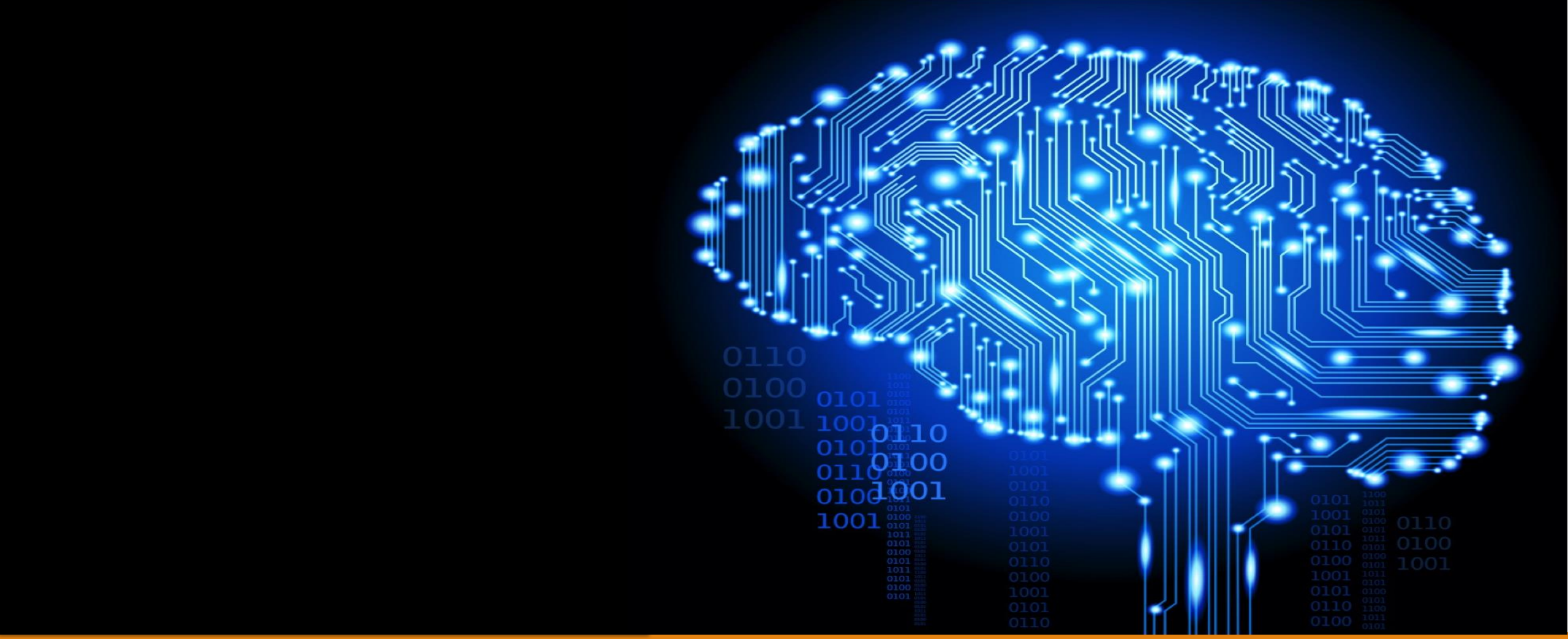


Lecture 2: Modular Learning

Deep Learning @ UvA

Lecture Overview

- Modularity in Deep Learning
- Popular Deep Learning modules
- Neural Network Cheatsheet
- Backpropagation

The Machine Learning Paradigm

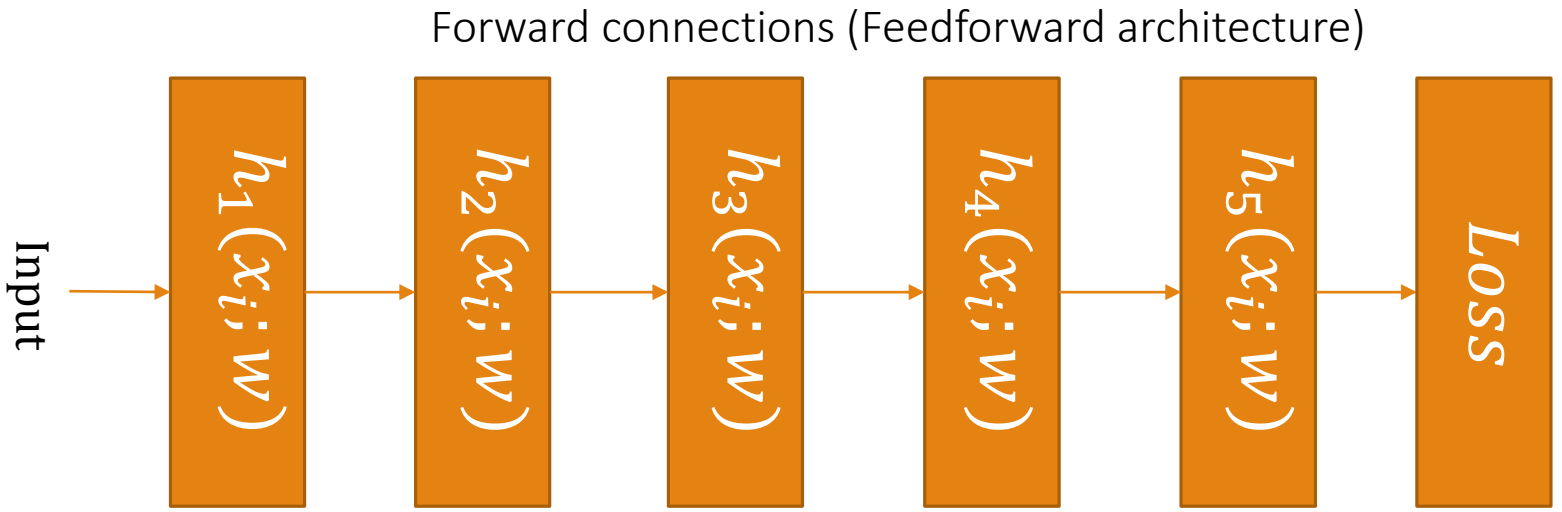
What is a neural network again?

- A family of parametric, non-linear and hierarchical representation learning functions, which are massively optimized with stochastic gradient descent to encode domain knowledge, i.e. domain invariances, stationarity.
- $a^L(x; w^1, \dots, w^L) = h^L(h^{L-1}(\dots h^1(x, w^1), w^{L-1}),\ w^L)$
  - x:input, w^l: parameters for layer l, $a^l = h^l(x, w^l)$: (non-)linear function
- Given training corpus $\{X, Y\}$ find optimal parameters

$$\mathrm{w}^* \leftarrow \arg\min_w \sum_{(x,y)\subseteq(X,Y)} \mathcal{L}(y, a^L(x))$$

Neural network models

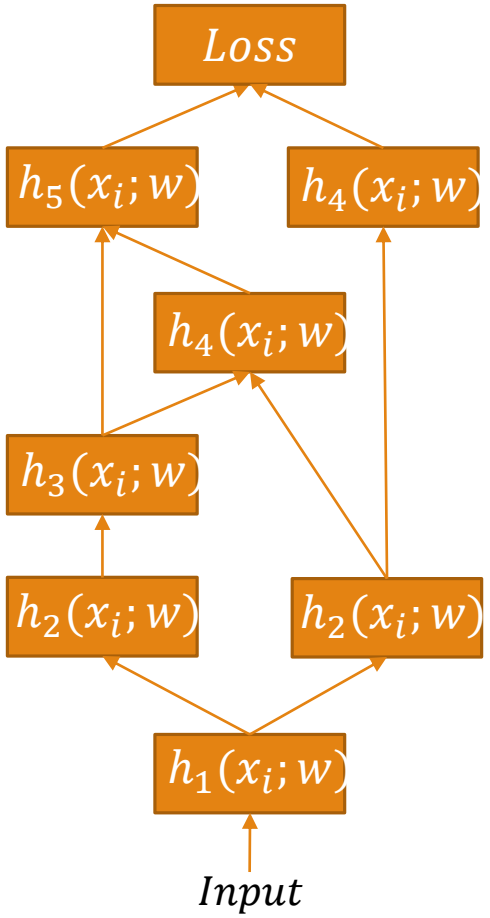
- A neural network model is a series of hierarchically connected functions
- This hierarchies can be very, very complex



Neural network models

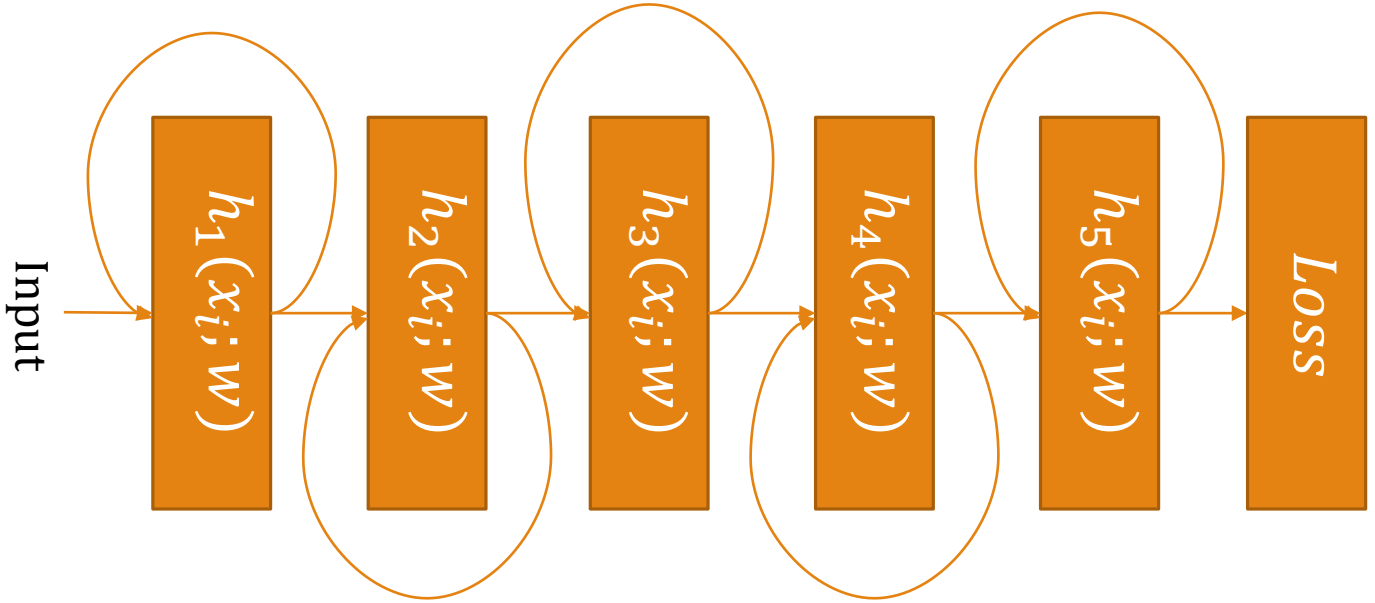
- A neural network model is a series of hierarchically connected functions
- This hierarchies can be very, very complex

Interweaved connections (Directed Acyclic Graphs- DAGNN)



Neural network models

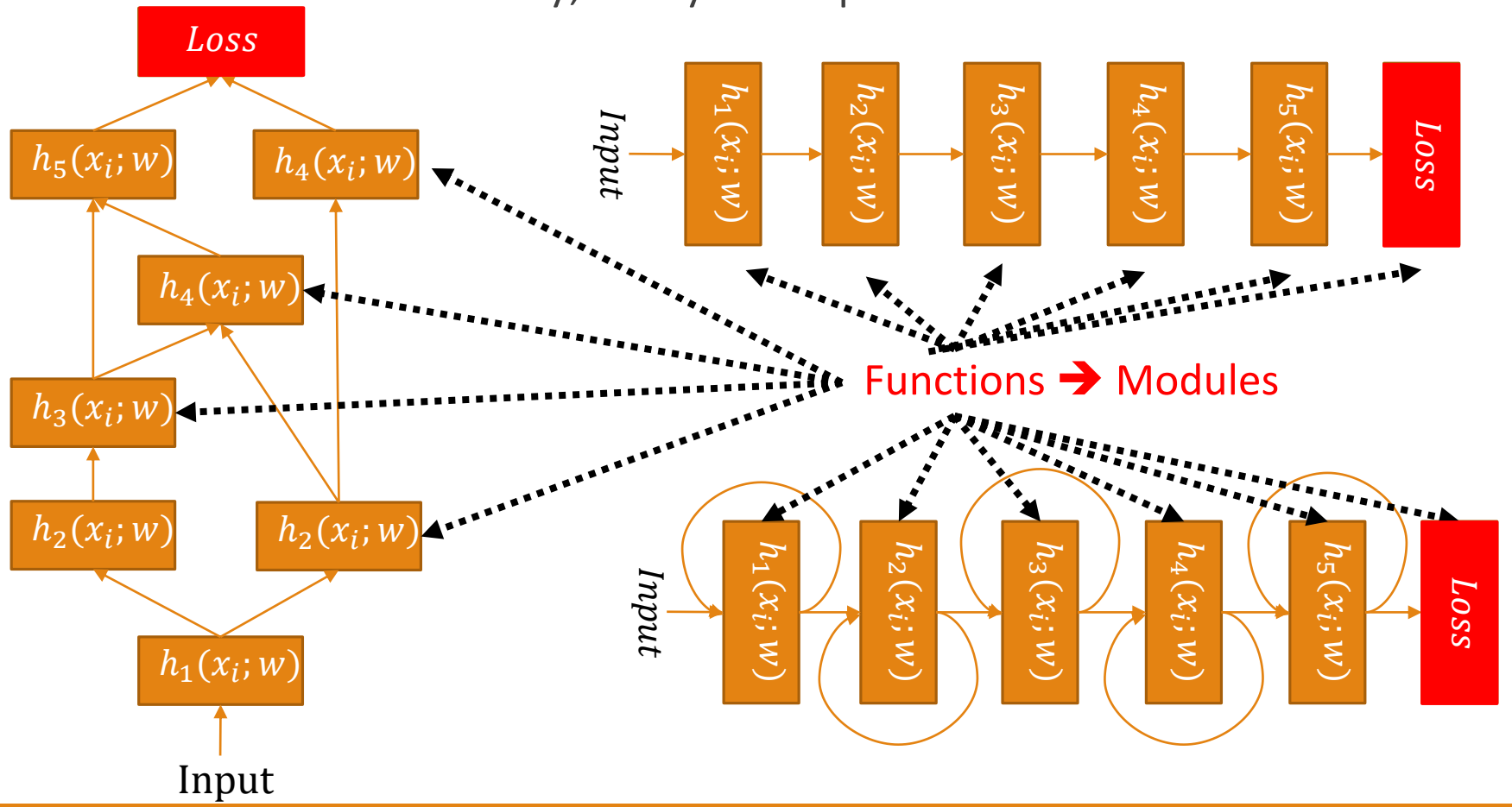
- A neural network model is a series of hierarchically connected functions
- This hierarchies can be very, very complex



Loopy connections (Recurrent architecture, special care needed)

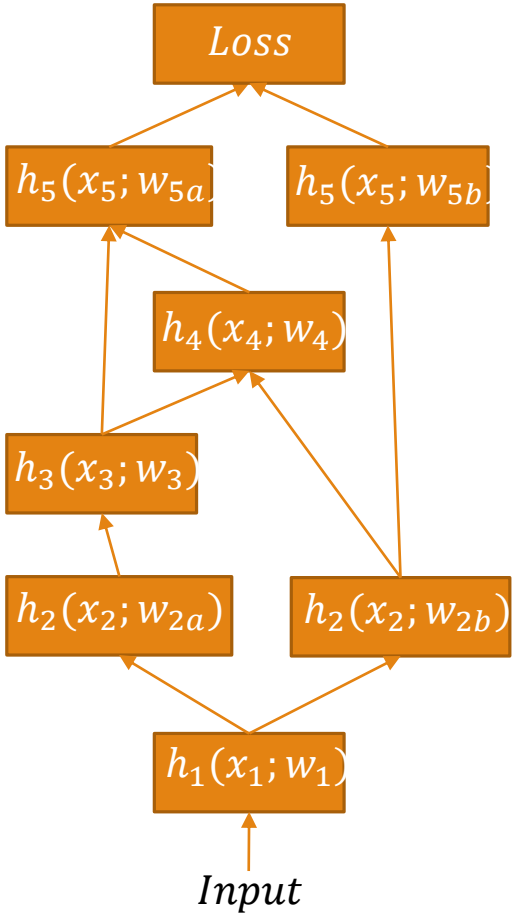
Neural network models

- A neural network model is a series of hierarchically connected functions
- This hierarchies can be very, very complex



What is a module?

- A module is a building block for our network
- Each module is an object/function $a = h(x; w)$ that
  - Contains trainable parameters w
  - Receives as an argument an input x
  - And returns an output a based on the activation function $h(\dots)$
- The activation function should be (at least) **first order differentiable (almost) everywhere**
- For easier/more efficient backpropagation → store module input
  - easy to get module output fast
  - easy to compute derivatives



Anything goes or do special constraints exist?

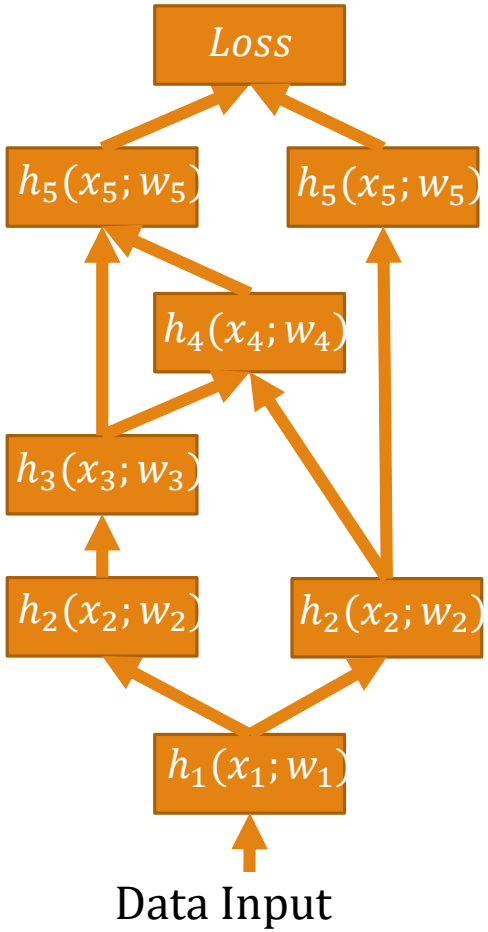
- A neural network is a composition of modules (building blocks)
- Any architecture works
- If the architecture is a feedforward cascade, no special care
- If acyclic, there is right order of computing the forward computations
- If there are loops, these form **recurrent** connections (revisited later)

Forward computations for neural networks

- Simply compute the activation of each module in the network

$$a^l = h^l(x^l; w), \text{ where } a^l = x^{l+1}$$

- We need to know the precise function behind each module $h^l(\dots)$
- Recursive operations
  - One module's output is another's input
- Steps
  - Visit modules one by one starting from the data input
  - Some modules might have several inputs from multiple modules
- Compute modules activations **with the right order**
  - Make sure all the inputs computed at the right time



How to get w? **Gradient-based learning**

- Usually Maximum Likelihood on the training set

$$\mathrm{w}^* = \arg\max_w \prod_{x,y} p_{model}(y|x;w)$$

- Taking the logarithm, the Maximum Likelihood is equivalent to minimizing the negative log-likelihood cost function

$$\mathcal{L}(w) = -\mathbb{E}_{x,y\sim\tilde{p}_{data}} \log p_{model}(y|x;w)$$

- $p_{model}(y|x)$ is last layer output

Quiz

If our last layer is the Gaussian function $N(y;\ h(w;\ x), I)$ what could be our cost function like? (Multiple answers possible)

- $\sim|y - h(w; x)|^2$
- $\sim \max\{0, 1 - y\ h(w; x)\}$
- $\sim|y - h(w; x)|_1$
- $\sim|y - h(w; x)|^2 + \lambda\Omega(\mathrm{w})$

How to get w? **Gradient-based learning**

- Usually **M**aximum **L**ikelihood in the train set

$$\mathrm{w}^* = \arg\max_{\theta} \prod_{x,y} p(y|x;w)$$

- Taking the logarithm, this means minimizing the cost function

$$\mathcal{L}(\theta) = -\mathbb{E}_{x,y\sim\tilde{p}_{data}} \log p_{model}(y|x;w)$$

- $p_{model}(y|x;w)$ is the last layer output

$$\log p_{model}(y|x) = \log \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{-|y-h(x;w)|^2}{2\sigma^2}\right)$$
$$\propto C + |y - h(x;w)|^2$$

Quiz

Why should we choose a cost function that matches the form of the last layer of the neural network?

- Otherwise one cannot use standard tools, like automatic differentiation, in packages like Tensorflow or Pytorch
- It makes the math simpler
- It avoids numerical instabilities
- It makes gradients large by avoiding functions saturating, thus learning is stable

Quiz

Why should the last network layer “click” with our cost function?

- Otherwise one cannot use standard tools, like automatic differentiation, in packages like Tensorflow or Pytorch
- It makes the math simpler
- It avoids numerical instabilities
- It makes gradients large by avoiding functions saturating, thus learning is stable

How to get w? **Gradient-based learning**

- In a neural net $p_{model}(y|x)$ is the module of the last layer (output layer)

$$\log p_{model}(y|x) = \log \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{-|y-f(\theta;x)|^2}{2\sigma^2}\right) \Longrightarrow$$

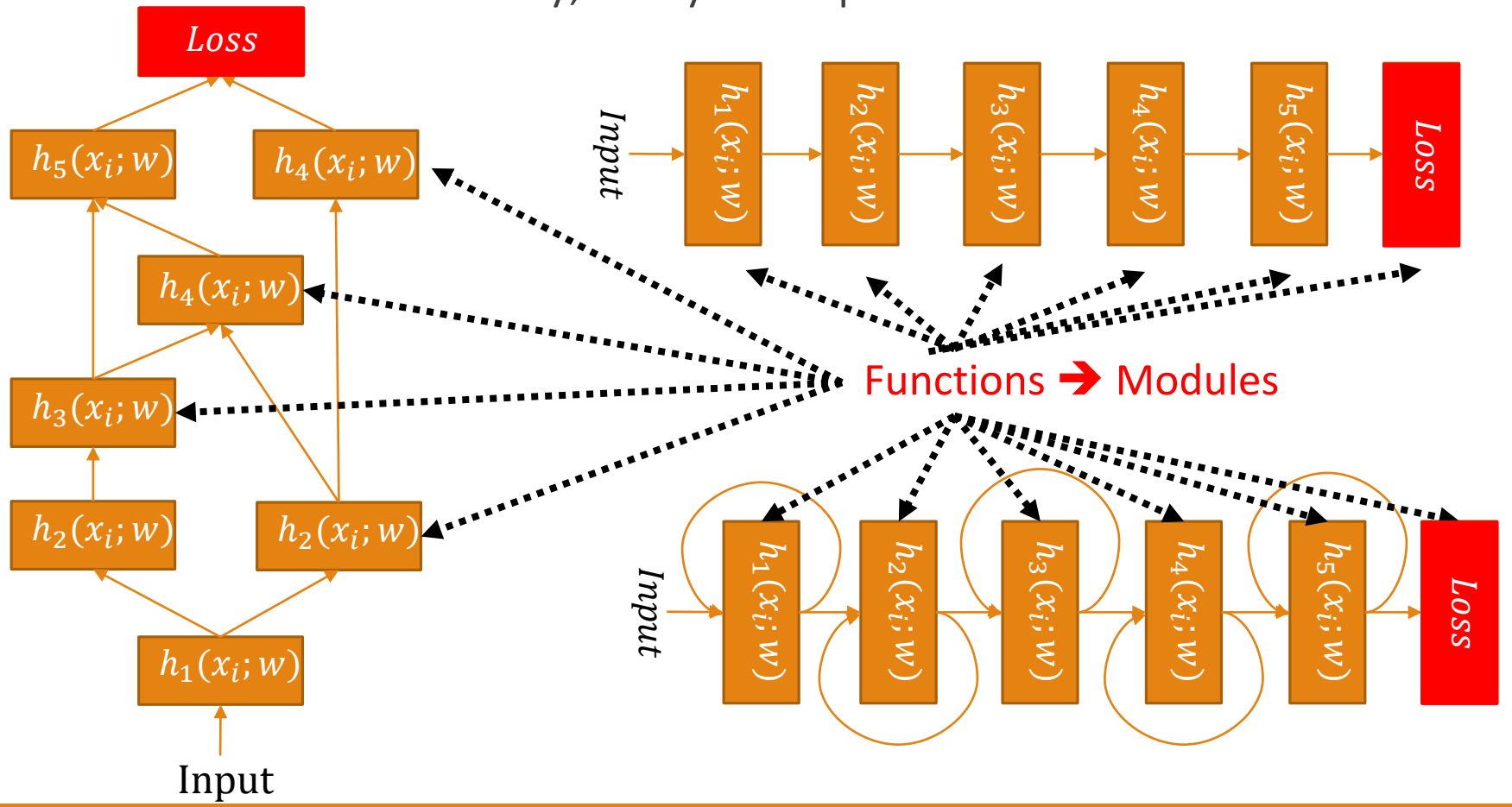
$$\log p_{model}(y|x) \propto C + |y - f(\theta; x)|^2$$

- Everything gets much simpler when the learned (neural network) function p_{model} matches the cost function $\mathcal{L}(\mathrm{w})$
- E.g the $\log$ of the negative log-likelihood cancels out the $\exp$ of the Gaussian
  - Easier math
  - Better numerical stability
  - Exponential-like activations often lead to saturation, which means gradients are almost 0, which means no learning
- That said, combining any function that is differentiable is possible
  - just not always convenient or smart

Everything is a *module*

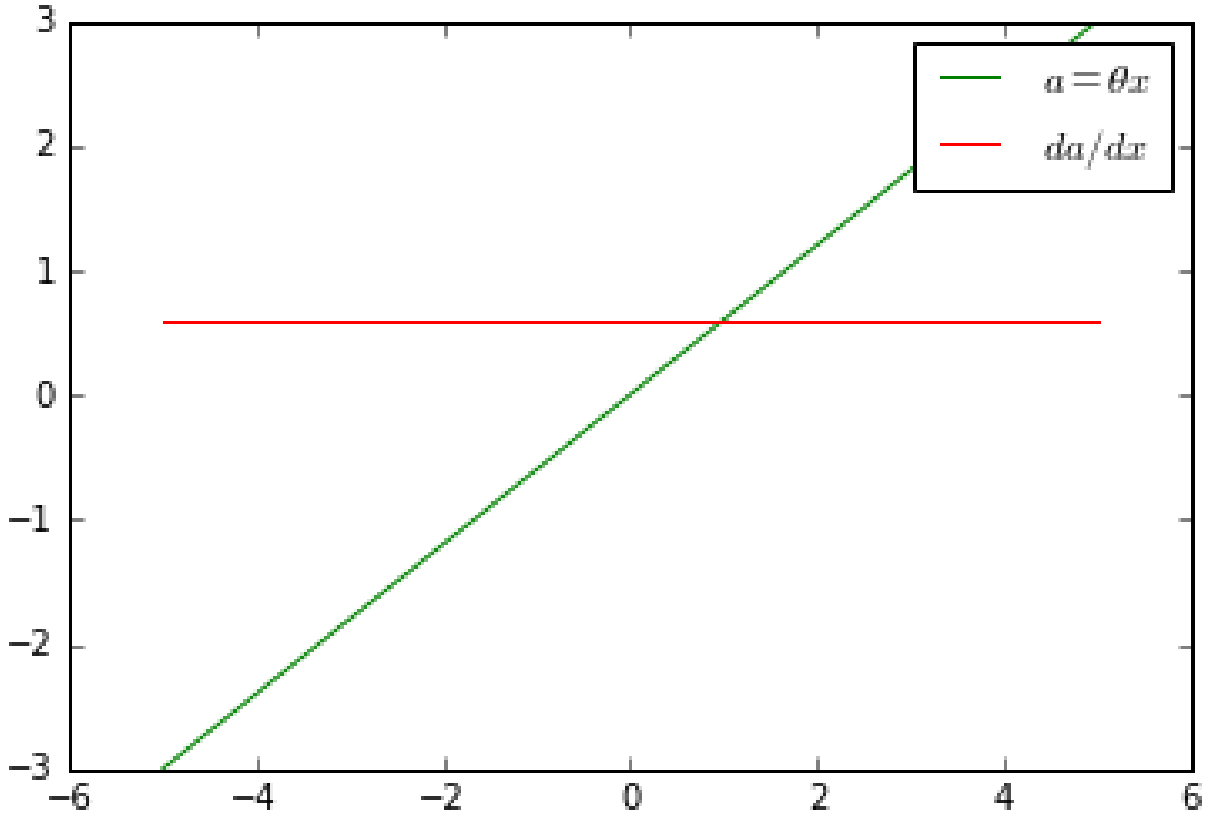
Neural network models

- A neural network model is a series of hierarchically connected functions
- This hierarchies can be very, very complex



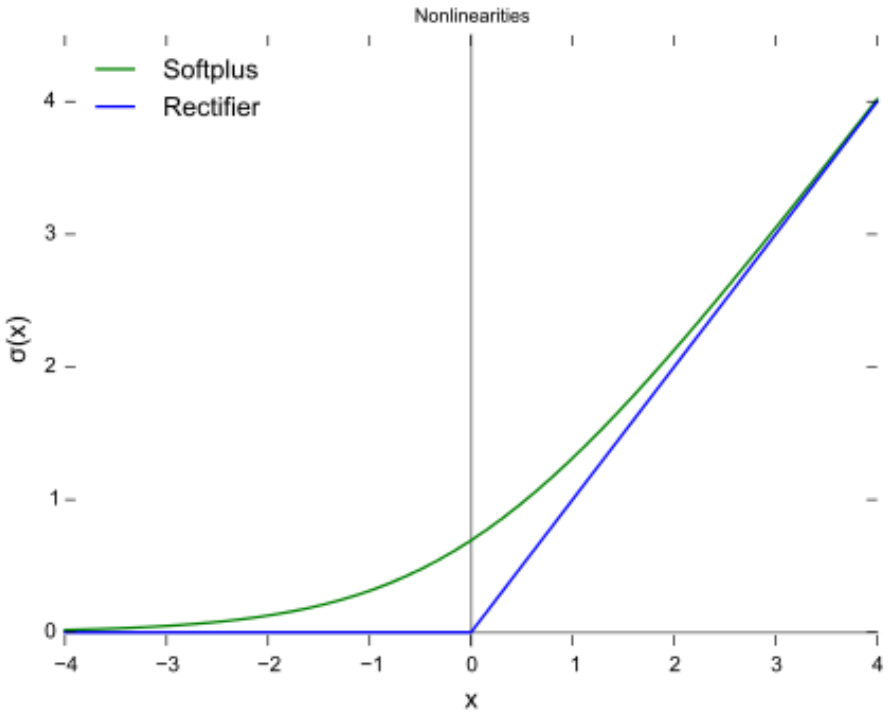
Linear module

- Activation: $a = wx$
- Gradient: $\frac{\partial a}{\partial w} = x$
- No activation saturation
- Hence, strong & stable gradients
  - Reliable learning with linear modules



Rectified Linear Unit (ReLU) module

- Activation: $a = h(x) = \max(0, x)$
- Gradient: $\frac{\partial a}{\partial x} = \begin{cases} 0, if\ x \leq 0 \\ 1, if x > 0 \end{cases}$



Quiz

What characterizes the Rectified Linear Unit?

- There is the danger the input x is consistently 0 because of a glitch. This would cause "dead neurons" that always are 0 with 0 gradient.
- It is discontinuous, so it might cause numerical errors during training
- It is piece-wise linear, so the "piece"-gradients are stable and strong
- Since they are linear, their gradients can be computed very fast and speed up training.
- They are more complex to implement, because an if condition needs to be introduced.

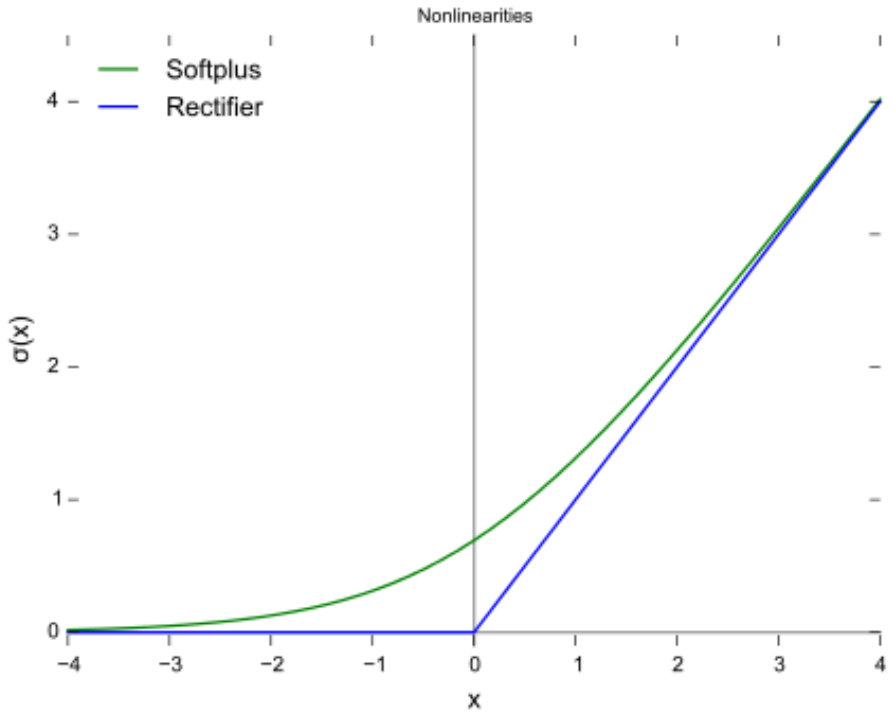
Quiz

What characterizes the Rectified Linear Unit?

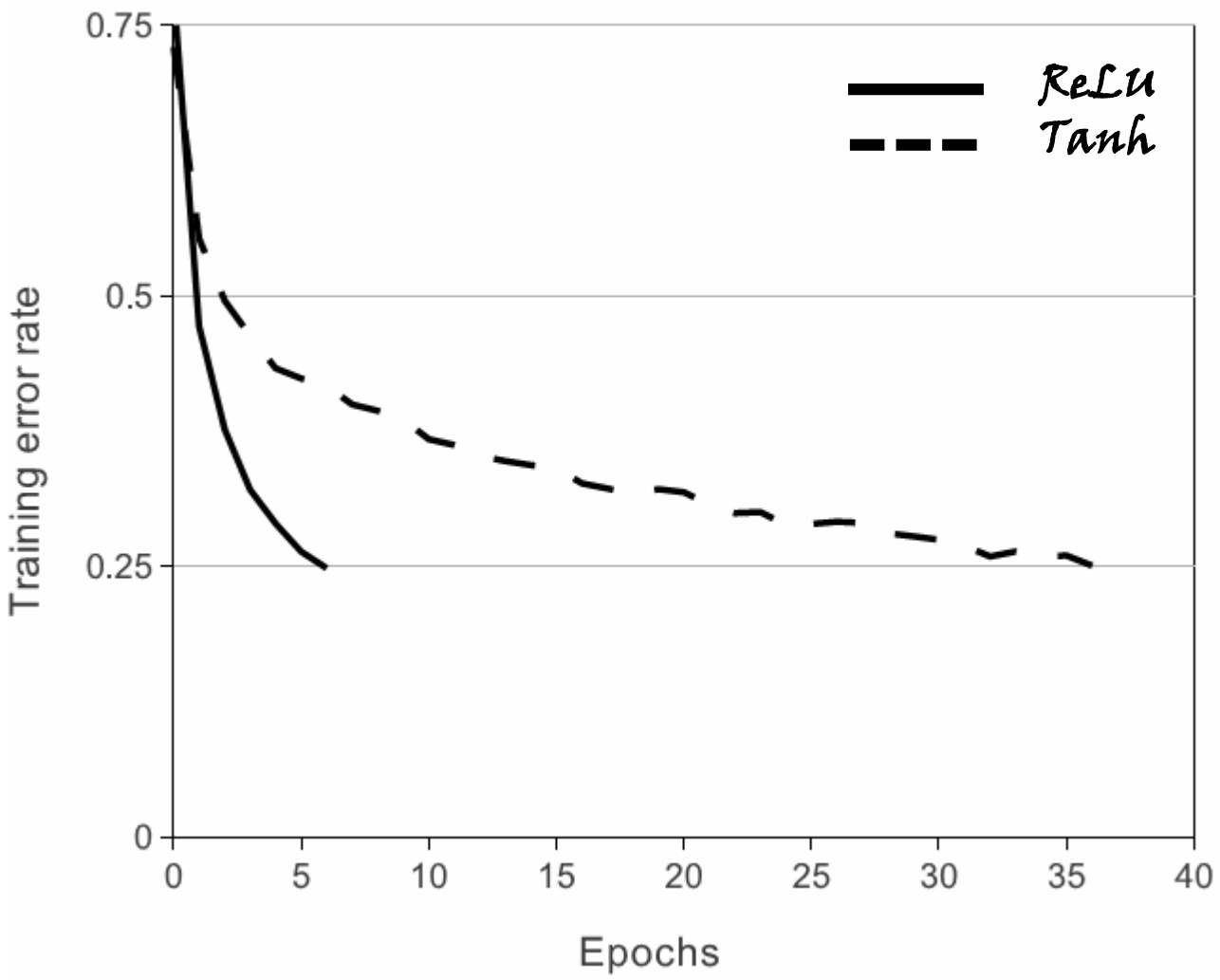
- There is the danger the input x is consistently 0 because of a glitch. This would cause "dead neurons" that always are 0 with 0 gradient.
- It is discontinuous, so it might cause numerical errors during training
- It is piece-wise linear, so the "piece"-gradients are stable and strong
- Since they are linear, their gradients can be computed very fast and speed up training.
- They are more complex to implement, because an if condition needs to be introduced.

Rectified Linear Unit (ReLU) module

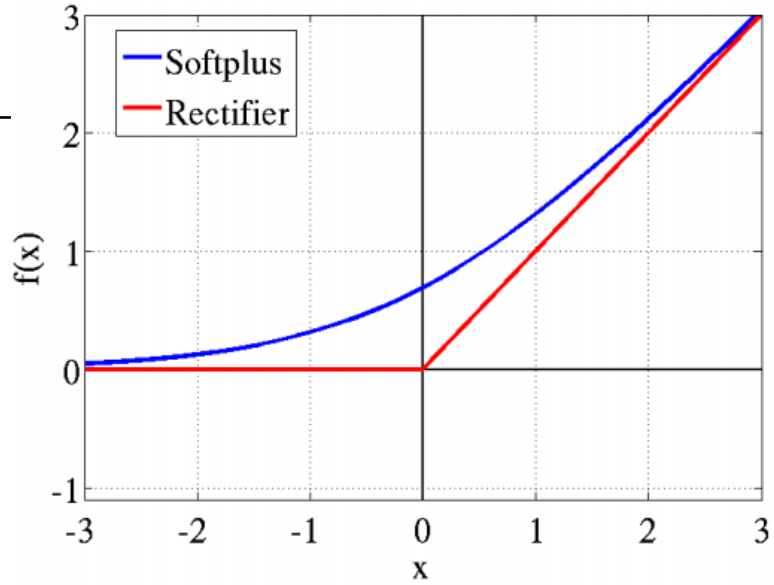
- Activation: $a = h(x) = \max(0, x)$
- Gradient: $\frac{\partial a}{\partial x} = \begin{cases} 0, if\ x \leq 0 \\ 1, if x > 0 \end{cases}$
- Strong gradients: either 0 or 1 ☺
- Fast gradients: just a binary comparison ☺
- It is not differentiable at 0, but not a big problem ☺
  - An activation of precisely 0 rarely happens with non-zero weights, and if it happens we choose a convention
- Dead neurons is an issue
  - Large gradients might cause a neuron to die. Higher learning rates might be beneficial
  - Assuming a linear layer before ReLU $h(x) = \max(0, wx + b)$, make sure the bias term b is initialized with a small initial value, $e.g. 0.1$ → more likely the ReLU is positive and therefore there is non zero gradient
- Nowadays ReLU is the default non-linearity



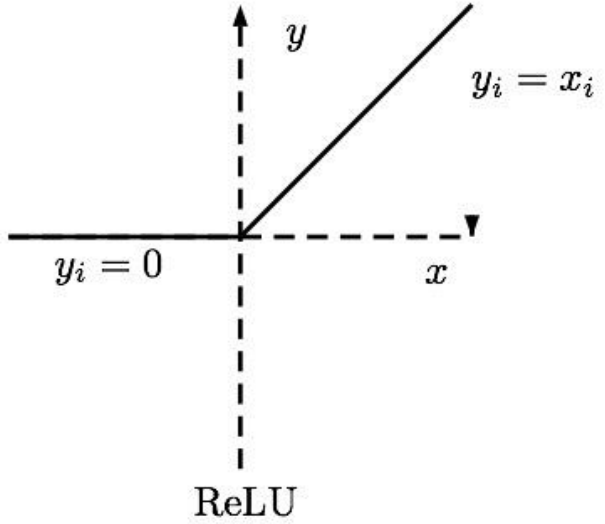
ReLU convergence rate

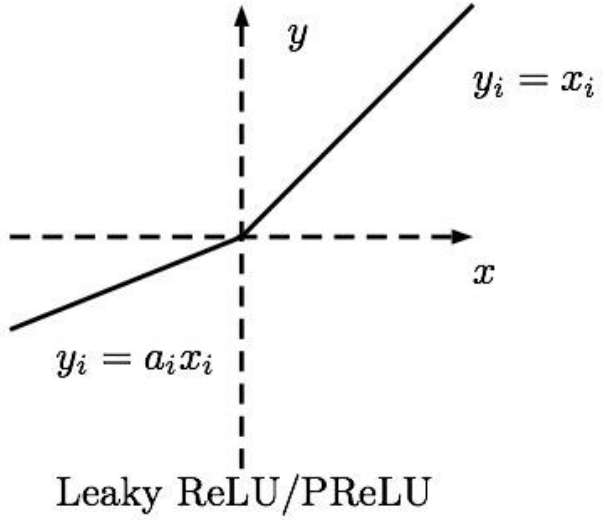


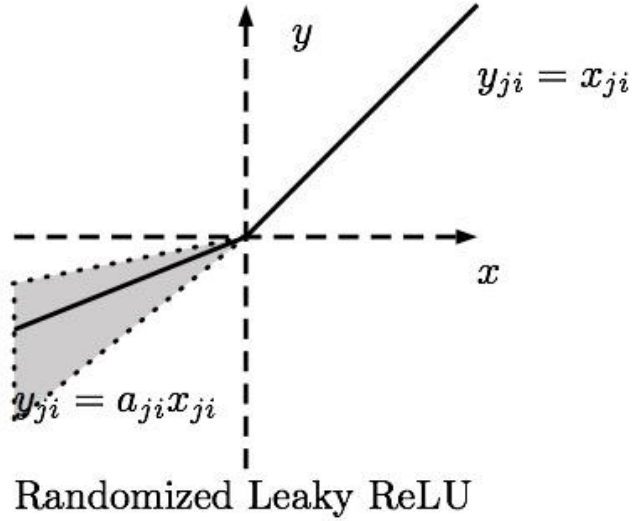
Other ReLUs



- Soft approximation (softplus): $a = h(x) = \ln(1 + e^x)$
- Noisy ReLU: $a = h(x) = \max(0, \mathrm{x} + \varepsilon), \varepsilon \sim N(0, \sigma(\mathrm{x}))$
- Leaky ReLU: $a = h(x) = \begin{cases} x, if\ x > 0 \\ 0.01x\ otherwise \end{cases}$
- Parametric ReLu: $a = h(x) = \begin{cases} x, if\ x > 0 \\ \beta x\ otherwise \end{cases}$ (parameter β is trainable)



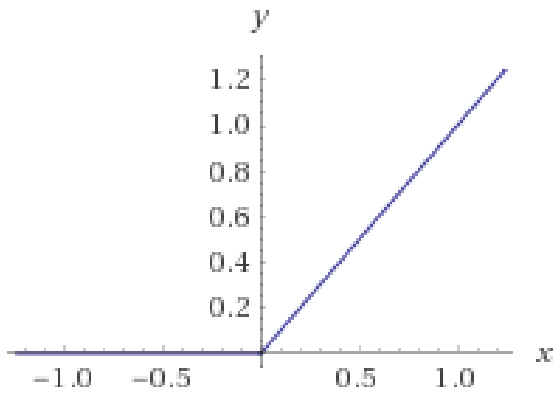




Quiz

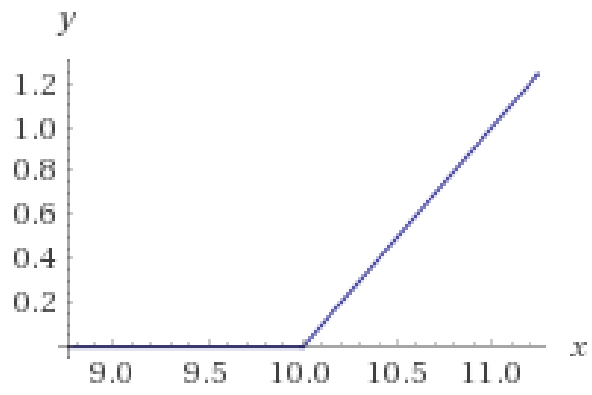
How would you compare the two non-linearities?

- They are equivalent for training
- They are not equivalent for training



(x from −1.2 to 1.2)

Computed by Wolfram|Alpha



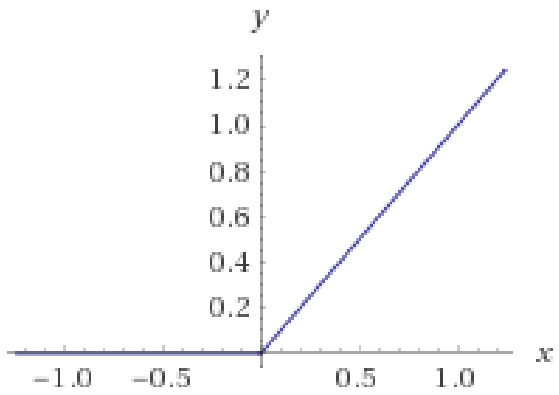
(x from 8.8 to 11.2)

Computed by Wolfram|Alpha

Quiz

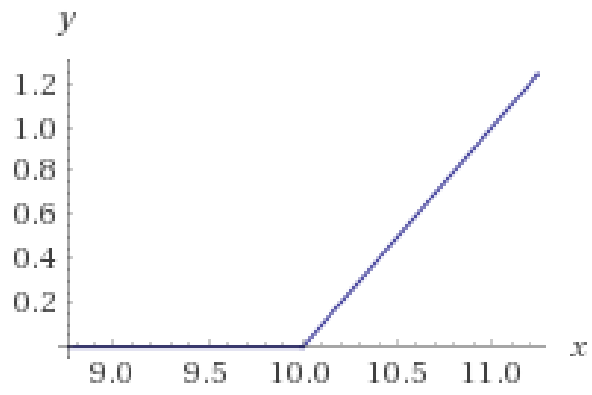
How would you compare the two non-linearities?

- They are equivalent for training
- They are not equivalent for training



(x from −1.2 to 1.2)

Computed by Wolfram|Alpha



(x from 8.8 to 11.2)

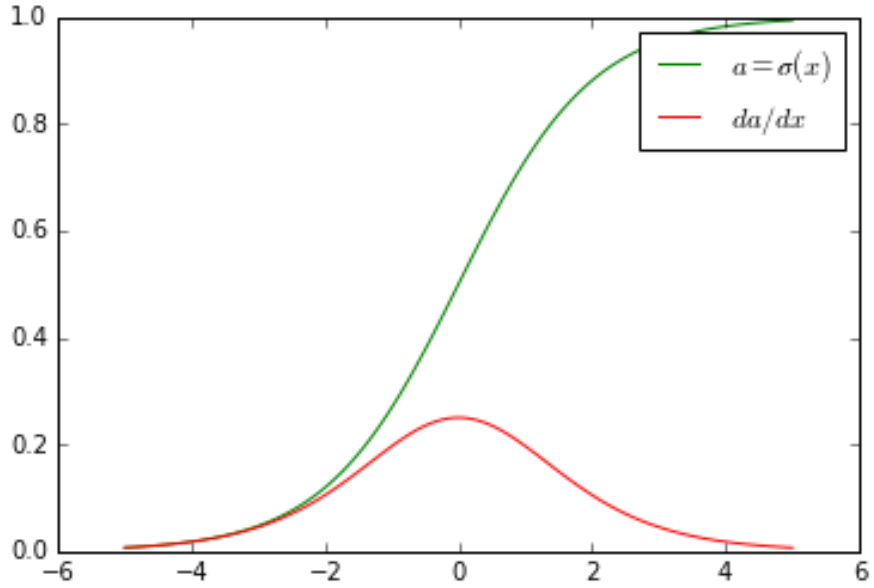
Computed by Wolfram|Alpha

Centered non-linearities

- Remember: a deep network is a hierarchy of similar modules
  - One ReLU is the input to the next ReLU
- Consistent behavior → input/output distributions must match
  - Otherwise, you will soon have inconsistent behavior
  - If ReLU-1 returns always highly positive numbers, e.g. $\sim 10{,}000$ → the next ReLU-2 biased towards highly positive or highly negative values (depending on the sign of the weight w)→ ReLU (2) essentially becomes a linear unit.
- We want our non-linearities to be mostly activated around the origin (centered activations)
  - the only way to encourage consistent behavior not matter the architecture

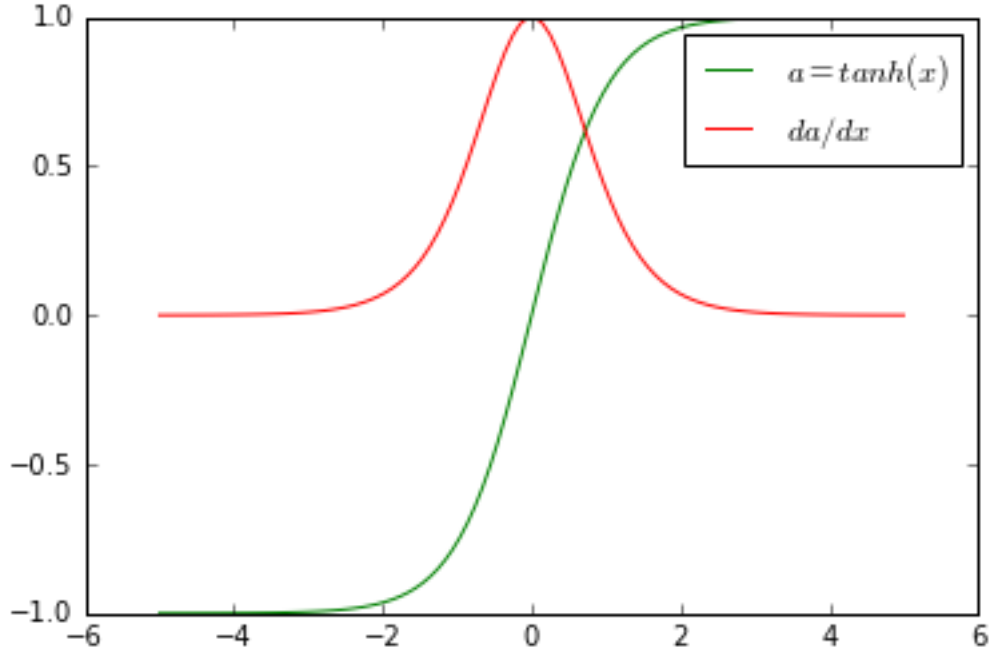
Sigmoid module

- Activation: $a = \sigma(x) = \frac{1}{1+e^{-x}}$
- Gradient: $\frac{\partial a}{\partial x} = \sigma(x)(1 - \sigma(x))$



Tanh module

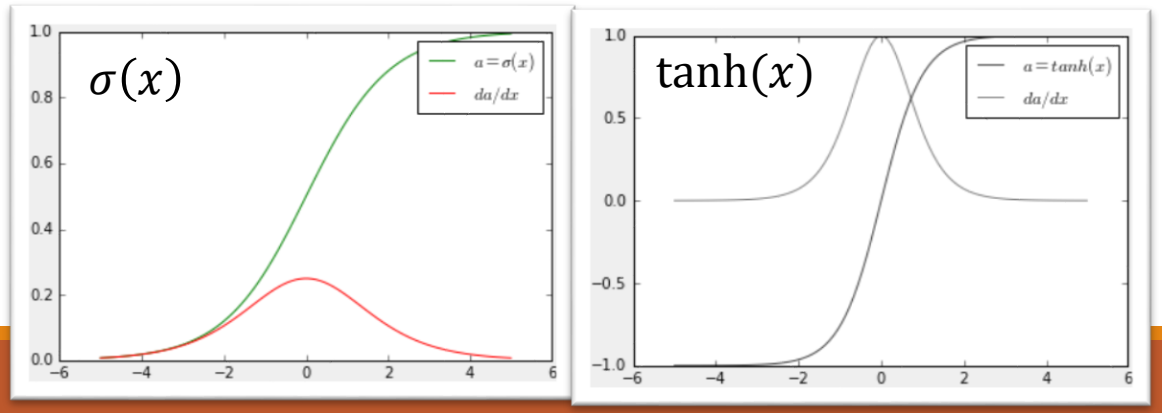
- Activation: $a = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- Gradient: $\frac{\partial a}{\partial x} = 1 - tanh^2(x)$



Quiz

Which non-linearity is better, the sigmoid or the tanh?

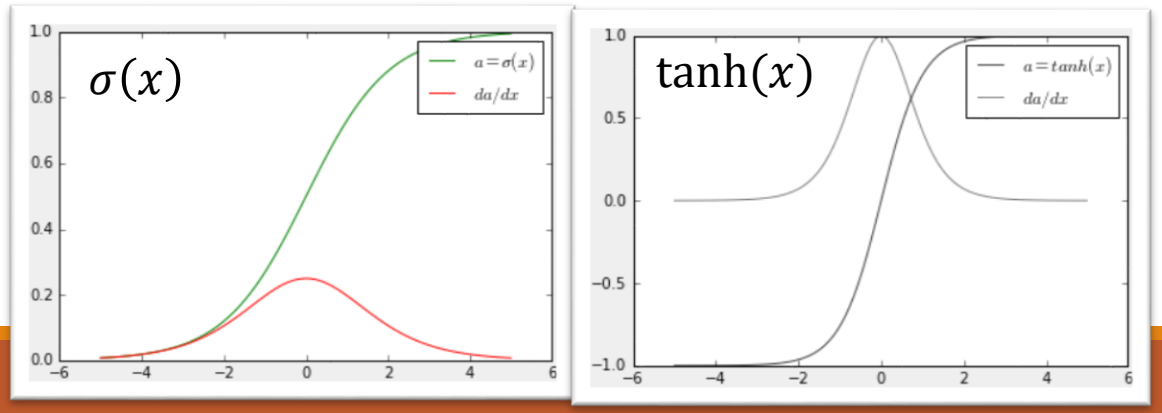
- The tanh, because on the average activation case it has stronger gradients
- The sigmoid, because it's output range [0, 1] resembles the range of probability values
- The tanh, because the sigmoid can be rewritten as a tanh
- The sigmoid, because it has a simpler implementation of gradients
- None of them are that great, they saturate for large or small inputs
- The tanh, because it's mean activation is around 0 and it is easier to combine with other modules



Quiz

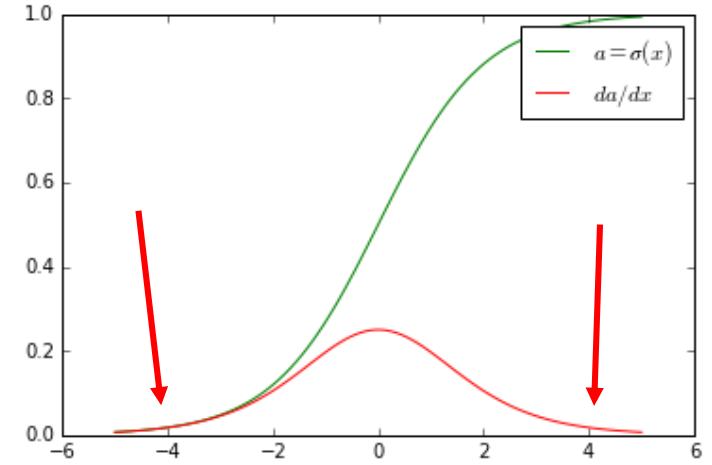
Which non-linearity is better, the sigmoid or the tanh?

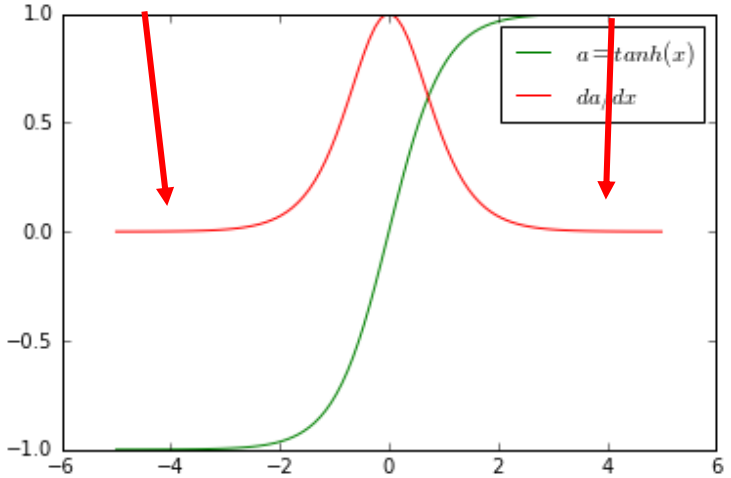
- The tanh, because on the average activation case it has stronger gradients
- The sigmoid, because it's output range [0, 1] resembles the range of probability values
- The tanh, because the sigmoid can be rewritten as a tanh
- The sigmoid, because it has a simpler implementation of gradients
- None of them are that great, they saturate for large or small inputs
- The tanh, because it's mean activation is around 0 and it is easier to combine with other modules



Tanh vs Sigmoids

- Functional form is very similar: $tanh(x) = 2\sigma(2x) - 1$
- $tanh(x)$ has better output $[-1, +1]$ range
  - Stronger gradients, because data is centered around 0 (not 0.5)
  - Less "positive" bias to hidden layer neurons as now outputs can be both positive and negative (more likely to have zero mean in the end)
- Both saturate at the extreme → 0 gradients
  - "Overconfident", without necessarily being correct
  - Especially bad when in the middle layers: why should a neuron be overconfident, when it represents a latent variable
- The gradients are < 1, so in deep layers the chain rule returns very small total gradient
- From the two, $tanh(x)$ enables better learning
  - But still, not a great choice





Sigmoid: An exception

- An exception for sigmoids is …

Sigmoid: An exception

- An exception for sigmoids is when used as the final output layer
- Sigmoid outputs can return very small or very large values (saturate)
  - Output units are not latent variables (have access to ground truth labels)
  - Still “overconfident”, but at least towards true values

Softmax module

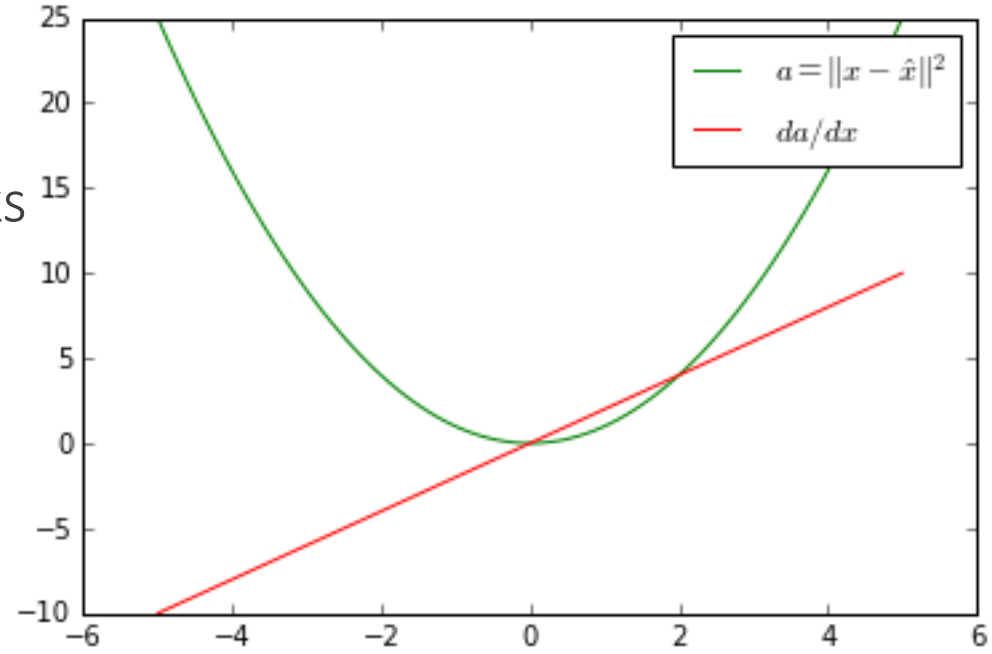
- Activation: $a^{(k)} = softmax(x^{(k)}) = \frac{e^{x^{(k)}}}{\sum_j e^{x^{(j)}}}$
  - Outputs probability distribution, $\sum_{k=1}^{K} a^{(k)} = 1$ for K classes
- Avoid exponentianting too large/small numbers → better stability

$$a^{(k)} = \frac{e^{x^{(k)}-\mu}}{\sum_j e^{x^{(j)}-\mu}}, \mu = \max_k x^{(k)} \text{ because}$$

$$\frac{e^{x^{(k)}-\mu}}{\sum_j e^{x^{(j)}-\mu}} = \frac{e^{\mu} e^{x^{(k)}}}{e^{\mu} \sum_j e^{x^{(j)}}} = \frac{e^{x^{(k)}}}{\sum_j e^{x^{(j)}}}$$

Euclidean loss module

- Activation: $a(x) = 0.5\ \|y - x\|^2$
  - Mostly used to measure the loss in regression tasks
- Gradient: $\frac{\partial a}{\partial x} = x - y$



Cross-entropy loss (log-likelihood) module

- Activation: $a(x) = -\sum_{k=1}^{K} y^{(k)} \log x^{(k)}, \quad y^{(k)} = \{0, 1\}$
- Gradient: $\frac{\partial a}{\partial x^{(k)}} = -\frac{y^{(k)}}{x^{(k)}}$
- The cross-entropy loss is the most popular classification loss for classifiers that output probabilities
- Cross-entropy loss couples well softmax/sigmoid module
  - The $\log$ of the cross-entropy cancels out the $\exp$ of the softmax/sigmoid
  - Often the modules are combined and joint gradients are computed
- Generalization of logistic regression for more than 2 outputs

New modules

- Everything can be a module, given some ground rules
- How to make our own module?
  - Write a function that follows the ground rules
- Needs to be (at least) first-order differentiable (almost) everywhere
- Hence, we need to be able to compute the

$$\frac{\partial a(x;w)}{\partial x} \text{ and } \frac{\partial a(x;w)}{\partial w}$$

A module of modules

- As everything can be a module, a module of modules could also be a module
- We can therefore make new building blocks as we please, if we expect them to be used frequently
- Of course, the same rules for the eligibility of modules still apply

1 sigmoid == 2 modules?

- Assume the sigmoid $\sigma(\dots)$ operating on top of wx

$$a = \sigma(wx)$$

- Directly computing it → complicated backpropagation equations
- Since **everything is a module**, we can decompose this **to 2 modules**

$$\boxed{a_1 = wx} \rightarrow \boxed{a_2 = \sigma(a_1)}$$

1 sigmoid == 2 modules?

- Two backpropagation steps instead of one

+ But now our gradients are simpler

- Algorithmic way of computing gradients
- We avoid taking more gradients than needed in a (complex) non-linearity

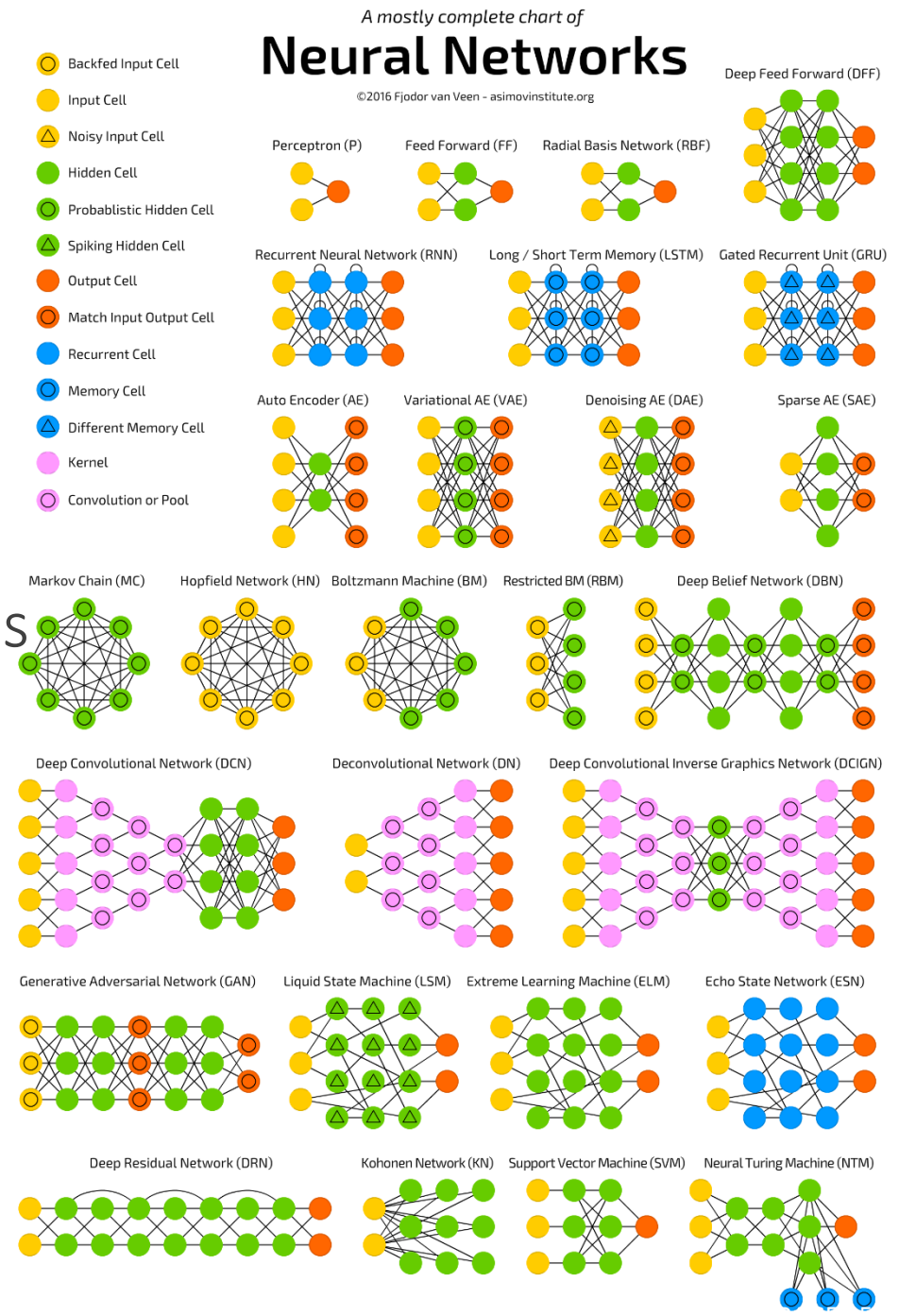
$$\boxed{a_1 = wx} \rightarrow \boxed{a_2 = \sigma(a_1)}$$

Many, many more modules out there …

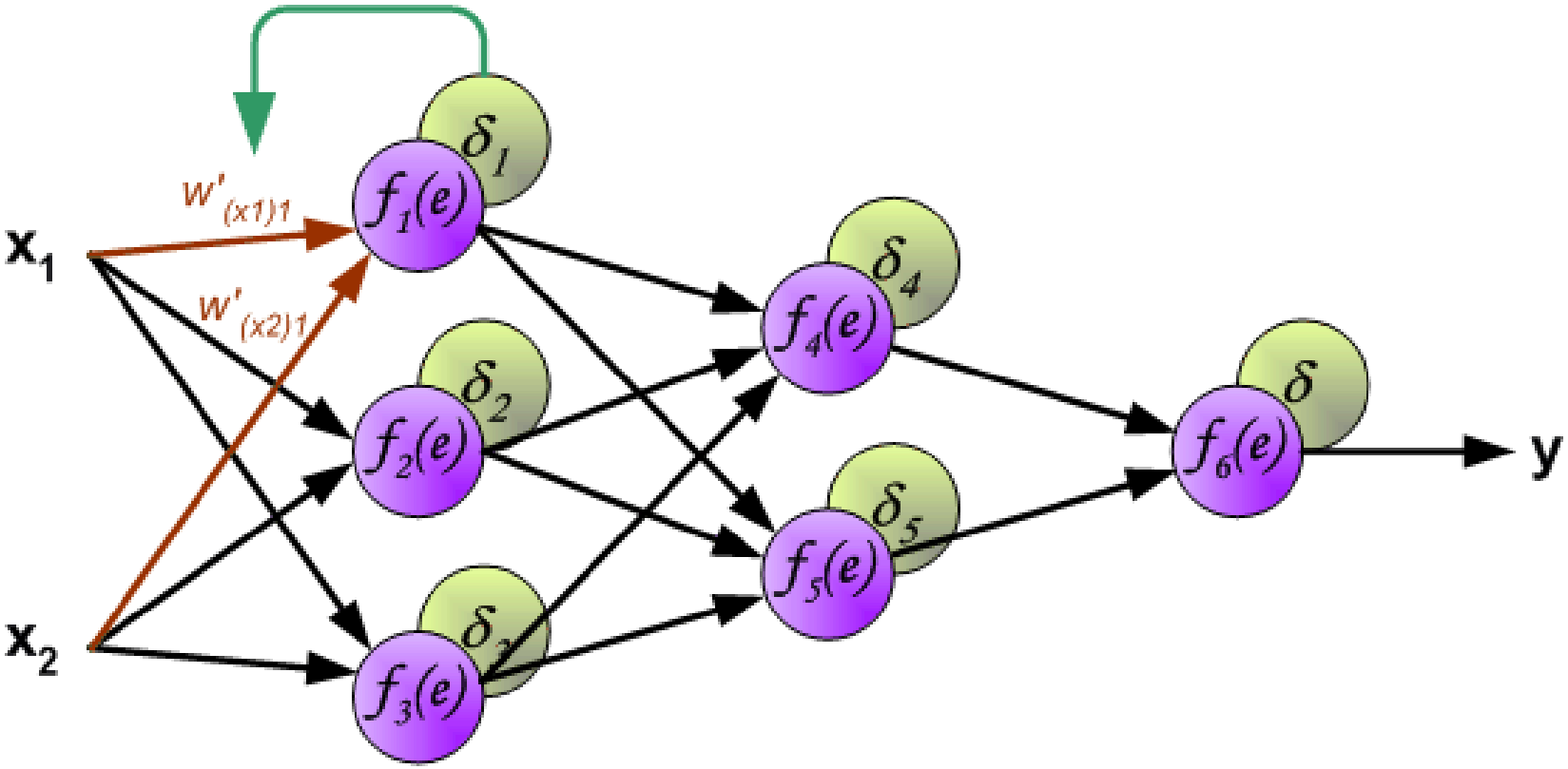
- Many will work comparably to existing ones
  - Not interesting, unless they work consistently better and there is a reason
- Regularization modules
  - Dropout
- Normalization modules
  - ℓ_2-normalization, ℓ_1-normalization
- Loss modules
  - Hinge loss
- Most of concepts discussed in the course can be casted as modules

Neural Network Cheatsheet

- Perceptrons, MLPs
- RNNs, LSTMs, GRUs
- Vanilla, Variational, Denoising Autoencoders
- Hopfield Nets, Restricted Boltzmann Machines
- Convolutional Nets, Deconvolutional Nets
- Generative Adversarial Nets
- Deep Residual Nets, Neural Turing Machines

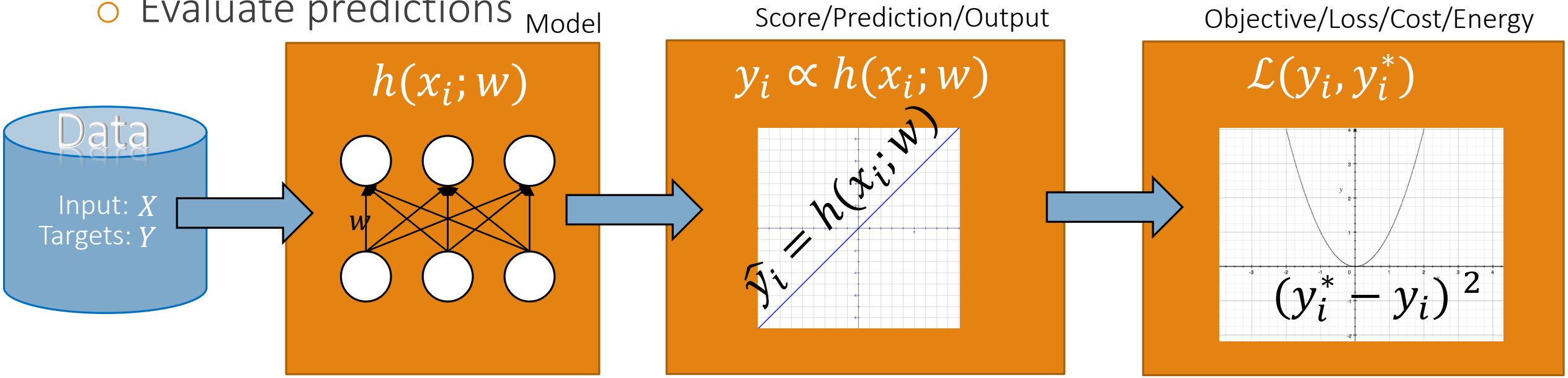


Backpropagation



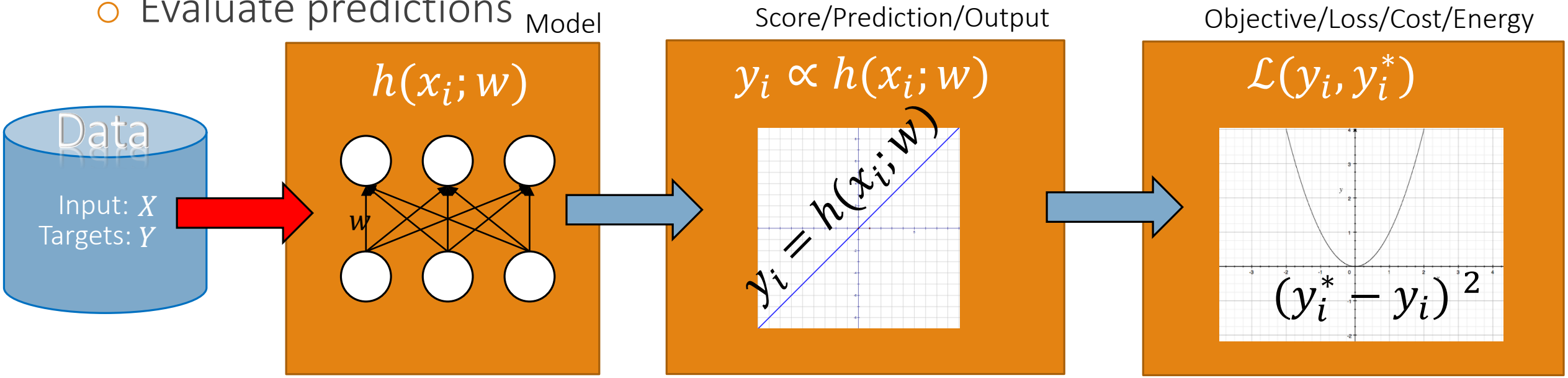
Forward computations

- Collect annotated data
- Define model and initialize randomly
- Predict based on current model
  - In neural network jargon **"forward propagation"**
- Evaluate predictions



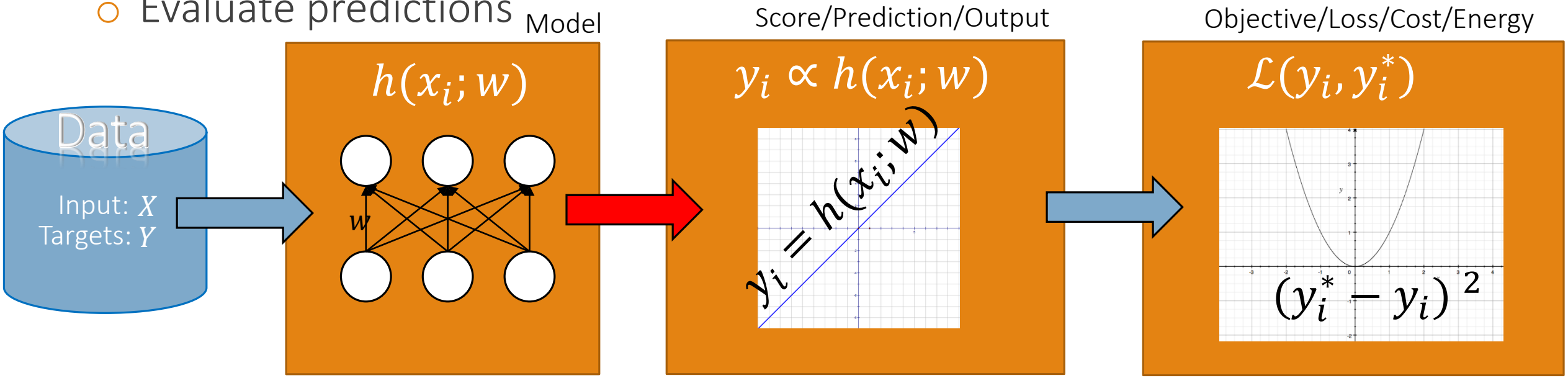
Forward computations

- Collect annotated data
- Define model and initialize randomly
- Predict based on current model
  - In neural network jargon **"forward propagation"**
- Evaluate predictions



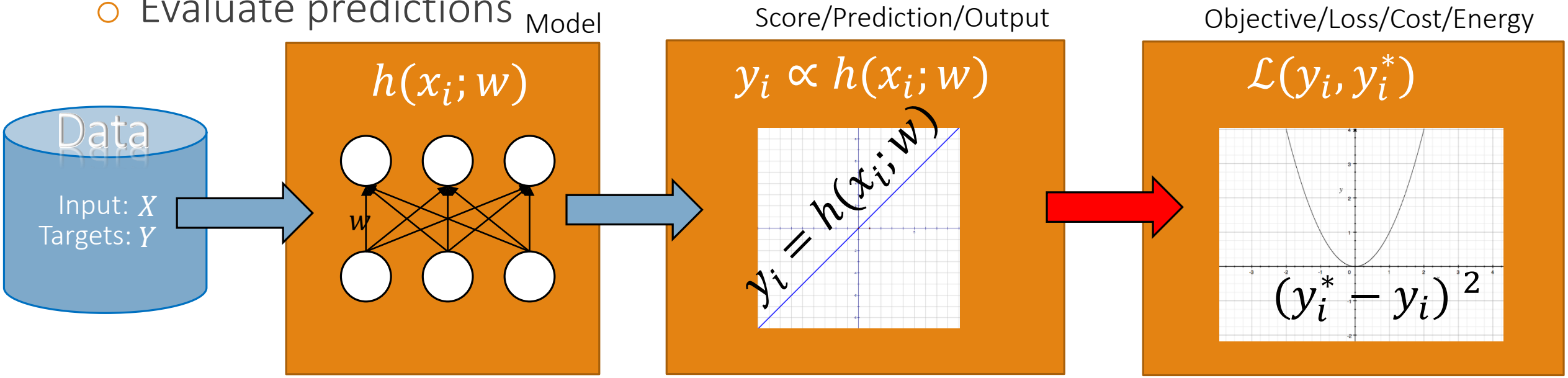
Forward computations

- Collect annotated data
- Define model and initialize randomly
- Predict based on current model
  - In neural network jargon **"forward propagation"**
- Evaluate predictions



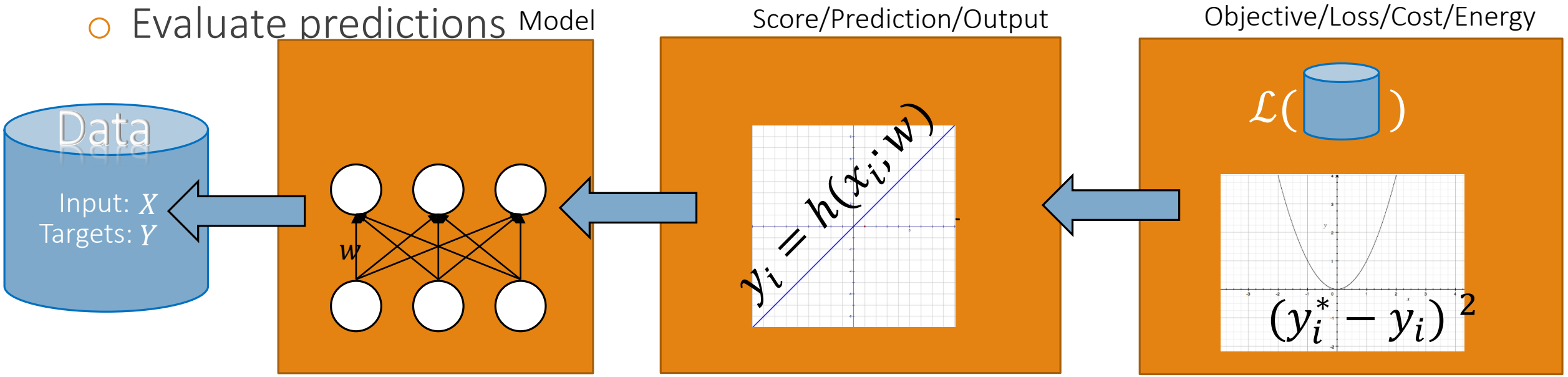
Forward computations

- Collect annotated data
- Define model and initialize randomly
- Predict based on current model
  - In neural network jargon **"forward propagation"**
- Evaluate predictions



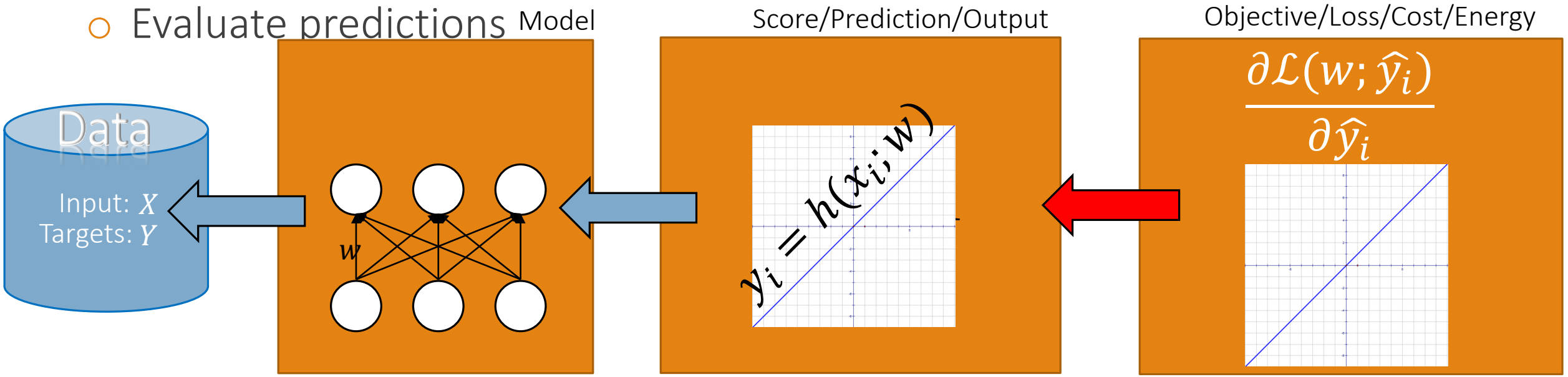
Backward computations

- Collect gradient data
- Define model and initialize randomly
- Predict based on current model
  - In neural network jargon **"backpropagation"**
- Evaluate predictions



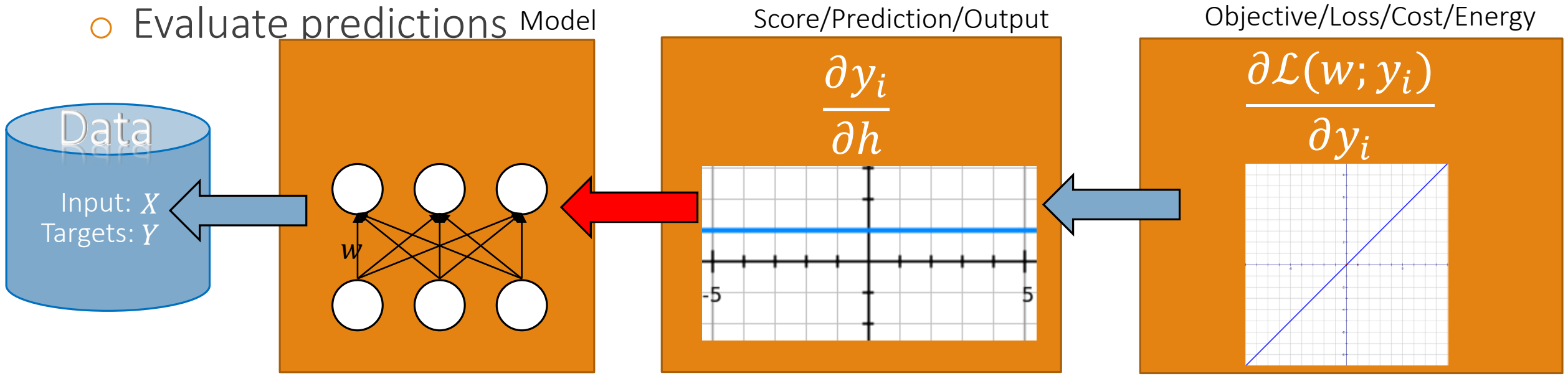
Backward computations

- Collect gradient data
- Define model and initialize randomly
- Predict based on current model
  - In neural network jargon **"backpropagation"**
- Evaluate predictions



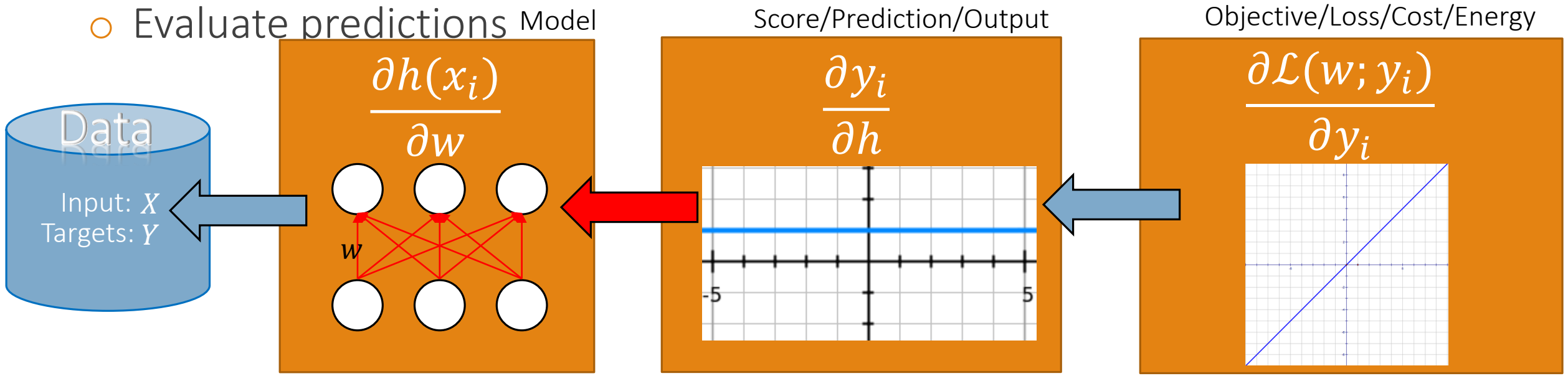
Backward computations

- Collect gradient data
- Define model and initialize randomly
- Predict based on current model
  - In neural network jargon **"backpropagation"**
- Evaluate predictions



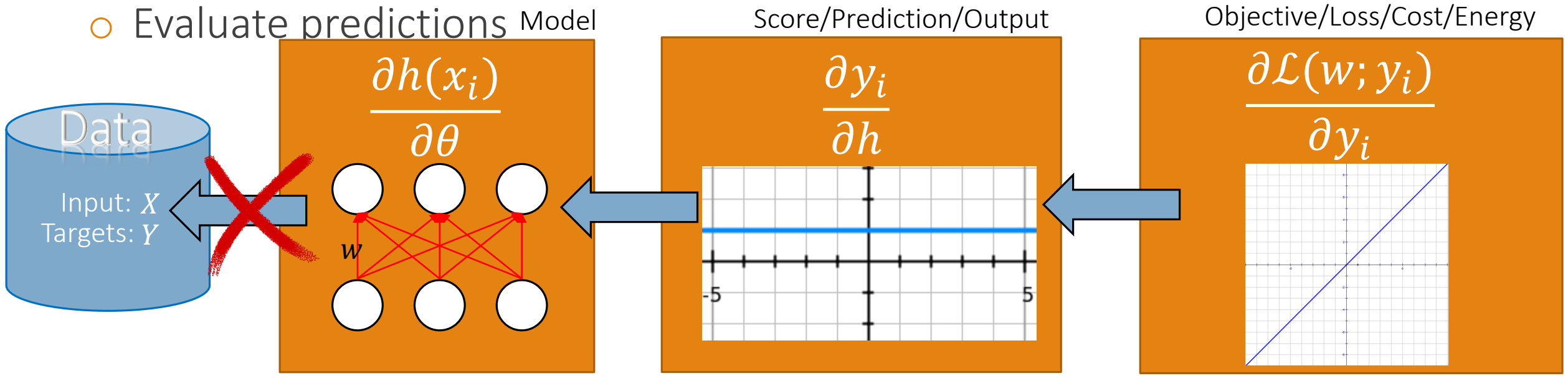
Backward computations

- Collect gradient data
- Define model and initialize randomly
- Predict based on current model
  - In neural network jargon **"backpropagation"**
- Evaluate predictions



Backward computations

- Collect gradient data
- Define model and initialize randomly
- Predict based on current model
  - In neural network jargon **"backpropagation"**
- Evaluate predictions

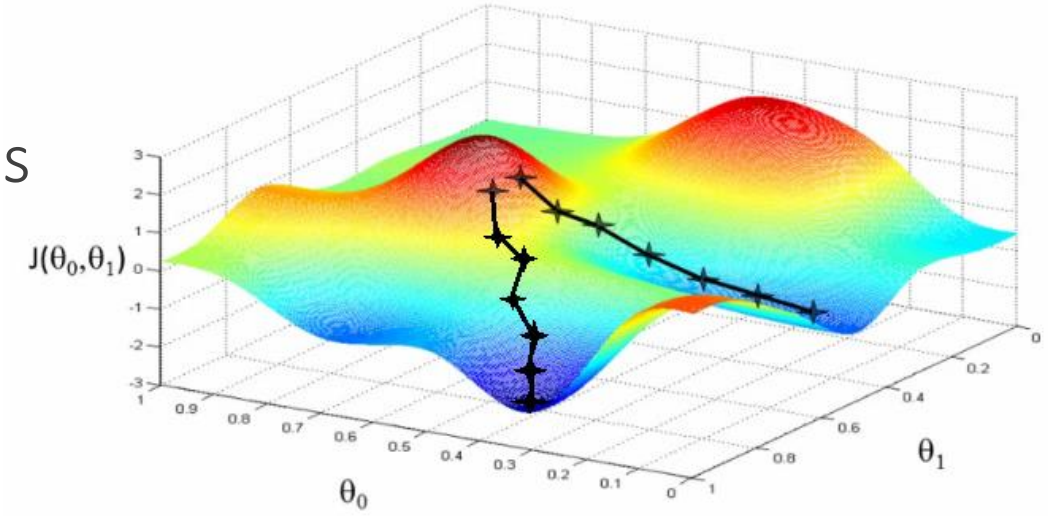


Optimization through Gradient Descent

- As for many models, we optimize our neural network with Gradient Descent

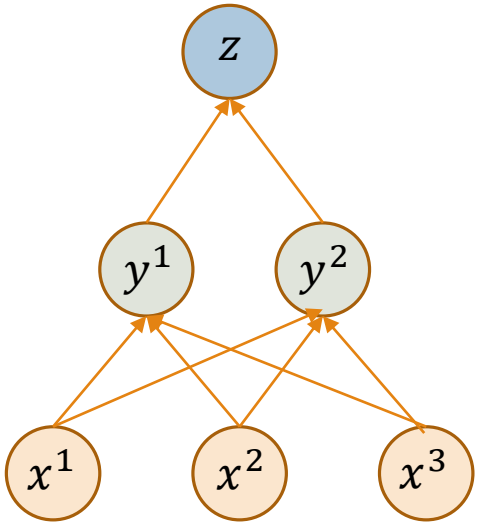
$$w^{(t+1)} = w^{(t)} - \eta_t \nabla_w \mathcal{L}$$

- The most important component in this formulation is the gradient
- How to compute the gradients for such a complicated function enclosing other functions, like $a^L(\dots)$?
  - Hint: Backpropagation
- Let's see, first, how to compute gradients with nested functions



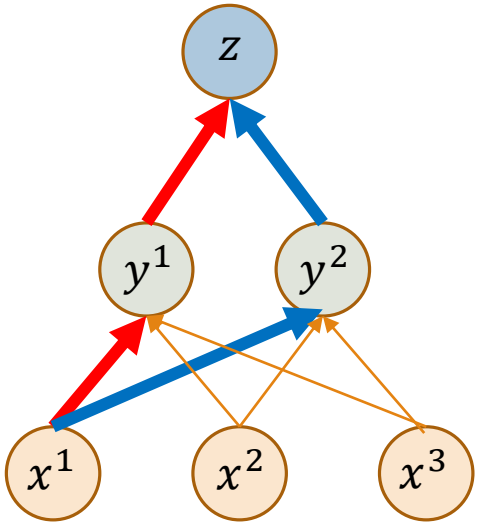
Chain rule

- Assume a nested function, $z = f(y)$ and $y = g(x)$
- Chain Rule for scalars x, y, z
  - $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$
- When $x \in \mathcal{R}^m, y \in \mathcal{R}^n, z \in \mathcal{R}$
  - $\frac{dz}{dx_i} = \sum_j \frac{dz}{dy_j}\frac{dy_j}{dx_i}$ → gradients from all possible paths



Chain rule

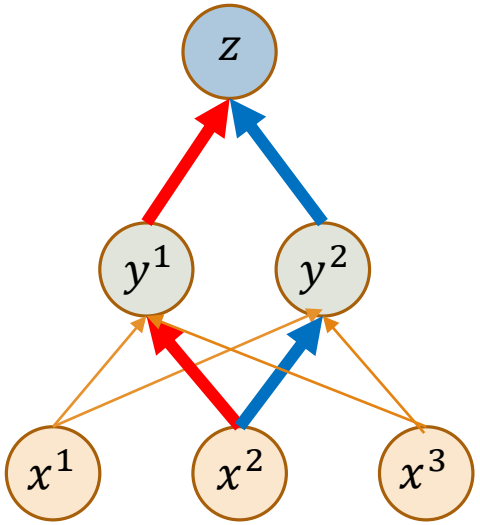
- Assume a nested function, $z = f(y)$ and $y = g(x)$
- Chain Rule for scalars x, y, z
  - $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$
- When $x \in \mathcal{R}^m, y \in \mathcal{R}^n, z \in \mathcal{R}$
  - $\frac{dz}{dx_i} = \sum_j \frac{dz}{dy_j}\frac{dy_j}{dx_i}$ → gradients from all possible paths



$$\frac{dz}{dx^1} = \frac{dz}{dy^1}\frac{dy^1}{dx^1} + \frac{dz}{dy^2}\frac{dy^2}{dx^1}$$

Chain rule

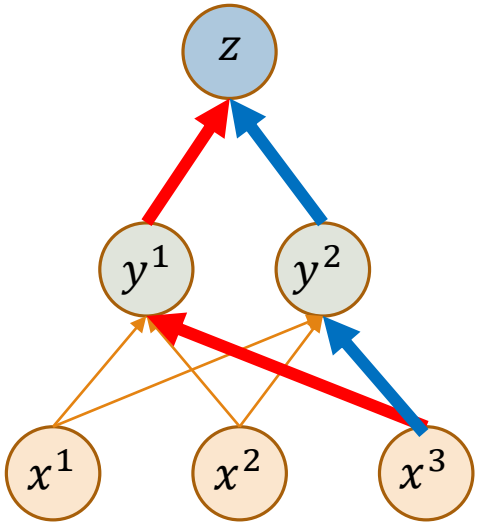
- Assume a nested function, $z = f(y)$ and $y = g(x)$
- Chain Rule for scalars x, y, z
  - $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$
- When $x \in \mathcal{R}^m, y \in \mathcal{R}^n, z \in \mathcal{R}$
  - $\frac{dz}{dx_i} = \sum_j \frac{dz}{dy_j}\frac{dy_j}{dx_i}$ → gradients from all possible paths



$$\frac{dz}{dx^2} = \frac{dz}{dy^1}\frac{dy^1}{dx^2} + \frac{dz}{dy^2}\frac{dy^2}{dx^2}$$

Chain rule

- Assume a nested function, $z = f(y)$ and $y = g(x)$
- Chain Rule for scalars x, y, z
  - $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$
- When $x \in \mathcal{R}^m, y \in \mathcal{R}^n, z \in \mathcal{R}$
  - $\frac{dz}{dx_i} = \sum_j \frac{dz}{dy_j}\frac{dy_j}{dx_i}$ → gradients from all possible paths



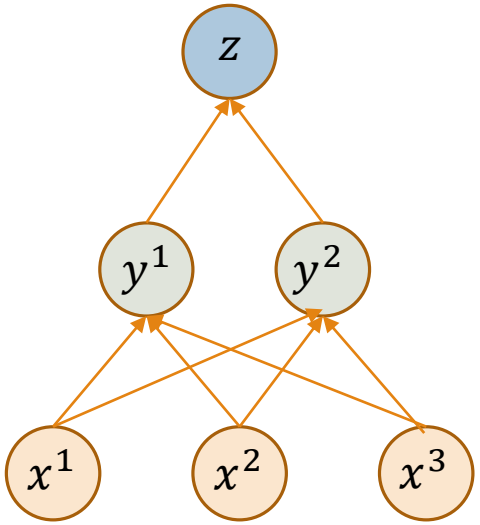
$$\frac{dz}{dx^3} = \frac{dz}{dy^1}\frac{dy^1}{dx^3} + \frac{dz}{dy^2}\frac{dy^2}{dx^3}$$

Chain rule

- Assume a nested function, $z = f(y)$ and $y = g(x)$
- Chain Rule for scalars x, y, z
  - $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$
- When $x \in \mathcal{R}^m, y \in \mathcal{R}^n, z \in \mathcal{R}$
  - $\frac{dz}{dx_i} = \sum_j \frac{dz}{dy_j}\frac{dy_j}{dx_i}$ → gradients from all possible paths
  - or in vector notation

$$\nabla_x(z) = \left(\frac{d\boldsymbol{y}}{d\boldsymbol{x}}\right)^T \cdot \nabla_y(z)$$

  - $\frac{d\boldsymbol{y}}{d\boldsymbol{x}}$ is the Jacobian



The Jacobian

- When $x \in \mathcal{R}^3, y \in \mathcal{R}^2$

$$J\big(y(x)\big) = \frac{d\boldsymbol{y}}{d\boldsymbol{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \frac{\partial y_1}{\partial x_3} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \frac{\partial y_2}{\partial x_3} \end{bmatrix}$$

Chain rule in practice

- $\mathrm{a} = \mathrm{h}(x) = \sin(0.5\mathrm{x}^2)$
- $\mathrm{t} = \mathrm{f}(\mathrm{y}) = \sin(y)$
- $y = g(x) = 0.5\, x^2$

$$\frac{df}{dx} = \frac{d\,[\sin(y)]}{dg}\frac{d[0.5x^2]}{dx}$$
$$= \cos(0.5x^2) \cdot x$$

Backpropagation ⟺ Chain rule!!!

- The loss function $\mathcal{L}(y, a^L)$ depends on a^L, which depends on a^{L-1}, …, which depends on a_l
- Gradients of parameters of layer l → Chain rule

$$\frac{\partial \mathcal{L}}{\partial w^l} = \frac{\partial \mathcal{L}}{\partial a^L} \cdot \frac{\partial a^L}{\partial a^{L-1}} \cdot \frac{\partial a^{L-1}}{\partial a^{L-2}} \cdot \ \dots \cdot \frac{\partial a^l}{\partial w^l}$$

- When shortened, we need to two quantities

$$\frac{\partial \mathcal{L}}{\partial w^l} = \left(\frac{\partial a^l}{\partial w^l}\right)^T \cdot \frac{\partial \mathcal{L}}{\partial a^l}$$

Gradient of a module w.r.t. its parameters

Gradient of loss w.r.t. the module output

Backpropagation ⟺ Chain rule!!!

$$\frac{\partial \mathcal{L}}{\partial w^l} = \left(\frac{\partial a^l}{\partial w^l}\right)^T \cdot \frac{\partial \mathcal{L}}{\partial a^l}$$

- For $\frac{\partial a^l}{\partial w^l}$ we only need the Jacobian of the l-th module output a^l w.r.t. to the module's parameters w^l
- Very local rule → every module looks for its own
  - No need to know what other modules do
- Since computations can be very local, this means that …

Backpropagation ⟺ Chain rule!!!

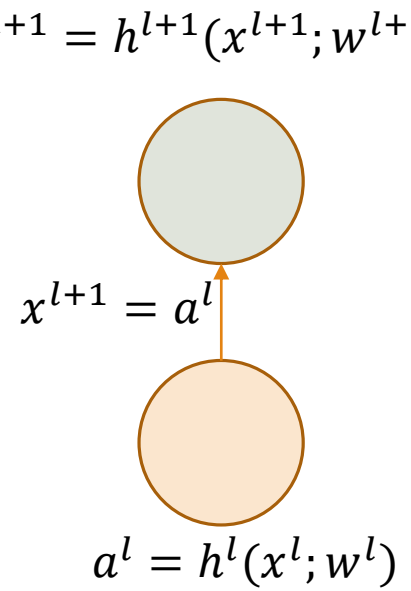
$$\frac{\partial \mathcal{L}}{\partial w^l} = \left(\frac{\partial a^l}{\partial w^l}\right)^T \cdot \frac{\partial \mathcal{L}}{\partial a^l}$$

- For $\frac{\partial a^l}{\partial w^l}$ we only need the Jacobian of the l-th module output a^l w.r.t. to the module's parameters w^l
- Very local rule → every module looks for its own
  - No need to know what other modules do
- Since computations can be very local, this means that …
  - graphs can be very complicated
  - modules can be complicated (as long as they are differentiable)

Backpropagation ⟺ Chain rule!!!

- For $\frac{\partial \mathcal{L}}{\partial a^l}$ we apply chain rule again

$$\frac{\partial \mathcal{L}}{\partial a^l} = \left(\frac{\partial a^{l+1}}{\partial a^l}\right)^T \cdot \frac{\partial \mathcal{L}}{\partial a^{l+1}}$$



- We can rewrite $\frac{\partial a_{l+1}}{\partial a_l}$ as gradient of module w.r.t. to input
  - Remember, the output of a module is the input for the next one: $a_l = x_{l+1}$

$$\frac{\partial \mathcal{L}}{\partial a^l} = \left(\frac{\partial a^{l+1}}{\partial x^{l+1}}\right)^T \cdot \frac{\partial \mathcal{L}}{\partial a^{l+1}}$$

Gradient w.r.t. the module input

Recursive rule (good for us)!!!

Quiz

How do we compute the gradient of multivariate activation functions, like softmax: $a^j = \exp x_j / (x_1 + x_2 + x_3)$?

- We vectorize the inputs and the outputs and compute the gradient as before
- We compute the Hessian matrix of the second-order derivatives: $d^2 a_j / (dx_i dx_j)$
- We compute the Jacobian matrix containing all the partial derivatives: da_j / dx_i

Quiz

How do we compute the gradient of multivariate activation functions, like softmax: $a^j = \exp x_j/(x_1 + x_2 + x_3)$?

- We vectorize the inputs and the outputs and compute the gradient as before
- We compute the Hessian matrix of the second-order derivatives: $d^2 a_j/(dx_i dx_j)$
- We compute the Jacobian matrix containing all the partial derivatives: da_j/dx_i

Multivariate functions $f(\boldsymbol{x})$

- Often module functions depend on multiple input variables
  - For instance, softmax!
  - Each output dimension depends on multiple input dimensions

$$a_j = \frac{e^{x_j}}{e^{x_1} + e^{x_2} + e^{x_3}}, j = 1,2,3$$

- For these cases there are **multiple paths** for each a_j
- So, for the $\frac{\partial a^l}{\partial x^l}$ (or $\frac{\partial a^l}{\partial w^l}$) we must compute the Jacobian matrix
- The Jacobian is the generalization of gradient for multivariate functions
  - e.g. in softmax a_2 depends on all e^{x_1}, e^{x_2} and e^{x_3}, not just on e^{x_2}

Diagonal Jacobians

- Quite often one output dim depends only on a single input dim
  - e.g. a sigmoid $a = \sigma(x)$, or $a = \tanh(x)$, or $a = \exp(x)$

$$a(x) = \sigma(\boldsymbol{x}) = \sigma\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) = \begin{bmatrix} \sigma(x_1) \\ \sigma(x_2) \\ \sigma(x_3) \end{bmatrix}$$

- Not need for full Jacobian, only the diagonal: anyways $\frac{\mathrm{d}a_i}{dx_j} = 0, \ \forall \ \mathrm{i} \neq \mathrm{j}$

$$\frac{d\boldsymbol{a}}{d\boldsymbol{x}} = \frac{d\boldsymbol{\sigma}}{d\boldsymbol{x}} = \begin{bmatrix} \sigma(x_1)(1-\sigma(x_1)) & 0 & 0 \\ 0 & \sigma(x_2)(1-\sigma(x_2)) & 0 \\ 0 & 0 & \sigma(x_3)(1-\sigma(x_3)) \end{bmatrix} \sim \begin{bmatrix} \sigma(x_1)(1-\sigma(x_1)) \\ \sigma(x_2)(1-\sigma(x_2)) \\ \sigma(x_3)(1-\sigma(x_3)) \end{bmatrix}$$

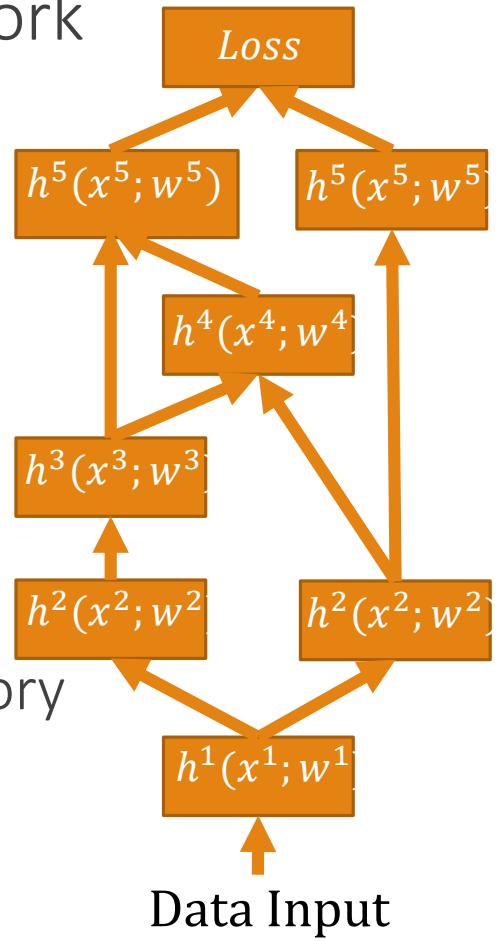
- Can rewrite equations as inner products to save computations

Forward graph

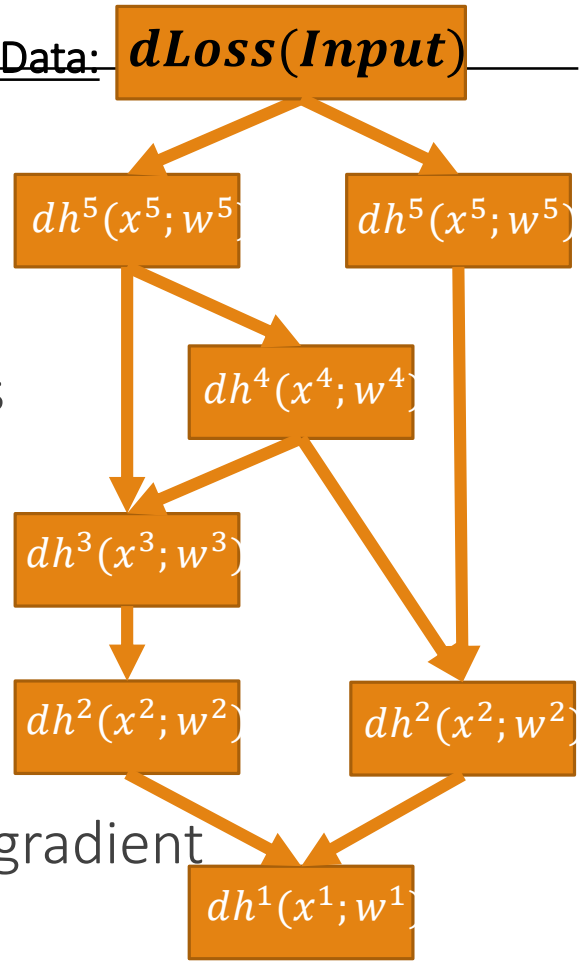
- Simply compute the activation of each module in the network

$$a^l = h^l(x^l; w)$$

  - Then, set $x^{l+1} := a^l$
  - Must know the precise function behind each module $h^l(\dots)$
- Then, repeat recursively
  - Visit modules one by one starting from the data input
  - One module's output is another module's input
  - Some modules might have several inputs from multiple modules
  - Store intermediate values → save compute time at the cost of memory
- Compute modules activations **with the right order**
  - Make sure all the inputs are computed at the right time



Backward graph



- Same story but in reverse
  - Take the reverse network (reverse connections) and go backwards
  - Instead of activation functions, use gradients of activation functions
- Requirements
  - Must know the gradient formulation of each module $\partial h^l(x^l; w^l)$
  - w.r.t. both their inputs x^l and parameters w^l
  - Note: We need the **forward computations first.**
    Activations are part of the gradients, including the final loss and its gradient
    The total loss is the sum of losses for all inputs in our batch
- The whole process can be described very neatly and concisely with the **backpropagation algorithm**

Backpropagation: Recursive chain rule

- **Step 1.** Compute forward propagations for all layers recursively

$$a^l = h^l(x^l) \text{ and } x^{l+1} = a^l$$

- **Step 2.** Once done with forward propagation, follow the reverse path.
  - Start from the last layer and for each new layer compute the gradients
  - Cache computations when possible to avoid redundant operations

$$\frac{\partial \mathcal{L}}{\partial a^l} = \left(\frac{\partial a^{l+1}}{\partial x^{l+1}}\right)^T \cdot \frac{\partial \mathcal{L}}{\partial a^{l+1}} \qquad \frac{\partial \mathcal{L}}{\partial w^l} = \frac{\partial a^l}{\partial w^l} \cdot \left(\frac{\partial \mathcal{L}}{\partial a^l}\right)^T$$

- **Step 3.** Use the gradients $\frac{\partial \mathcal{L}}{\partial w^l}$ with Stochastic Gradient Descend to train

Backpropagation: Recursive chain rule

- **Step 1.** Compute forward propagations for all layers recursively

$$a^l = h^l(x^l) \text{ and } x^{l+1} = a^l$$

- **Step 2.** Once done with forward propagation, follow the reverse path.
  - Start from the last layer and for each new layer compute the gradients
  - Cache computations when possible to avoid redundant operations

Vector with dimensions $[d_l \times 1]$

$$\frac{\partial \mathcal{L}}{\partial a^l} = \left(\frac{\partial a^{l+1}}{\partial x^{l+1}}\right)^T \cdot \frac{\partial \mathcal{L}}{\partial a^{l+1}}$$

Vector with dimensions $[d_{l-1} \times 1]$

$$\frac{\partial \mathcal{L}}{\partial w^l} = \frac{\partial a^l}{\partial w^l} \cdot \left(\frac{\partial \mathcal{L}}{\partial a^l}\right)^T$$

- **Step 3.** Use the gradients $\frac{\partial \mathcal{L}}{\partial w_l}$ with Stochastic Gradient Descend to train

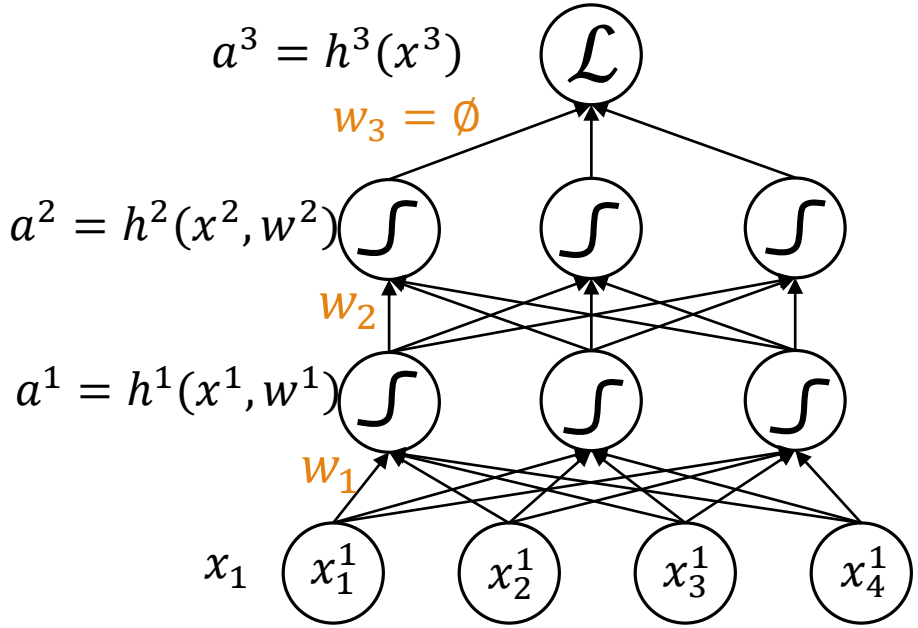
Jacobian matrix with dimensions $[d_{l+1} \times d_l]^T$

Vector with dimensions $[d_{l+1} \times 1]$

Matrix with dimensions $[d_{l-1} \times d_l]$

Vector with dimensions $[1 \times d_l]$

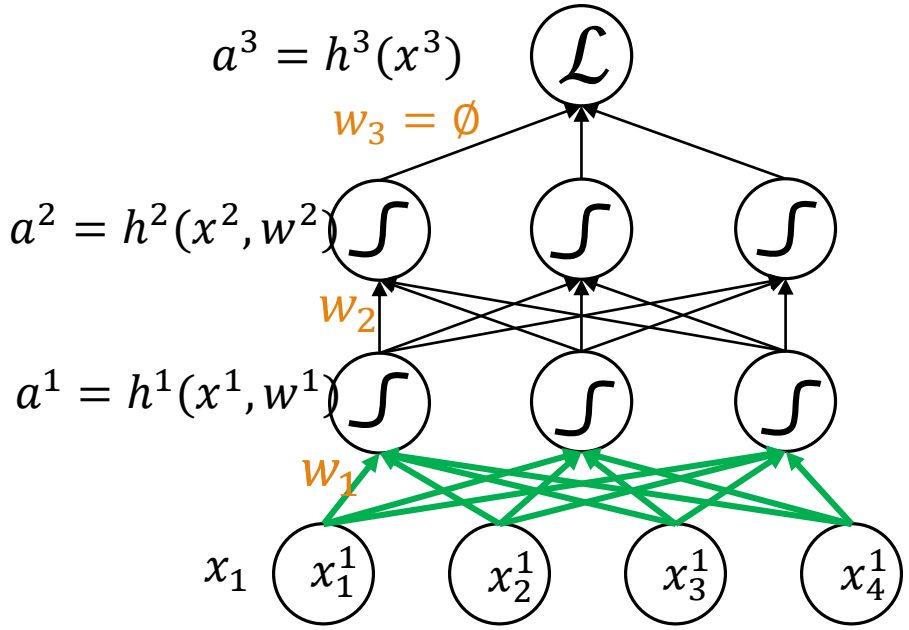
Backpropagation visualization



Backpropagation visualization at epoch (t)

Forward propagations

Compute and store $a_1 = h_1(x_1)$



Example

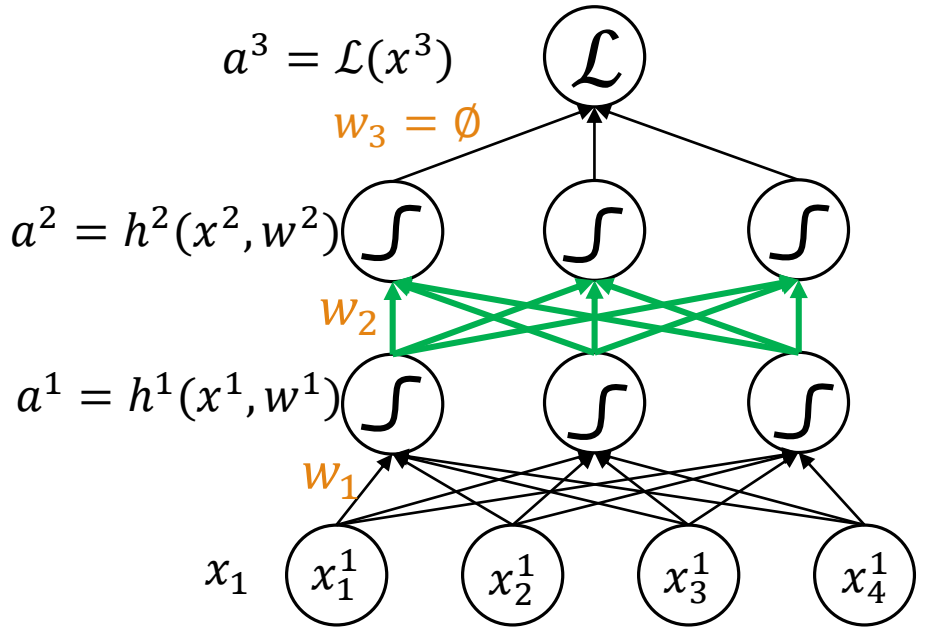
$a^1 = \sigma(w^1 x^1)$

Store!!!

Backpropagation visualization at epoch (t)

Forward propagations

Compute and store $a_2 = h_2(x_2)$



Example

$a^1 = \sigma(w^1 x^1)$

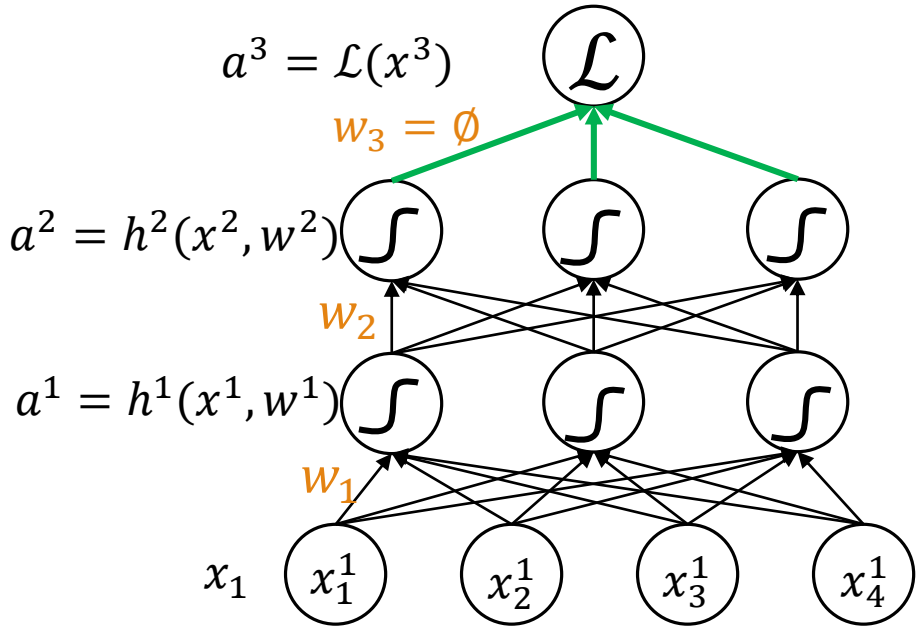
$a^2 = \sigma(w^2 x^2)$

Store!!!

Backpropagation visualization at epoch (t)

Forward propagations

Compute and store $a_3 = h_3(x_3)$



Example

$a^1 = \sigma(w^1 x^1)$

$a^2 = \sigma(w^2 x^2)$

$\mathcal{L}(y, a^2) = \|y - a^2\|^2 = \|y - x^3\|^2$

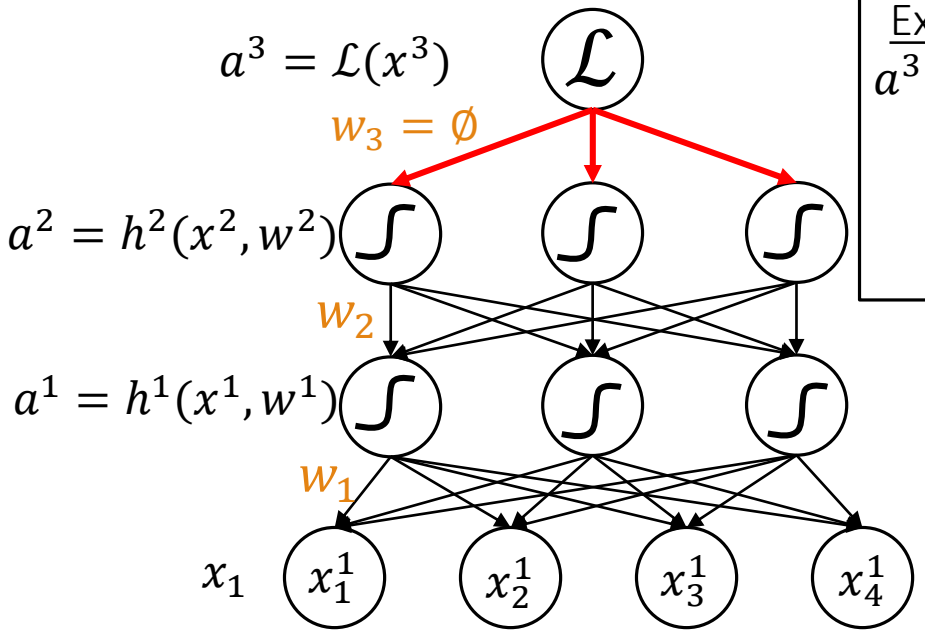
Store!!!

Backpropagation visualization at epoch (t)

Backpropagation

$\frac{\partial \mathcal{L}}{\partial a^3} =$ ... ← Direct computation

~~$\frac{\partial \mathcal{L}}{\partial w^3}$~~



Example

$a^3 = \mathcal{L}(y, x^3) = 0.5 \|y - x^3\|^2$

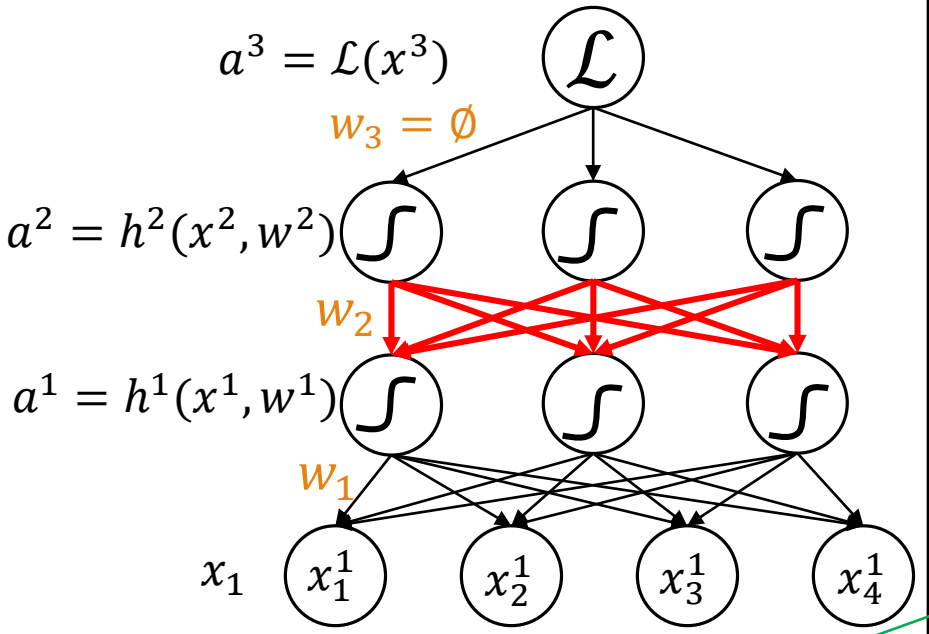
$\frac{\partial \mathcal{L}}{\partial x^3} = -(y - x^3)$

Backpropagation visualization at epoch (t)

Backpropagation

$$\frac{\partial \mathcal{L}}{\partial a^2} = \frac{\partial \mathcal{L}}{\partial a^3} \cdot \frac{\partial a^3}{\partial a^2}$$

$$\frac{\partial \mathcal{L}}{\partial w^2} = \frac{\partial \mathcal{L}}{\partial a^2} \cdot \frac{\partial a^2}{\partial w^2}$$



Stored during forward computations

Exampl

e $\mathcal{L}(y, x_3) = 0.5 \|y - x_3\|^2$

$$a^2 = \sigma(w^2 x^2)$$

$$\frac{\partial \mathcal{L}}{\partial a^2} = \frac{\partial \mathcal{L}}{\partial x^3} = -(y - x_3)$$

$$\partial\sigma(x) = \sigma(x)(1 - \sigma(x))$$

$$\frac{\partial a^2}{\partial w^2} = x^2 \sigma(w^2 x^2)(1 - \sigma(w^2 x^2))$$

$$= x^2 a^2 (1 - a^2)$$

$$\frac{\partial \mathcal{L}}{\partial a^2} = -(y - x^3)$$

$$\frac{\partial \mathcal{L}}{\partial w^2} = \frac{\partial \mathcal{L}}{\partial a^2} x^2 a^2 (1 - a^2)$$

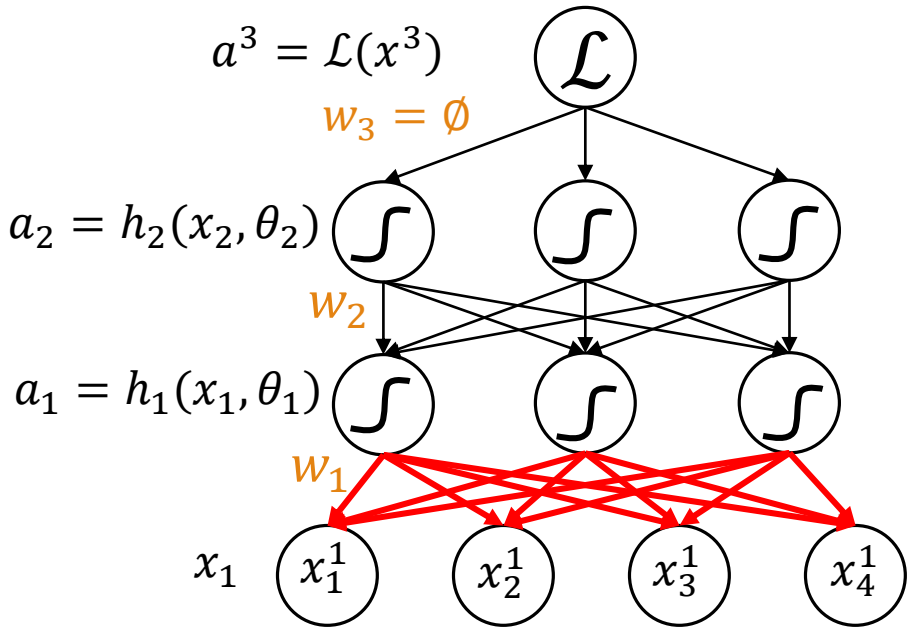
Store!!!

Backpropagation visualization at epoch (t)

Backpropagation

$$\frac{\partial \mathcal{L}}{\partial a_1} = \frac{\partial \mathcal{L}}{\partial a_2} \cdot \frac{\partial a_2}{\partial a_1}$$

$$\frac{\partial \mathcal{L}}{\partial \theta_1} = \frac{\partial \mathcal{L}}{\partial a_1} \cdot \frac{\partial a_1}{\partial \theta_1}$$



Exampl

e $\mathcal{L}(y, a^3) = 0.5 \|y - a^3\|^2$

$$a^2 = \sigma(w^2 x^2)$$

$$a^1 = \sigma(w^1 x^1)$$

$$\frac{\partial a^2}{\partial a^1} = \frac{\partial a^2}{\partial a^2} = w^2 a^2 (1 - a^2)$$

$$\frac{\partial a^1}{\partial w^1} = x^1 a^1 (1 - a^1)$$

$$\frac{\partial \mathcal{L}}{\partial a^1} = \frac{\partial \mathcal{L}}{\partial a^2} w^2 a^2 (1 - a^2)$$

$$\frac{\partial \mathcal{L}}{\partial w^1} = \frac{\partial \mathcal{L}}{\partial w^1} x^1 a^1 (1 - a^1)$$

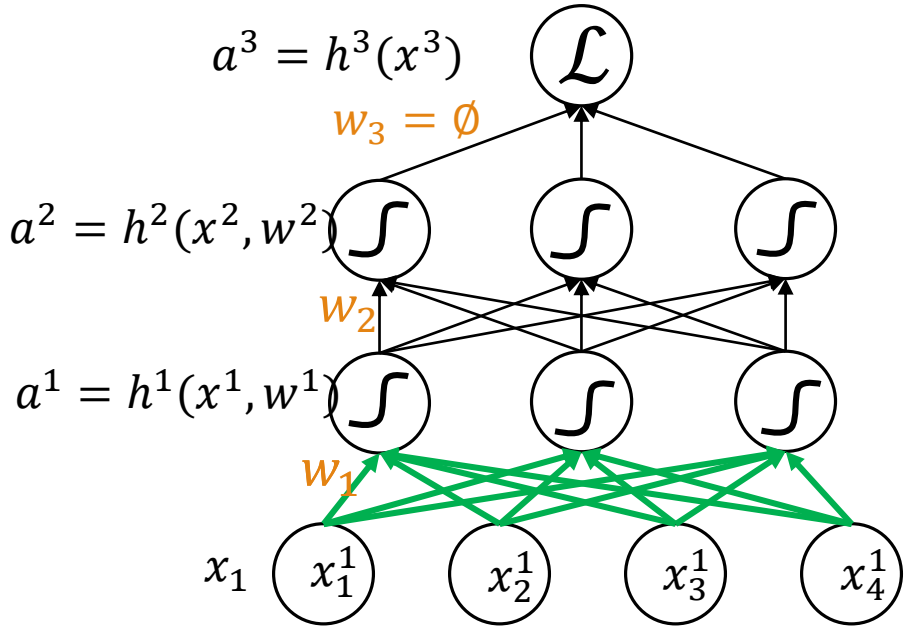
Computed from the exact previous backpropagation step (Remember, recursive rule)

Backpropagation visualization at epoch $(t+1)$?

Backpropagation visualization at epoch $(t+1)$

Forward propagations

Compute and store $a_1 = h_1(x_1)$



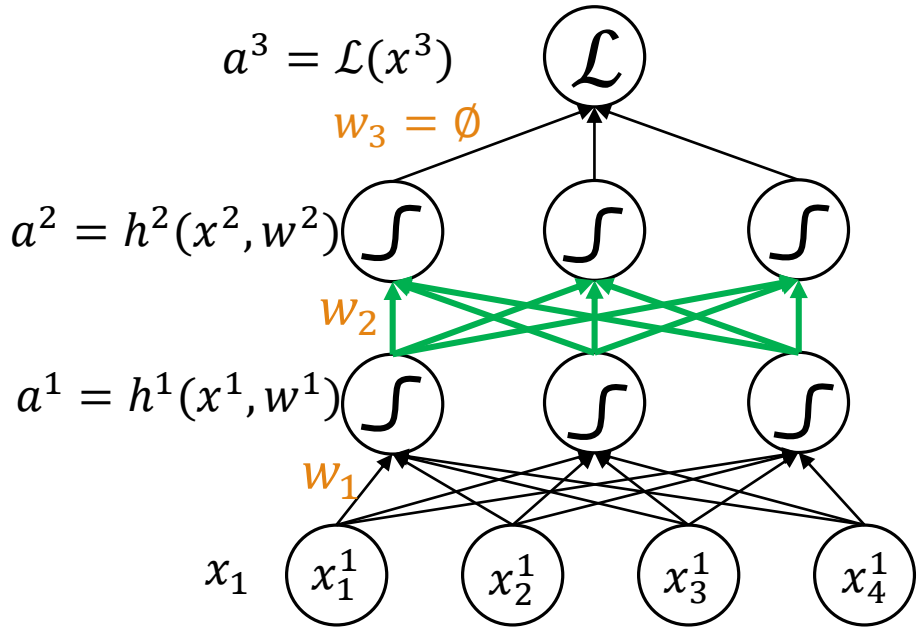
$a^1 = \sigma(w^1 x^1)$

Store!!!

Backpropagation visualization at epoch $(t+1)$

Forward propagations

Compute and store $a_2 = h_2(x_2)$



$a^1 = \sigma(w^1 x^1)$

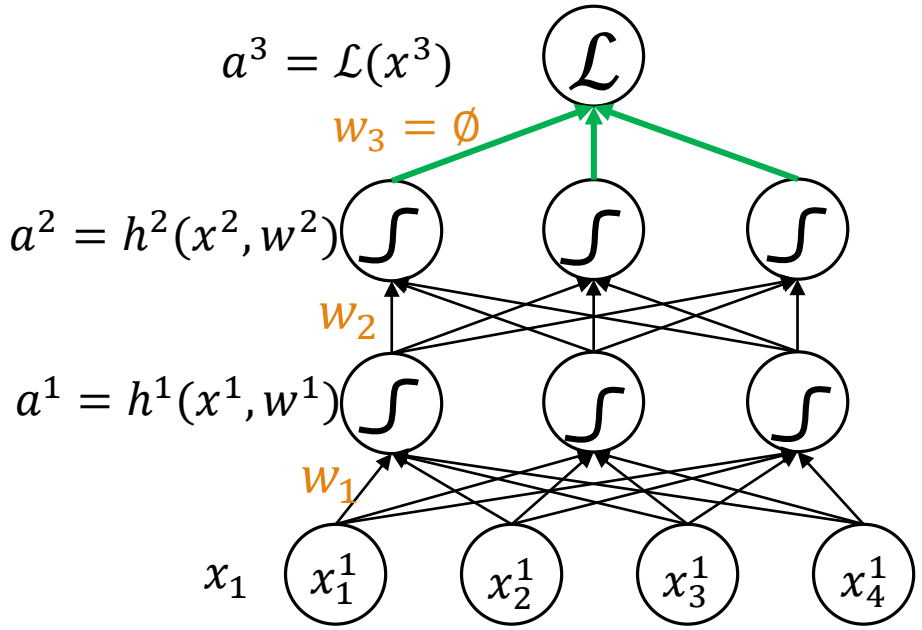
$a^2 = \sigma(w^2 x^2)$

Store!!!

Backpropagation visualization at epoch $(t+1)$

Forward propagations

Compute and store $a_3 = h_3(x_3)$



$$a^1 = \sigma(w^1 x^1)$$
$$a^2 = \sigma(w^2 x^2)$$
$$\mathcal{L}(y, a^2) = \|y - a^2\|^2 = \|y - x^3\|^2$$

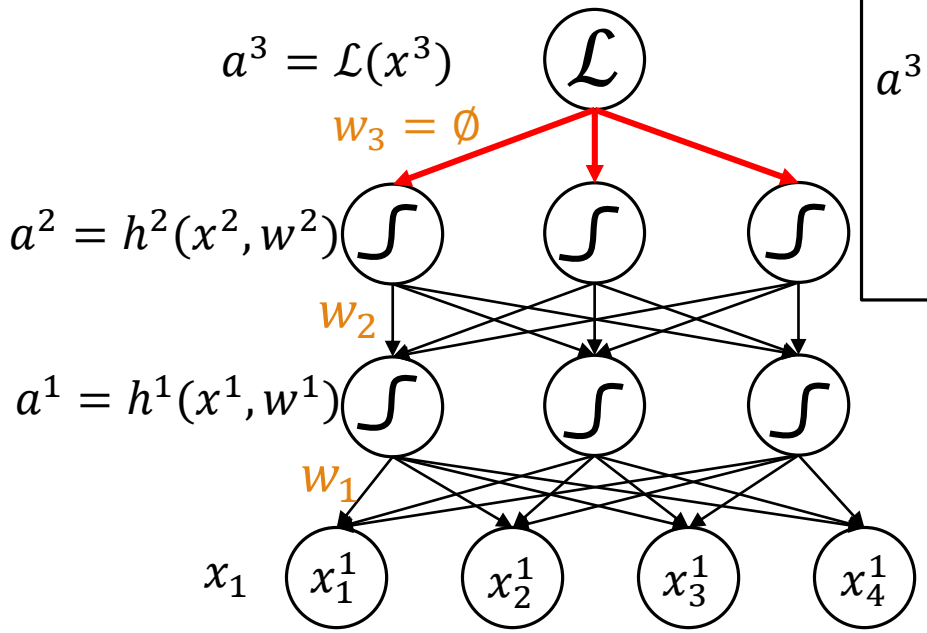
Store!!!

Backpropagation visualization at epoch $(t+1)$

Backpropagation

$\frac{\partial \mathcal{L}}{\partial a^3} =$ ... ← Direct computation

~~$\frac{\partial \mathcal{L}}{\partial w^3}$~~



$a^3 = \mathcal{L}(y, x^3) = 0.5 \|y - x^3\|^2$

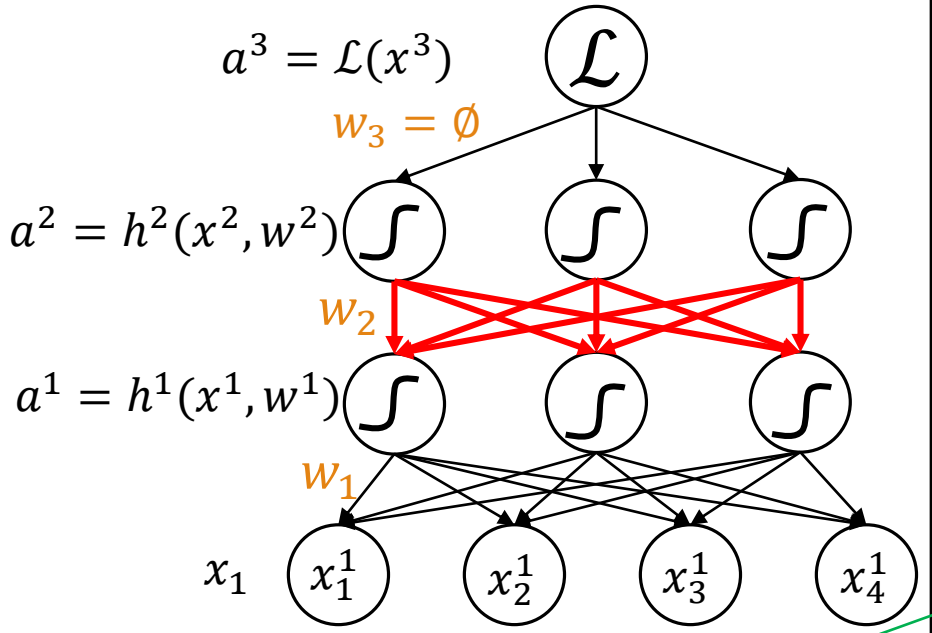
$\frac{\partial \mathcal{L}}{\partial x^3} = -(y - x^3)$

Backpropagation visualization at epoch $(t+1)$

Backpropagation

$$\frac{\partial \mathcal{L}}{\partial a^2} = \frac{\partial \mathcal{L}}{\partial a^3} \cdot \frac{\partial a^3}{\partial a^2}$$

$$\frac{\partial \mathcal{L}}{\partial w^2} = \frac{\partial \mathcal{L}}{\partial a^2} \cdot \frac{\partial a^2}{\partial w^2}$$



$$\mathcal{L}(y, x_3) = 0.5 \, \|y - x_3\|^2$$

$$a^2 = \sigma(w^2 x^2)$$

$$\frac{\partial \mathcal{L}}{\partial a^2} = \frac{\partial \mathcal{L}}{\partial x^3} = -(y - x_3)$$

$$\partial\sigma(x) = \sigma(x)(1 - \sigma(x))$$

$$\frac{\partial a^2}{\partial w^2} = x^2 \sigma(w^2 x^2)(1 - \sigma(w^2 x^2))$$

$$= x^2 a^2 (1 - a^2)$$

$$\frac{\partial \mathcal{L}}{\partial a^2} = -(y - x^3)$$

$$\frac{\partial \mathcal{L}}{\partial w^2} = \frac{\partial \mathcal{L}}{\partial a^2} x^2 a^2 (1 - a^2)$$

Stored during forward computations

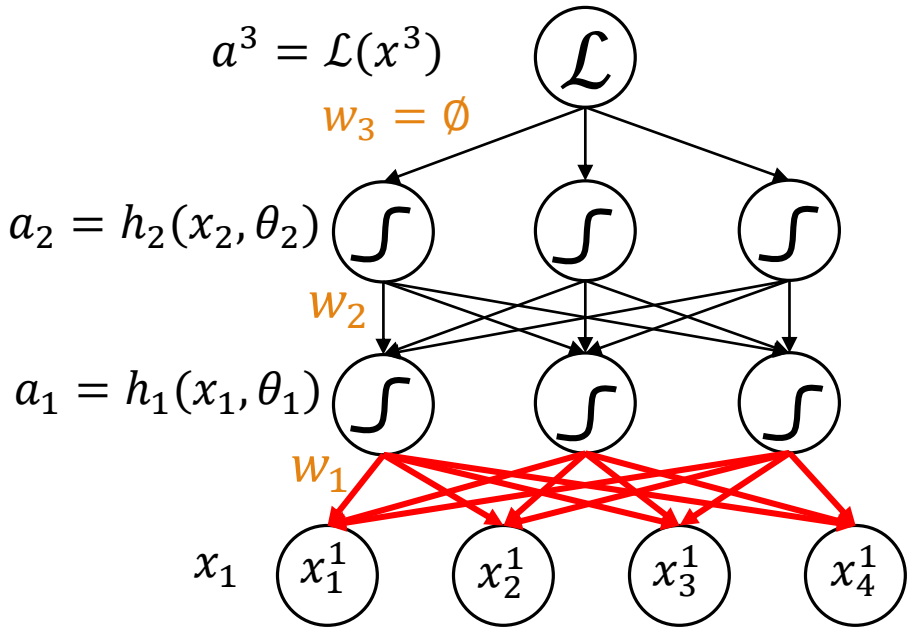
Store!!!

Backpropagation visualization at epoch $(t+1)$

Backpropagation

$$\frac{\partial \mathcal{L}}{\partial a_1} = \frac{\partial \mathcal{L}}{\partial a_2} \cdot \frac{\partial a_2}{\partial a_1}$$

$$\frac{\partial \mathcal{L}}{\partial \theta_1} = \frac{\partial \mathcal{L}}{\partial a_1} \cdot \frac{\partial a_1}{\partial \theta_1}$$



$$\mathcal{L}(y, a^3) = 0.5 \, \|y - a^3\|^2$$

$$a^2 = \sigma(w^2 x^2)$$

$$a^1 = \sigma(w^1 x^1)$$

$$\frac{\partial a^2}{\partial a^1} = \frac{\partial a^2}{\partial a^2} = w^2 a^2 (1 - a^2)$$

$$\frac{\partial a^1}{\partial w^1} = x^1 a^1 (1 - a^1)$$

$$\frac{\partial \mathcal{L}}{\partial a^1} = \frac{\partial \mathcal{L}}{\partial a^2} w^2 a^2 (1 - a^2)$$

$$\frac{\partial \mathcal{L}}{\partial w^1} = \frac{\partial \mathcal{L}}{\partial w^1} x^1 a^1 (1 - a^1)$$

Computed from the exact previous backpropagation step (Remember, recursive rule)

Dimension analysis

- To make sure everything is done correctly → "Dimension analysis"
- The dimensions of the gradient w.r.t. w^l must be equal to the dimensions of the respective weight w^l

$$\dim\left(\frac{\partial \mathcal{L}}{\partial a^l}\right) = \dim(a^l)$$

$$\dim\left(\frac{\partial \mathcal{L}}{\partial w^l}\right) = \dim(w^l)$$

Dimension analysis

- For $\frac{\partial \mathcal{L}}{\partial a^l} = \left(\frac{\partial a^{l+1}}{\partial x^{l+1}}\right)^T \frac{\partial \mathcal{L}}{\partial a^{l+1}}$

$$\dim(a^l) = d_l$$
$$\dim(w^l) = d_{l-1} \times d_l$$

$$[d_l \times 1] = [d_{l+1} \times d_l]^T \cdot [d_{l+1} \times 1]$$

- For $\frac{\partial \mathcal{L}}{\partial w^l} = \frac{\partial a^l}{\partial w^l} \cdot \left(\frac{\partial \mathcal{L}}{\partial w^l}\right)^T$

$$[d_{l-1} \times d_l] = [d_{l-1} \times 1] \cdot [1 \times d_l]$$

So, Backprop, what's the big deal?

- Backprop is as simple as it is complicated
- Mathematically, just the chain rule
  - Found some time around the 1700s by I. Newton and Leibniz, who invented calclulus
  - That simple, that we can even automate it ("reverse-mode differentiation")

- However, why is it that we can train a highly non-complex machine with many local optima, like neural nets, with a strongly local learning algorithm like Backprop?
  - Why even is it a good choice?
  - Not really known, even today

Summary

- Modularity in Neural Networks
- Neural Network Modules
- Neural Network Cheatsheet
- Backpropagation

Reading material

- Chapter 6
- Efficient Backprop, LeCun et al., 1998