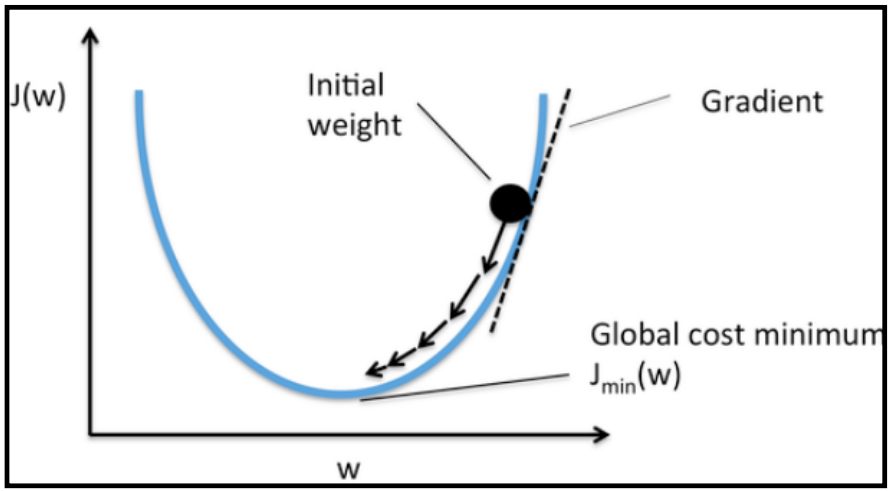
$$J(\theta) = -\frac{1}{2}\mathbb{E}_{\mathbf{x},y\sim\hat{p}_{\text{data}}} \|y - f(\mathbf{x};\theta)\|^2 + const$$

4. Gradient:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\mathbf{x},y\sim\hat{p}_{\text{data}}}(y - \tilde{y})\nabla_\theta \tilde{y}$$

Desirable Property of Gradient

1. Gradients in neural networks should be large and predictable enough to make learning algorithms progressed.
2. Functions that saturate (become very flat) undermine this objective because the gradient becomes very small.
   - Happens when activation functions producing output of hidden/output units saturate.
3. Negative log-likelihood helps avoid saturation for many models
   - Log function in Negative log likelihood cost function undoes exp of some units.

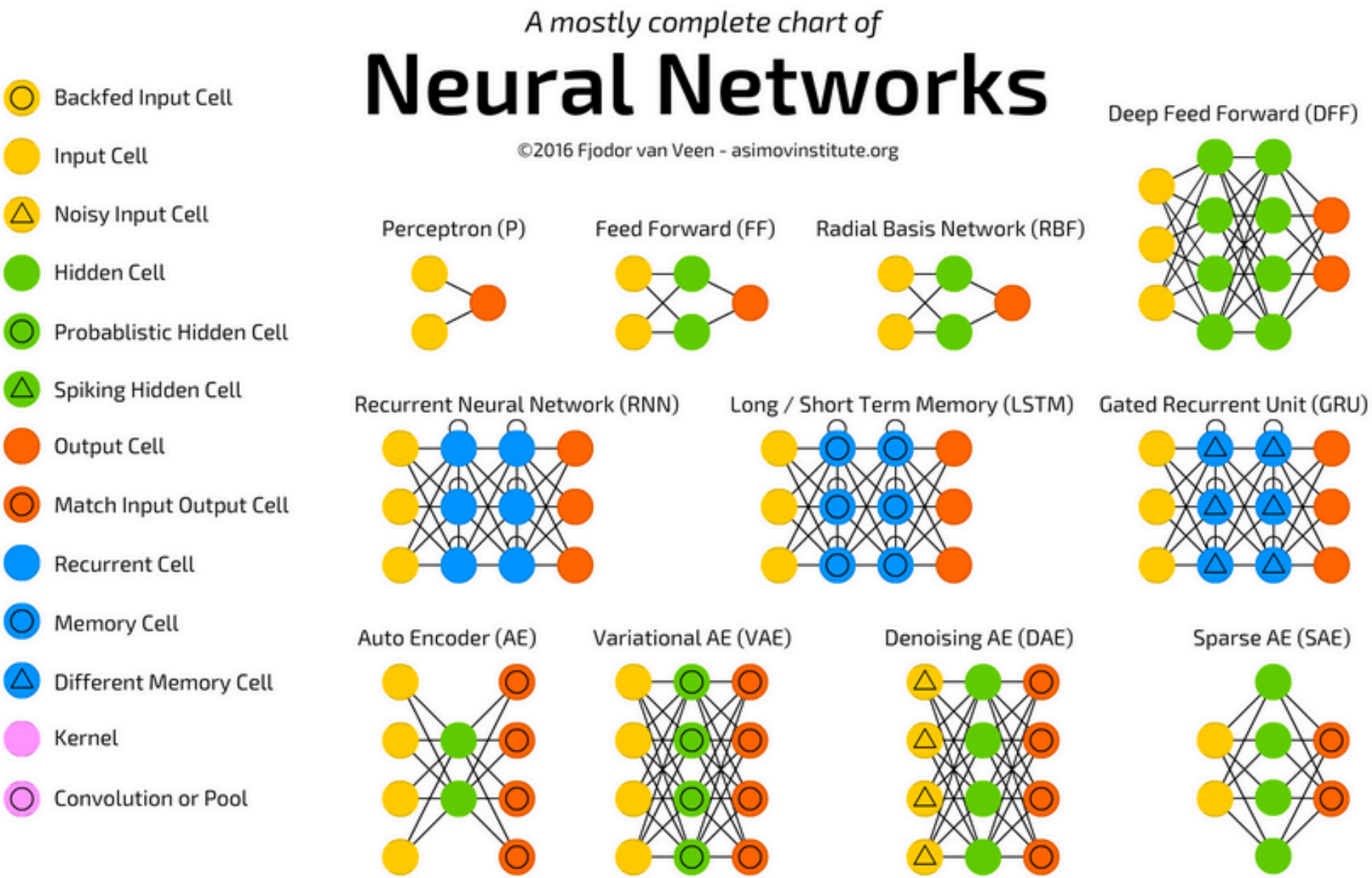


Why deep?

What is a computational graph?

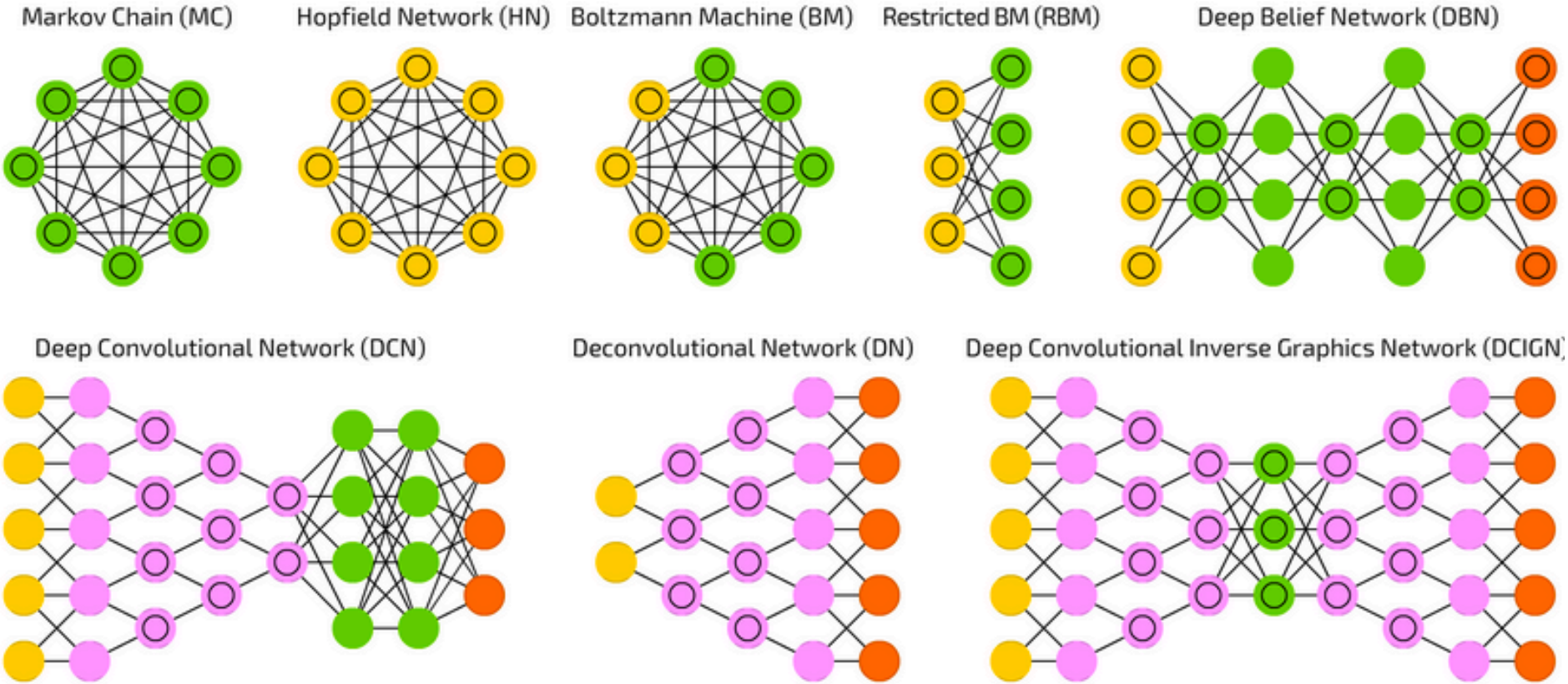
Architecture Design for Deep Learning

Generic Neural Architectures (1-11)



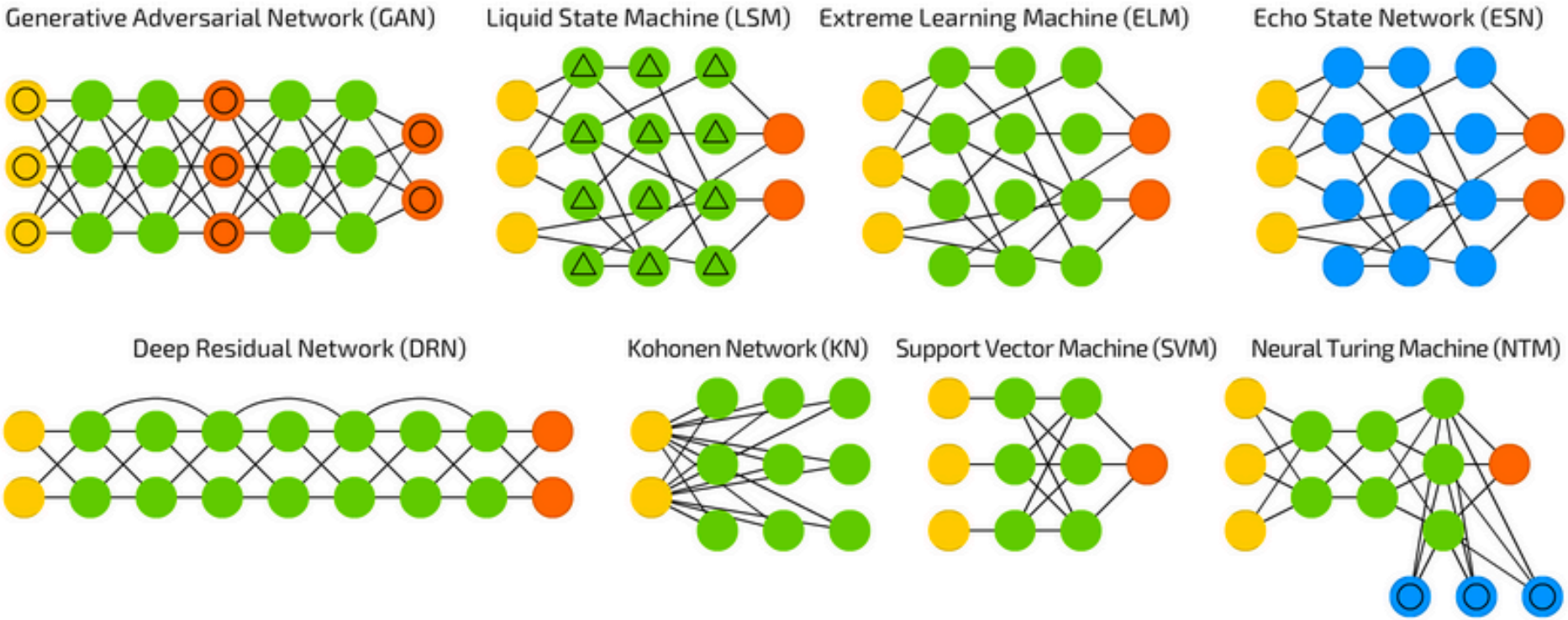
http://www.asimovinstitute.org/neural-network-zoo/

Generic Neural Architectures (12-19)



http://www.asimovinstitute.org/neural-network-zoo/

Generic Neural Architectures (20-27)



http://www.asimovinstitute.org/neural-network-zoo/

Architecture Terminology

1. The word architecture refers to the overall structure of the network.
   - How many units should it have?
   - How the units should be connected to each other?
2. Most neural networks are organized into groups of units called layers.
   - Most architectures arrange these layers in a chain structure.
   - Each layer is a function of the layer that preceded it, *e.g.*,
     - ★ first layer is given by $\mathbf{h}^{(1)} = g^{(1)}\left(\mathbf{W}^{(1)^T}\mathbf{x} + \mathbf{b}^{(1)}\right)$
     - ★ second layer is given by $\mathbf{h}^{(2)} = g^{(2)}\left(\mathbf{W}^{(2)^T}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}\right)$

Advantage of Deeper Networks

1. Main architectural considerations: depth and width of a network.
2. Deeper networks have:
   - far fewer units in each layer.
   - far fewer parameters.
   - often generalize well to the test set.
   - but are often more difficult to optimize.
3. Ideal network architecture should be found via experimentation guided by validation set error.

Universal Approximation Theorem[1]

Theorem

A feed-forward network with a single hidden layer containing a finite number of neurons can approximate any continuous functions on compact subsets of $\mathbb{R}^n$, under mild assumptions on the activation function.

1. Simple neural networks can represent a wide variety of interesting functions when given appropriate parameters.
2. However, it does not touch upon the algorithmic learnability of those parameters.
   - Optimizing algorithms may not find the parameters.
   - May choose wrong function due to overfitting.

[1] More details here: https://www.deep-mind.org/2023/03/26/the-universal-approximation-theorem/.

FNN & No Free Lunch[2]

FNN

Feed-forward networks provide a universal system for representing functions:

- Given a continuous function, there is a feed-forward network that approximates the function.

No Free Lunch

There is no universal procedure for examining a training set of specific examples and choosing a function that will generalize to points not in training set.

[2] More details: https://en.wikipedia.org/wiki/No_free_lunch_in_search_and_optimization

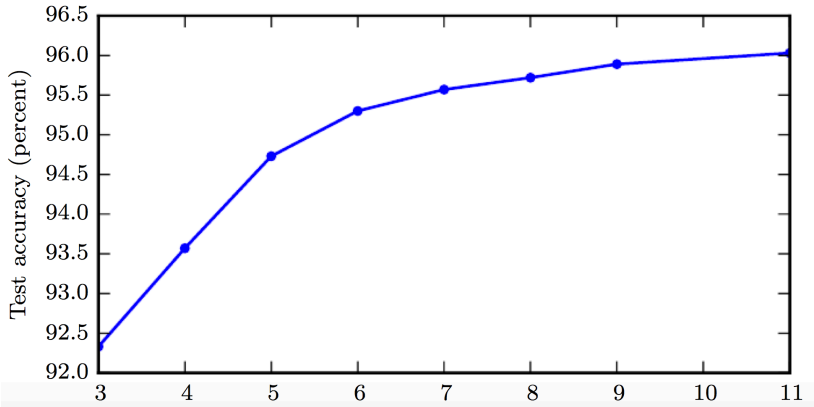
On Size of Network[3]

1. Universal Approximation Theorem implies there is a network large enough to achieve any degree of accuracy:
   - but does not say how large the network will be.
2. Some bounds on size of the single-layer network exist for a broad class of functions, but worst case is exponential number of hidden units w.r.t. data dimension.
3. Using deeper models can reduce number of units required and reduce generalization error:
   - Some families of functions can be represented efficiently if depth $> d$ but require much larger model if depth $< d$.
   - Still a challenging task for learning theory community.

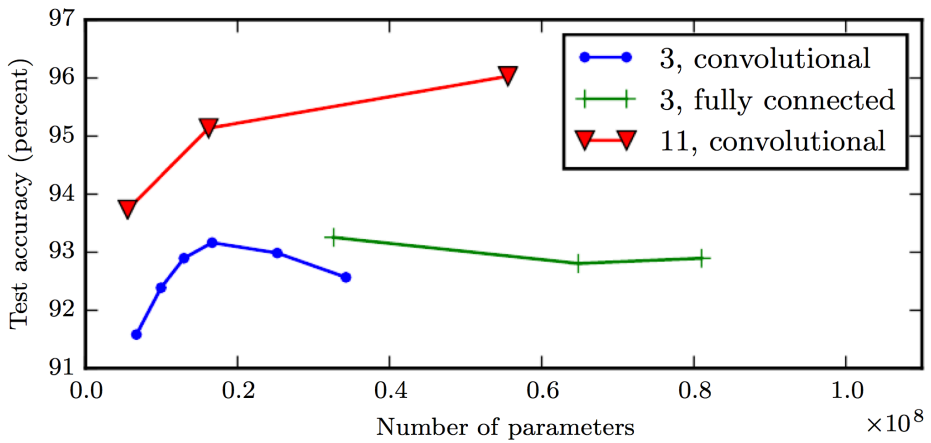
[3] More details: https://blog.claytonsanford.com/2021/08/15/hssv21.html

Empirical Justification for Deep

- CNN for street-view image recognition.
- Deeper networks perform better.



Test accuracy consistently increases with depth



Increasing parameters without increasing depth is not as effective

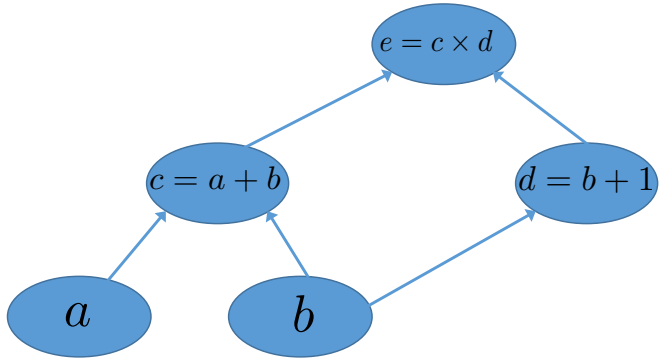
Computational Graphs

How to apply SGD in Deep Neural Networks?

1. Until now we have learned how to defined an FNN, but how to learn the model given data?
   - Learning by stochastic gradient descent (SGD).
2. To apply SGD, we need the gradients of the parameters of the neural networks for the loss function.
3. How to calculate the gradients in a general framework?
   - Backpropagation (BP).

Graph of a Math Expression

1. How to make computers understand math expressions?
2. Computational graphs are a nice way to express math expressions.
3. Consider the expression: $e = (a + b) \times (b + 1)$



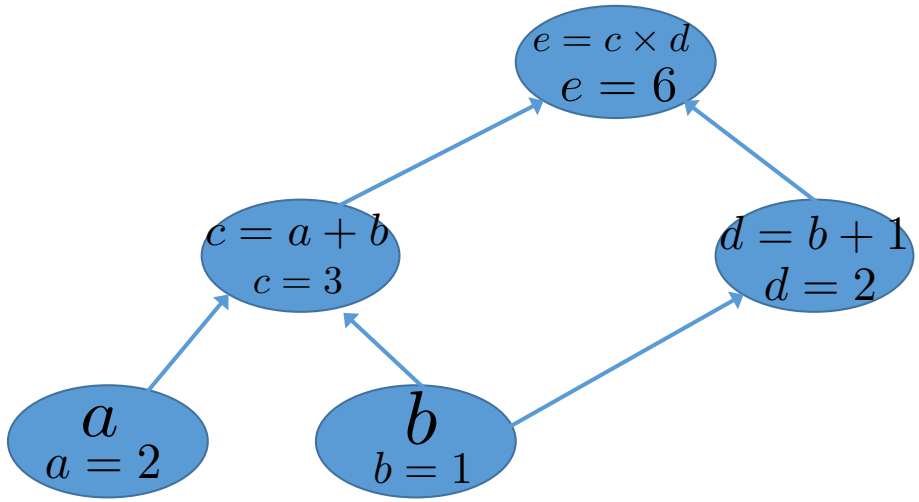
3. Introduce intermediate variables for results of each operation:

$$c = a + b; \quad d = b + 1; \quad e = c \times d$$

4. Construct a graph corresponding to these expressions:
   - operations and inputs are nodes
   - values used in operations are directed edges

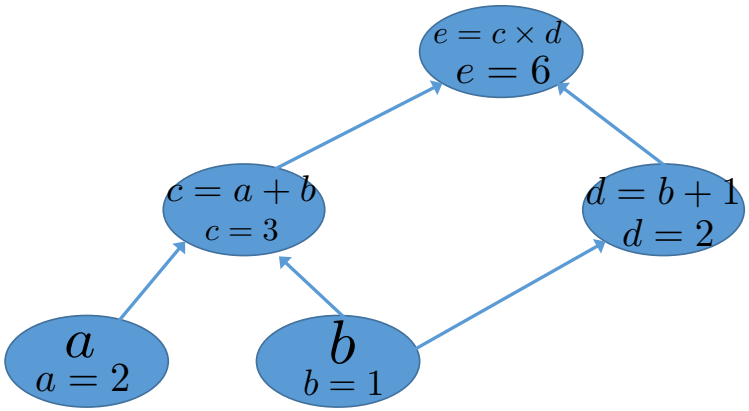
Evaluating the expression

1. Set the input variables to values and compute nodes up through the graph.
2. For $a = 2$ and $b = 1$:
3. Expression evaluates to 6.



Computational Graph Language

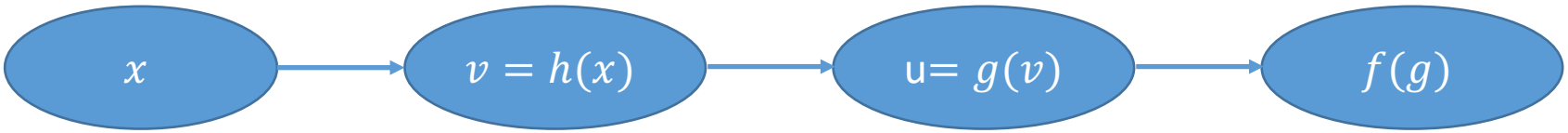
1. To describe backpropagation more precisely, we review the computational graph language.
2. Each node is either
   - a variable: scalar, vector, matrix, tensor, or other type
   - or an operation
     - ★ simple function of one or more variables
     - ★ functions more complex than operations are obtained by composing operations
   - if variable y is computed by applying operation to variable x then draw directed edge from x to y.



Derivatives of Composite function with Chain Rule

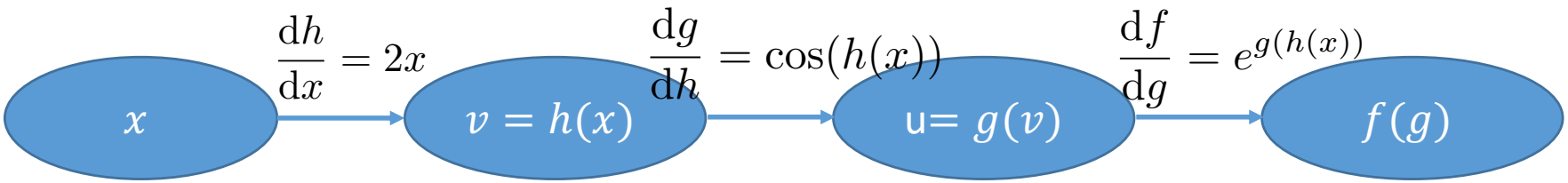
1. Computational graph provides a convenient way to calculate derivatives.
2. Let a composite function be $f(g(h(x))) \triangleq f \circ g \circ h(x)$, where $f(x) = e^x$, $g(x) = \sin(x)$, $h(x) = x^2$.
3. Using chain rule, we have

$$\begin{aligned}\frac{\mathrm{d}f}{\mathrm{d}x} &= \frac{\mathrm{d}f}{\mathrm{d}g} \cdot \frac{\mathrm{d}g}{\mathrm{d}h} \cdot \frac{\mathrm{d}h}{\mathrm{d}x} \\ &= e^{g(h(x))} \cdot \cos(h(x)) \cdot 2x \\ &= e^{\sin(x^2)} \cdot \cos(x^2) \cdot 2x\end{aligned}$$



Derivatives using Computational Graph

1. All we need to do is get the derivative of each node w.r.t. each of its inputs.



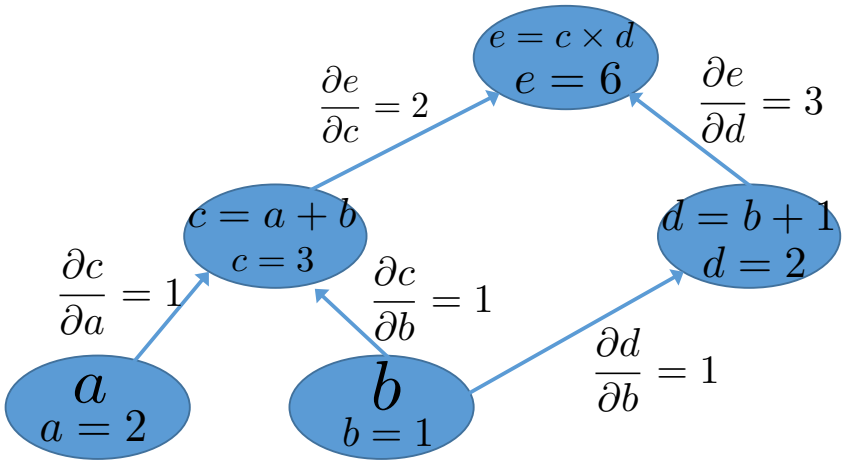
2. Get the derivative by multiplying the "connection" derivatives.

$$\frac{df}{dx} = \frac{df}{dg} \cdot \frac{dg}{dh} \cdot \frac{dh}{dx} = e^{\sin(x^2)} \cdot \cos(x^2) \cdot 2x$$

Derivatives for $e = (a + b) \times (b + 1)$ with Computational Graph

1. Indirection connection by chain rule:

$$\frac{\partial e}{\partial a} = \frac{\partial e}{\partial c} \cdot \frac{\partial c}{\partial a} = 2 \times 1 = 2$$



2. The general rule (with multiple paths) is: sum over all possible paths from one node to the other while multiplying derivatives on each path

$$\frac{\partial f(a(x), b(x))}{\partial x} = \frac{\partial f(a, b)}{\partial a}\frac{\partial a(x)}{\partial x} + \frac{\partial f(a, b)}{\partial b}\frac{\partial b(x)}{\partial x}$$

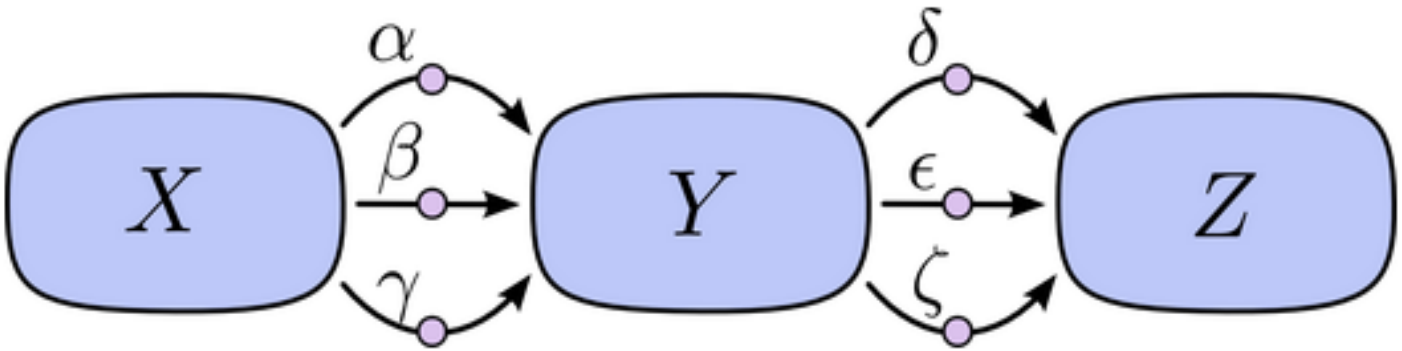
   - *e.g.*, two paths from e to b

$$\frac{\partial e}{\partial b} = \frac{\partial e}{\partial c} \cdot \frac{\partial c}{\partial b} + \frac{\partial e}{\partial d} \cdot \frac{\partial d}{\partial b}$$
$$= 2 \times 1 + 3 \times 1 = 5$$

Naive Implementation of Derivative on Computational Graph

1. To compute the derivative of a node "*a*" w.r.t. another node "*b*", we need to find the number of paths from *b* to *a*.
   - This would lead to combinatorial explosion, *i.e.*, the number of path grows exponentially.
2. To get derivative $\frac{\partial Z}{\partial X}$, we need to sum over $3 \times 3 = 9$ paths:

$$\frac{\partial Z}{\partial X} = \alpha\delta + \alpha\epsilon + \alpha\varsigma + \beta\delta + \beta\epsilon + \beta\varsigma + \gamma\delta + \gamma\epsilon + \gamma\varsigma$$



3. Backpropagation is an efficient way of computing the derivatives on computational graphs, especially for deep neural networks.

How to calculate gradients in any deep neural network?

How to implement backpropagation?

Backpropagation

Importance of Backpropagation

1. Backprop is a technique for computing derivatives quickly by avoiding duplicated computations.
2. It is the key algorithm that makes training deep models computationally tractable.
3. For modern neural networks it can make training gradient descent 10 million times faster relative to naive implementation:
   - it is the difference between a model that takes a week to train instead of 200,000 years.
4. It is based on the computational graph.
5. Simply called backprop.

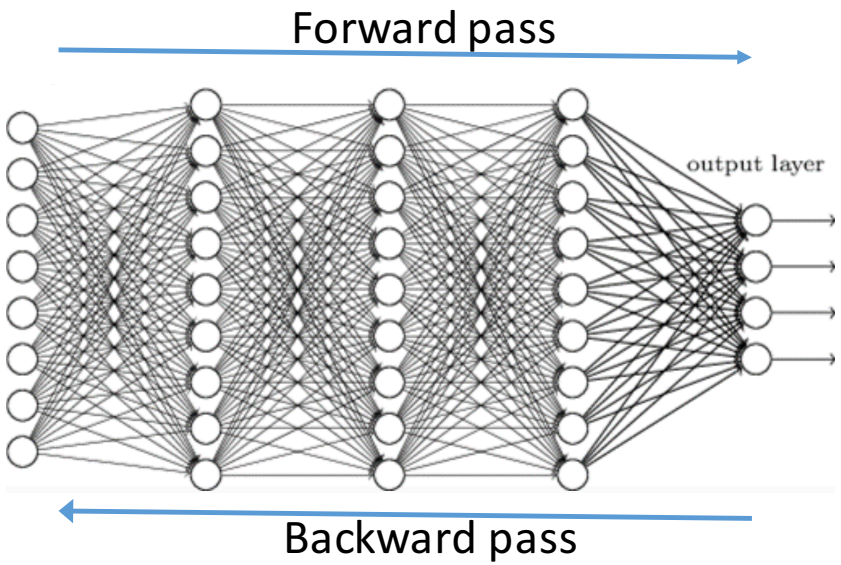
Anecdote

Geoffery Hinton and BP

- **1986**: Invented Backpropagation.
- **2017**: “Discard it, do it from the beginning!”
  - Capsule networks.
- **2022**: The forward-forward algorithm.

Basic Steps in Backpropagation

1. Two passes through the computational graph:
   - forward pass: compute the outputs of each layer.
   - backward pass: compute the gradients of parameters in each layer, based on results from the forward pass.
2. Backprop algorithm does this using a simple and inexpensive procedure.



Backprop vs. Learning

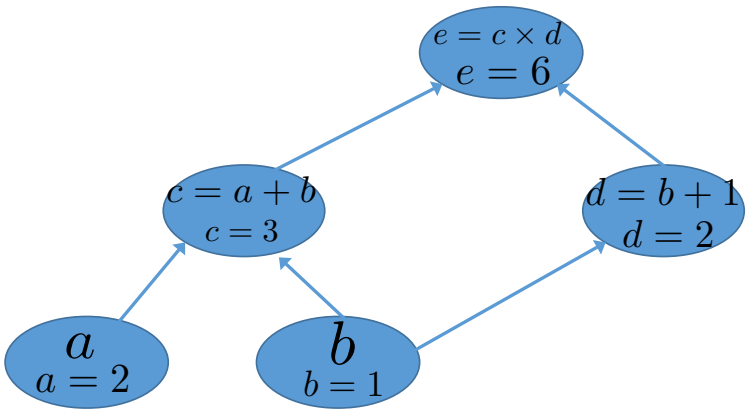
1. Backprop does not mean the whole learning algorithm.
2. Backprop only refers to the method of computing the gradient.
   - Another algorithm, such as SGD, is used to perform leaning using this gradient:

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \eta \nabla_{\mathbf{w}} J(\mathbf{w})$$

3. Often misunderstood to being specific to multilayer neural networks.
   - It can be used to compute derivatives for any function.
   - It also can be used to compute Jacobian of a function with multiple outputs.
   - We restrict to case where f has a single output.

Recap: Computational Graph

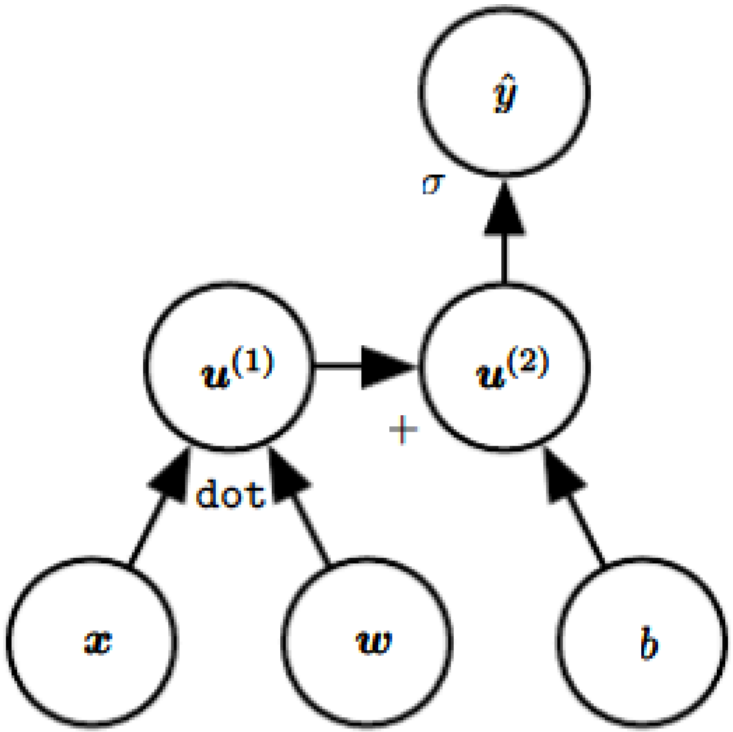
- Each node is either
  - a variable: scalar, vector, matrix, tensor, or other type
  - or an operation
    - ★ simple function of one or more variables
    - ★ functions more complex than operations are obtained by composing operations
  - if variable y is computed by applying operation to variable x then draw directed edge from x to y.



Example: Graph of Logistic Regression

$$\hat{y} = \sigma\left(\mathbf{w}^T \mathbf{x} + b\right)$$

- Variables in graph $\mathbf{u}^{(1)}$ and $\mathbf{u}^{(2)}$ are not in the original expression, but are needed in the graph.



Example: Graph of ReLU

$$\mathbf{H} = \max\left\{0, \mathbf{W}^T \mathbf{X} + \mathbf{b}\right\}$$

- Variables in graph $\mathbf{U}^{(1)}$ and $\mathbf{U}^{(2)}$ are not in the original expression, but are needed in the graph.

Recap: Calculus' Chain Rule for Scalars

- Let x be a real number; f and g be functions mapping from a real number to a real number.
- Let $y = g(x)$, $z = f(g(x)) = f(y)$, then

$$\frac{\mathrm{d}z}{\mathrm{d}x} = \frac{\mathrm{d}z}{\mathrm{d}y} \cdot \frac{\mathrm{d}y}{\mathrm{d}x}$$

Generalizing Chain Rule to Vectors

- Let $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{y} \in \mathbb{R}^n$; $f : \mathbb{R}^n \to \mathbb{R}$, and $g : \mathbb{R}^m \to \mathbb{R}^n$.
- Let $\mathbf{y} = g(\mathbf{x})$, $z = f(g(\mathbf{x})) = f(\mathbf{y})$, then

$$\frac{\mathrm{d}z}{\mathrm{d}x_i} = \sum_j \frac{\mathrm{d}z}{\mathrm{d}y_j} \cdot \frac{\mathrm{d}y_j}{\mathrm{d}x_i}$$

  - Like multiple paths when each node represents one element of a vector.
- In vector notation, this is

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

  where $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ is the $n \times m$ Jacobian matrix of g.
- Backprop algorithm consists of performing "Jacobian-gradient" product for each step of graph.

Generalizing Chain Rule to Tensors

1. Backpropagation is usually applied to tensors with arbitrary dimensionality.
2. This is almost the same as with vectors:
   - we could flatten each tensor into a vector, compute a vector-valued gradient and reshape it back to a tensor.
3. In this view backpropagation is still multiplying Jacobians by gradients:

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

Chain Rule for Tensors

1. To denote gradient of value z w.r.t. a tensor $\mathbf{X}$, we write as if $\mathbf{X}$ were a vector:
   - an element in a k-order tensor is indexed by a tuple $(i_1, \cdots, i_k)$, we abstract this away by a single (vector) variable $i \triangleq (i_1, \cdots, i_k)$ to represent the tuple.
2. For all possible tuples i, $(\nabla_{\mathbf{X}} z)_i$ gives $\frac{\partial z}{\partial X_i}$:
   - exactly same as how for all possible indices i in a vector, $(\nabla_{\mathbf{X}} z)_i$ gives $\frac{\partial z}{\partial X_i}$.
3. Chain rule for tensors: let $\mathbf{Y} = g(\mathbf{X})$ and $z = f(\mathbf{Y})$:

$$\nabla_{\mathbf{X}} z = \sum_i \underbrace{(\nabla_{\mathbf{X}} Y_i)}_{\text{tensor}} \underbrace{\frac{\partial z}{\partial Y_i}}_{\text{scalar}}$$

Backprop is Recursive Chain Rule

1. Backprop is obtained by recursively applying the chain rule.
2. Using the chain rule, it is straightforward to write expression for gradient of a scalar w.r.t. any node in graph producing that scalar.
3. However, evaluating that expression on a computer has some extra considerations:
   - *e.g.*, many subexpressions may be repeated several times within overall expression.
   - should we store subexpressions or recompute them?

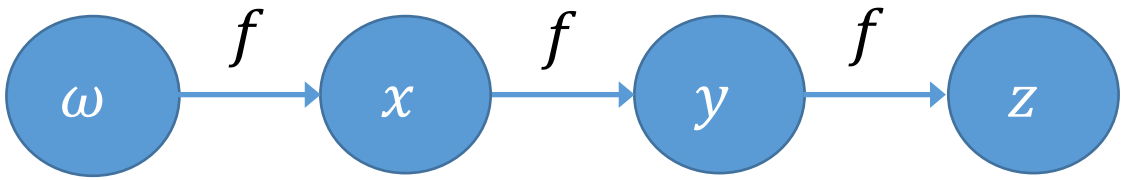
Example of repeated subexpressions

1. Let ω be the input to the graph; Use same function $f : \mathbb{R} \to \mathbb{R}$ at every step: $x = f(\omega), \quad y = f(x), \quad z = f(y)$.

$$\frac{\partial z}{\partial \omega} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x} \cdot \frac{\partial y}{\partial \omega} = f'(y)f'(x)f'(\omega) \qquad (1)$$

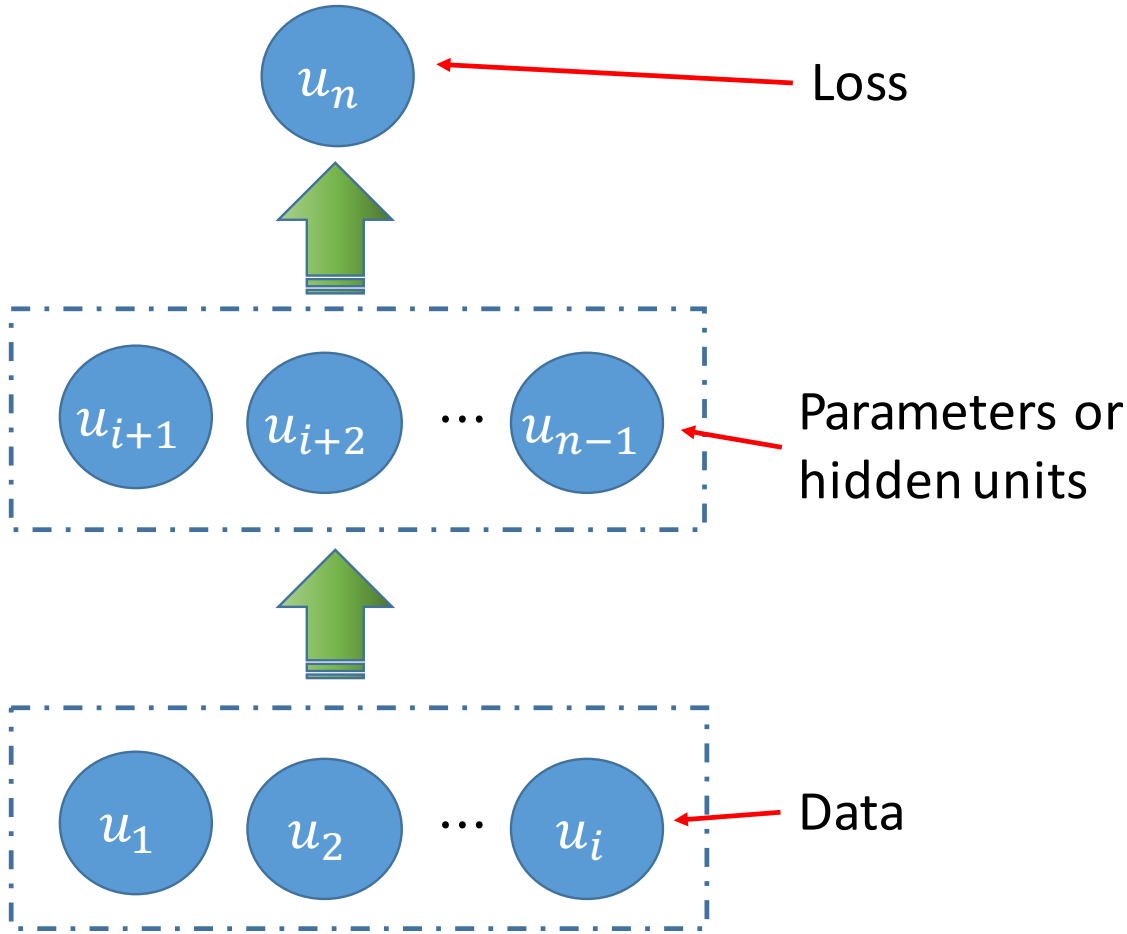
$$= f'(f(f(\omega)))f'(f(\omega))f'(\omega) \qquad (2)$$

2. We can also use (2) to compute the derivative, where $f(\omega)$ is recomputed each time it is needed.
3. We can use (1) to compute the derivative, where $f(\omega)$ is computed once and stored it in x.
   - For low memory, (1) preferable: reduced runtime.
   - (2) is also valid chain rule, useful for limited memory.
   - (1) is the approach taken by backprop.
4. For complicated graphs as in deep neural networks, (2) exponentially wastes computations for repeated subexpressions.



Basic Backprop Algorithm

- Assume each u_i is a scalar.
- Each $\{u_j\}$ is connected by a subset of other $\{u_k\}_{k \leq j}$, dented as $Pa(u_j)$.
- For Each $\{u_j\}_{j>i}$, $u_j = f^{(j)}(Pa(u_j))$.
- The basic backprop consists of two procedures: the forward pass and the backward pass.

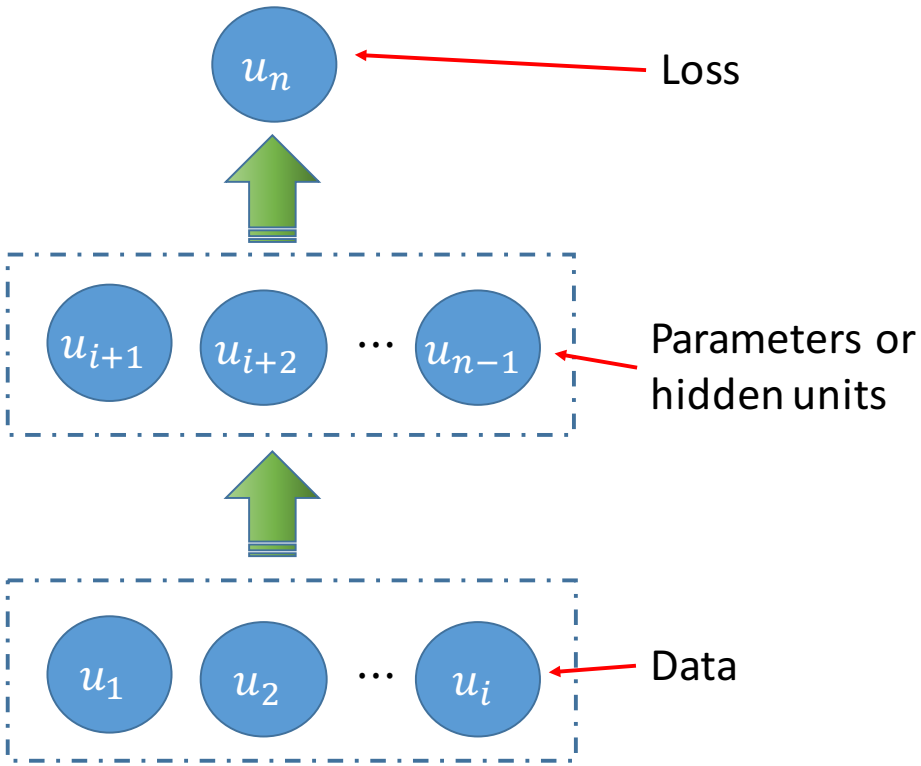


Forward Propagation Computation

Input: Data $\{x_i\}$
Output: Loss $\{u_n\}$

for $j = 1, \ldots, i$ **do**
  $u_j = x_j$
end
for $j = i + 1, \ldots, n$ **do**
  $u_j = f^{(j)}(Pa(u_j))$
end

Algorithm 1: Forward Propagation



- Algorithm specifies a computational graph $\mathcal{G}$.
- Computation in reverse order gives back-propagation computational graph $\mathcal{B}$.

Backward Propagation Computation

1. Proceeds exactly in reverse order of computation in $\mathcal{G}$.
2. Each node u_k in $\mathcal{B}$ computes the derivative $\frac{\partial u_n}{\partial u_k}$.
3. Done by using the chain rule w.r.t. the scalar output u_n:

$$\frac{\partial u_n}{\partial u_k} = \sum_{j:u_k \in Pa(u_j)} \underbrace{\frac{\partial u_n}{\partial u_j}}_{(1)} \cdot \underbrace{\frac{\partial u_j}{\partial u_k}}_{(2)},$$

where all derivatives in the summation can be computed:

- (1) has been computed in previous step.
- (2) is easily computed given the mapping from u_k to u_j.
- acting like dynamic programming.

Backpropagation Algorithm

Input: Data $\{x_i\}$
Output: All derivatives $\{\frac{\partial u_n}{\partial u_k}\}$

Run forward propagation to obtain network activations (outputs)
Initialize grad-table, a data structure that will store derivatives that have been computed, The entry grad-table[u_k] will store the computed value of $\frac{\partial u_n}{\partial u_k}$
grad-table[u_k] $\leftarrow 1$
for $k = n-1, \ldots, 1$ **do**
  grad-table[u_k] $= \sum_{j:u_k \in Pa(u_j)}$ grad-table[u_j] $\frac{\partial u_j}{\partial u_k}$ (*)
end

Algorithm 2: Backpropagation Algorithm

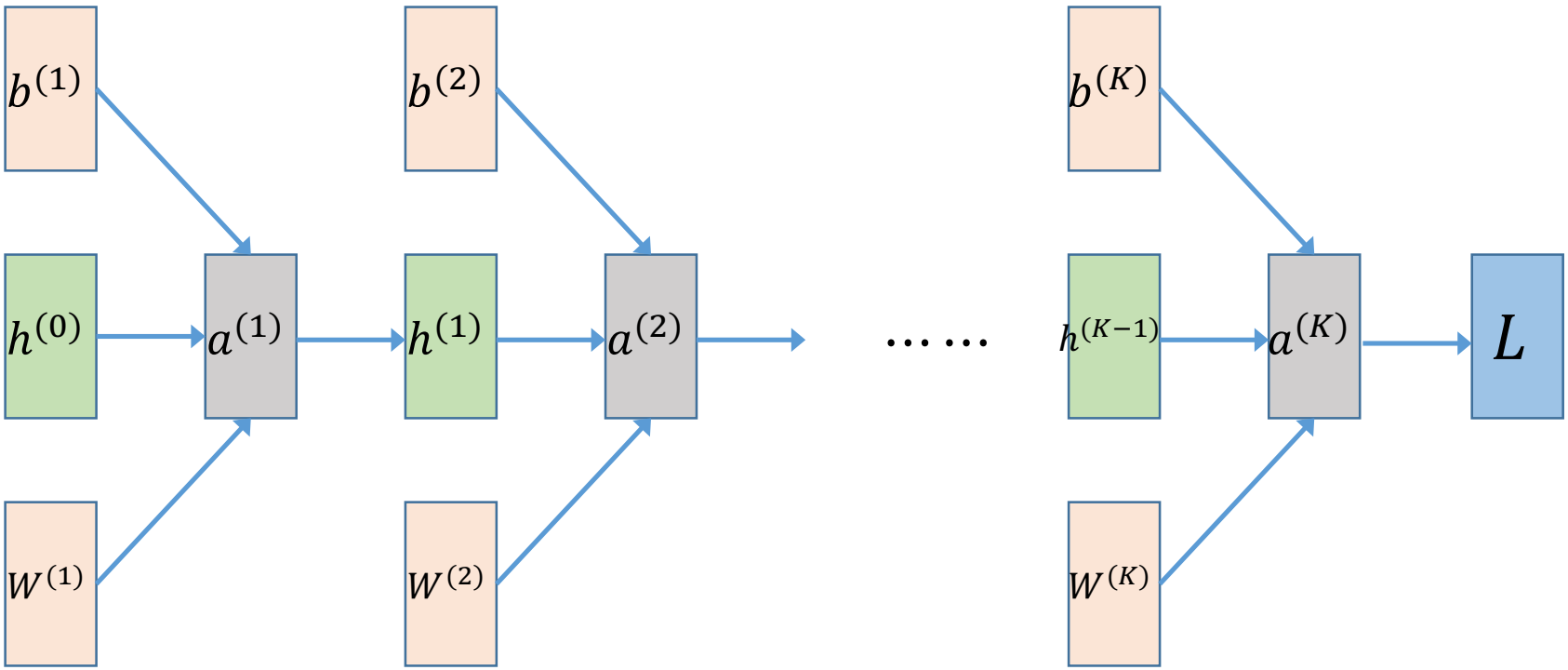
Step (*) computes $\frac{\partial u_n}{\partial u_k} = \sum_{j:u_k \in Pa(u_j)} \frac{\partial u_n}{\partial u_j} \cdot \frac{\partial u_j}{\partial u_k}$.

Computational Complexity & Generalization

1. Computational cost is proportional to the number of edges in the graph (same as for forward pass):
   - each edge is visited a limited number of times.
   - complexity proportional to #edges in the computational graph.
2. Backpropagation thus avoids exponential explosion in repeated subexpressions
   - by simplifications on the computational graph.
3. It can be generalized to the case of intermediate tensor-output.

Backprop in Fully Connected MLP

- Maps parameters to supervised loss $L(\hat{y}, y)$ associated with a single training example $(\mathbf{x}, y)$ with $\hat{y}$ the output when $\mathbf{x}$ is the input.



Forward Prop for MLP (without regularization)

Input: Network depth K; Weight matrices $\mathbf{W}^{(i)}, i \in \{1, \ldots, K\}$; bias parameters $\mathbf{b}^{(i)}, i \in 1, \ldots, K$; Input data $\mathbf{x}$ and target output y;
Input: Activation function $f^{(k)}, k \in \{1, \ldots, K\}$
Output: Loss function L

$\hbar^{(0)} = \mathbf{x}$
for $k = 1, \ldots, K$ **do**
  $\mathbf{a}^{(k)} = \mathbf{W}^{(k)^T} h^{(k-1)} + \mathbf{b}^{(k)}$
  $\mathbf{h}^{(k)} = f^{(k)}(\mathbf{a}^{(k)})$
end
$\bar{y} = \mathbf{h}^{(l)}$
Return $L(\bar{y}, y)$

Algorithm 3: Forward prop through typical deep NN

- $L(\bar{y}, y)$ is the loss function w.r.t. the network output $\bar{y}$ and the target y, *e.g.*, the cross entropy.

Backward Prop for MLP

Forward prop through typical deep NN

$\mathbf{g} \leftarrow \nabla_{\bar{y}} J = \nabla_{\bar{y}} L(\bar{y}, y)$ **(gradient of hidden layers, *e.g.*, derivative of softmax on the top-layer)**

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

Caculate the gradient of the pre-nonlinearity activation $\mathbf{a}^{(k)}$ **(element-wise multiplication if** f **is elementwise)**

//g stores either $\nabla_{\mathbf{a}^{(k)}} J$ **or** $\nabla_{\mathbf{h}^{(k)}} J$

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

Compute the gradients on weights and biases (including the regularization term: $\mathbf{a}^{(k)} = \mathbf{W}^{(k)} \mathbf{h}^{(k-1)} + b^{(k)}$**)**

$$\nabla_{b^{(k)}} J = \nabla_{\mathbf{a}^{(k)}} J \odot \frac{\partial \mathbf{a}^{(k)}}{\partial b^{(k)}} = \mathbf{g}$$

$$\nabla_{\mathbf{W}^{(k)}} J = \nabla_{\mathbf{a}^{(k)}} J \odot \frac{\partial \mathbf{a}^{(k)}}{\partial \mathbf{W}^{(k)}} = \mathbf{g}\, \mathbf{h}^{k-1\,T}$$

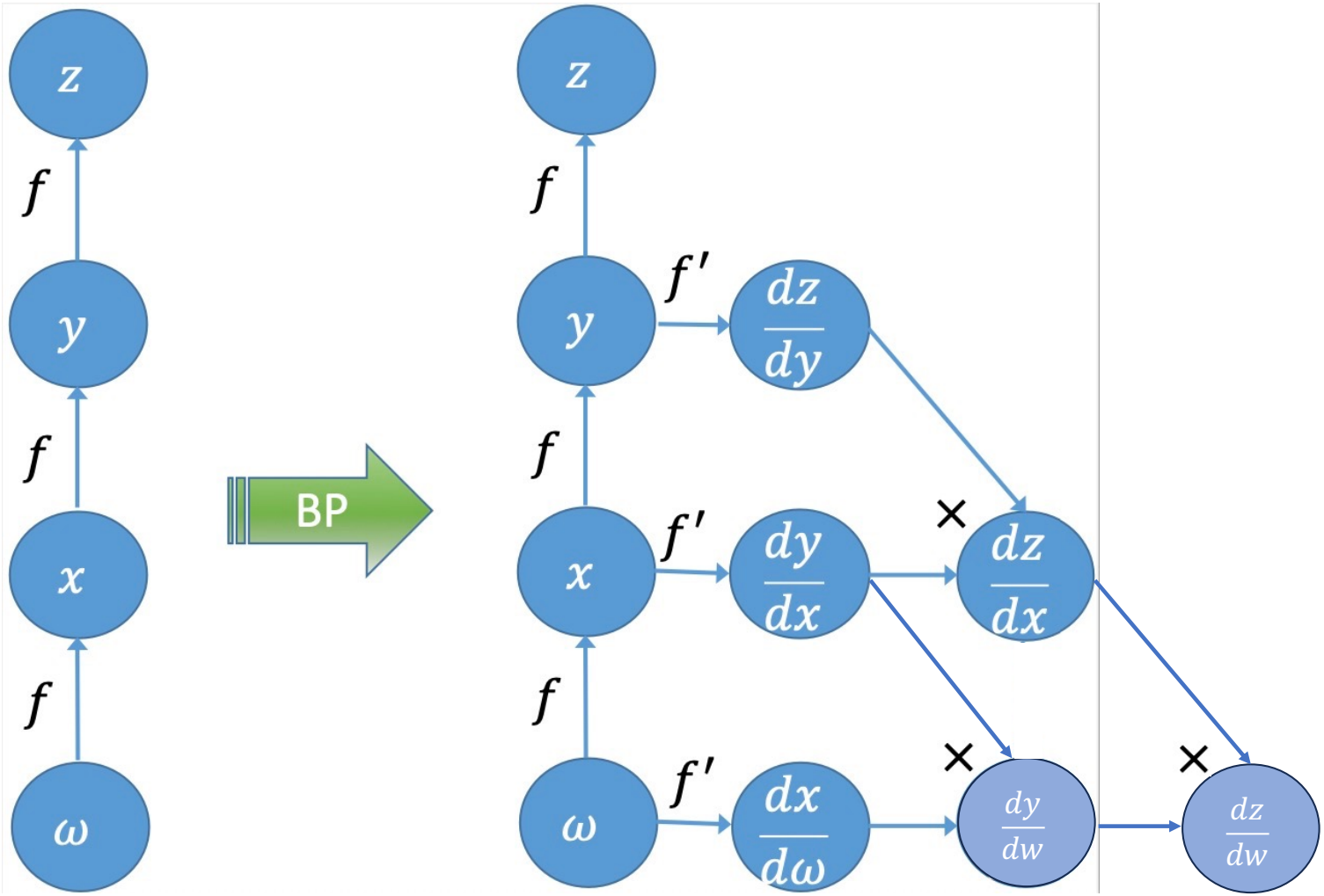
Propagate the gradients w.r.t. the next lower-level hidden layer's activations: $\mathbf{g} = \nabla_{\mathbf{h}^{(k-1)}} J = \frac{\partial \mathbf{a}^{(k)}}{\partial \mathbf{h}^{(k-1)}} \cdot \nabla_{\mathbf{a}^{(k)}} J = \mathbf{W}^{(k)\,T}\, \mathbf{g}$

end

Two Approaches to Implement BP

1. Symbol-to-number differentiation:
   - Input: computational graph and numerical values of inputs to graph.
   - Output: a set of numerical values describing gradient at the input values.
   - Used by libraries: PyTorch/Torch and Caffe.
   - See previous examples.
2. Symbol-to-symbol differentiation:
   - Input: computational graph.
   - Output: Add additional nodes to the graph.
   - Used by libraries: Theano and Tensorflow.
   - BP does not need to ever access any actual numerical values:
     - ★ instead it adds nodes to a computational graph describing how to compute the derivatives for any specific numerical values.
     - ★ a generic graph evaluation engine can later compute derivatives for any specific numerical values.

Example: Symbol-to-symbol Derivatives



Advantages/Disadvantages of Symbol-to-Symbol Approach

- Advantages:
  1. Derivatives are described in the same language as the original expression.
  2. Because the derivatives are just another computational graph, it is possible to run back-propagation again:
     - ★ differentiating the derivatives.
     - ★ yields higher-order derivatives.
- Disadvantages:
  - More computational overhead
  - More complex in implementation and usage

How is BP Implemented in General

Constructing the Graph

1. Each node in graph $\mathcal{G}$ corresponds to a variable.
2. Each variable is described by a tensor $\mathbf{V}$, subsuming scalars, vectors and matrices.
3. For each node, several routines are implemented associated with the node.

Subroutines Associated with V

- get_operation ($\mathbf{V}$):
  - returns the operation that computes $\mathbf{V}$ represented by the edges coming into $\mathbf{V}$.
  - *e.g.*, suppose we have a variable that is computed by matrix multiplication $\mathbf{C} = \mathbf{A}\,\mathbf{B}$, then get_operation ($\mathbf{V}$) returns a pointer to an instance of the corresponding C++ class implemented the multiplication.
- get_consumers ($\mathbf{V}, \mathcal{G}$):
  - returns list of variables that are children of $\mathbf{V}$ in the computational graph $\mathcal{G}$.
- get_inputs ($\mathbf{V}, \mathcal{G}$):
  - returns list of variables that are parents of $\mathbf{V}$ in the computational graph $\mathcal{G}$.

bprop Operation

1. Each operation (node) *op* is also associated with a *bprop* operation.
2. *op.bprop* operation can compute a Jacobian-vector product as described by: $\nabla_{\mathbf{x}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T \nabla_{\mathbf{y}} z$ for input $\mathbf{x}$ and output $\mathbf{y}$.
3. This is how the backpropagation algorithm can achieve great generality:
   - each operation is responsible for knowing how to backpropagate through the edges in the graph that it participates in (local operation).

Inputs, Outputs of bprop

1. Backprogation algorithm itself does not need to know any differentiation rules:
   - op.bprop implements it.
   - it only needs to call each operation's op.bprop rules with the right arguments.
2. Formally op.bprop(inputs, $\mathbf{X}$, $\mathbf{D}$) must return:

$$\nabla_{\mathbf{X}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{X}}\right)^T \nabla_{\mathbf{y}} z = \sum_i \left(\nabla_{\mathbf{X}} y_i\right) \mathbf{D}_i$$

   - "inputs" is a list of inputs that are supplied to the operation, $\mathbf{y}$ is a math function that the operation implements.
   - $\mathbf{X}$ is the input whose gradient we wish to compute.
   - $\mathbf{D}$ is the gradient on the output of the operation, *e.g.*, gradient of last layer.
   - Just an implementation of the chain rule: $\nabla_{\mathbf{X}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{X}}\right)^T \underbrace{\nabla_{\mathbf{y}} z}_{\mathbf{D}_i}$.

Software Implementations

1. Usually provide both:
   - operations.
   - their *bprop* methods.
2. Users of software libraries are able to backpropagate through graphs built using common operations like:
   - matrix multiplication, exponents, logarithms, *etc*.
3. To add a new operation to existing library, one must derive ob.prop method manually.

Formal Backpropagation Algorithm: Outermost Skeleton

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

Inner Loop: build-grad ($\mathbf{V}, \mathcal{G}, \mathcal{G}', \mathbf{grad\text{-}table}$)

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

Computational Graph of Gradient

1. It would be large and tedious for this example.
2. One benefit of back-propagation algorithm is that it can automatically generate gradients that would be straightforward but tedious manually for a software engineer to derive.

After gradient computation, it is the responsibility of SGD or other optimization algorithm to use gradients to update parameters.