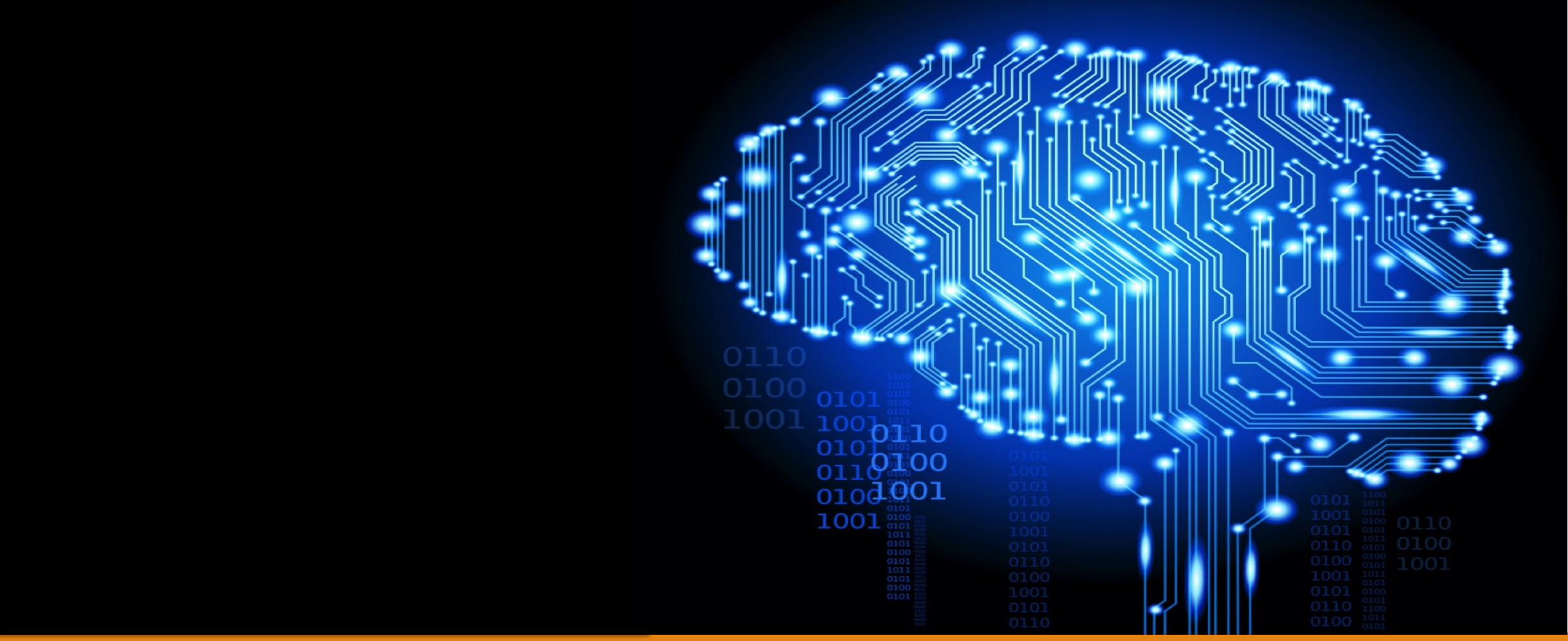


Lecture 2: Modular Learning

Deep Learning @ UvA

Lecture Overview

- Machine Learning Paradigm for Neural Networks
- Neural Networks as modular architectures
- Neural Network Modules and Theory
- The Backpropagation algorithm for learning with a neural network
- How to implement and check your very own module

The Machine Learning Paradigm

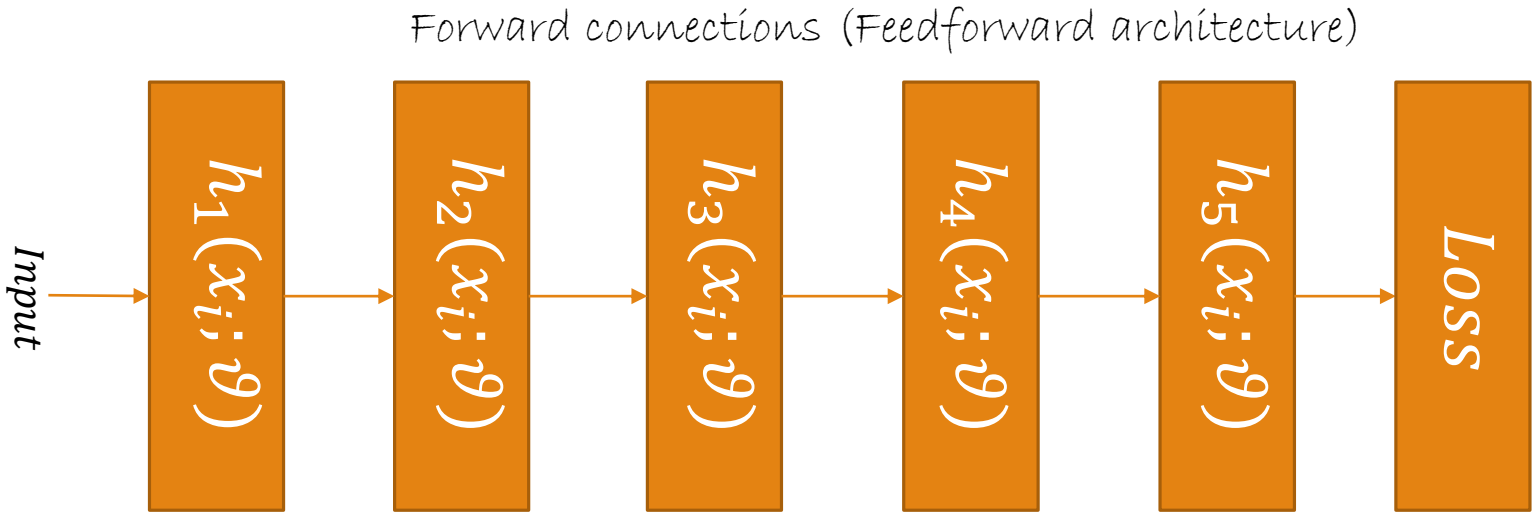
What is a neural network again?

- 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.
- $a_L\left(x; \theta_{1,\dots,\mathrm{L}}\right) = h_L\left(h_{L-1}(\dots h_1(x, \theta_1), \theta_{L-1}),\ \theta_L\right)$
  - x:input, θ_l: parameters for layer l, $a_l = h_l(x, \theta_l)$: (non-)linear function
- Given training corpus $\{X, Y\}$ find optimal parameters

$$\theta^* \leftarrow \arg\min_\theta \sum_{(x,y)\subseteq(X,Y)} \ell\left(y, a_L\left(x; \theta_{1,\dots,\mathrm{L}}\right)\right)$$

Neural network models

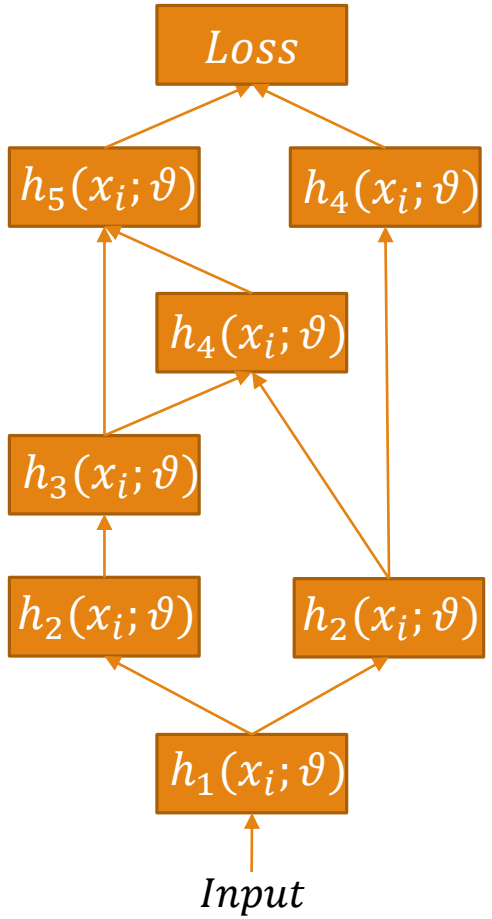
- A neural network model is a series of hierarchically connected functions
- This hierarchies can be very, very complex



Neural network models

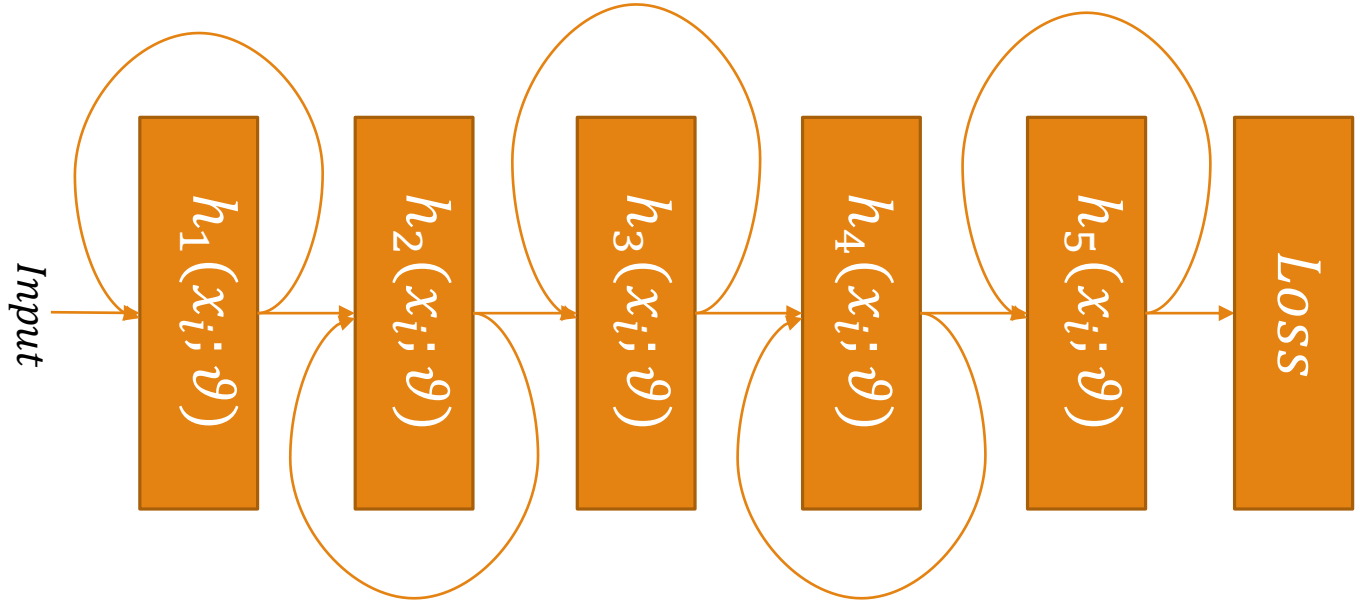
- A neural network model is a series of hierarchically connected functions
- This hierarchies can be very, very complex

Interweaved connections (Directed Acyclic Graph architecture- DAGNN)



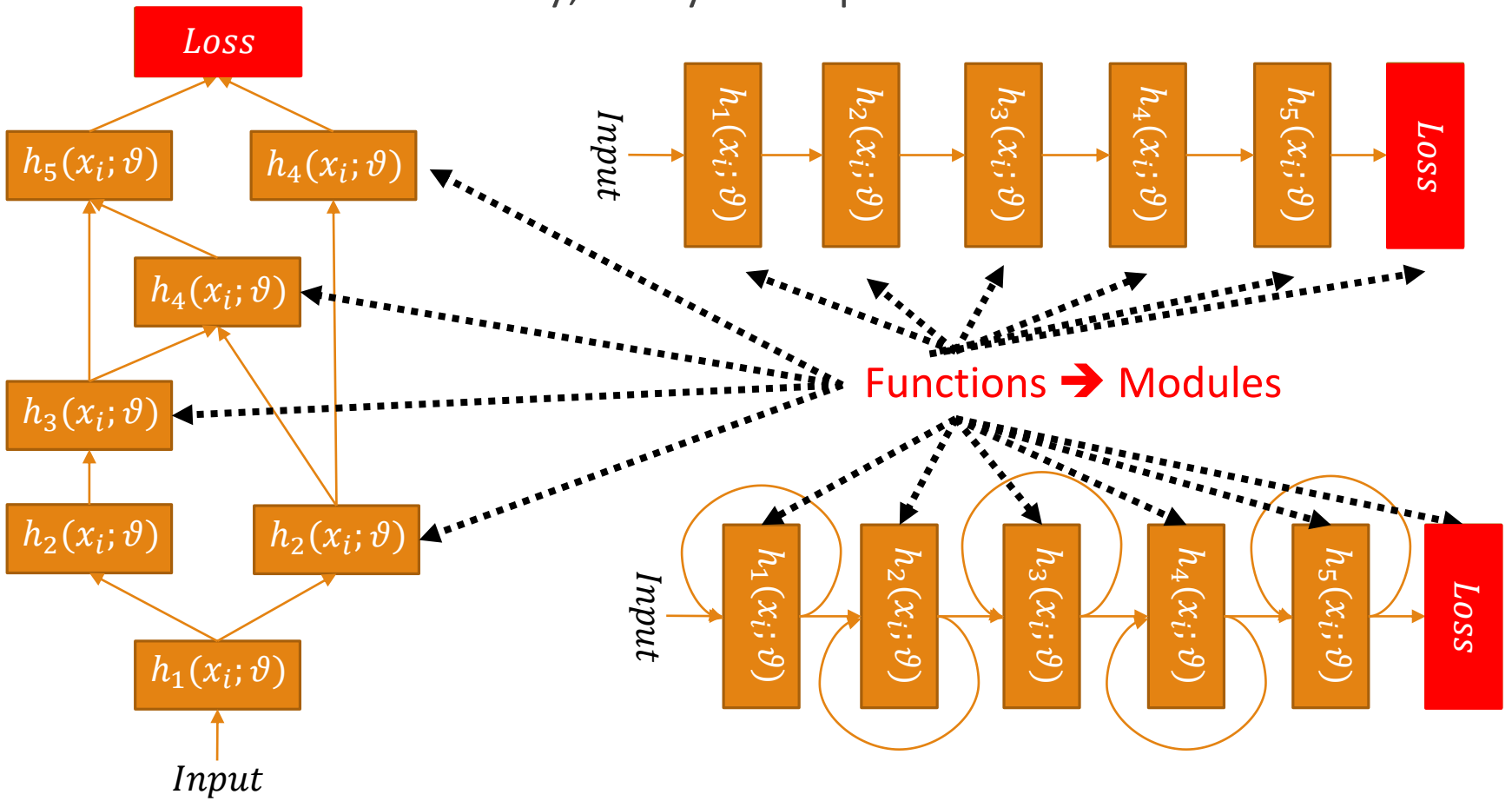
Neural network models

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



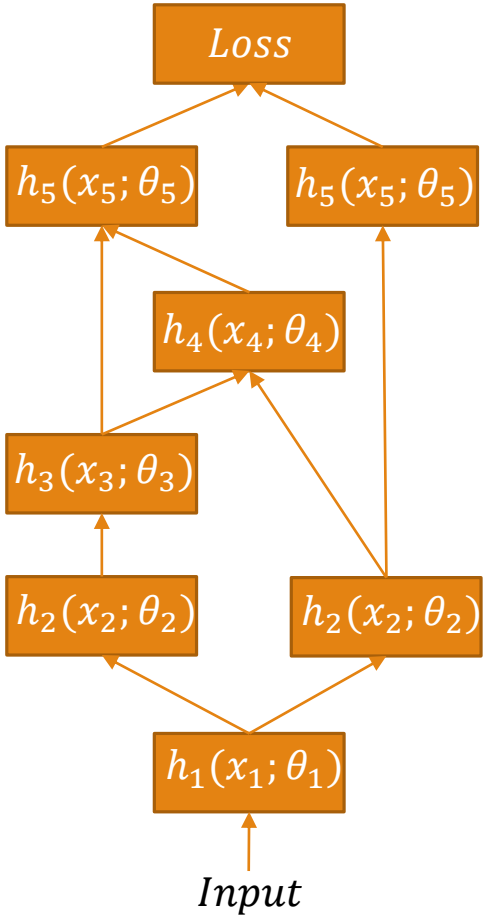
Neural network models

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



What is a module?

- A module is a building block for our network
- Each module is an object/function $a = h(x; \theta)$ that
  - Contains trainable parameters (θ)
  - 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?

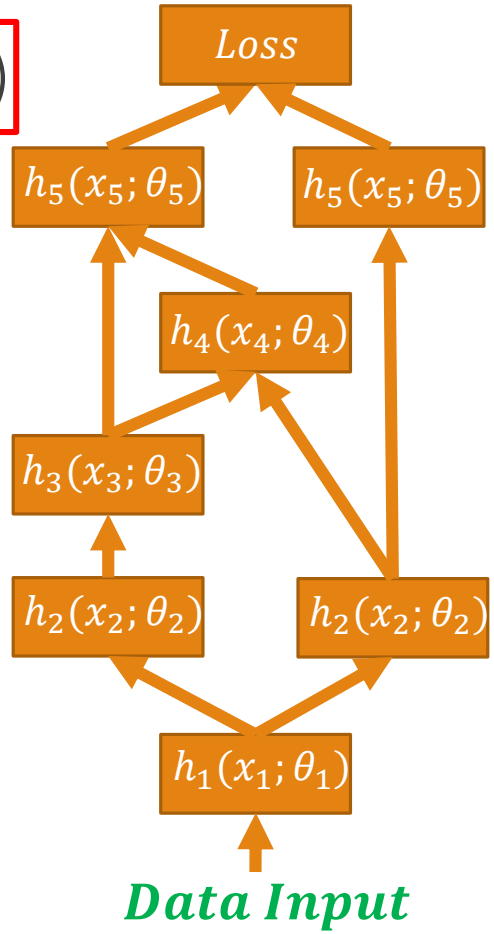
- A neural network is a composition of modules (building blocks)
- Any architecture works
- If the architecture is a feedforward cascade, no special care
- If acyclic, there is right order of computing the forward computations
- If there are loops, these form **recurrent** connections (revisited later)

Forward computations for neural networks

- Simply compute the activation of each module in the network

$$a_l = h_l(x_l; \vartheta), \text{ where } a_l = x_{l+1} (\text{or } x_l = a_{l-1})$$

- 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 θ? **Gradient-based learning**

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

$$\theta^* = \arg\max_{\theta} \prod_i p(\theta; x_i, y_i)$$

- Taking the logarithm, this means minimizing the negative log-likelihood cost function

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

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

Prepare to vote

Internet

1

2

This presentation has been loaded without the Shakespeak add-in.
Want to download the add-in for free? Go to http://shakespeak.com/en/free-download/.

TXT

1

2

Voting is anonymous

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

A. ~ |y-f(θ; x)|^2
B. ~ max{0, 1-y f(θ; x)}
C. ~ |y-f(θ; x)|_1
D. ~ |y-f(θ; x)|^2+λΩ(θ)

The question will open when you start your session and slideshow.

Closed

shakespeak

Internet

TXT

This presentation has been loaded without the Shakespeak add-in.
Want to download the add-in for free? Go to http://shakespeak.com/en/free-download/.

Votes: 0

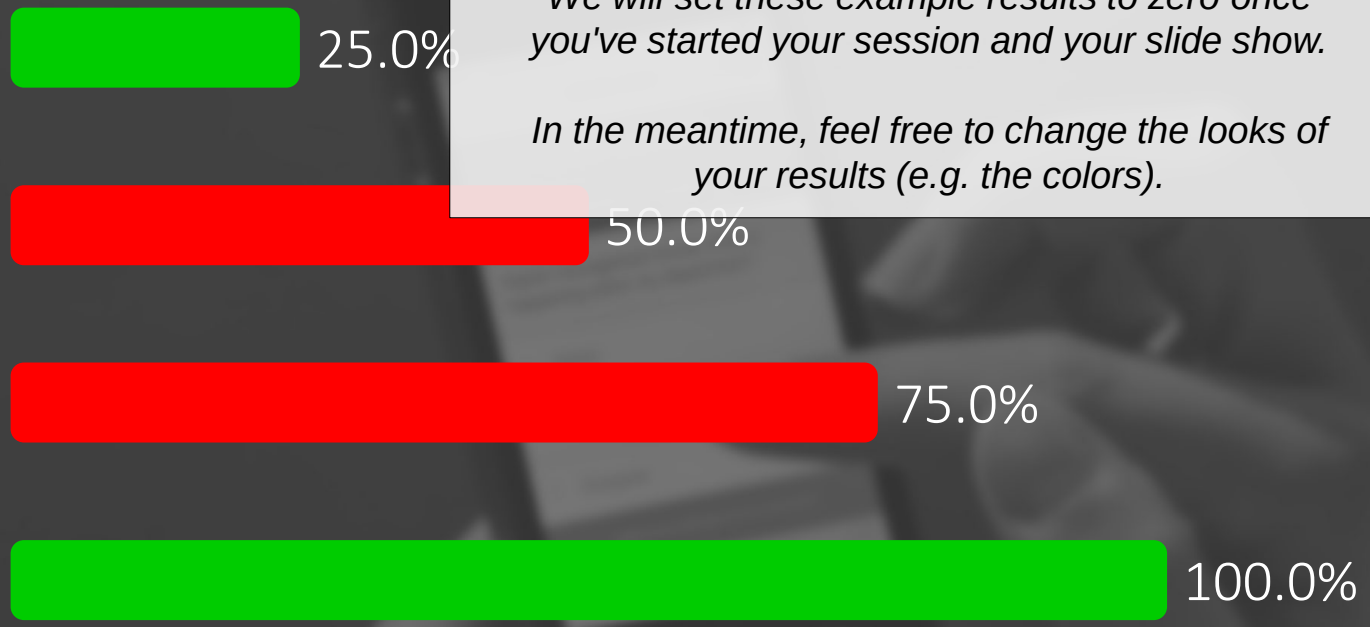
If our last layer is the Gaussian function N(y; f(θ; x), I) what could be our cost function like? (Multiple answers possible)

A. ~ |y-f(θ; x)|^2

B. ~ max{0, 1-y f(θ; x)}

C. ~ |y-f(θ; x)|_1

D. ~ |y-f(θ; x)|^2+λΩ(θ)



We will set these example results to zero once you've started your session and your slide show.

In the meantime, feel free to change the looks of your results (e.g. the colors).

Closed

shakespeak

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

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

$$\theta^* = \arg\max_{\theta} \prod_i p(\theta; x_i, y_i)$$

- Taking the logarithm, this means minimizing the cost function

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

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

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

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

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

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

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

Why do we want to match the form of the cost function with the form of learnt neural network function? (Multiple answers possible)

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

The question will open when you start your session and slideshow.

Closed

shakespeak

Internet

TXT

This presentation has been loaded without the Shakespeak add-in.
Want to download the add-in for free? Go to http://shakespeak.com/en/free-download/.

Votes: 0

Why do we want to match the form of the cost function with the form of learnt neural network function? (Multiple answers possible)

A. Otherwise one cannot use standard tools, like automatic differentiation, in packages like Tensorflow or Pytorch

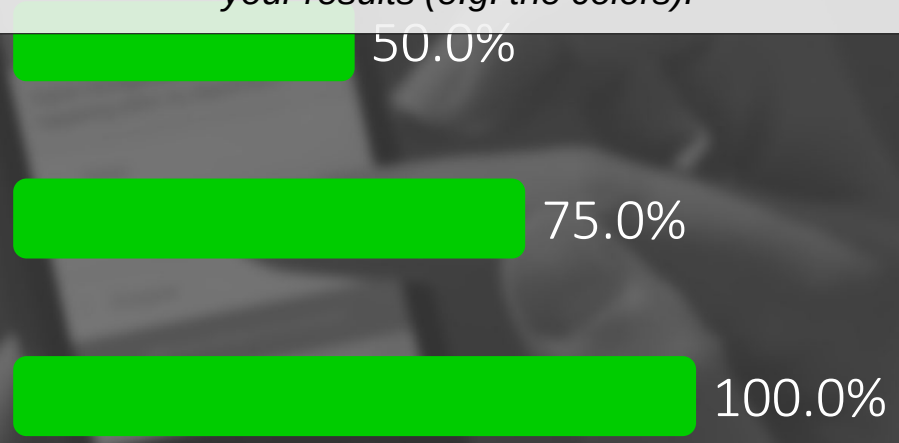
B. It makes the math simpler

C. It avoids numerical instabilities

D. It makes gradients large by avoiding functions saturating, thus learning is stable

We will set these example results to zero once you've started your session and your slide show.

In the meantime, feel free to change the looks of your results (e.g. the colors).



Closed

shakespeak

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

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

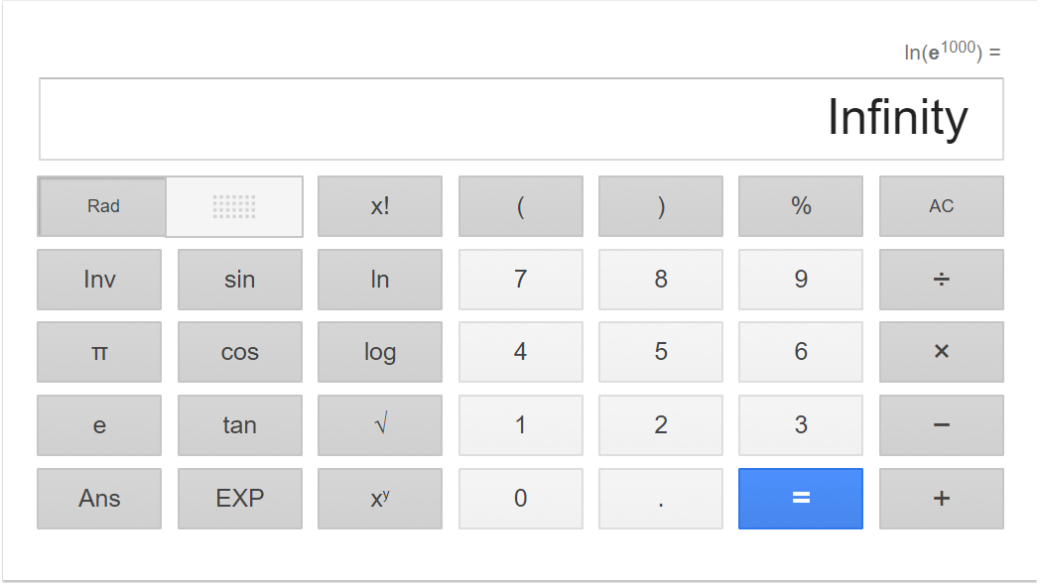
$$\log p_{model}(y|x) = \log \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{-|y-f(\theta;x)|^2}{2\sigma^2}\right) \Longrightarrow$$

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

- Everything gets much simpler when the learned (neural network) function p_{model} matches the cost function $\mathcal{L}(\theta)$
- 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

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

- $\exp(1000) = \infty \Rightarrow \log \infty$

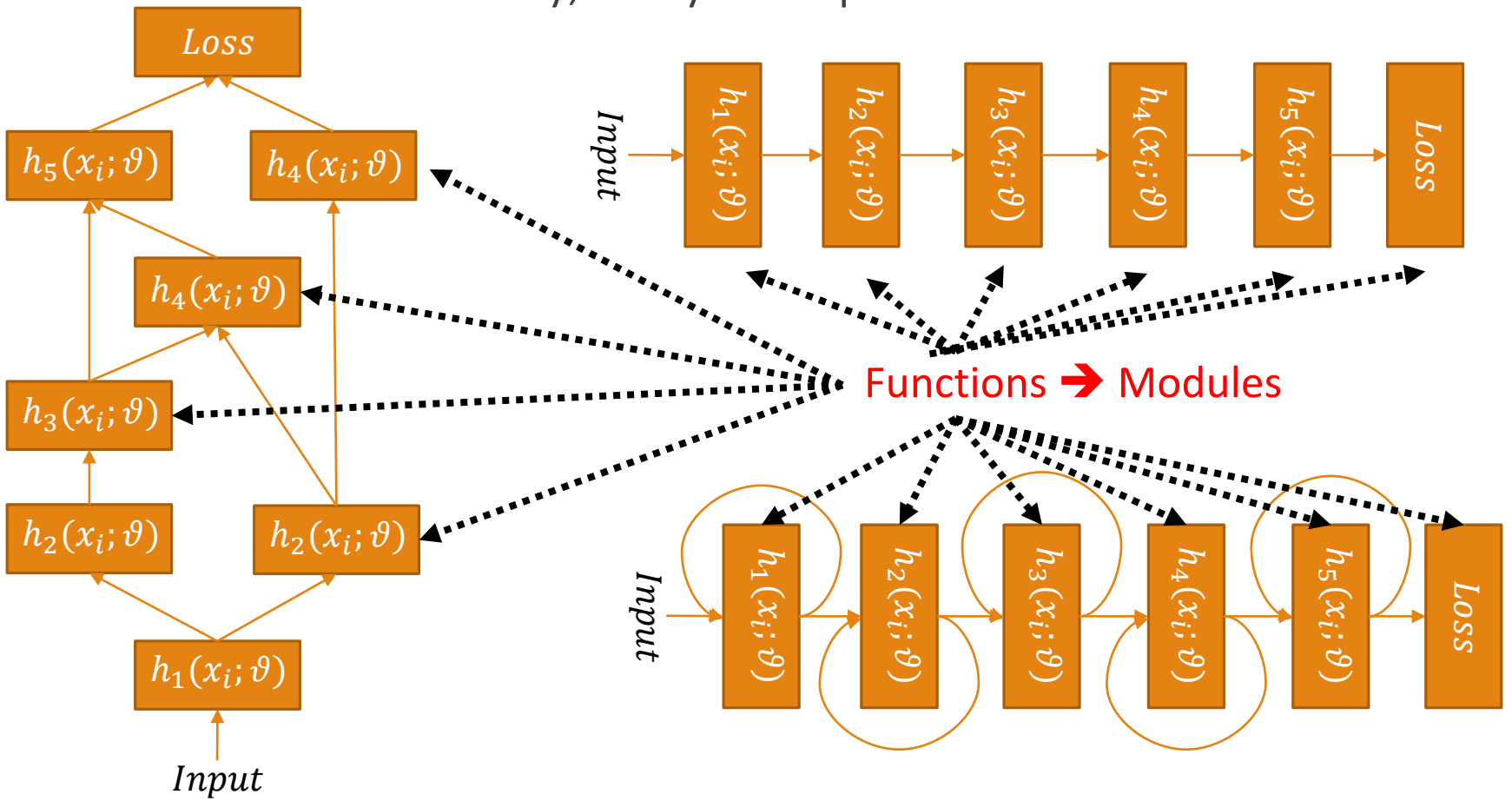


- $\log\big(\exp(1000)\big) = 1000$

Everything is a *module*

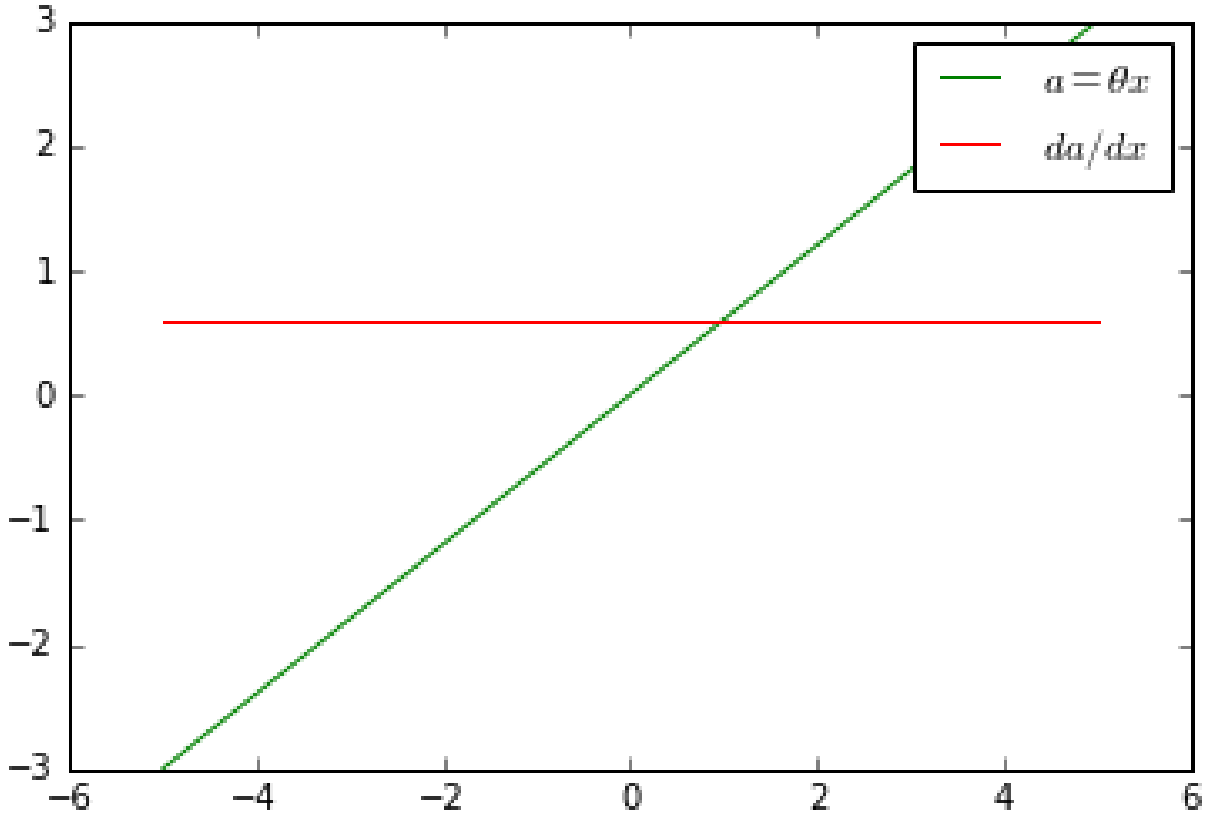
Neural network models

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



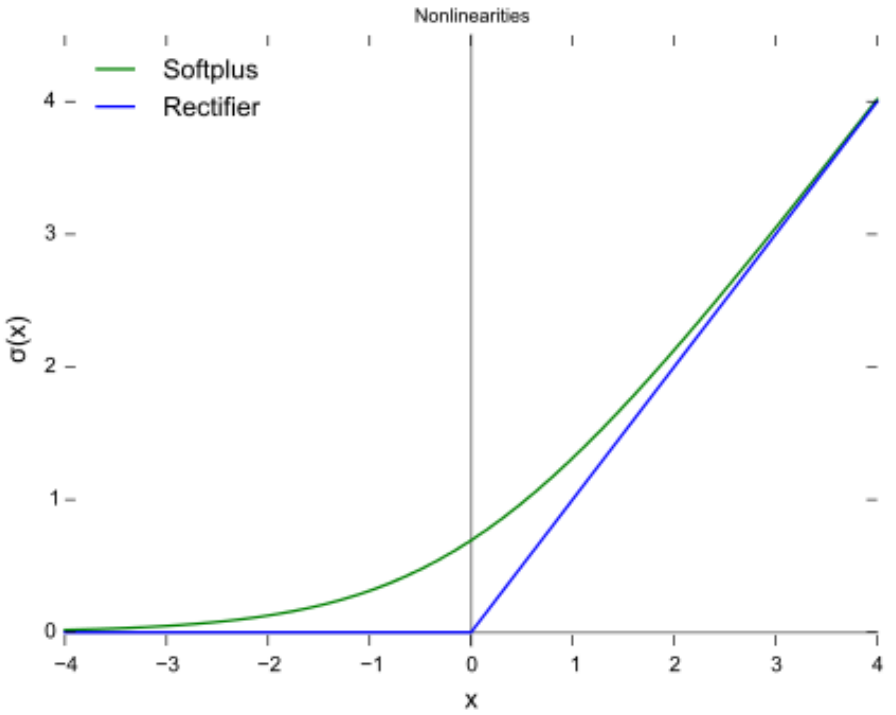
Linear module

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



What characterizes the Rectified Linear Unit?

A. 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.
B. It is discontinuous, so it might cause numerical errors during training
C. It is piece-wise linear, so the "piece"-gradients are stable and strong
D. Since they are linear, their gradients can be computed very fast and speed up training.
E. They are more complex to implement, because an if condition needs to be introduced.

The question will open when you start your session and slideshow.

Closed

shakespeak

Internet This text box will be used to describe the different message coding methods.

TXT The applicable explanation will be selected after you have started the session.

This presentation has been loaded without the Shakespeak add-in.
Want to download the add-in for free? Go to http://shakespeak.com/en/free-download/.

Votes: 0

What characterizes the Rectified Linear Unit?

A. 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.

B. It is discontinuous, so it might cause numerical errors during training

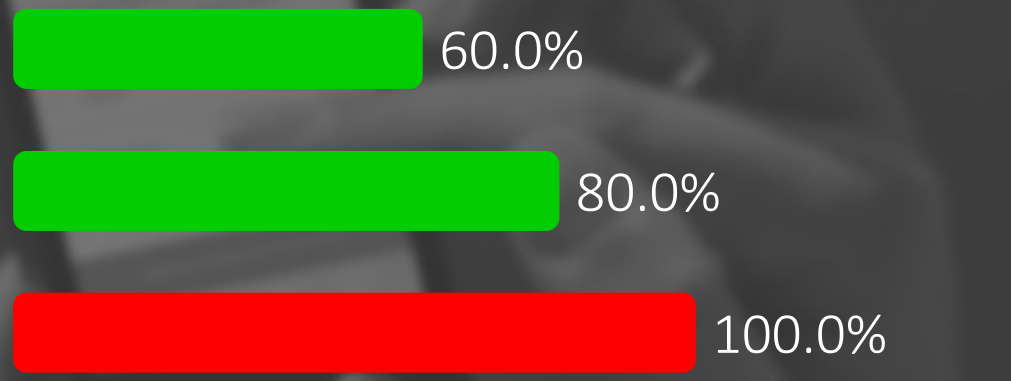
C. It is piece-wise linear, so the "piece"-gradients are stable and strong

D. Since they are linear, their gradients can be computed very fast and speed up training.

E. They are more complex to implement, because an if condition needs to be introduced.

We will set these example results to zero once you've started your session and your slide show.

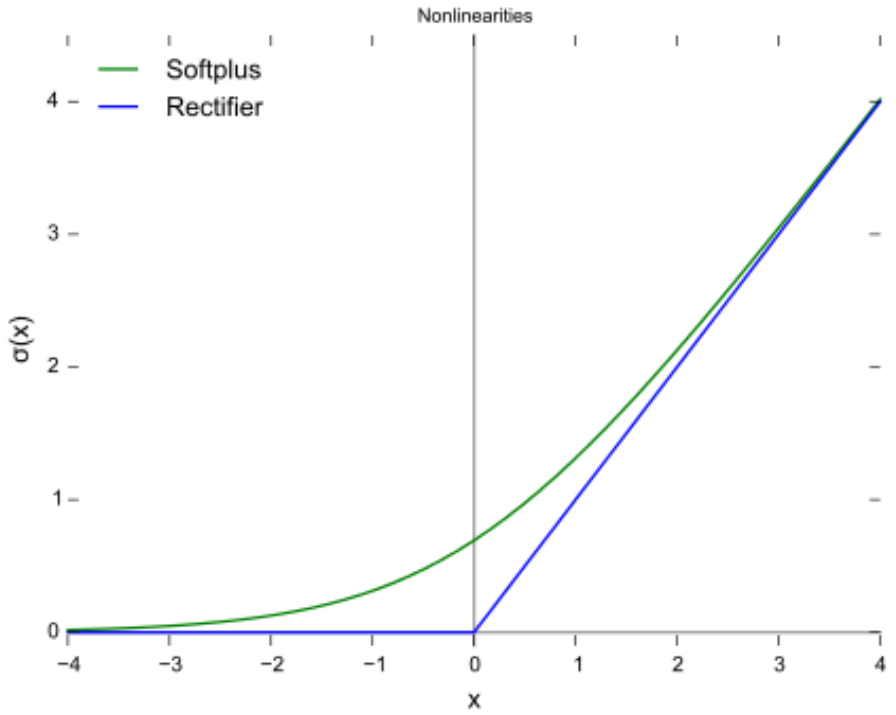
In the meantime, feel free to change the looks of your results (e.g. the colors).



shakespeak

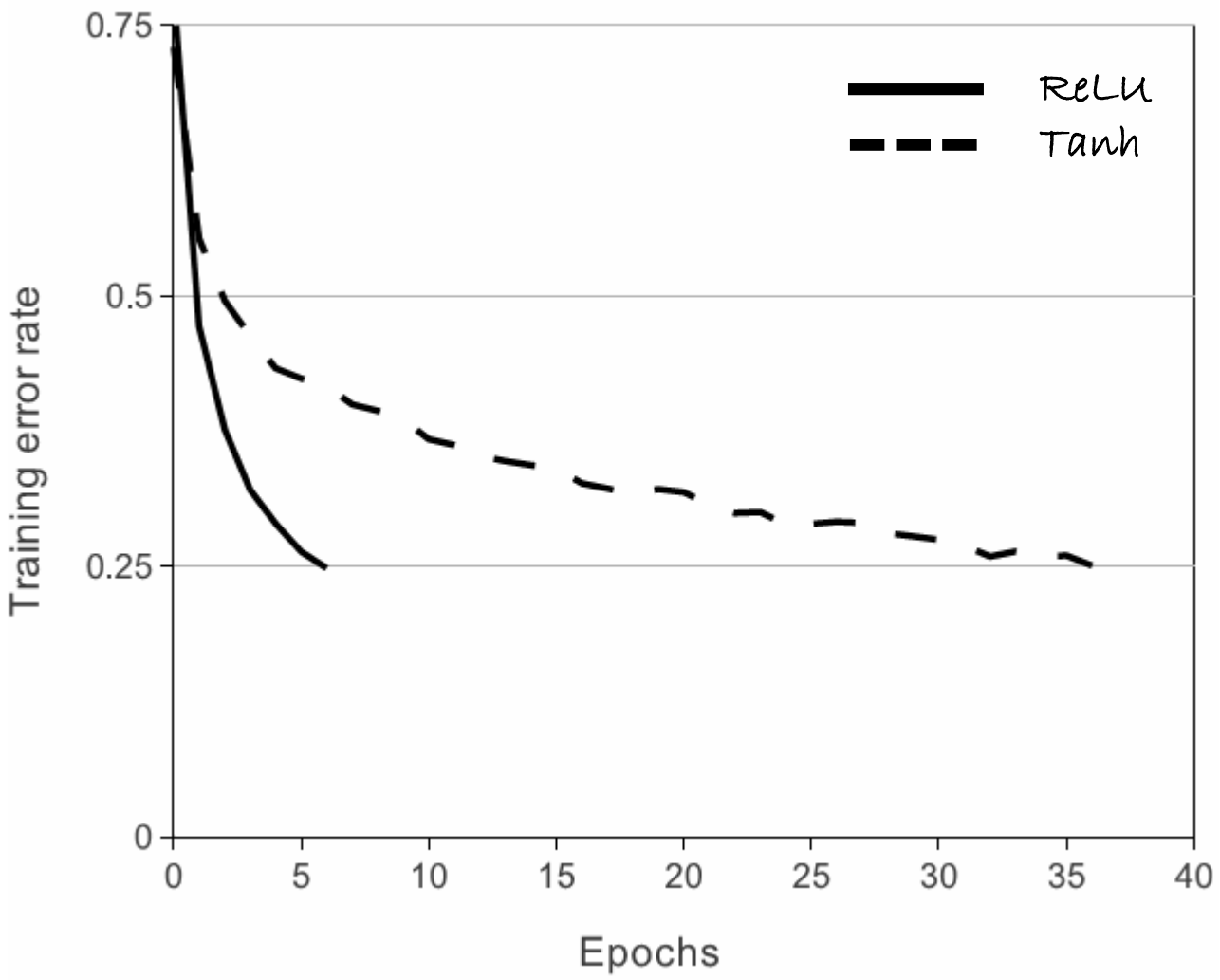
Rectified Linear Unit (ReLU) module

- 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 this is not a very big problem ☺
  - An activation of precisely 0 rarely happens with non-zero weights, and if it happens we choose a convention



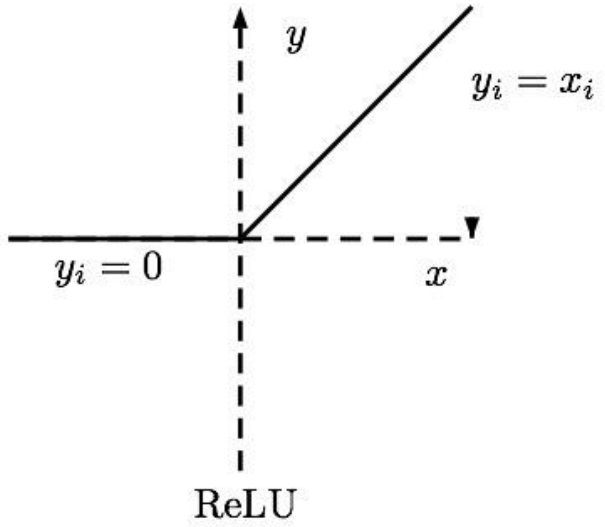
- Dead neurons is an issue
  - Large gradients might cause a neuron to die. Higher learning rates might be beneficial
  - Assuming a linear layer before ReLU $h(x) = \max(0, wx + b)$, make sure the bias term b is initialized with a small initial value, $e.g. 0.1$ → more likely the ReLU is positive and therefore there is non zero gradient
- Nowadays ReLU is the default non-linearity

ReLU convergence rate

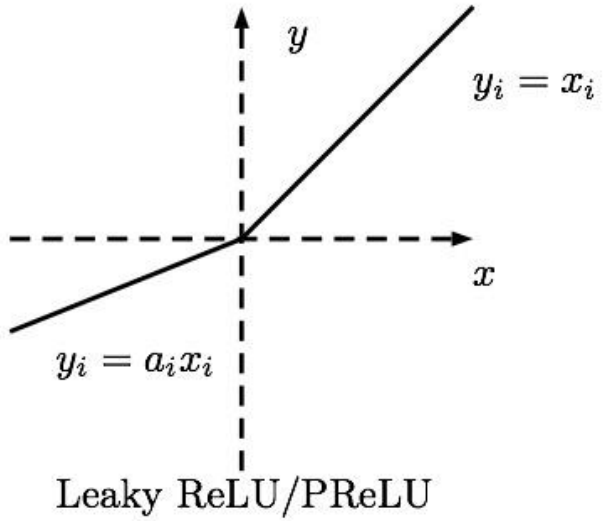


Other ReLUs

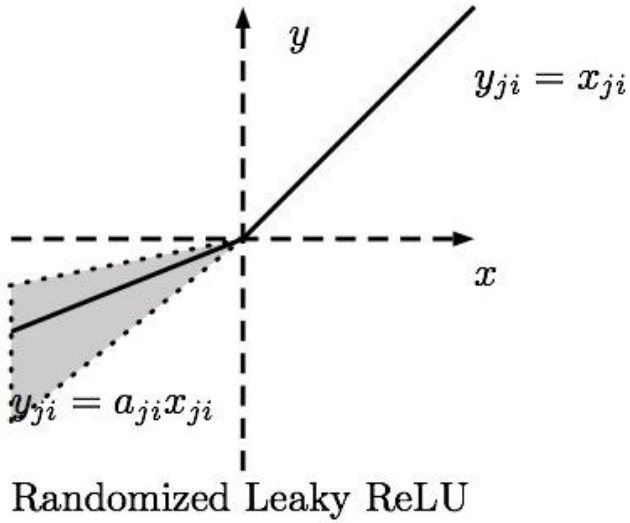
- Soft approximation (softplus): $a = h(x) = \ln(1 + e^x)$
- Noisy ReLU: $a = h(x) = \max(0, \mathrm{x} + \varepsilon), \varepsilon \sim N(0, \sigma(\mathrm{x}))$
- Leaky ReLU: $a = h(x) = \begin{cases} x, if\ x > 0 \\ 0.01x\ otherwise \end{cases}$
- Parametric ReLu: $a = h(x) = \begin{cases} x, if\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



ReLU



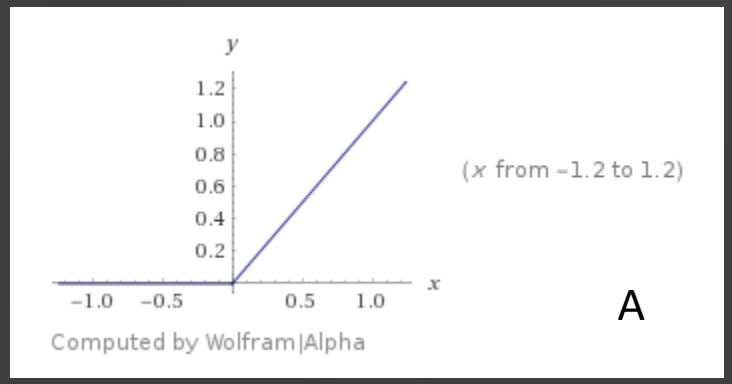
Leaky ReLU/PReLU



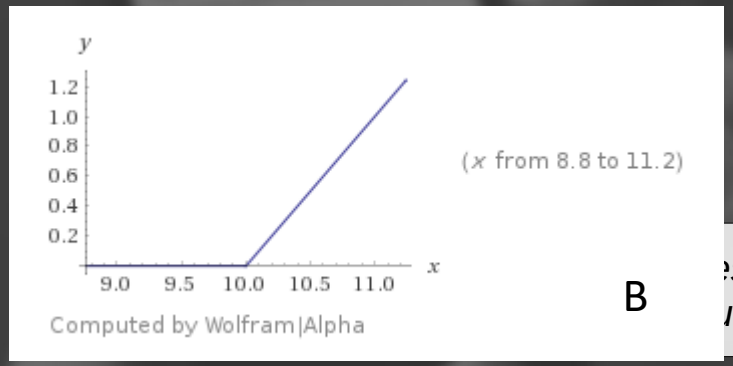
Randomized Leaky ReLU

How would you compare the two non-linearities?

A. They are equivalent for training because it's the same non-linearity, so the same type of gradient is computed.
B. They are not equivalent for training, because one of the two is shifted and creates mean activation imbalance



A



B

estion will open when you
ur session and slideshow.

Closed

shakespeak

Internet This text box will be used to describe the different message coding methods

TXT The applicable explanation will be selected after you start the question session

This presentation has been loaded without the Shakespeak add-in.
Want to download the add-in for free? Go to http://shakespeak.com/en/free-download/.

Votes: 0

How would you compare the two non-linearities?

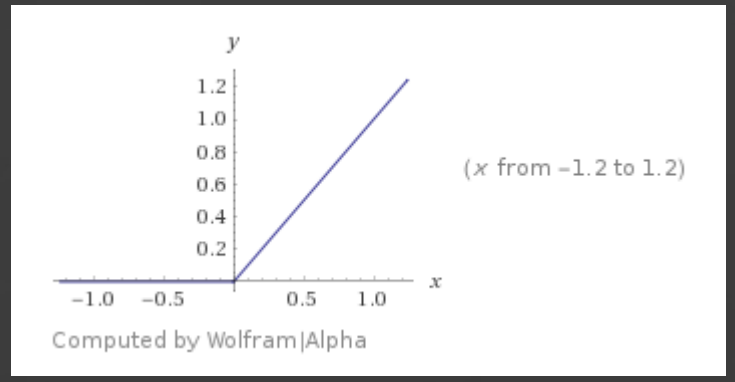
A. They are equivalent for training because it's the same non-linearity, so the same type of gradient is computed.

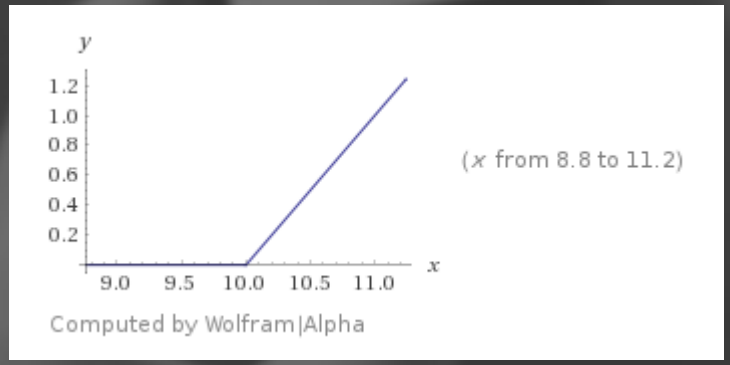
We will set these example results to zero once you've started your session and your slide show.

In the meantime, feel free to change the looks of your results (e.g. the colors).

B. They are not equivalent for training, because one of the two is shifted and creates mean activation imbalance

100.0%

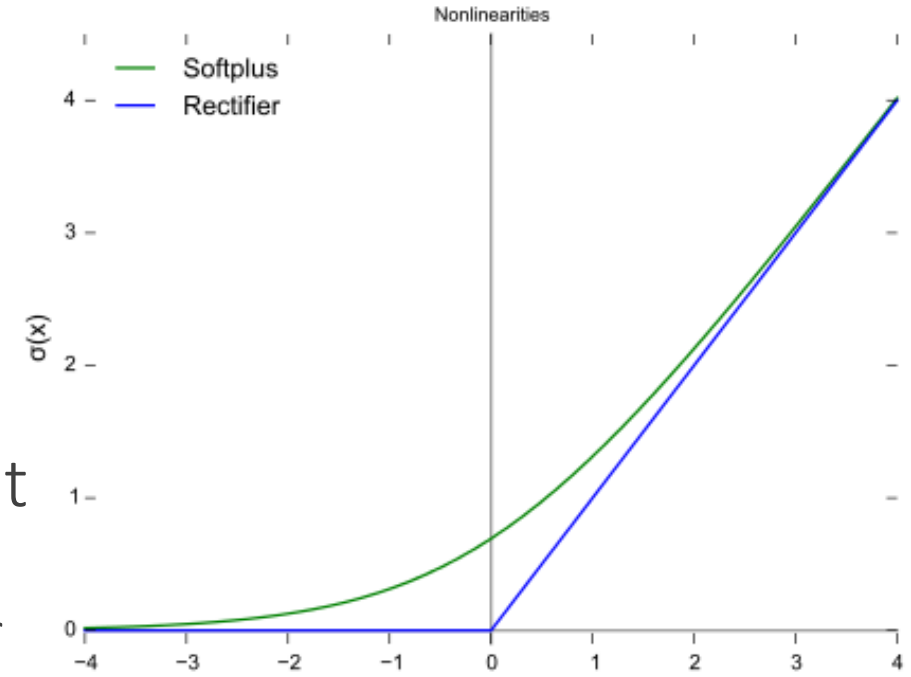




shakespeak

Closed

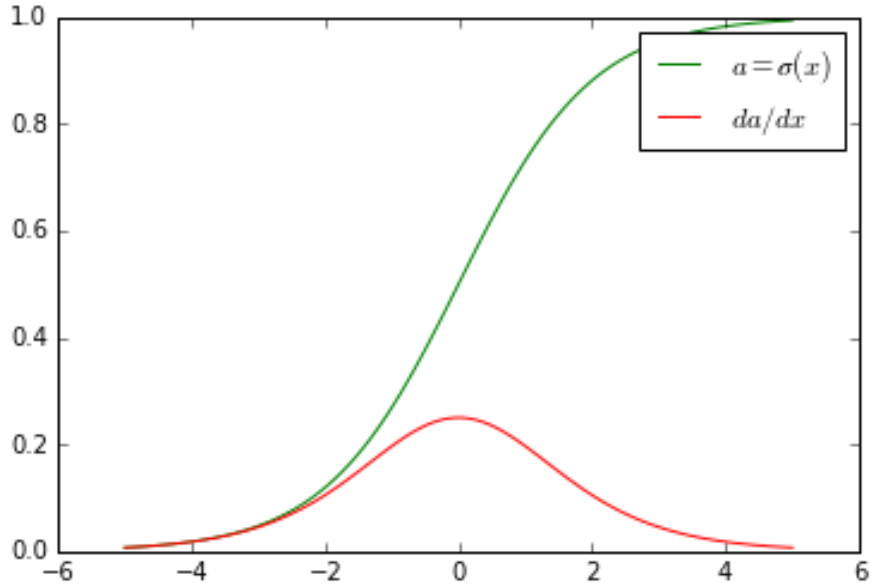
Centered non-linearities



- Remember: a deep network is a hierarchy of modules that are functionally the similar
  - Hence, they heavily affect one another, as one ReLU would be an input to another ReLU (roughly)
- To have consistent behavior, the input/output distributions must match
  - Otherwise, you will soon have inconsistent behavior
  - If one ReLU (1) neuron returns always highly positive numbers, e.g. $\sim 10{,}000$, then the subsequent ReLU (2) would be biased towards highly positive or highly negative values (depending on the weight w). So, the ReLU (2) would essentially become a linear unit.
- We want our non-linearities to be mostly activated around the origin (centered activations), as is the only way to encourage consistent behavior not matter the architecture

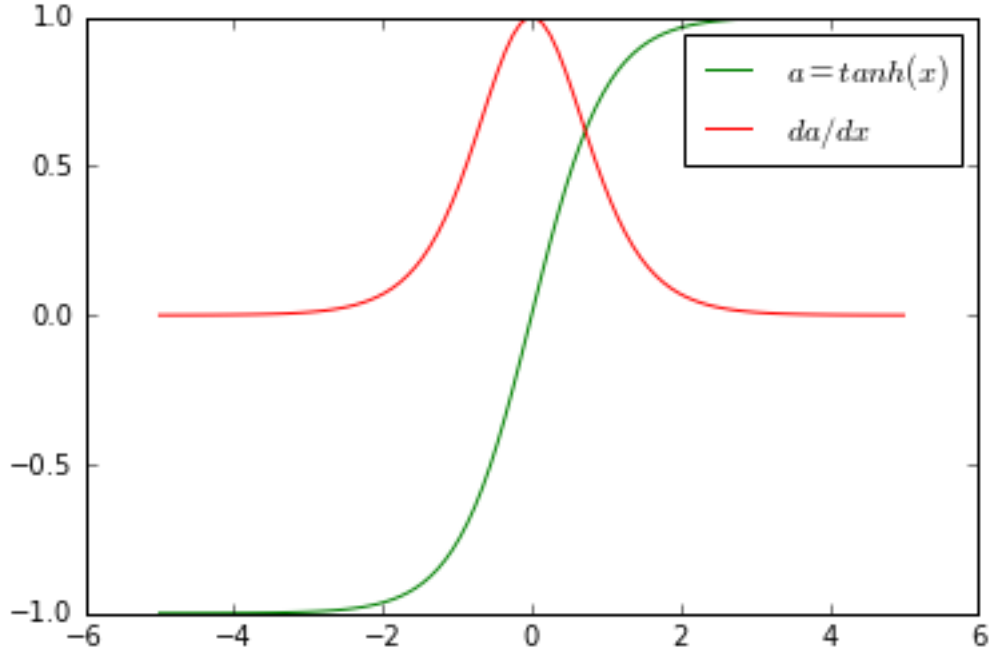
Sigmoid module

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



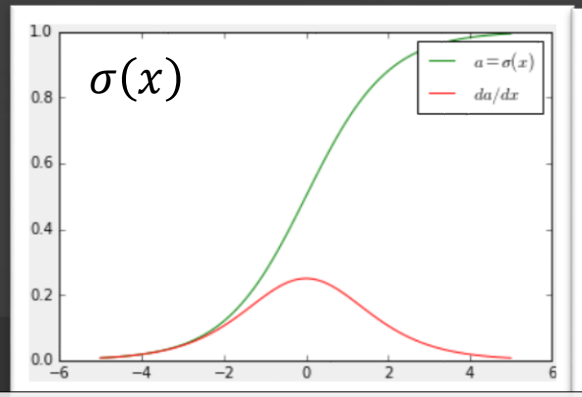
Tanh module

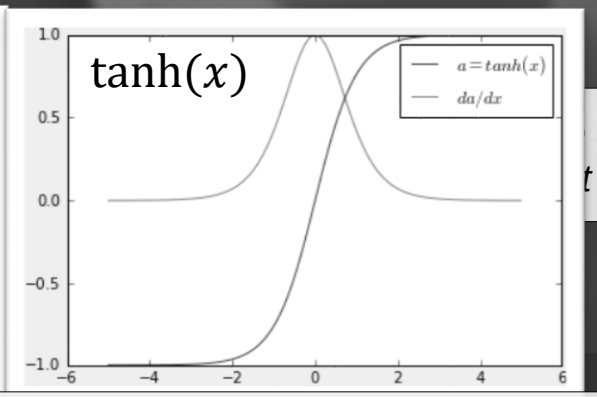
- Activation: $a = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- Gradient: $\frac{\partial a}{\partial x} = 1 - tanh^2(x)$



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

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





question will open when you t your session and slideshow.

Closed

shakespeak

Internet
TXT

This presentation has been loaded without the Shakespeak add-in.
Want to download the add-in for free? Go to http://shakespeak.com/en/free-download/.

Votes: 0

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

A. The tanh, because on the average activation case it has stronger gradients

B. The sigmoid, because it's output range [0, 1] resembles the range of probability values

C. The tanh, because the sigmoid can be rewritten as a tanh

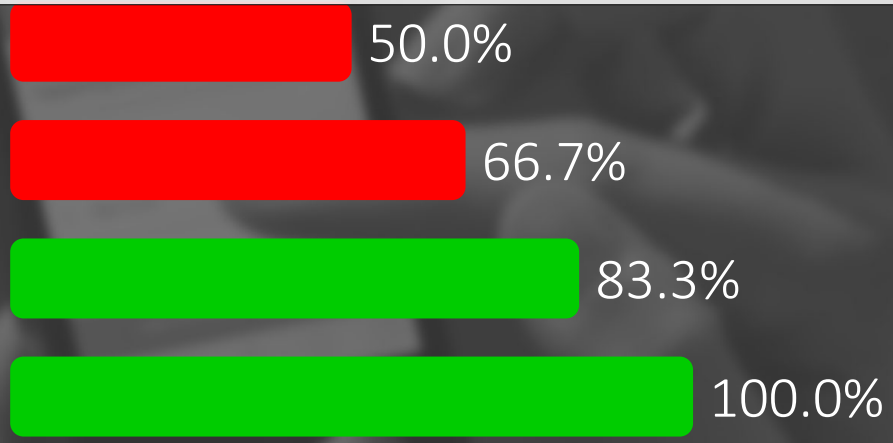
D. The sigmoid, because it has a simpler implementation of gradients

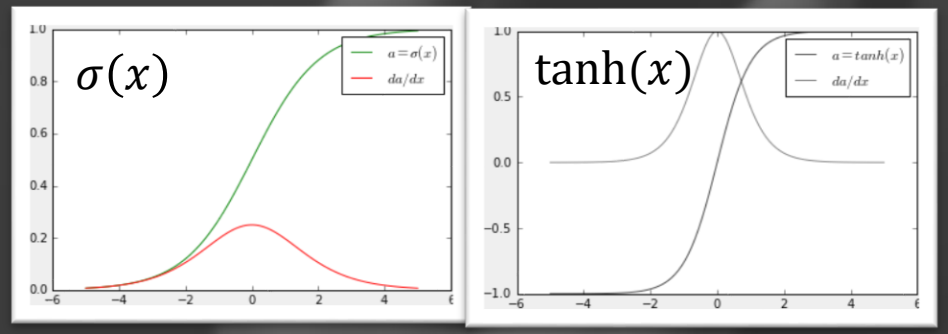
E. None of them are that great, they saturate for large or small inputs

F. The tanh, because it's mean activation is around 0 and it is easier to combine with other modules

We will set these example results to zero once you've started your session and your slide show.

In the meantime, feel free to change the looks of your results (e.g. the colors).





Closed

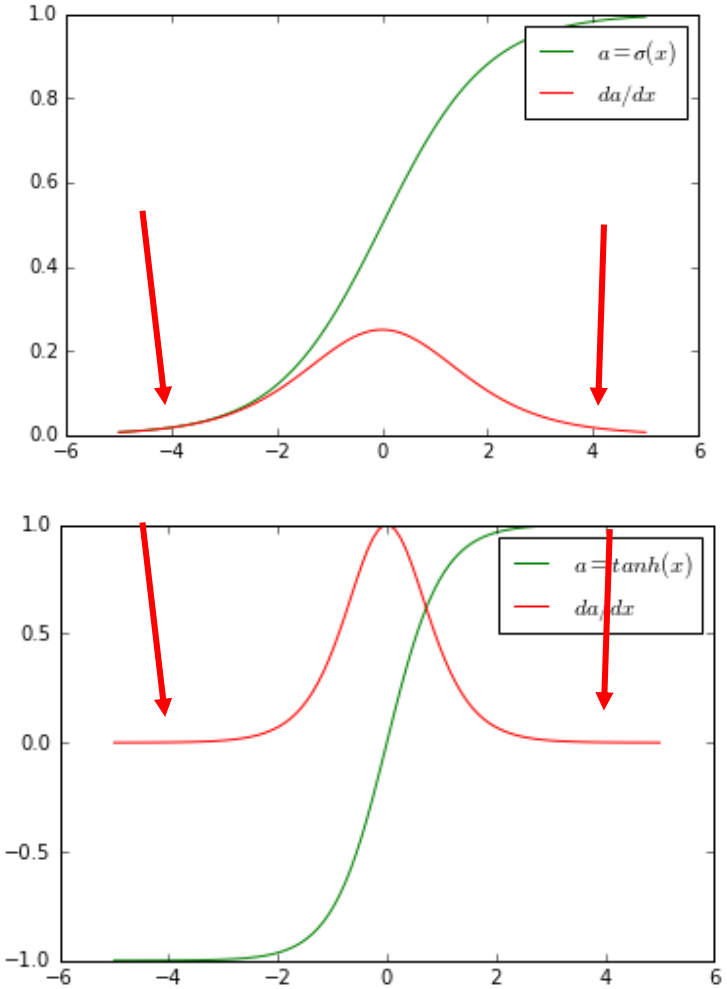
shakespeak

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 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
- 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 when used as the final output layer
- In that case, if they return very small or very large values (saturate) for negative or positive samples, it's ok because their activation is correct
  - "Overconfident", but at least correct

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
- Because $e^{a+b} = e^a e^b$, we usually compute

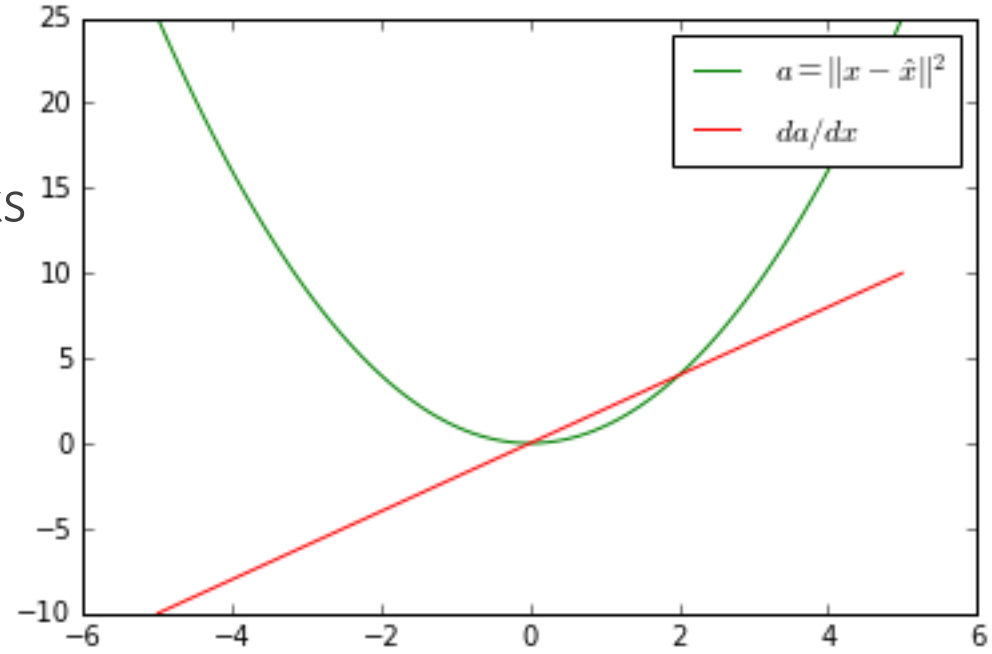
$$a^{(k)} = \frac{e^{x^{(k)}-\mu}}{\sum_j e^{x^{(j)}-\mu}}, \mu = \max_k x^{(k)} \text{ because}$$

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

- Remember: Avoid exponentianting large numbers → better stability

Euclidean loss module

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



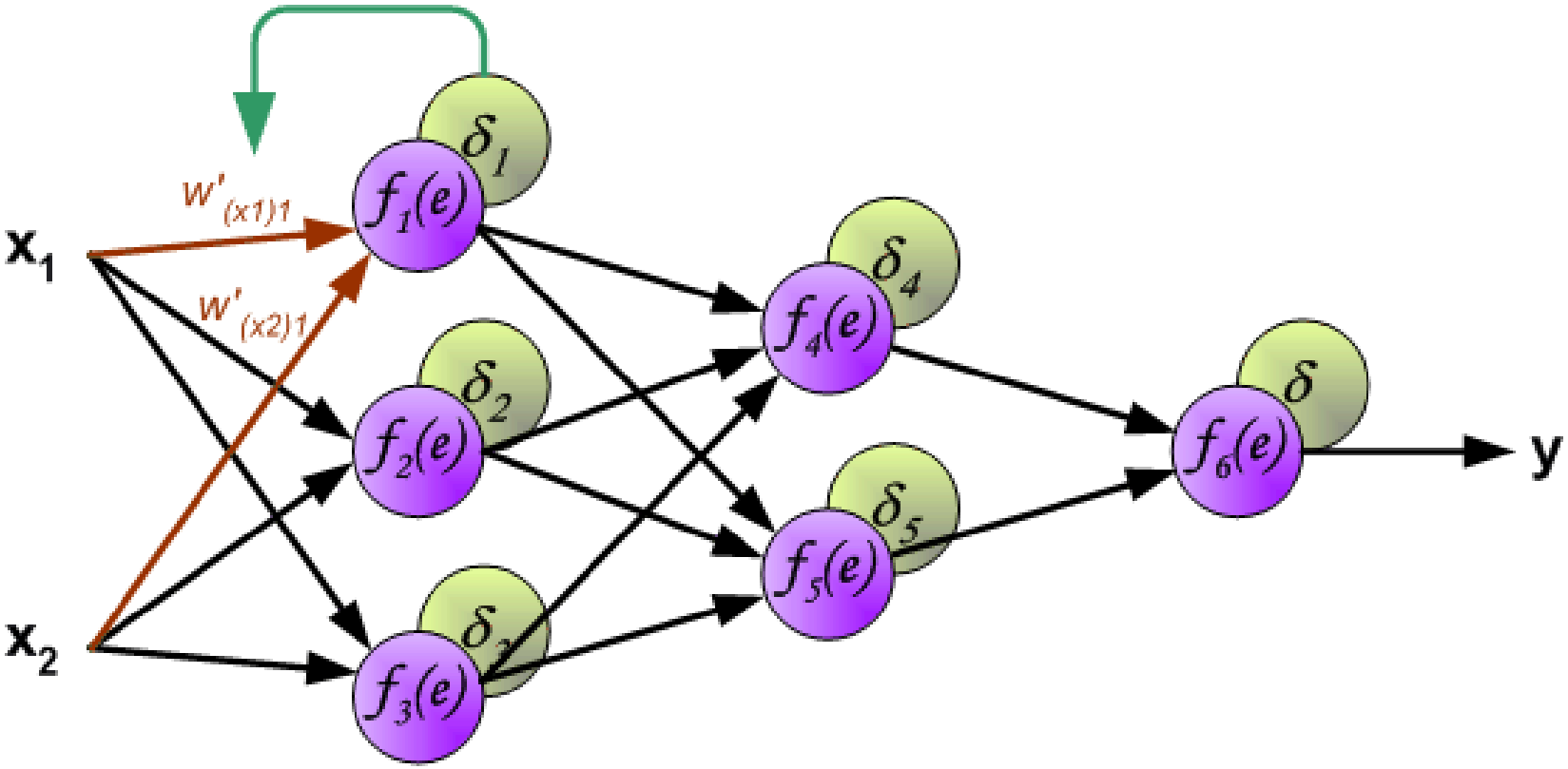
Cross-entropy loss (log-likelihood) module

- Activation: $a(x) = -\sum_{k=1}^{K} y^{(k)} \log x^{(k)}, \quad y^{(k)} = \{0, 1\}$
- Gradient: $\frac{\partial a}{\partial x^{(k)}} = -\frac{1}{x^{(k)}}$
- The cross-entropy loss is the most popular classification loss for classifiers that output probabilities (not SVM)
- 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

Many, many more modules out there …

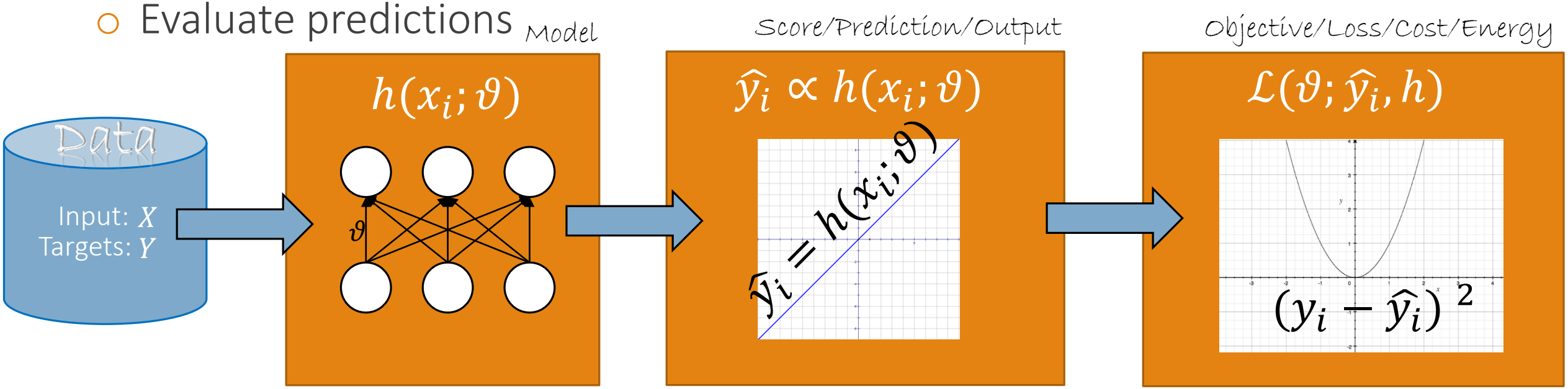
- Many will work comparably to existing ones
  - Not interesting, unless they work consistently better and there is a reason
- Regularization modules
  - Dropout
- Normalization modules
  - ℓ_2-normalization, ℓ_1-normalization
- Loss modules
  - Hinge loss
- Most of concepts discussed in the course can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

Backpropagation



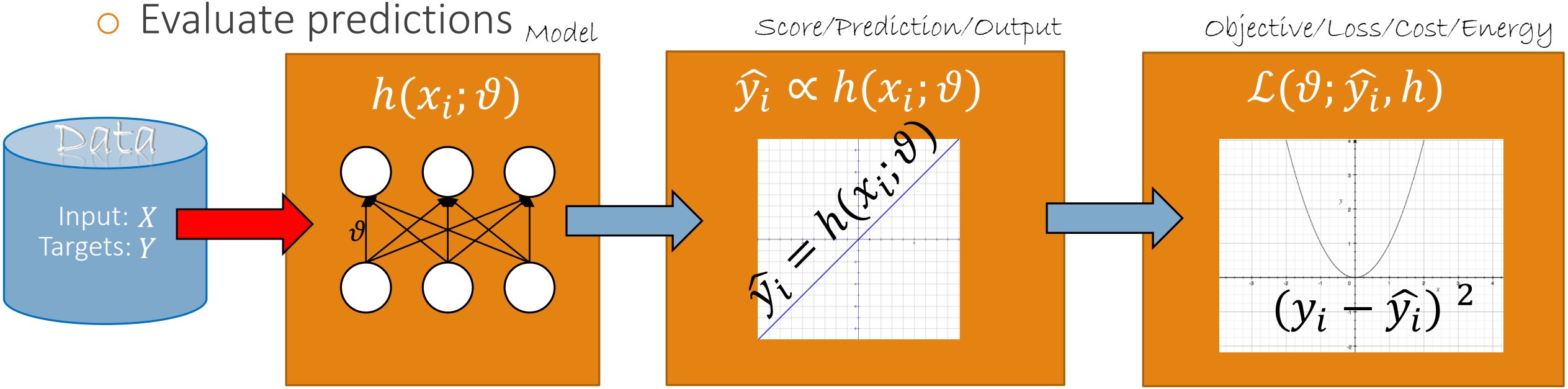
Forward computations

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



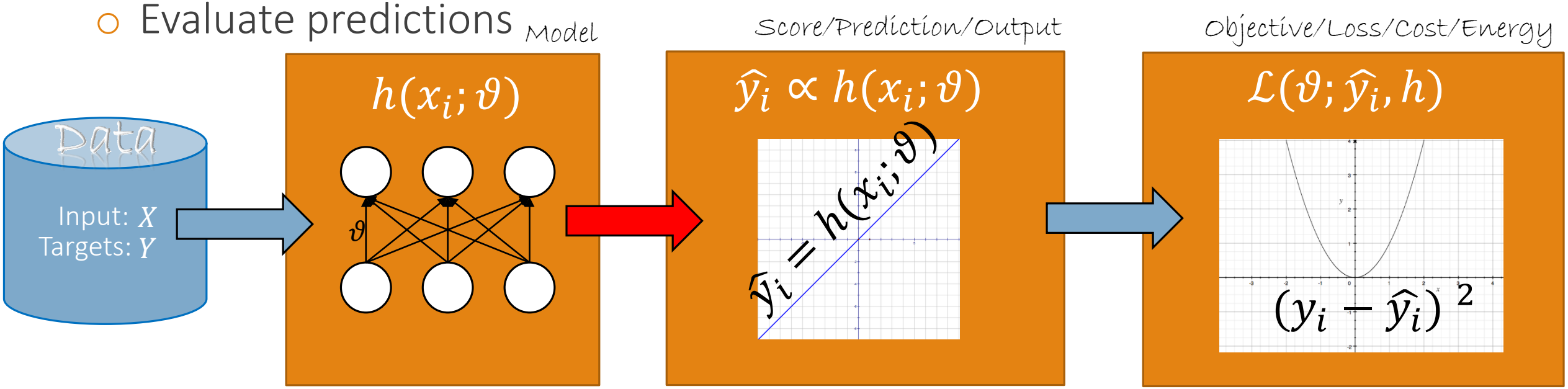
Forward computations

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



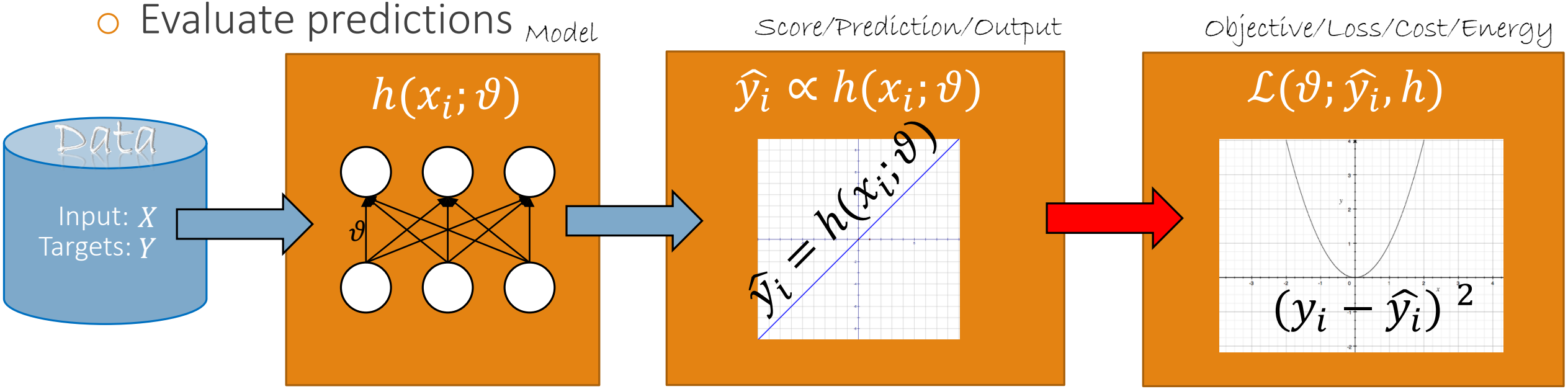
Forward computations

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



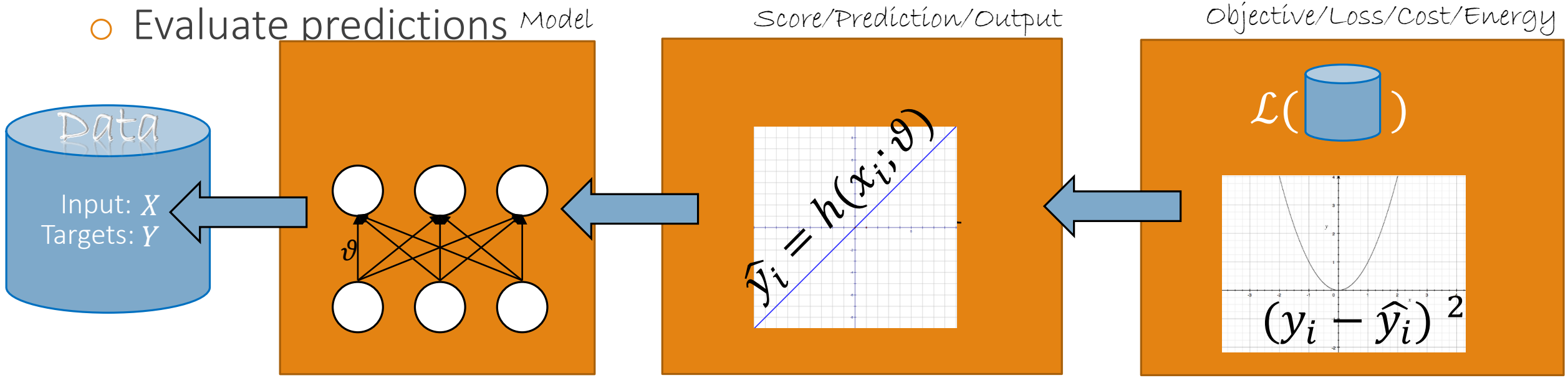
Forward computations

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



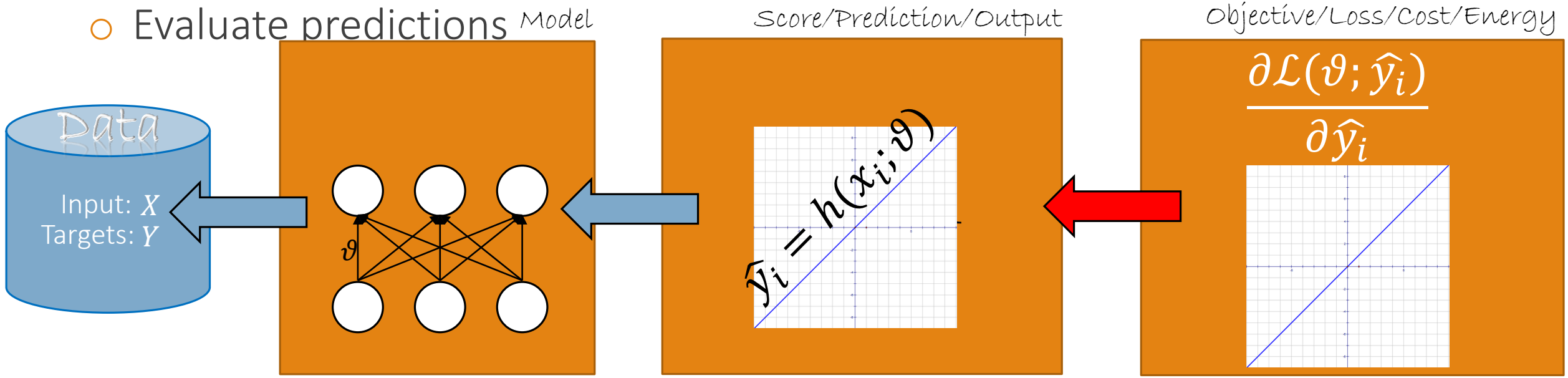
Backward computations

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



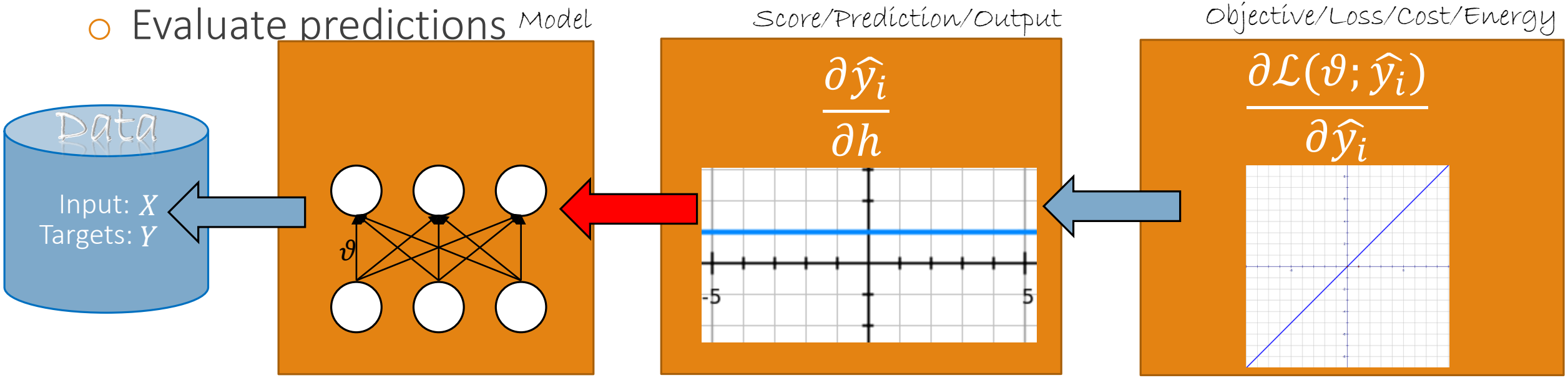
Backward computations

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



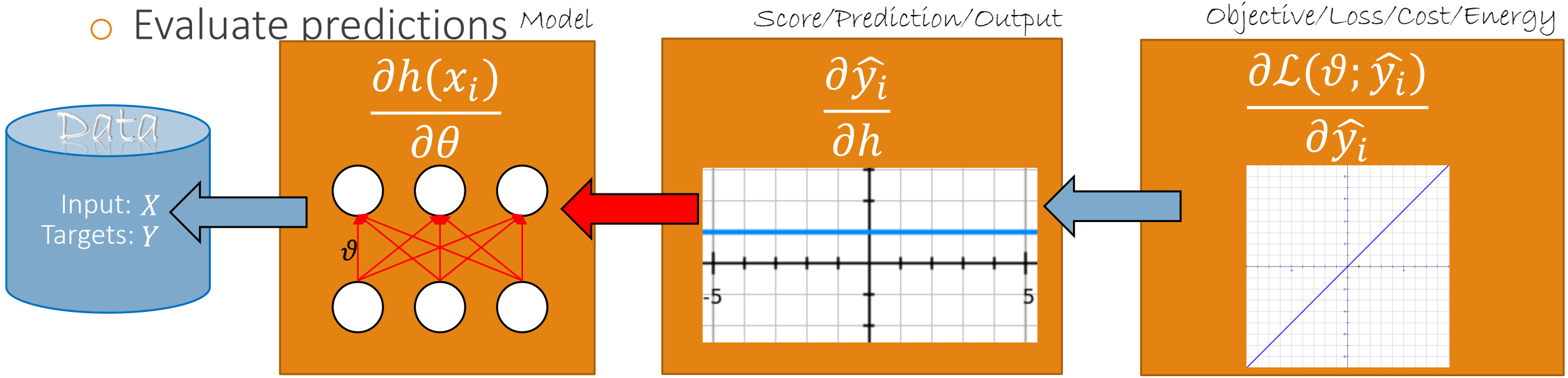
Backward computations

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



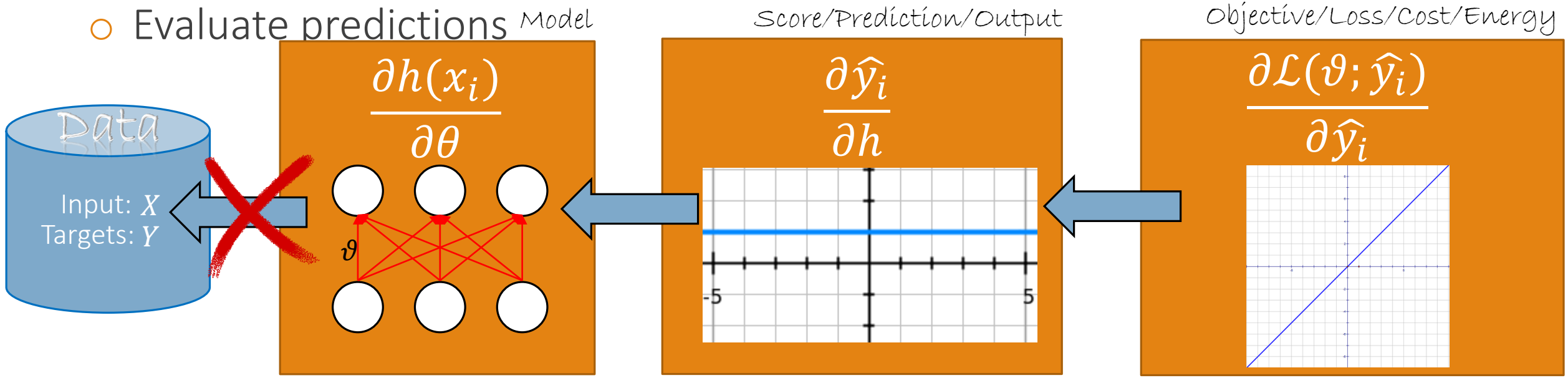
Backward computations

- 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

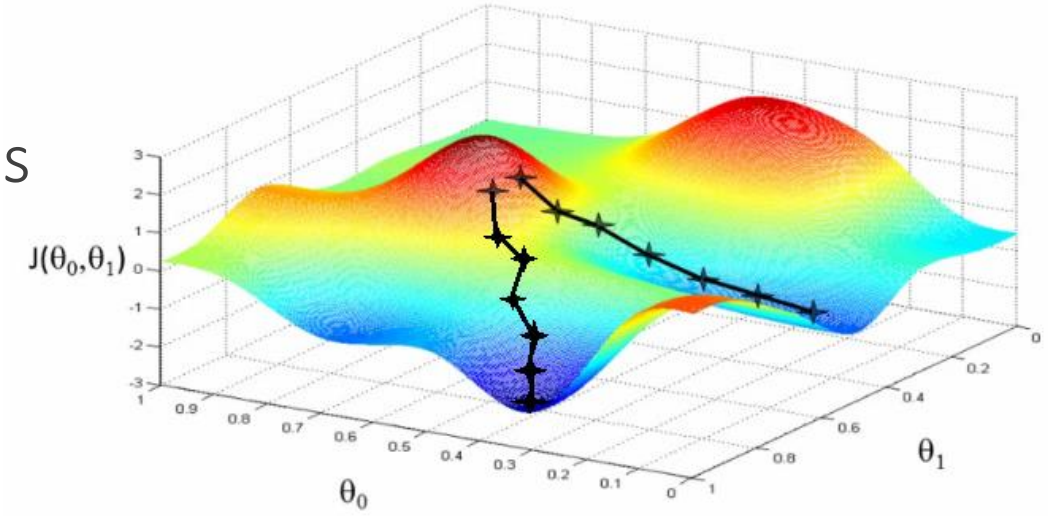


Optimization through Gradient Descent

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

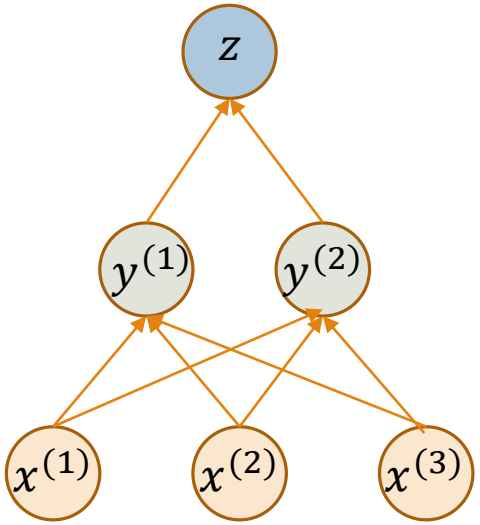
$$\theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla_\theta \mathcal{L}$$

- 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



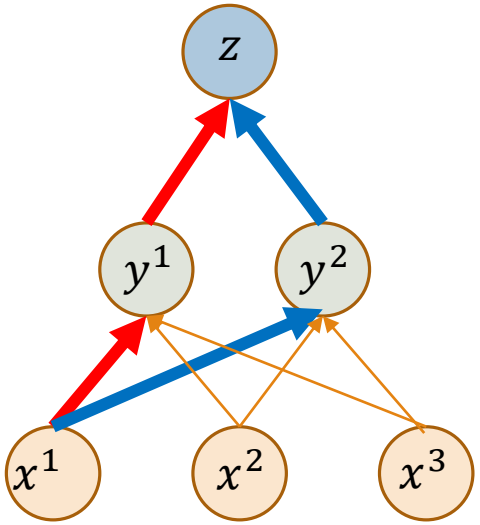
Chain rule

- 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

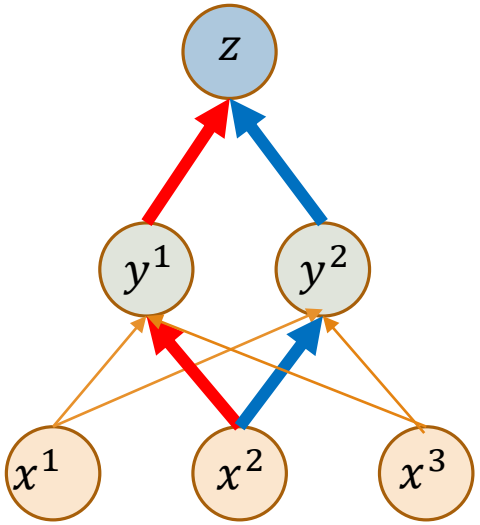
- Assume a nested function, $z = f(y)$ and $y = g(x)$
- Chain Rule for scalars x, y, z
  - $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$
- When $x \in \mathcal{R}^m, y \in \mathcal{R}^n, z \in \mathcal{R}$
  - $\frac{dz}{dx_i} = \sum_j \frac{dz}{dy_j}\frac{dy_j}{dx_i}$ → gradients from all possible paths



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

Chain rule

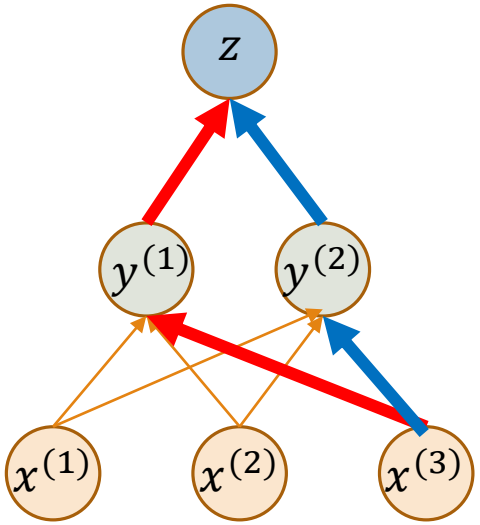
- Assume a nested function, $z = f(y)$ and $y = g(x)$
- Chain Rule for scalars x, y, z
  - $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$
- When $x \in \mathcal{R}^m, y \in \mathcal{R}^n, z \in \mathcal{R}$
  - $\frac{dz}{dx_i} = \sum_j \frac{dz}{dy_j}\frac{dy_j}{dx_i}$ → gradients from all possible paths



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

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



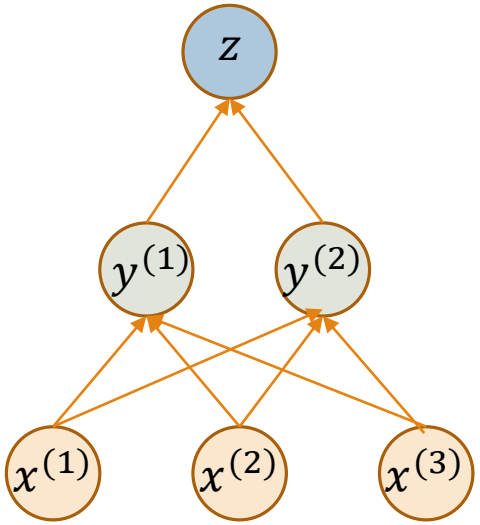
$$\frac{dz}{dx^3} = \frac{dz}{dy^1}\frac{dy^1}{dx^3} + \frac{dz}{dy^2}\frac{dy^2}{dx^3}$$

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

$$J(y(x)) = \frac{d\boldsymbol{y}}{d\boldsymbol{x}} = \begin{bmatrix} \frac{\partial y^{(1)}}{\partial x^{(1)}} & \frac{\partial y^{(1)}}{\partial x^{(2)}} & \frac{\partial y^{(1)}}{\partial x^{(3)}} \\ \frac{\partial y^{(2)}}{\partial x^{(1)}} & \frac{\partial y^{(2)}}{\partial x^{(2)}} & \frac{\partial y^{(2)}}{\partial x^{(3)}} \end{bmatrix}$$

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_2: $a_L(x; \theta_{1,\dots,L}) = h_L\,(h_{L-1}(\dots h_1(x, \theta_1), \dots, \theta_{L-1}), \theta_L)$
- Gradients of parameters of layer l → Chain rule

$$\frac{\partial \mathcal{L}}{\partial \theta_l} = \frac{\partial \mathcal{L}}{\partial a_L} \cdot \frac{\partial a_L}{\partial a_{L-1}} \cdot \frac{\partial a_{L-1}}{\partial a_{L-2}} \cdot \dots \cdot \frac{\partial a_l}{\partial \theta_l}$$

- When shortened, we need to two quantities

$$\frac{\partial \mathcal{L}}{\partial \theta_l} = \left(\frac{\partial a_l}{\partial \theta_l}\right)^T \cdot \frac{\partial \mathcal{L}}{\partial a_l}$$

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

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

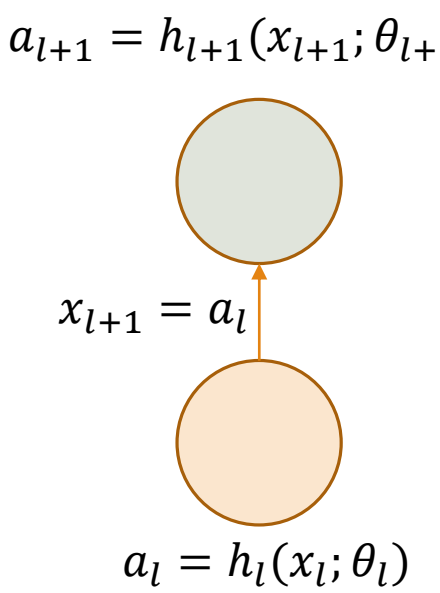
Backpropagation $\Leftrightarrow$ Chain rule!!!

- For $\frac{\partial a_l}{\partial \theta_l}$ in $\frac{\partial \mathcal{L}}{\partial \theta_l} = (\frac{\partial a_l}{\partial \theta_l})^T \cdot \frac{\partial \mathcal{L}}{\partial a_l}$ we only need the Jacobian of the l-th module output a_l w.r.t. to the module's parameters θ_l
- Very local rule, every module looks for its own
  - No need to know what other modules do
- Since computations can be very local
  - graphs can be very complicated
  - modules can be complicated (as long as they are differentiable)

Backpropagation ⟺ Chain rule!!!

- For $\frac{\partial \mathcal{L}}{\partial a_l}$ in $\frac{\partial \mathcal{L}}{\partial \theta_l} = \left(\frac{\partial a_l}{\partial \theta_l}\right)^T \cdot \frac{\partial \mathcal{L}}{\partial a_l}$ 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: $a_l = x_{l+1}$

Gradient w.r.t. the module input

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

Recursive rule (good for us)!!!

What happens to the gradient if the output of multivariate activation functions, which depend on multiple inputs, like softmax: a^j=\exp(x_1)/(\exp(x_1)+\exp(x_2)+\exp(x_3))?

A. We vectorize the inputs and the outputs and compute the gradient as before
B. We compute the Hessian matrix containing all the second-order derivatives: da^2_i/d^2x_j
C. We compute the Jacobian matrix containing all the partial derivatives: da_i/dx_j

The question will open when you start your session and slideshow.

Closed

What happens to the gradient if the output of multivariate activation functions, which depend on multiple inputs, like softmax: a^j=\exp(x_1)/(\exp(x_1)+\exp(x_2)+\exp(x_3))?

A. We vectorize the inputs and the outputs and compute the gradient as before

33.3%

We will set these example results to zero once you've started your session and your slide show.

In the meantime, feel free to change the looks of your results (e.g. the colors).

B. We compute the Hessian matrix containing all the second-order derivatives: da^2_i/d^2x_j

66.7%

C. We compute the Jacobian matrix containing all the partial derivatives: da_i/dx_j

100.0%

Closed

shakespeak

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

- Often module functions depend on multiple input variables
  - 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 for the $\frac{\partial a_l}{\partial x_l}$ (or $\frac{\partial a_l}{\partial \theta_l}$) we must compute Jacobian matrix as a_l depends on multiple input x_l (or θ_l)
  - The Jacobian is the generalization of the 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

- But, quite often in modules the output depends only in a single input
  - 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 $\frac{\mathrm{d}a^i}{dx^j} = 0, \ \forall \ \mathrm{i} \neq \mathrm{j}$

$$\frac{d\boldsymbol{a}}{d\boldsymbol{x}} = \frac{d\boldsymbol{\sigma}}{d\boldsymbol{x}} = \begin{bmatrix} \sigma(x^1)(1-\sigma(x^1)) & 0 & 0 \\ 0 & \sigma(x^2)(1-\sigma(x^2)) & 0 \\ 0 & 0 & \sigma(x^3)(1-\sigma(x^3)) \end{bmatrix} \sim \begin{bmatrix} \sigma(x^1)(1-\sigma(x^1)) \\ \sigma(x^2)(1-\sigma(x^2)) \\ \sigma(x^3)(1-\sigma(x^3)) \end{bmatrix}$$

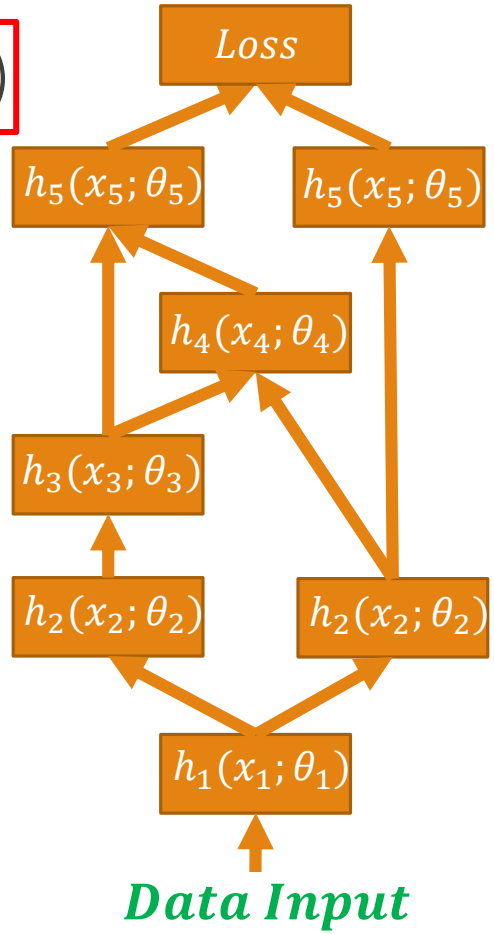
  - Can rewrite equations as inner products to save computations

Forward computations for neural networks

- Simply compute the activation of each module in the network

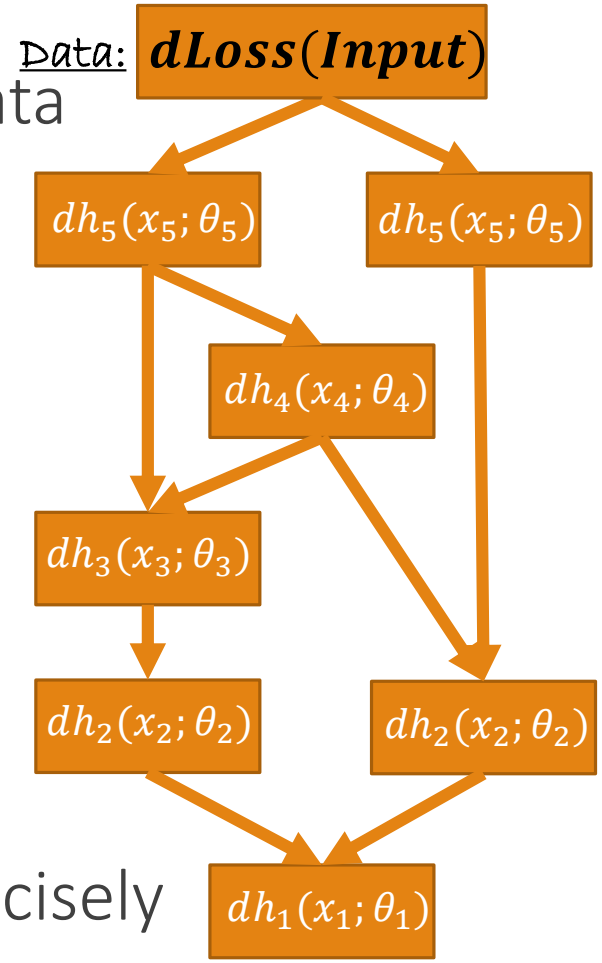
$$a_l = h_l(x_l; \vartheta), \text{ where } a_l = x_{l+1} (\text{or } x_l = a_{l-1})$$

- 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



Backward computations for neural networks

- Simply compute the gradients of each module for our data
  - We need to know the gradient formulation of each module $\partial h_l(x_l; \theta_l)$ w.r.t. their inputs x_l and parameters θ_l
- We need the **forward computations first**
  - Their result is the sum of losses for our input data
- Then take the reverse network (reverse connections) and traverse it backwards
- Instead of using the activation functions, we use their gradients
- The whole process can be described very neatly and concisely with the **backpropagation algorithm**



Dimension analysis

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

$$\dim\left(\frac{\partial \mathcal{L}}{\partial a_l}\right) = \dim(a_l)$$

$$\dim\left(\frac{\partial \mathcal{L}}{\partial \theta_l}\right) = \dim(\theta_l)$$

Dimension analysis

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

$$\dim(a_l) = d_l$$
$$\dim(\theta_l) = d_{l-1} \times d_l$$

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

- For $\frac{\partial \mathcal{L}}{\partial \theta_l} = \frac{\partial a_l}{\partial \theta_l} \cdot \left(\frac{\partial \mathcal{L}}{\partial a_l}\right)^T$

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

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

$$\frac{\partial \mathcal{L}}{\partial a_l} = \left(\frac{\partial a_{l+1}}{\partial x_{l+1}}\right)^T \cdot \frac{\partial \mathcal{L}}{\partial a_{l+1}} \qquad \frac{\partial \mathcal{L}}{\partial \theta_l} = \frac{\partial a_l}{\partial \theta_l} \cdot \left(\frac{\partial \mathcal{L}}{\partial a_l}\right)^T$$

- **Step 3.** Use the gradients $\frac{\partial \mathcal{L}}{\partial \theta_l}$ 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 $[d_{l-1} \times 1]$

$$\frac{\partial \mathcal{L}}{\partial \theta_l} = \frac{\partial a_l}{\partial \theta_l} \cdot \left(\frac{\partial \mathcal{L}}{\partial a_l}\right)^T$$

- **Step 3.** Use the gradients $\frac{\partial \mathcal{L}}{\partial \theta_l}$ 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 $[1 \times d_l]$

Dimensionality analysis: An Example

- $d_{l-1} = 15$ (15 neurons), $d_l = 10$ (10 neurons), $d_{l+1} = 5$ (5 neurons)
- Let's say $a_l = \theta_l^{\mathrm{T}} x_l$ and $a_{l+1} = \theta_{l+1}^{\mathrm{T}} x_{l+1}$
- Forward computations
  - $a_{l-1}: [15 \times 1]$, $a_l: [10 \times 1]$, $a_{l+1}: [5 \times 1]$
  - $x_l: [15 \times 1]$, $x_{l+1}: [10 \times 1]$
  - $\theta_l: [15 \times 10]$
- Gradients
  - $\frac{\partial \mathcal{L}}{\partial a_l}: [5 \times 10]^T \cdot [5 \times 1] = [10 \times 1]$
  - $\frac{\partial \mathcal{L}}{\partial \theta_l}: [15 \times 1] \cdot [10 \times 1]^T = [15 \times 10]$

$$x_l = a_{l-1}$$

Intuitive Backpropagation

Backpropagation in practice

- Things are dead simple, just compute per module

$$\frac{\partial a(x;\theta)}{\partial x} \qquad \frac{\partial a(x;\theta)}{\partial \theta}$$

- Then follow iterative procedure

$$\boxed{\frac{\partial \mathcal{L}}{\partial a_l} = \left(\frac{\partial a_{l+1}}{\partial x_{l+1}}\right)^T \cdot \frac{\partial \mathcal{L}}{\partial a_{l+1}}} \qquad \boxed{\frac{\partial \mathcal{L}}{\partial \theta_l} = \frac{\partial a_l}{\partial \theta_l} \cdot \left(\frac{\partial \mathcal{L}}{\partial a_l}\right)^T}$$

Backpropagation in practice

- Things are dead simple, just compute per module
  - $\frac{\partial a(x;\theta)}{\partial x}$
  - $\frac{\partial a(x;\theta)}{\partial \theta}$

- Then follow iterative procedure [remember: $a_l = x_{l+1}$]

Derivatives from layer above

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

$$\frac{\partial \mathcal{L}}{\partial \theta_l} = \frac{\partial a_l}{\partial \theta_l} \cdot \left(\frac{\partial \mathcal{L}}{\partial a_l}\right)^T$$

Module derivatives

Forward propagation

- For instance, let's consider our module is the function $\cos(\theta x) + \pi$
- The forward computation is simply

```
import numpy as np
def forward(x):
    return np.cos(self.theta*x)+np.pi
```

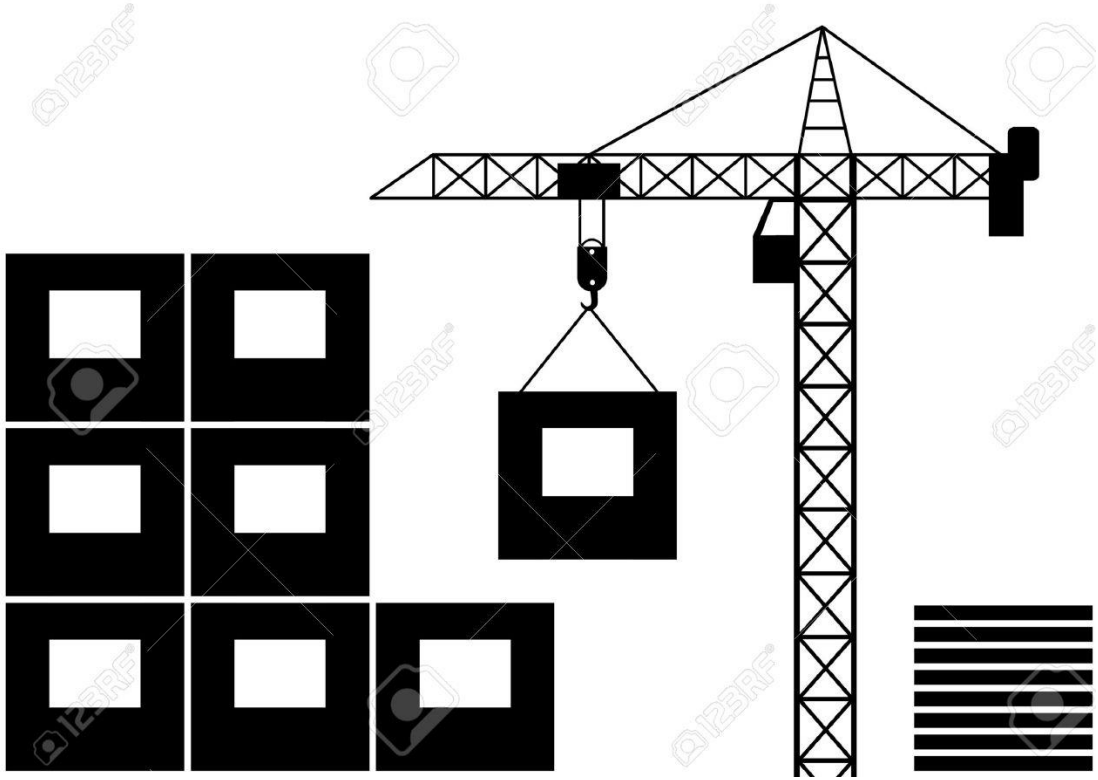
Backward propagation

- The backpropagation for the function $\cos(x) + \pi$

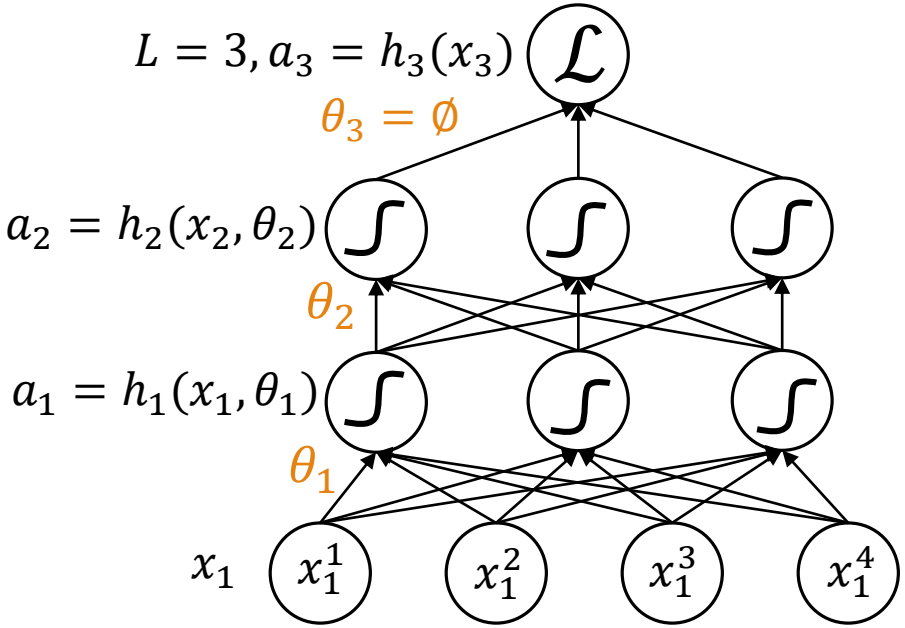
```
import numpy as np
def backward_dx(x):
    return -self.theta*np.sin(self.theta*x)
```

```
import numpy as np
def backward_dtheta(x):
    return -x*np.sin(self.theta*x)
```

Backpropagation: An example



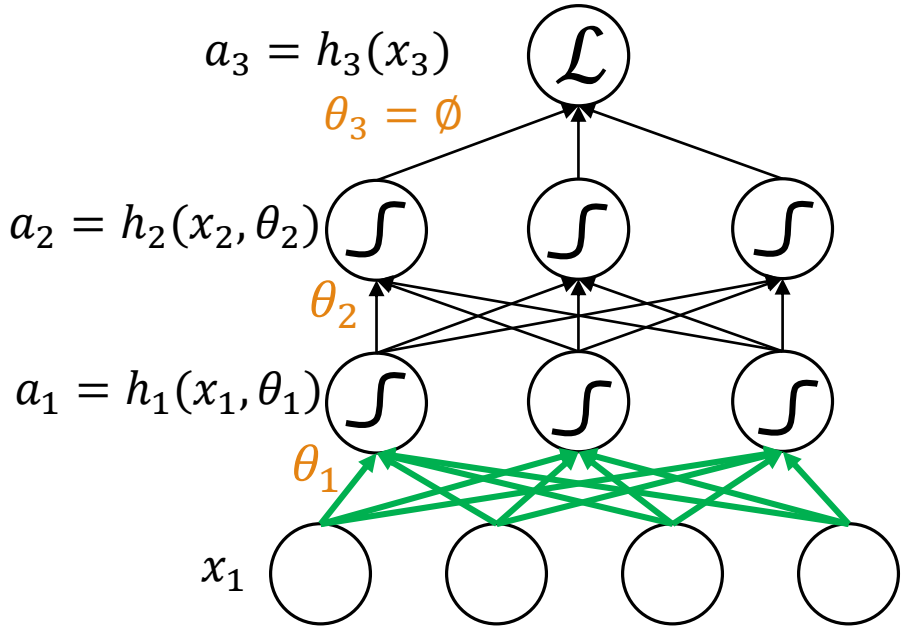
Backpropagation visualization



Backpropagation visualization at epoch (t)

Forward propagations

Compute and store $a_1 = h_1(x_1)$



Example

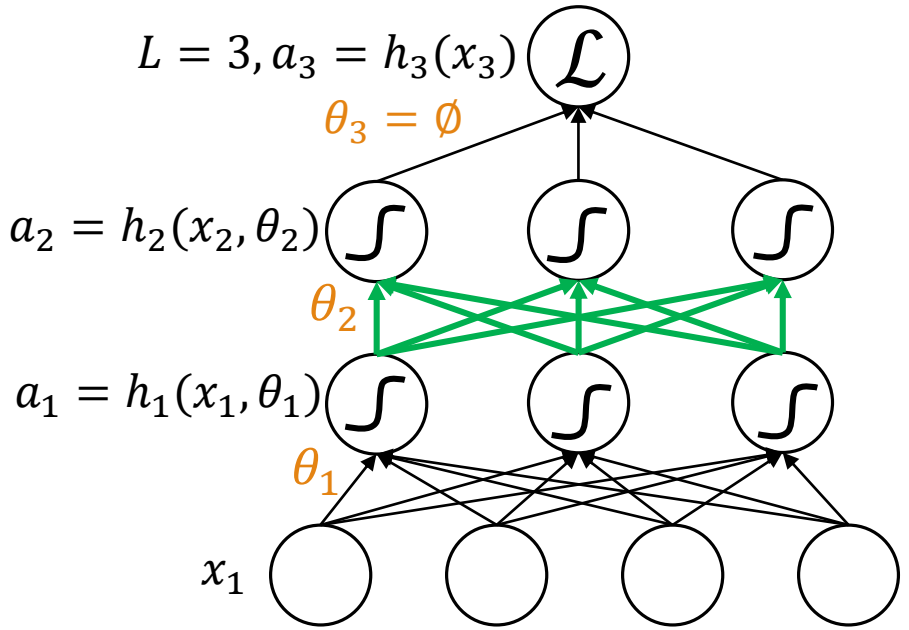
$a_1 = \sigma(\theta_1 x_1)$

Store!!!

Backpropagation visualization at epoch (t)

Forward propagations

Compute and store $a_2 = h_2(x_2)$



Example

$a_1 = \sigma(\theta_1 x_1)$

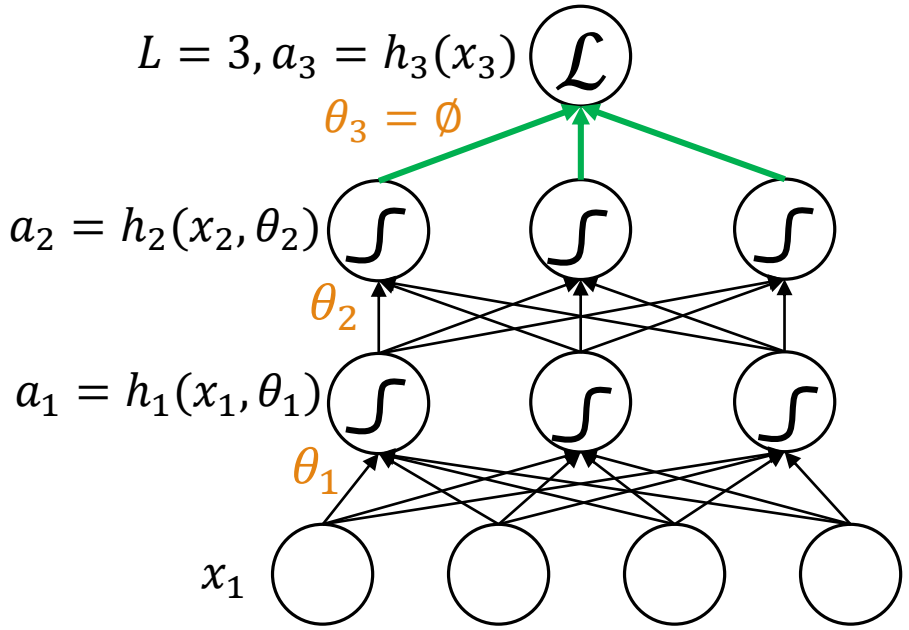
$a_2 = \sigma(\theta_2 x_2)$

Store!!!

Backpropagation visualization at epoch (t)

Forward propagations

Compute and store $a_3 = h_3(x_3)$



Example

$a_1 = \sigma(\theta_1 x_1)$

$a_2 = \sigma(\theta_2 x_2)$

$a_3 = \|y - x_3\|^2$

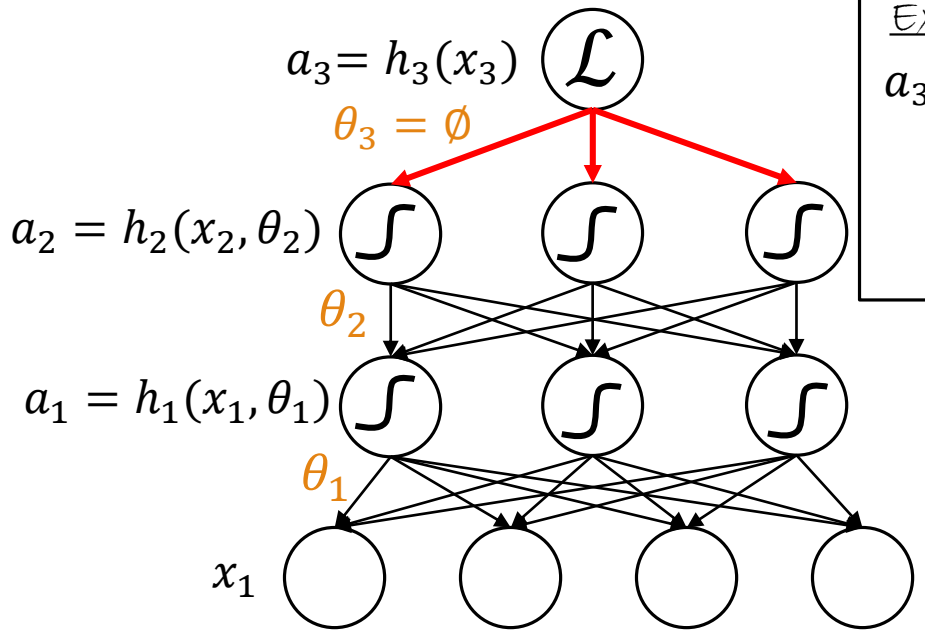
Store!!!

Backpropagation visualization at epoch (t)

Backpropagation

$$\frac{\partial \mathcal{L}}{\partial a_3} = \ldots \leftarrow \text{Direct computation}$$

~~$\frac{\partial \mathcal{L}}{\partial \theta_3}$~~



Example

$$a_3 = \mathcal{L}(y, x_3) = h_3(x_3) = 0.5 \, \|y - x_3\|^2$$

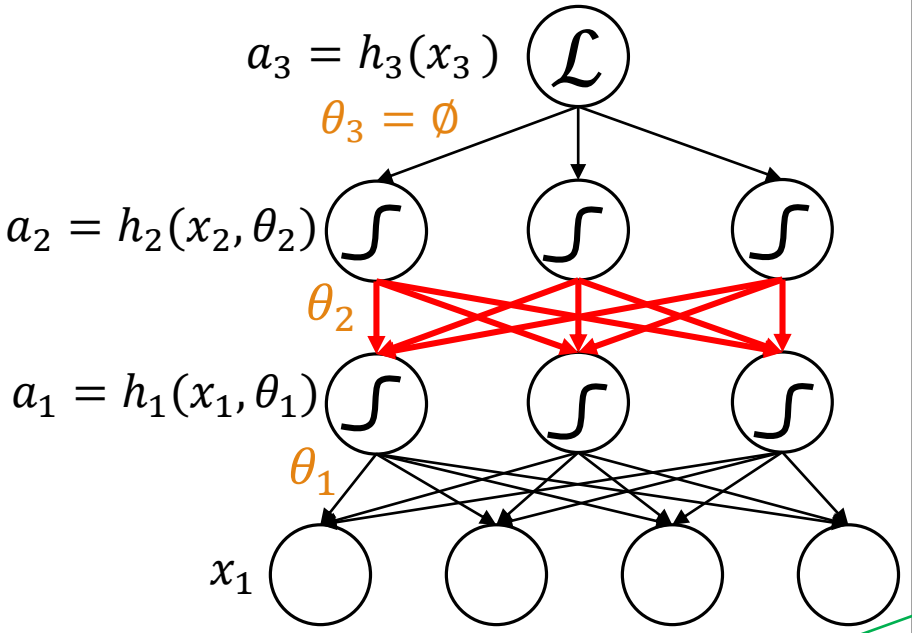
$$\frac{\partial \mathcal{L}}{\partial x_3} = -(y - x_3)$$

Backpropagation visualization at epoch (t)

Backpropagation

$$\frac{\partial \mathcal{L}}{\partial a_2} = \frac{\partial \mathcal{L}}{\partial a_3} \cdot \frac{\partial a_3}{\partial a_2}$$

$$\frac{\partial \mathcal{L}}{\partial \theta_2} = \frac{\partial \mathcal{L}}{\partial a_2} \cdot \frac{\partial a_2}{\partial \theta_2}$$



Stored during forward computations

Example

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

$$x_3 = a_2$$

$$a_2 = \sigma(\theta_2 x_2)$$

$$\frac{\partial \mathcal{L}}{\partial a_2} = \frac{\partial \mathcal{L}}{\partial x_3} = -(y - x_3)$$

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

$$\frac{\partial a_2}{\partial \theta_2} = x_2 \sigma(\theta_2 x_2)(1 - \sigma(\theta_2 x_2))$$

$$= x_2 a_2 (1 - a_2)$$

$$\frac{\partial \mathcal{L}}{\partial a_2} = -(y - x_3)$$

$$\frac{\partial \mathcal{L}}{\partial \theta_2} = \frac{\partial \mathcal{L}}{\partial a_2} x_2 a_2 (1 - a_2)$$

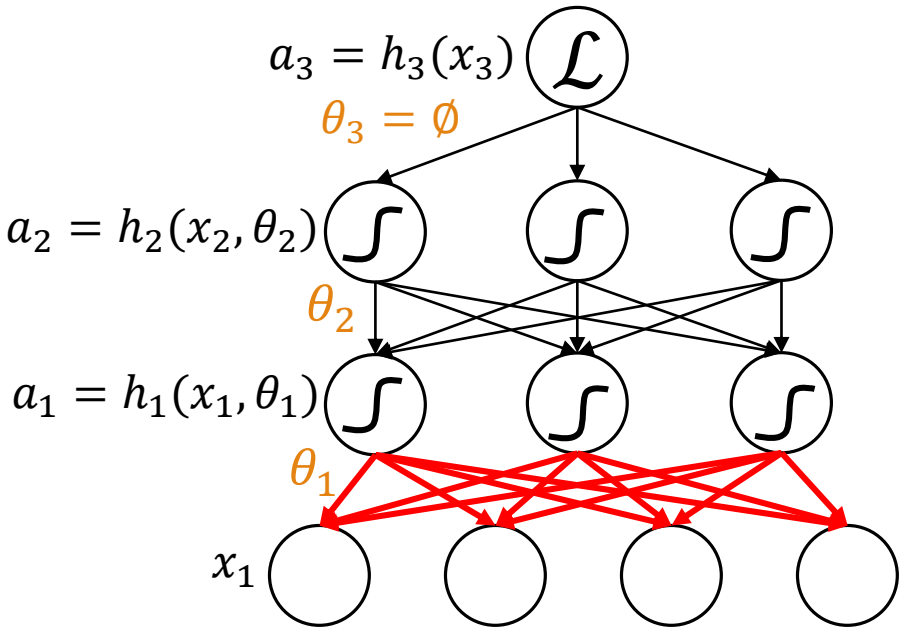
Store!!!

Backpropagation visualization at epoch (t)

Backpropagation

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

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



Example

$$\mathcal{L}(y, a_3) = 0.5 \, \|y - a_3\|^2$$

$$a_2 = \sigma(\theta_2 x_2)$$

$$x_2 = a_1$$

$$a_1 = \sigma(\theta_1 x_1)$$

$$\frac{\partial a_2}{\partial a_1} = \frac{\partial a_2}{\partial x_2} = \theta_2 a_2 (1 - a_2)$$

$$\frac{\partial a_1}{\partial \theta_1} = x_1 a_1 (1 - a_1)$$

$$\frac{\partial \mathcal{L}}{\partial a_1} = \frac{\partial \mathcal{L}}{\partial a_2} \theta_2 a_2 (1 - a_2)$$

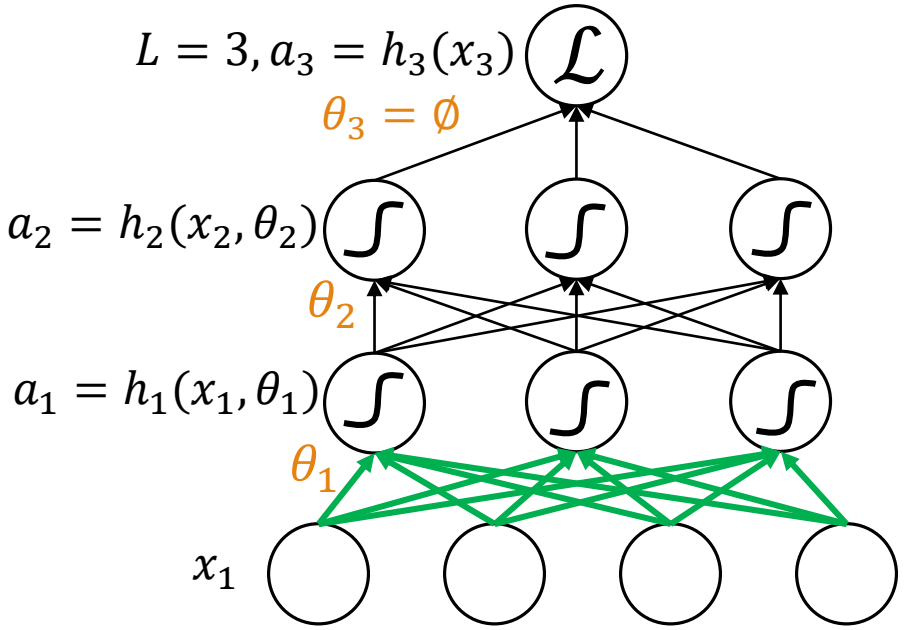
$$\frac{\partial \mathcal{L}}{\partial \theta_1} = \frac{\partial \mathcal{L}}{\partial a_1} x_1 a_1 (1 - a_1)$$

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

Backpropagation visualization at epoch $(t+1)$

Forward propagations

Compute and store $a_1 = h_1(x_1)$



Example

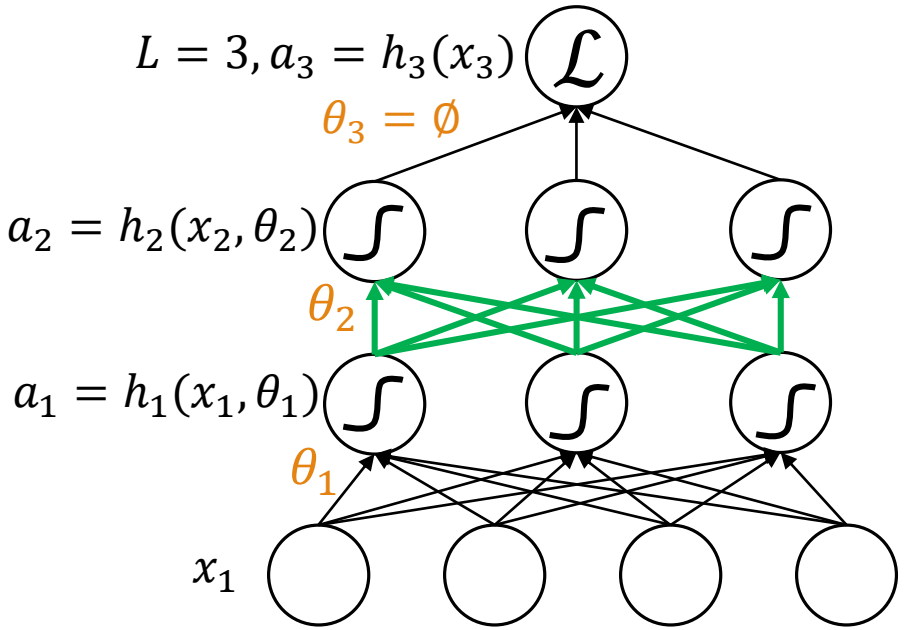
$a_1 = \sigma(\theta_1 x_1)$

Store!!!

Backpropagation visualization at epoch $(t+1)$

Forward propagations

Compute and store $a_2 = h_2(x_2)$



Example

$a_1 = \sigma(\theta_1 x_1)$

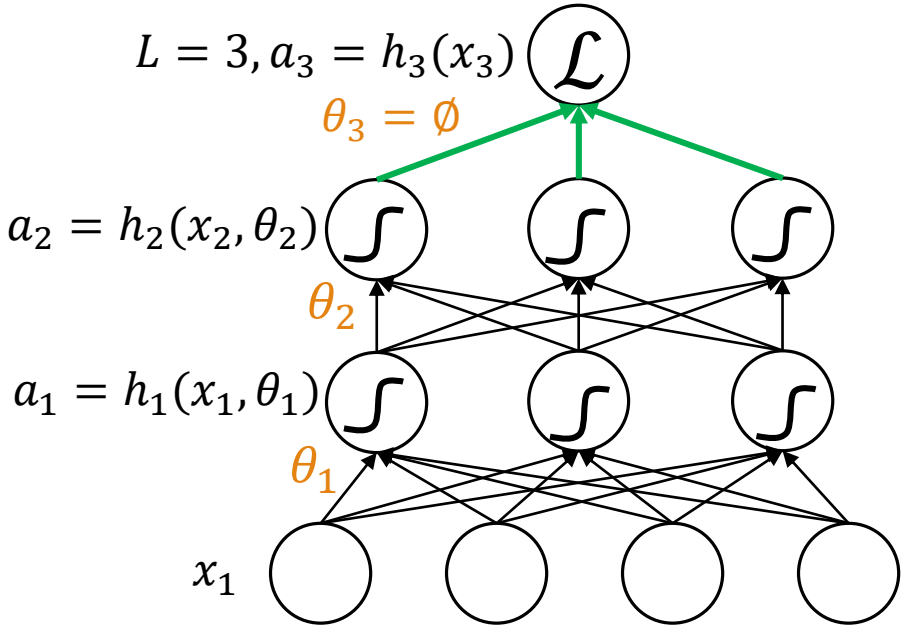
$a_2 = \sigma(\theta_2 x_2)$

Store!!!

Backpropagation visualization at epoch $(t+1)$

Forward propagations

Compute and store $a_3 = h_3(x_3)$



Example

$a_1 = \sigma(\theta_1 x_1)$

$a_2 = \sigma(\theta_2 x_2)$

$a_3 = \|y - x_3\|^2$

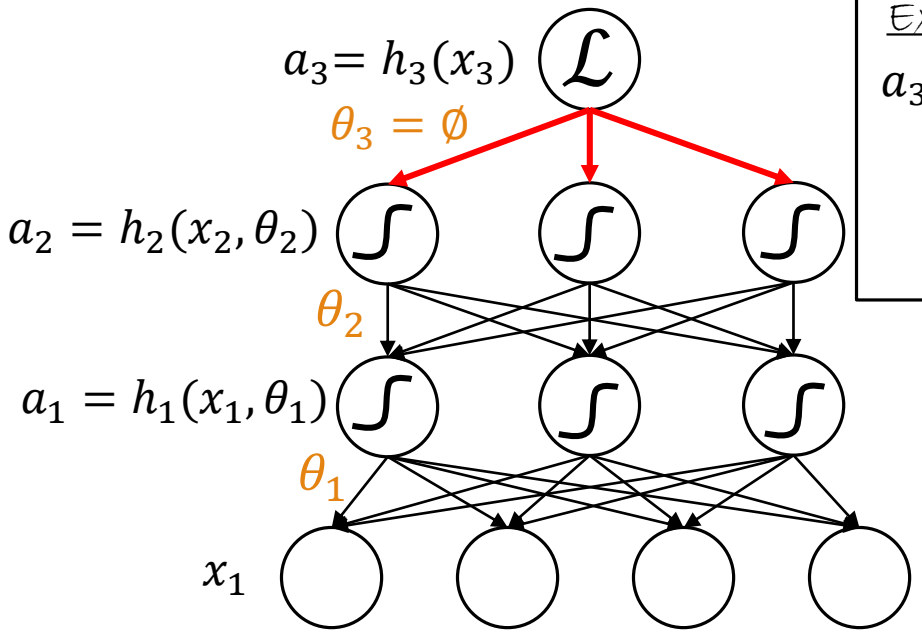
Store!!!

Backpropagation visualization at epoch $(t+1)$

Backpropagation

$$\frac{\partial \mathcal{L}}{\partial a_3} = \ldots \leftarrow \text{Direct computation}$$

~~$\frac{\partial \mathcal{L}}{\partial \theta_3}$~~



Example

$$a_3 = \mathcal{L}(y, x_3) = h_3(x_3) = 0.5\,\|y - x_3\|^2$$

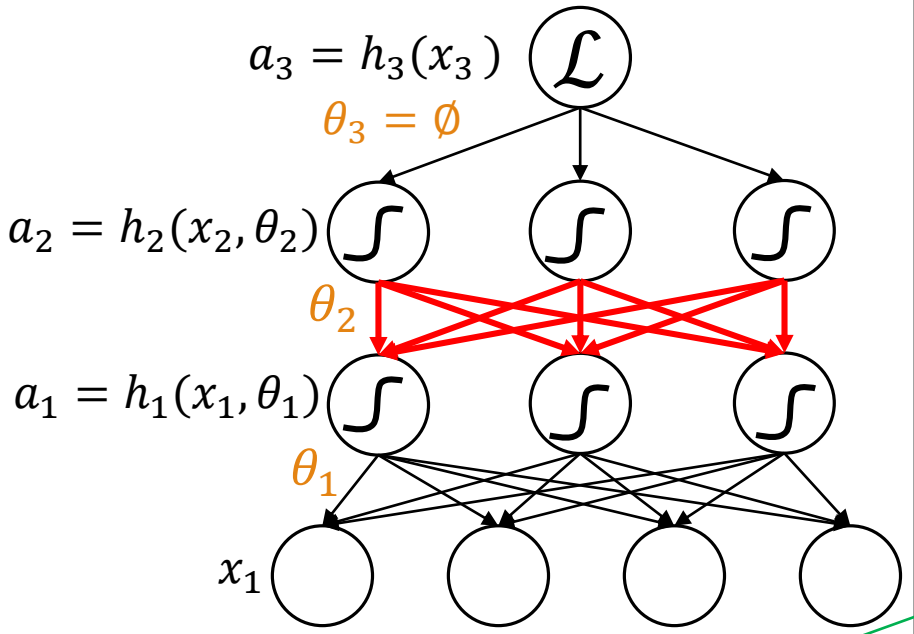
$$\frac{\partial \mathcal{L}}{\partial x_3} = -(y - x_3)$$

Backpropagation visualization at epoch $(t+1)$

Backpropagation

$$\frac{\partial \mathcal{L}}{\partial a_2} = \frac{\partial \mathcal{L}}{\partial a_3} \cdot \frac{\partial a_3}{\partial a_2}$$

$$\frac{\partial \mathcal{L}}{\partial \theta_2} = \frac{\partial \mathcal{L}}{\partial a_2} \cdot \frac{\partial a_2}{\partial \theta_2}$$



Stored during forward computations

Example

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

$$x_3 = a_2$$

$$a_2 = \sigma(\theta_2 x_2)$$

$$\frac{\partial \mathcal{L}}{\partial a_2} = \frac{\partial \mathcal{L}}{\partial x_3} = -(y - x_3)$$

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

$$\frac{\partial a_2}{\partial \theta_2} = x_2 \sigma(\theta_2 x_2)(1 - \sigma(\theta_2 x_2))$$

$$= x_2 a_2 (1 - a_2)$$

$$\frac{\partial \mathcal{L}}{\partial a_2} = -(y - x_3)$$

$$\frac{\partial \mathcal{L}}{\partial \theta_2} = \frac{\partial \mathcal{L}}{\partial a_2} x_2 a_2 (1 - a_2)$$

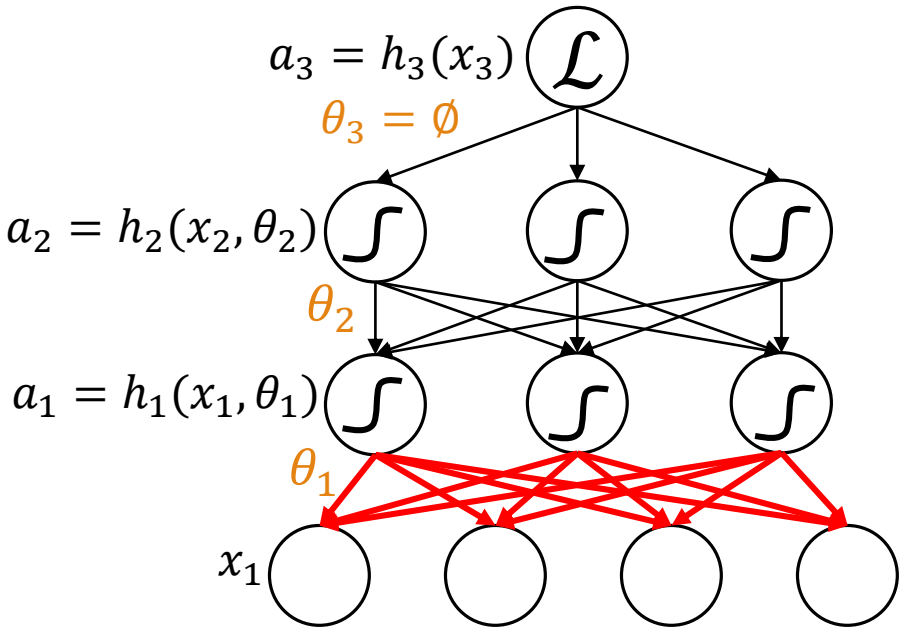
Store!!!

Backpropagation visualization at epoch $(t+1)$

Backpropagation

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

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



Example

$$\mathcal{L}(y, a_3) = 0.5 \, \|y - a_3\|^2$$

$$a_2 = \sigma(\theta_2 x_2)$$

$$x_2 = a_1$$

$$a_1 = \sigma(\theta_1 x_1)$$

$$\frac{\partial a_2}{\partial a_1} = \frac{\partial a_2}{\partial x_2} = \theta_2 a_2 (1 - a_2)$$

$$\frac{\partial a_1}{\partial \theta_1} = x_1 a_1 (1 - a_1)$$

$$\frac{\partial \mathcal{L}}{\partial a_1} = \frac{\partial \mathcal{L}}{\partial a_2} \theta_2 a_2 (1 - a_2)$$

$$\frac{\partial \mathcal{L}}{\partial \theta_1} = \frac{\partial \mathcal{L}}{\partial a_1} x_1 a_1 (1 - a_1)$$

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

Some practical tricks of the trade

- For classification use cross-entropy loss
- Use Stochastic Gradient Descent on mini-batches
- Shuffle training examples **at each** new epoch
- Normalize input variables
  - $(\mu, \sigma^2) = (0,1)$
  - $\mu = 0$

Composite modules

or …

“Make your own module”

Backpropagation again

- **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

$$\frac{\partial \mathcal{L}}{\partial a_l} = \left(\frac{\partial a_{l+1}}{\partial x_{l+1}}\right)^T \cdot \frac{\partial \mathcal{L}}{\partial a_{l+1}} \qquad \frac{\partial \mathcal{L}}{\partial \theta_l} = \frac{\partial a_l}{\partial \theta_l} \cdot \left(\frac{\partial \mathcal{L}}{\partial a_l}\right)^T$$

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

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

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

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 θx

$$a = \sigma(\theta x)$$

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

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

1 sigmoid == 2 modules?

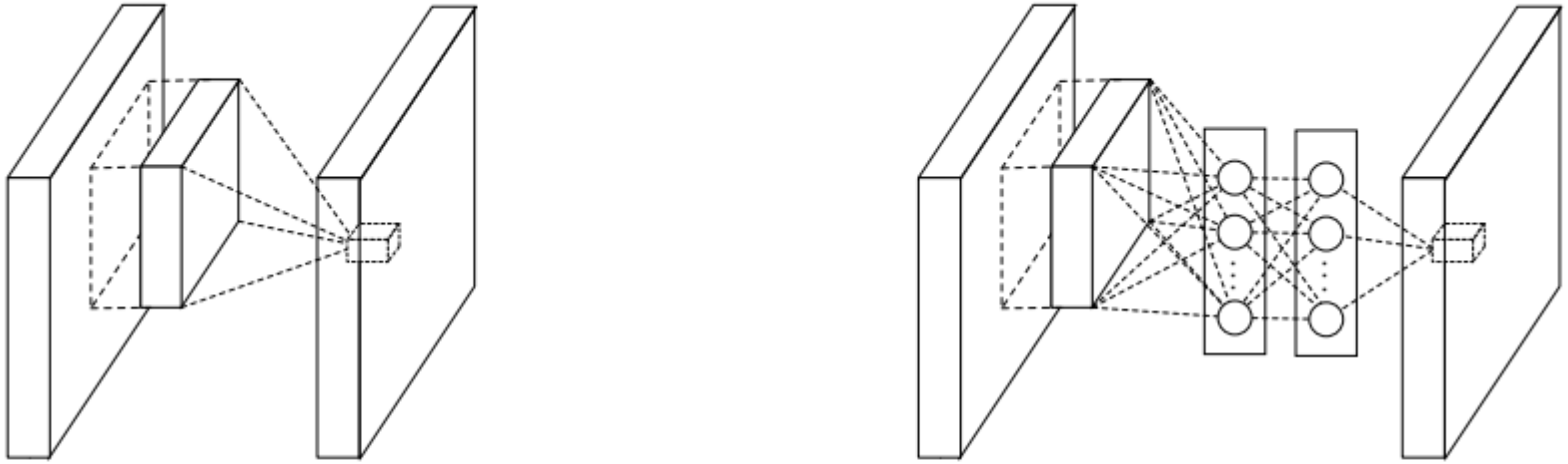
- Two backpropagation steps instead of one

+ But now our gradients are simpler

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

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

Network-in-network [Lin et al., arXiv 2013]



(a) Linear convolution layer

(b) Mlpconv layer

Figure 1: Comparison of linear convolution layer and mlpconv layer. The linear convolution layer includes a linear filter while the mlpconv layer includes a micro network (we choose the multilayer perceptron in this paper). Both layers map the local receptive field to a confidence value of the latent concept.

ResNet [He et al., CVPR 2016]

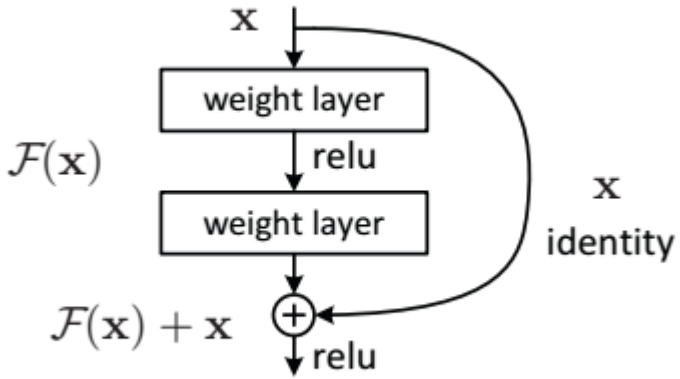
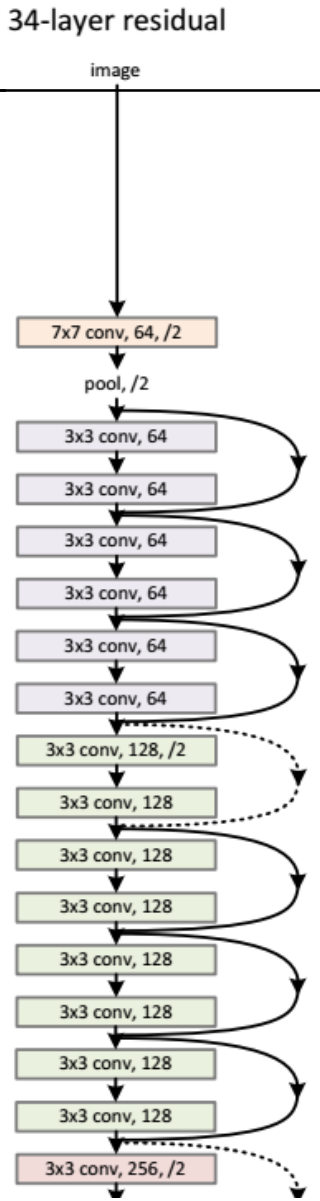


Figure 2. Residual learning: a building block.



Radial Basis Function (RBF) Network module

- RBF module

$$a = \sum_j u_j \exp(-\beta_j (x - w_j)^2)$$

- Decompose into cascade of modules

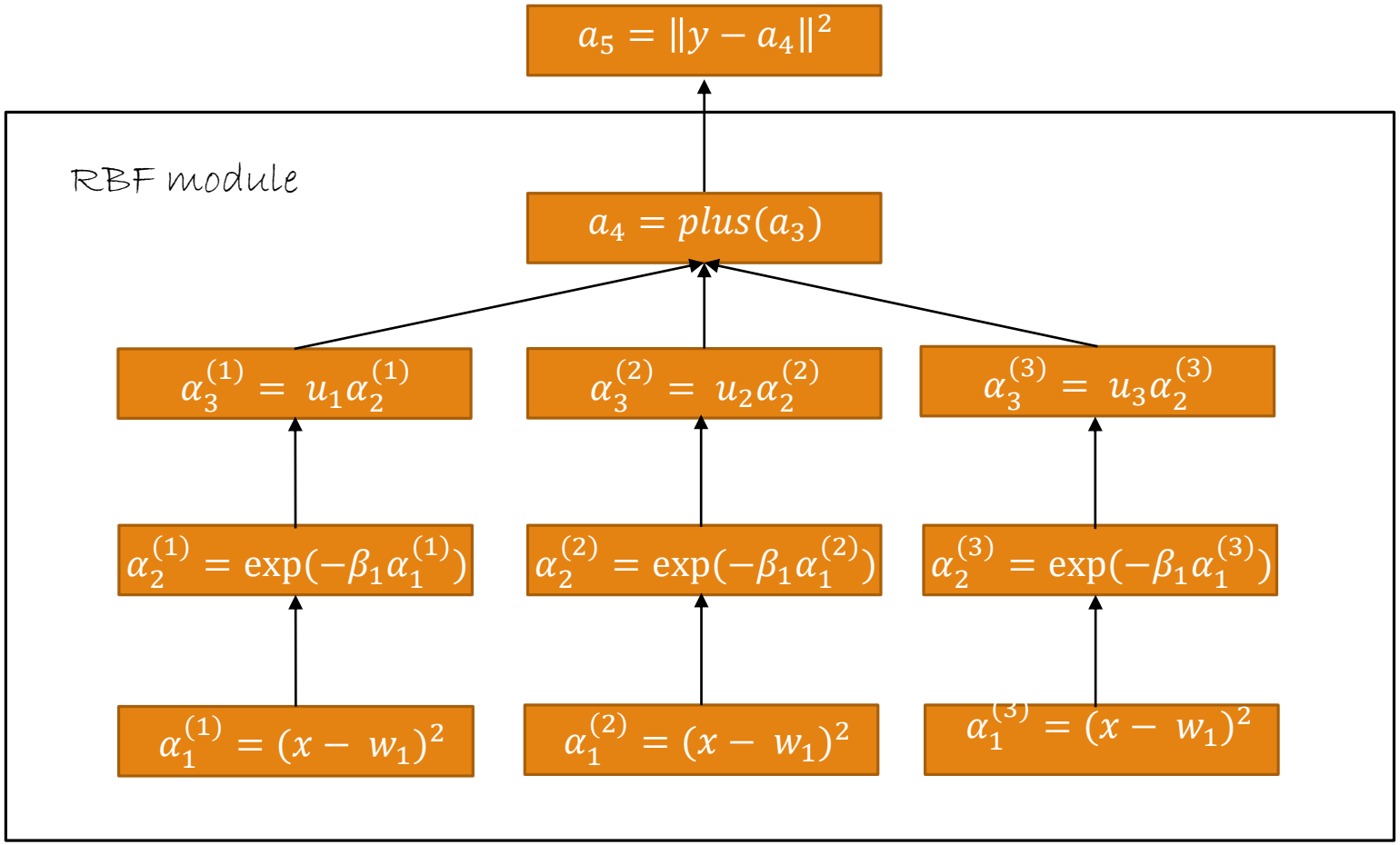
$$a_1 = (x - w)^2$$
$$a_2 = \exp(-\beta a_1)$$
$$a_3 = u a_2$$
$$a_4 = plus(\dots, a_3^{(j)}, \dots)$$

Radial Basis Function (RBF) Network module

- An RBF module is good for regression problems, in which cases it is followed by a Euclidean loss module
- The Gaussian centers w_j can be initialized externally, e.g. with k-means

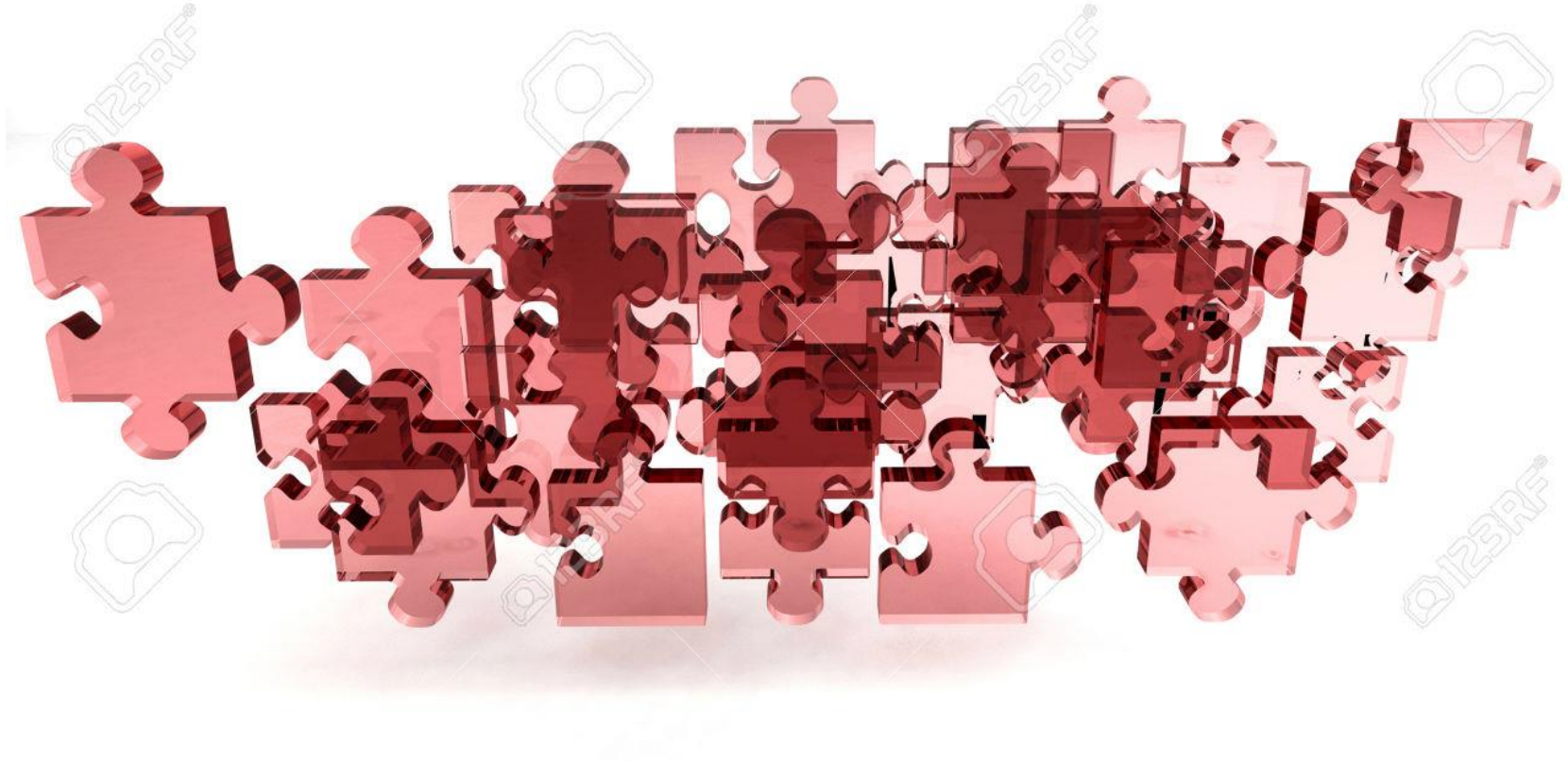
$$a_1 = (x - w)^2$$
$$a_2 = \exp(-\beta a_1)$$
$$a_3 = u a_2$$
$$a_4 = plus(\dots, a_3^{(j)}, \dots)$$

An RBF visually



$$a_1 = (x - w)^2 \rightarrow a_2 = \exp(-\beta a_1) \rightarrow a_3 = ua_2 \rightarrow a_4 = plus(\dots, a_3^{(j)}, \dots)$$

Unit tests



Unit test

- Always check your implementations
  - Not only for Deep Learning
- Does my implementation of the sin function return the correct values?
  - If I execute $\mathbf{sin}(\pi/2)$ does it return 1 as it should
- Even more important for gradient functions
  - not only our implementation can be wrong, but also our math
- Slightest sign of malfunction → ALWAYS RECHECK
  - Ignoring problems never solved problems

Gradient check

Original gradient definition: $\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h)}{\Delta h}$

- Most dangerous part for new modules → get gradients wrong
- Compute gradient analytically
- Compute gradient computationally $g(\theta^{(i)}) \approx \frac{a(\theta + \varepsilon) - a(\theta - \varepsilon)}{2\varepsilon}$
- Compare

$$\Delta(\theta^{(i)}) = \left\| \frac{\partial a(x; \theta^{(i)})}{\partial \theta^{(i)}} - g(\theta^{(i)}) \right\|^2$$

- Is difference in $(10^{-4}, 10^{-7})$ → thengradients are good

Gradient check

- Perturb one parameter $\theta^{(i)}$ at a time with $\theta^{(i)} + \varepsilon$
- Then check $\Delta\left(\theta^{(i)}\right)$ for that one parameter only
- **Do not** perturb the whole parameter vector $\theta + \varepsilon$
  - This will give **wrong results** (simple geometry)
- Sample dimensions of the gradient vector
  - If you get a few dimensions of an gradient vector good, all is good
  - Sample function and bias gradients equally, otherwise you might get your bias wrong

Numerical gradients

- Can we replace analytical gradients with numerical gradients?
- In theory, yes!
- In practice, no!
  - Too slow

Be creative!

- What about trigonometric modules?
- Or polynomial modules?
- Or new loss modules?

Summary

- Machine Learning Paradigm for Neural Networks
- Neural Networks as modular architectures
- Neural Network Modules and Theory
- The Backpropagation algorithm for learning with a neural network
- How to implement and check your very own module

Reading material & references

- Chapter 6