UNIVERSITÄT
BERN

Deep Feedforward Networks

Paolo Favaro

Contents

- Introduction to Feedforward Neural Networks: definition, design, training

- Based on **Chapter 6** (and 4) of Deep Learning by Goodfellow, Bengio, Courville

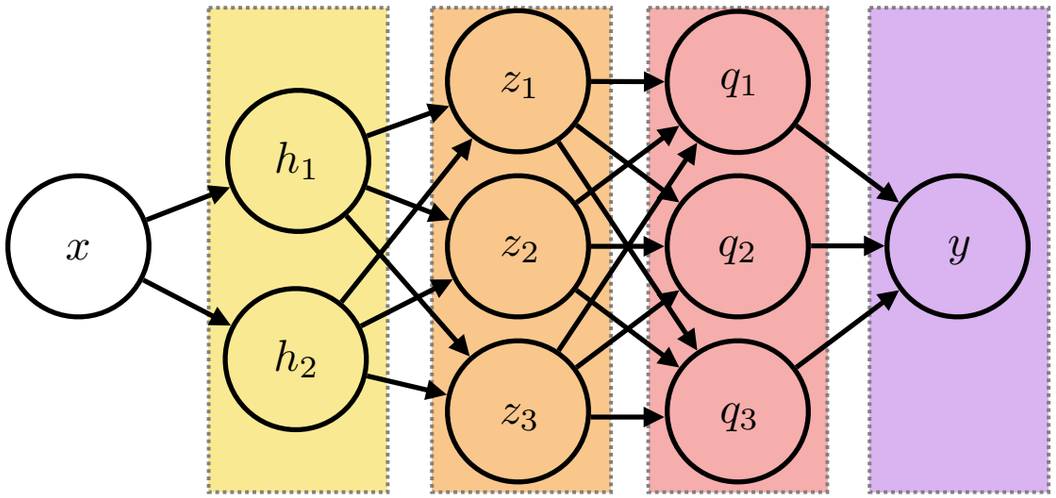
- References to Machine Learning and Pattern Recognition by Bishop

Resources

- Books and online material for further studies
  - CS231 @ Stanford (Fei-Fei Li)
  - **Pattern Recognition and Machine Learning**
    by Christopher M. Bishop
  - **Machine Learning: a Probabilistic Perspective**
    by Kevin P. Murphy

Feedforward Neural Networks

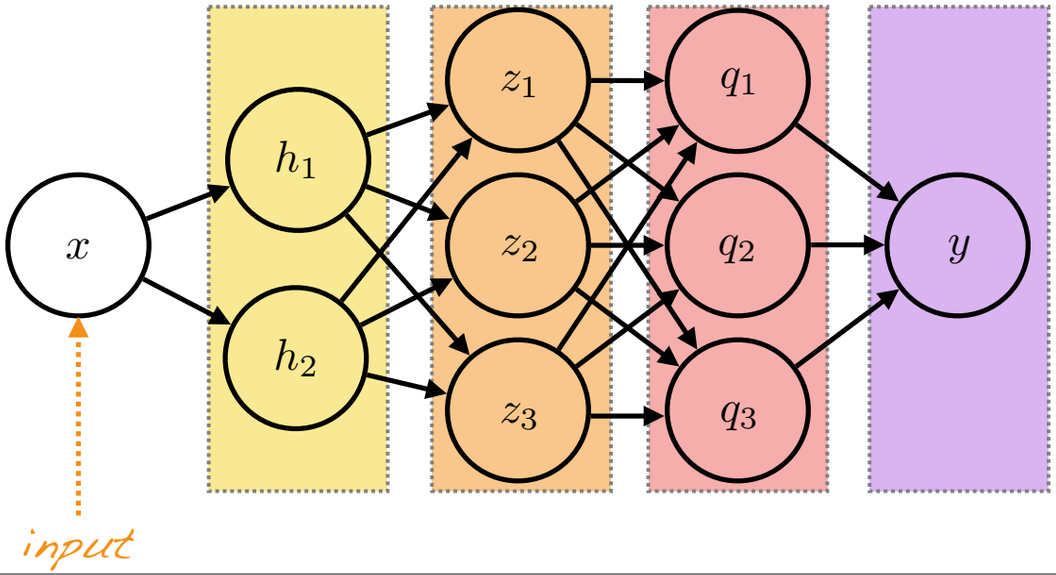
- Feedforward networks are a sequence of layers, each processing the output of the previous layer(s)



What is a feedforward neural network?
It is a chain of functions (typically "simple" functions) that have a limited scope (i.e., they depend only on a subset of the variables. The dependency is also hierarchical: each layer takes as inputs the outputs of the previous layer (there is no feedback).
These relations can also be specified by a directed acyclic graph (DAG).

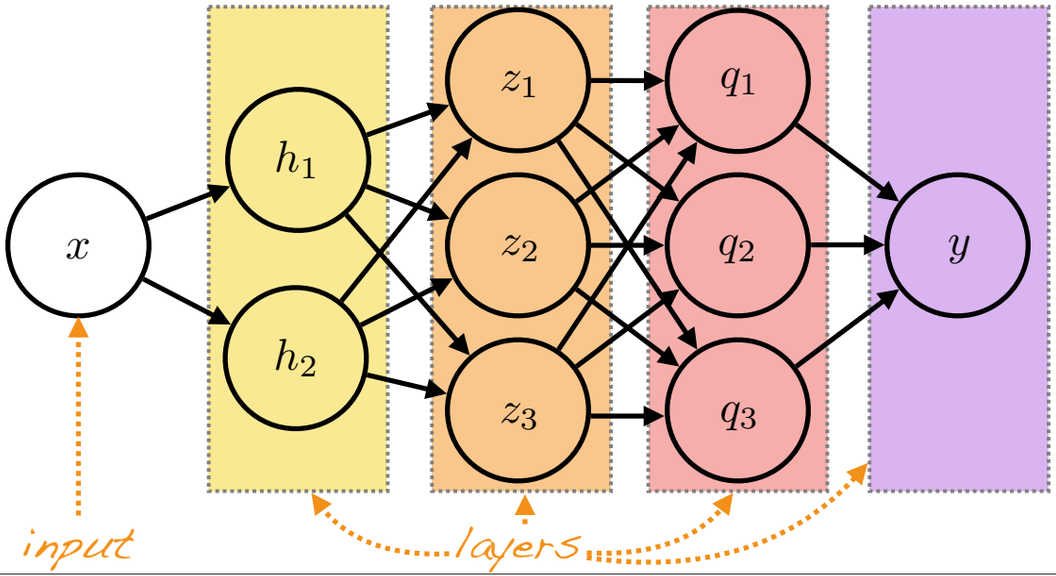
Feedforward Neural Networks

- Feedforward networks are a sequence of layers, each processing the output of the previous layer(s)



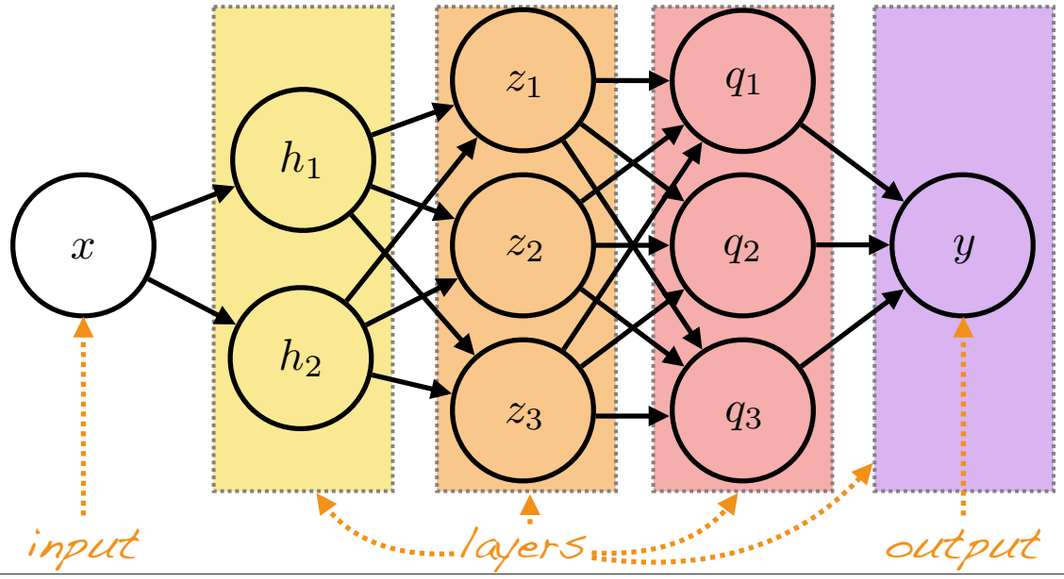
Feedforward Neural Networks

- Feedforward networks are a sequence of layers, each processing the output of the previous layer(s)

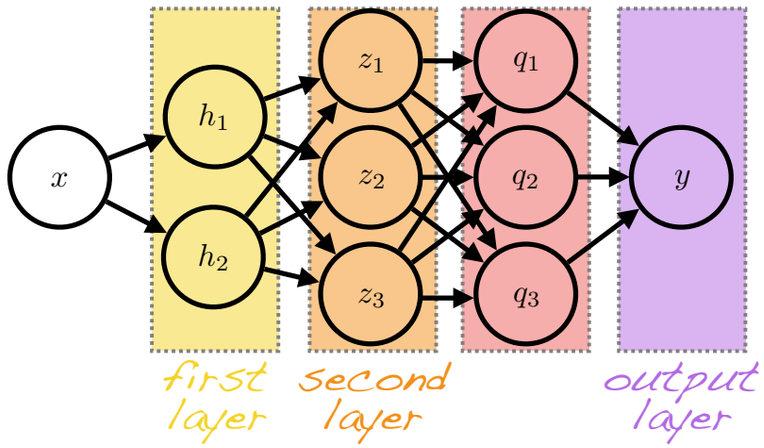


Feedforward Neural Networks

- Feedforward networks are a sequence of layers, each processing the output of the previous layer(s)

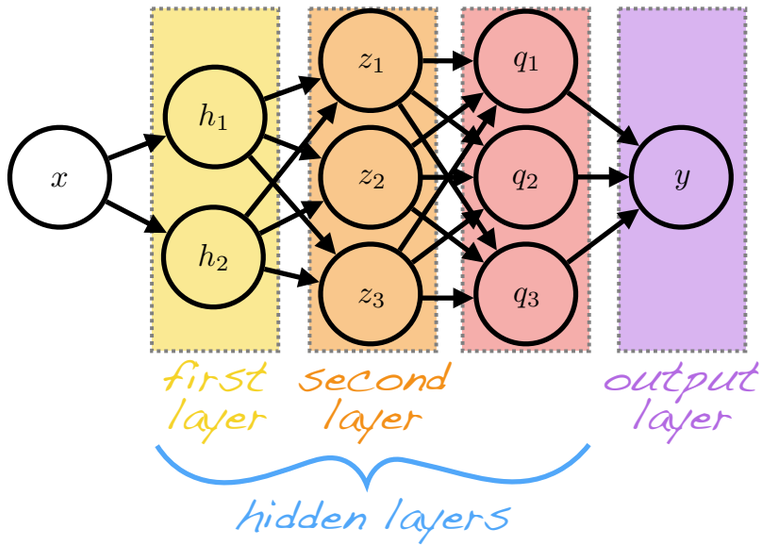


Feedforward Neural Networks

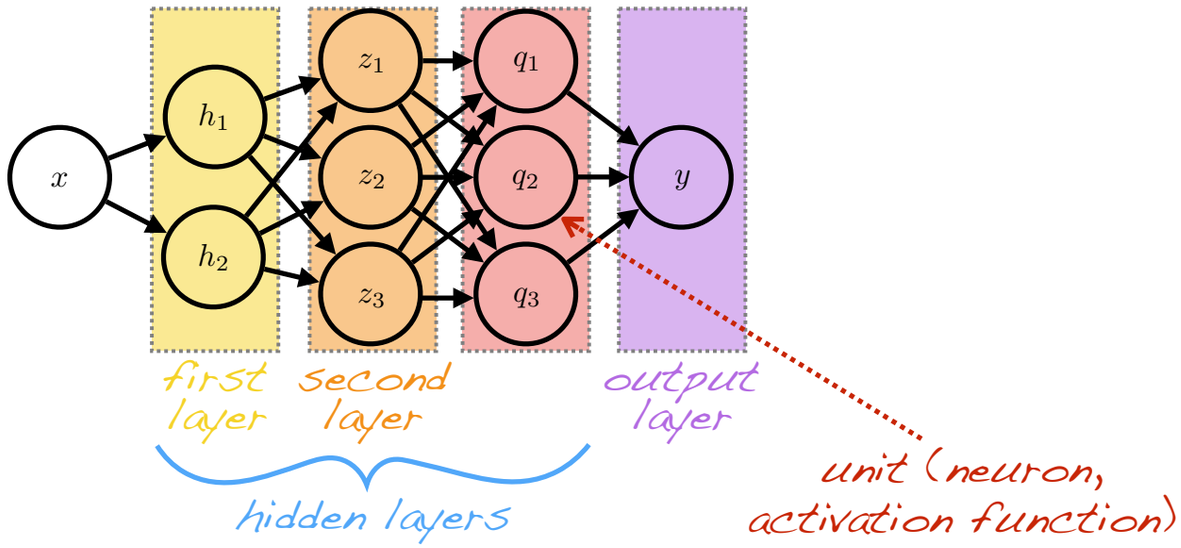


Some common terminology.
The width applies to each layer.
Deep learning refers to the large number of layers used in the latest version of neural networks.
Typically, hidden layers are not directly associated to a known output.
One unit is also referred to as a neuron. It is largely inspired by neuroscience
and the functional analogy of the neurons in the brain (an attempt to imitate them).
The function used in one unit is also referred to as activation function (again a loose reference to an analogy in neuroscience).

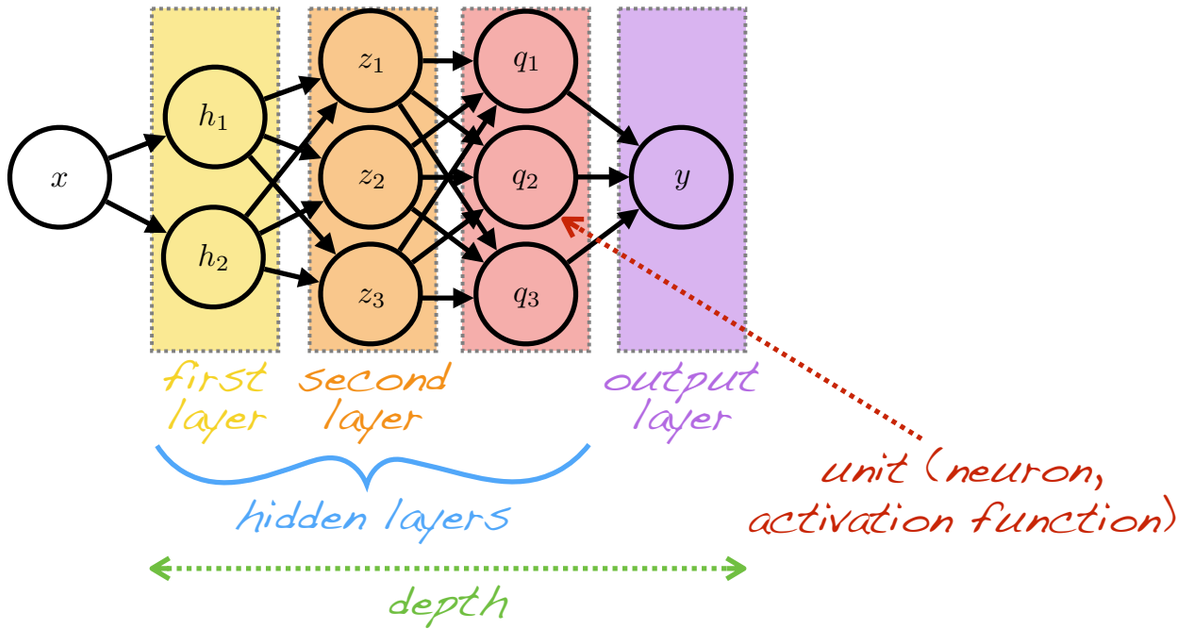
Feedforward Neural Networks



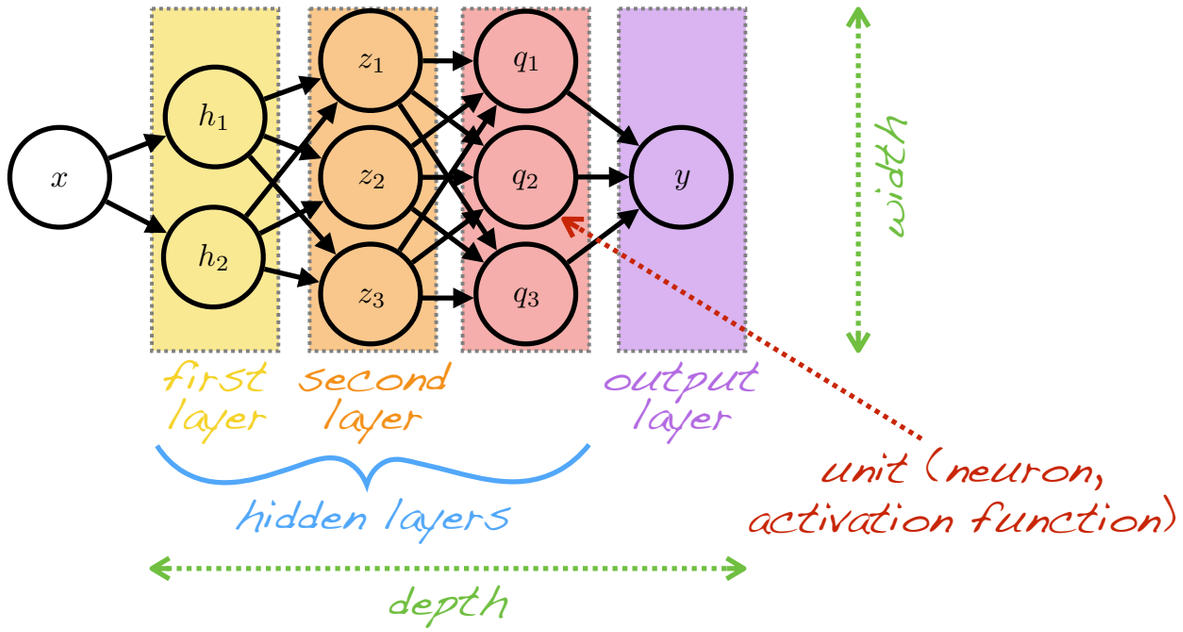
Feedforward Neural Networks



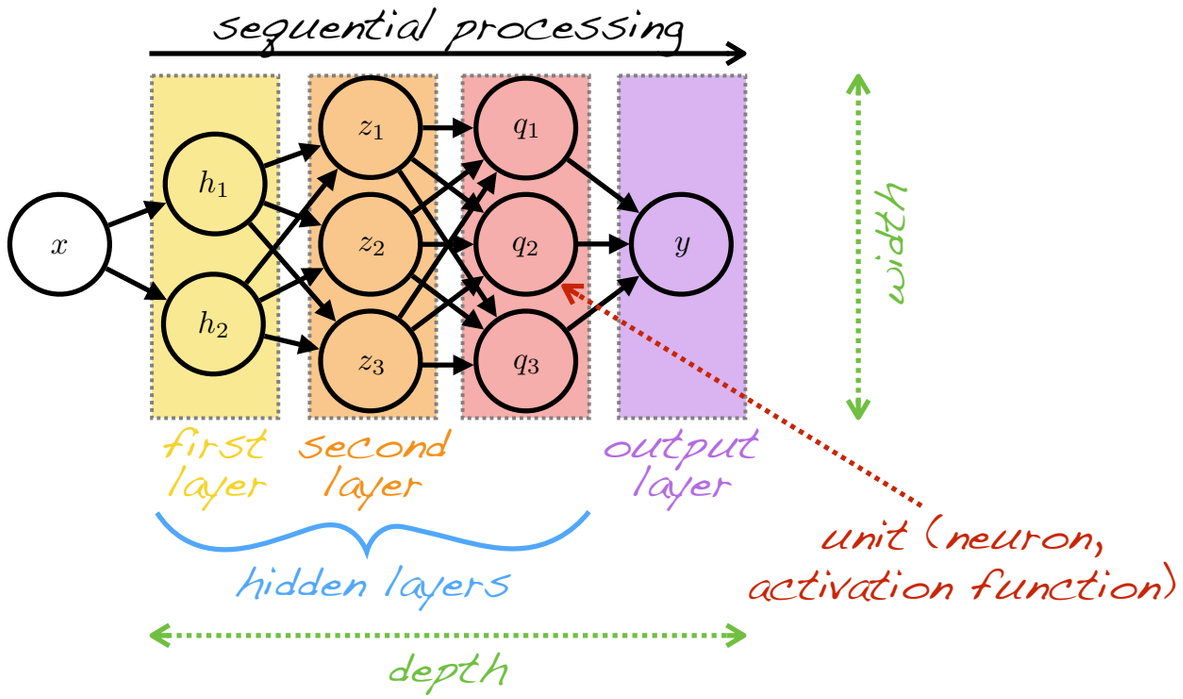
Feedforward Neural Networks



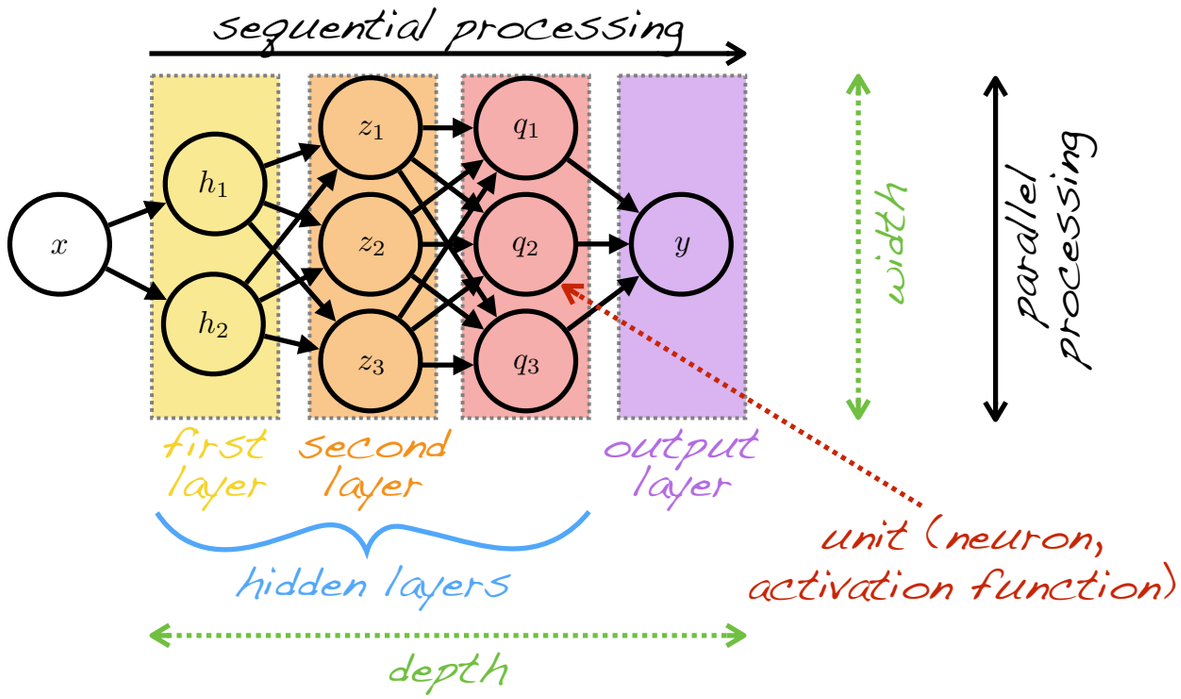
Feedforward Neural Networks



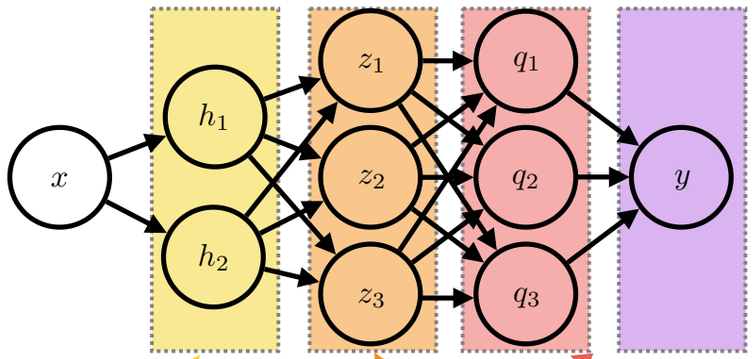
Feedforward Neural Networks



Feedforward Neural Networks



Feedforward Neural Networks



$$y = f_4(q_1, q_2, q_3)$$

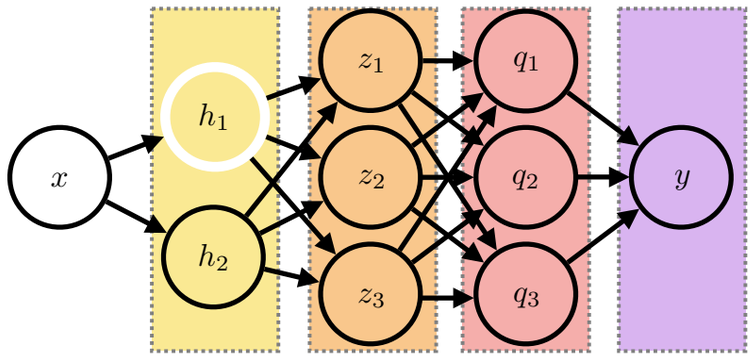
$$h_1 = f_{1,1}(x)$$
$$h_2 = f_{1,2}(x)$$

$$z_1 = f_{2,1}(h_1, h_2)$$
$$z_2 = f_{2,2}(h_1, h_2)$$
$$z_3 = f_{2,3}(h_1, h_2)$$

$$q_1 = f_{3,1}(z_1, z_2, z_3)$$
$$q_2 = f_{3,2}(z_1, z_2, z_3)$$
$$q_3 = f_{3,3}(z_1, z_2, z_3)$$

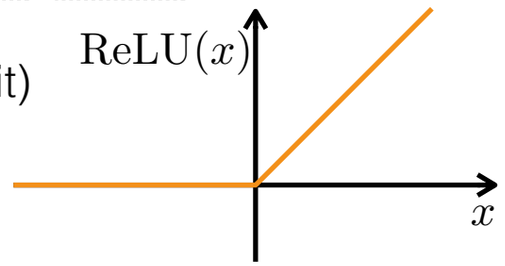
Mathematically we can express the relationships via functional composition

Feedforward Neural Networks



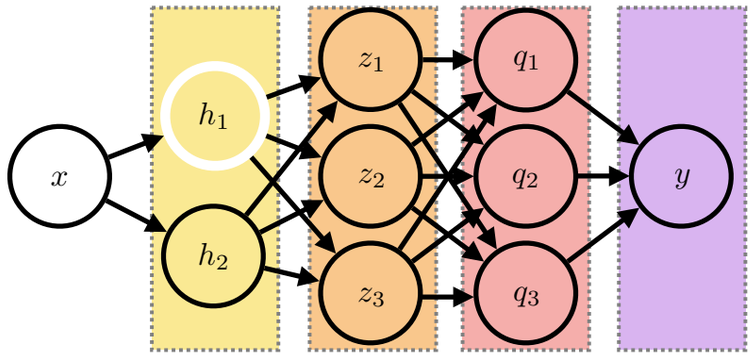
Example (rectified linear unit)

$f_{1,1}(x) = \mathrm{ReLU}(x)$



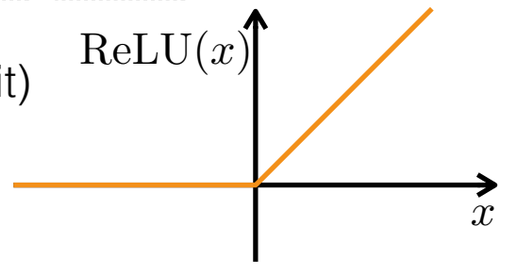
An example of simple function is the rectified linear unit (ReLU).
This is an example of a non linear function.
This function is parameter-free.

Feedforward Neural Networks

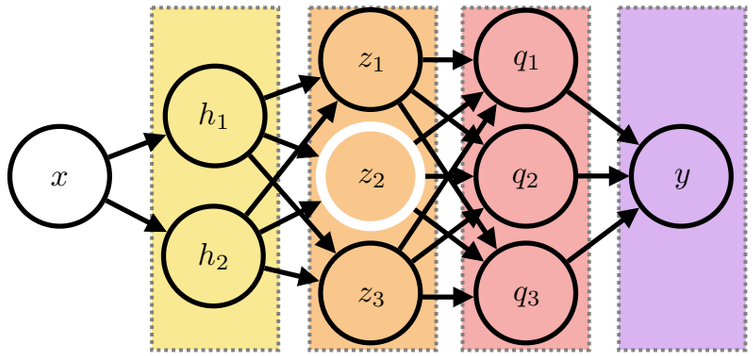


Example (rectified linear unit)

$f_{1,1}(x) = \mathrm{ReLU}(x)$



Feedforward Neural Networks

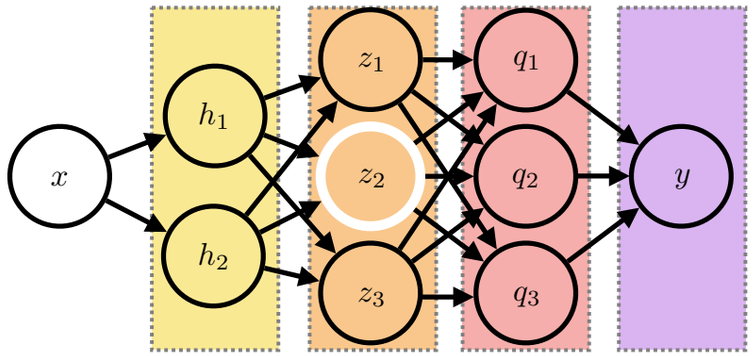


Example (fully connected unit)

$f_{2,2}(h_1, h_2) = w_1 h_1 + w_2 h_2$

Another example of simple function is the fully connected unit.
This is an example of a linear function.
This function is parametric (the parameters are w1 and w2).

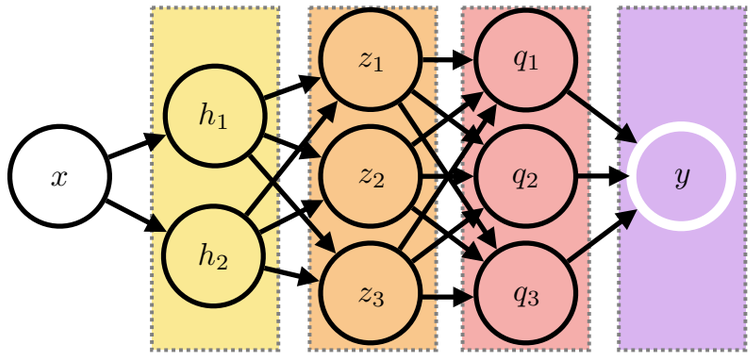
Feedforward Neural Networks



Example (fully connected unit)

$$f_{2,2}(h_1, h_2) = w_1 h_1 + w_2 h_2$$

Feedforward Neural Networks



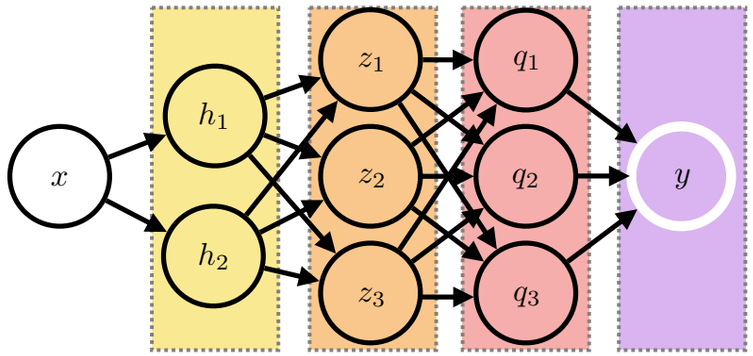
$$y = f_4(q_1, q_2, q_3)$$

Hierarchical composition of functions

$$y = f_4(f_{3,1}(f_{2,1}(f_{1,1}(x), f_{1,2}(x)), \dots), \dots)$$

Although each layer may implement a very simple function, the composition of several simple functions becomes quickly a very complex one.

Feedforward Neural Networks



$$y = f_4(q_1, q_2, q_3)$$

Hierarchical composition of functions

$$y = f_4(f_{3,1}(f_{2,1}(f_{1,1}(x), f_{1,2}(x)), \dots), \dots)$$

Feedforward Neural Networks

- Feedforward neural networks define a family of functions $f(x;\theta)$
- The goal is to find parameters θ that define the best mapping

$$y = f(x;\theta)$$

between input x and output y

- The key constraints are the I/O dependencies

The fundamental property of feedforward neural networks is the way they define the family of functions f. As in the previous slides, the main constraint is in the (compositional) dependencies.

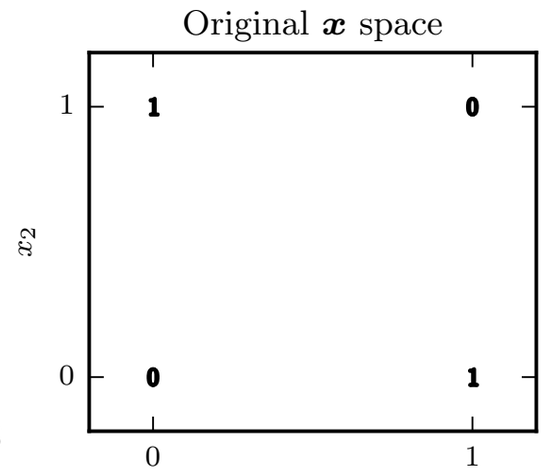
Deploying a Neural Network

- Given a **task** (in terms of I/O mappings)
- We need
  - **Cost function**
  - **Neural network model** (e.g., choice of units, their number, their connectivity)
  - **Optimization method** (back-propagation)

First, we need to have a clear objective. In ML we have already mentioned that this will be done by means of pairs (x,y) (input,output) (the training set).
Then we need to choose how we will penalize mistakes in the estimated mapping (e.g., 0-1 loss, L2 etc). Then we need to design the network. This is largely an art at this stage. Common practice is to start from networks that fit the task and that have been proven to work on similar problems. Then one modifies the network and uses diagnosis and error analysis tools to guide the changes. Finally, the training will require choosing an optimizer to minimize the cost function. We will use a gradient-based method, also referred to as back-propagation.

Example: Learning XOR

- Objective function is the XOR operation between two binary inputs x_1 and x_2



- Training set (x,y) pairs is

$$\left\{ \left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, 0 \right), \left(\begin{bmatrix} 0 \\ 1 \end{bmatrix}, 1 \right), \left(\begin{bmatrix} 1 \\ 0 \end{bmatrix}, 1 \right), \left(\begin{bmatrix} 1 \\ 1 \end{bmatrix}, 0 \right) \right\}$$

Let us use an example to get a sense for the whole process. We look at the case of data in a XOR configuration (an example that is often used as a toy problem to analyze classifiers).

Cost Function

- Let us use the Mean Squared Error (MSE) as a first attempt

$$J(\theta) = \frac{1}{4}\sum_{i=1}^{4}\left(y^i - f(x^i;\theta)\right)^2$$

Linear Model

- Let us try a linear model of the form

$$f(x; w, b) = w^\top x + b$$

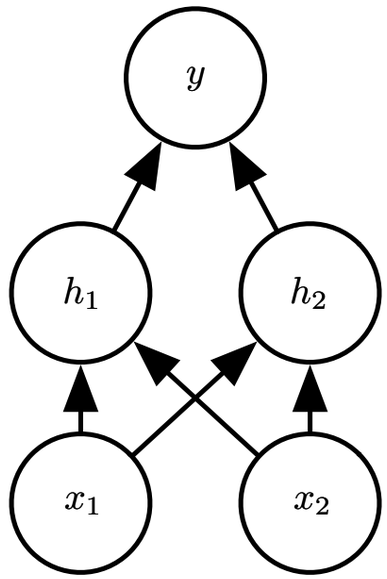
- This choice leads to the normal equations (see slides on Machine Learning Review) and the following values for the parameters

$$\omega = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \qquad b = \frac{1}{2}$$

So the output is a constant (b) for any input, which is largely unsatisfactory.
Graphically we can see that the plane with the smallest vertical distance from each 2D point in the training set is indeed the one with w=[0 0]^T and b=1/2.

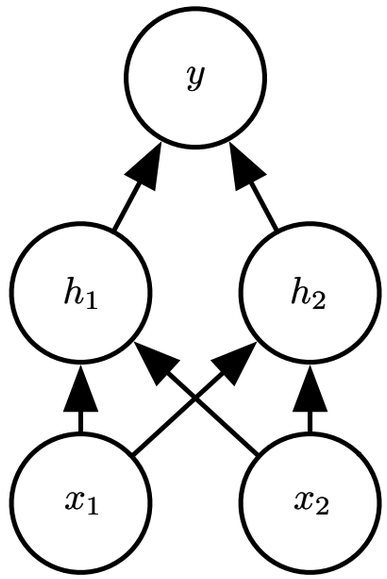
Nonlinear Model

- Let us try a simple feedforward network with one hidden layer and two hidden units



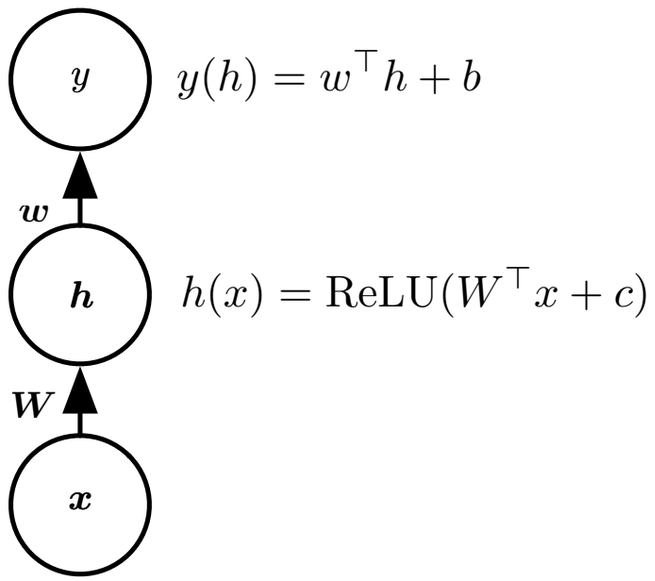
Nonlinear Model

- If each activation function is linear then the composite function would also be linear
- We would have the same poor result as before
- We must consider nonlinear activation functions



Nonlinear Model

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



For simplicity, let us group together all units in the same layer into vectors x and h.

Optimization

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

At this stage we would use optimization to fit f to the y in the training set. In this example, we skip this step and assume that some oracle gives us the parameters

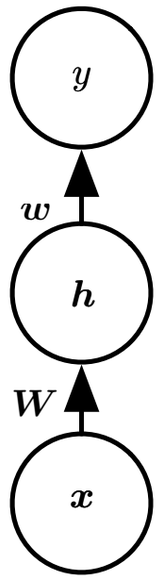
$$W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \qquad c = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

$$w = \begin{bmatrix} 1 \\ -2 \end{bmatrix} \qquad b = 0$$

Let us consider these settings and then compute the output of the function f on the training set.

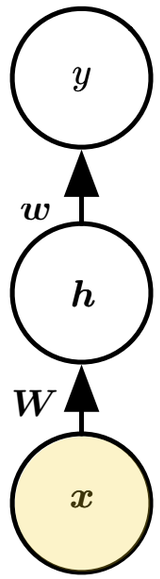
Simulation

$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$



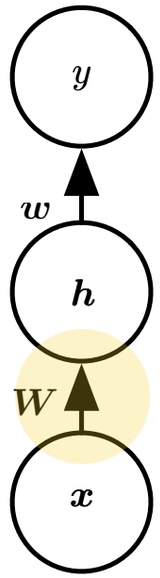
Simulation

$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$



Simulation

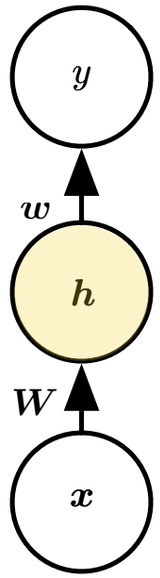
$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \Rightarrow XW + \mathbf{1}c = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$



Simulation

$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \rightarrow XW + \mathbf{1}c = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

$$\rightarrow \max\{0, XW + \mathbf{1}c\} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

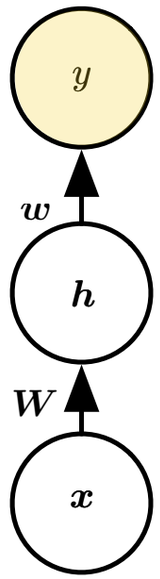


Simulation

$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \rightarrow XW + \mathbf{1}c = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

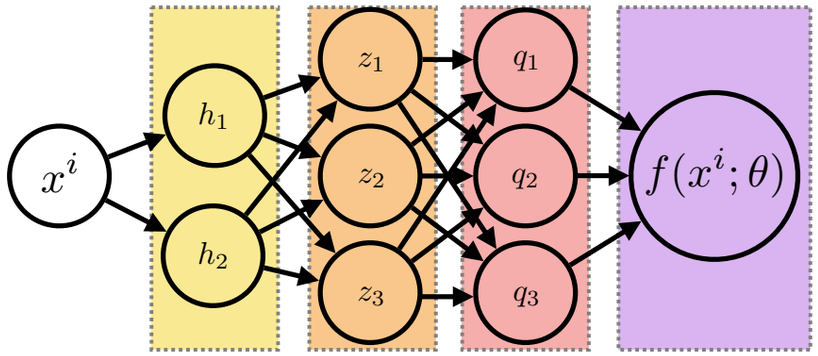
$$\rightarrow \max\{0, XW + \mathbf{1}c\} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

$$\rightarrow \max\{0, XW + \mathbf{1}c\}w + \mathbf{1}b = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$



the XOR function
(matches Y)

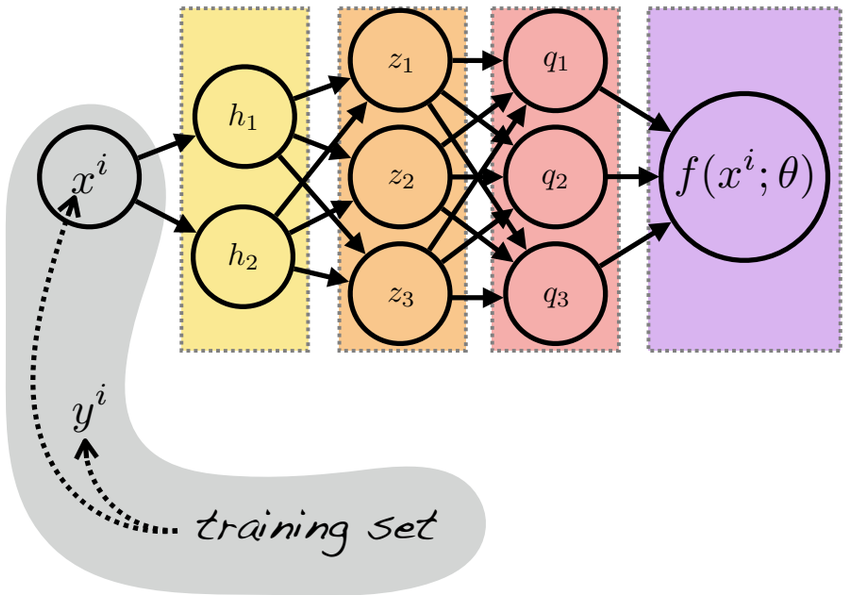
Step-by-Step Analysis



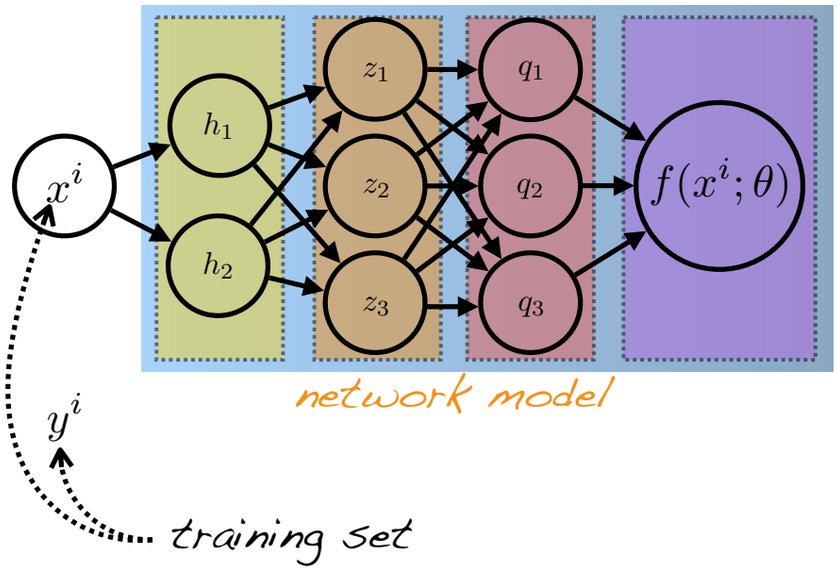
y^i

Now that we have seen an example of how to design the cost function, a model (and motivated the need for nonlinearity), and analysed the performance, we can present more in depth each item in the design of a machine learning algorithm: the cost function, the model (the neural network and, in particular, its hidden layers), and the optimization procedure.

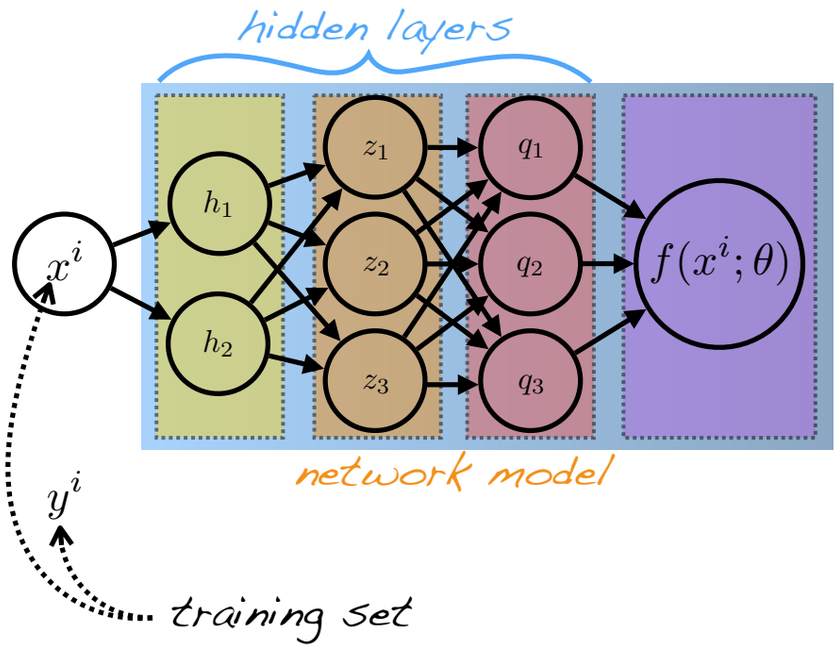
Step-by-Step Analysis



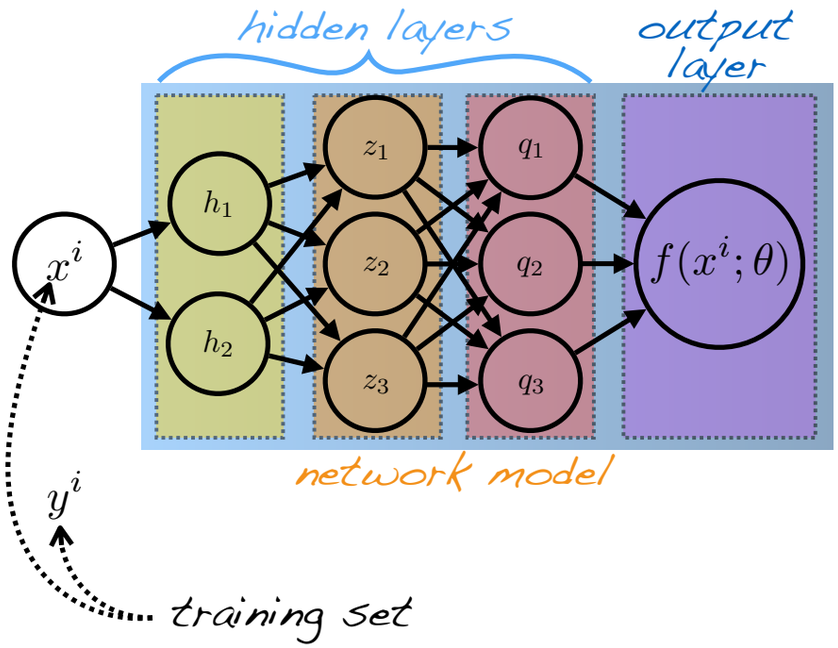
Step-by-Step Analysis



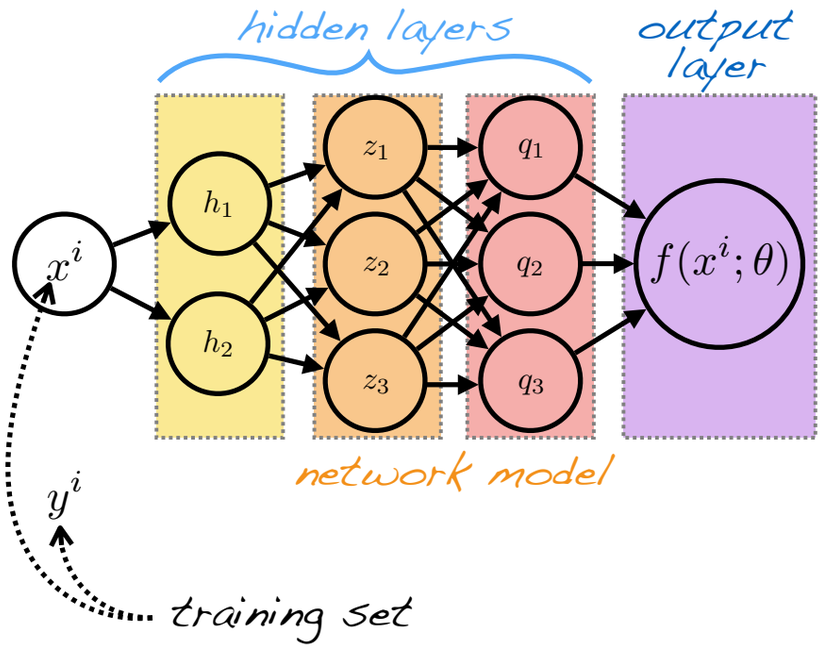
Step-by-Step Analysis



Step-by-Step Analysis



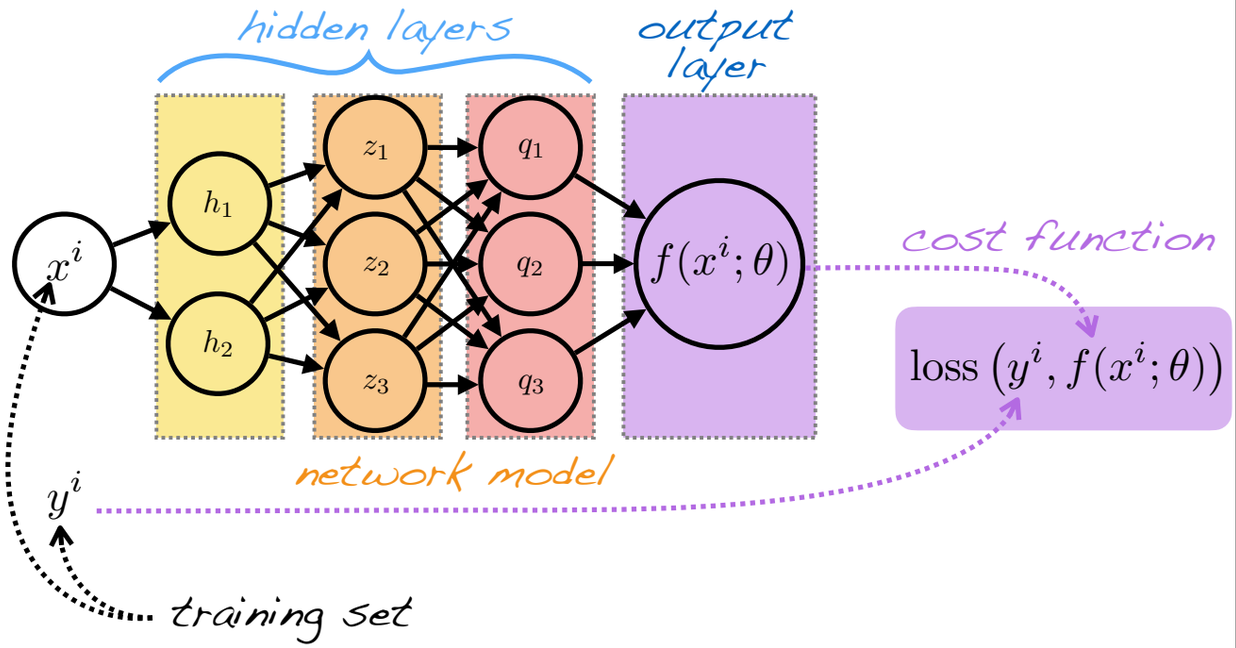
Step-by-Step Analysis



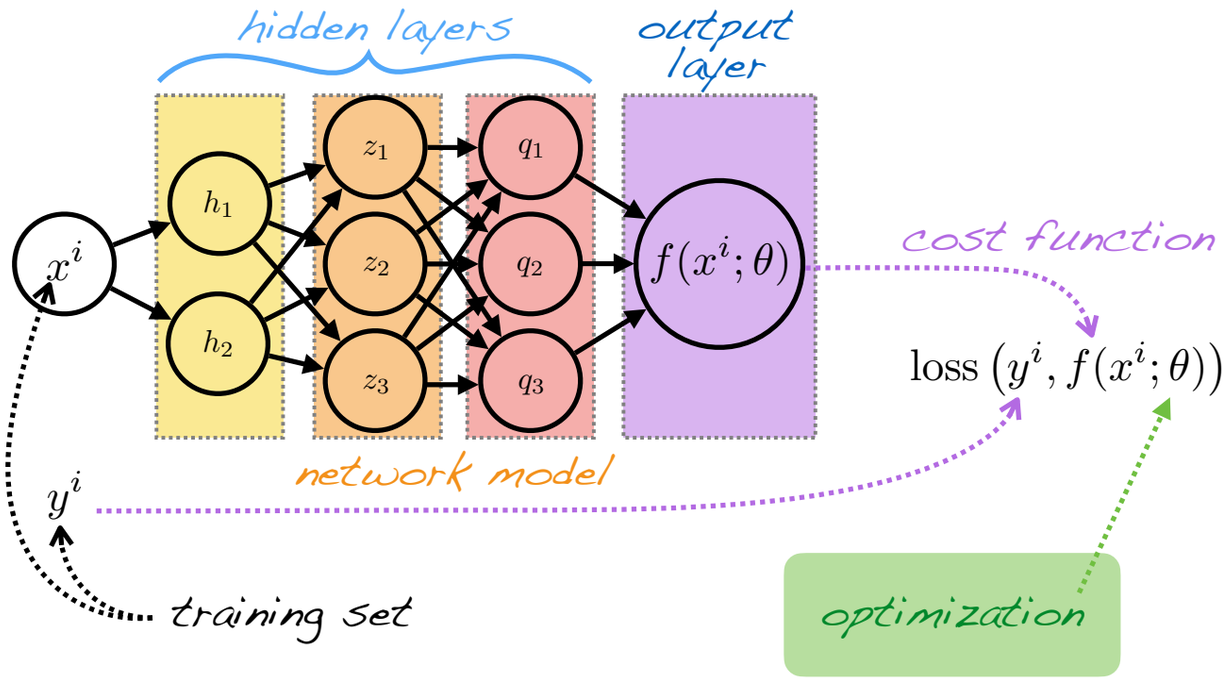
cost function

$$\text{loss}\left(y^i, f(x^i;\theta)\right)$$

Step-by-Step Analysis



Step-by-Step Analysis



Cost Function

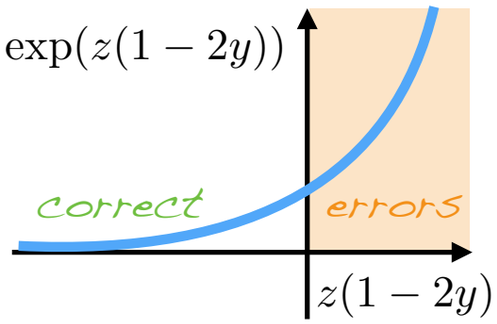
- Based on the **conditional distribution** $p_{\text{model}}(y|x;\theta)$
  - Maximum Likelihood (i.e., **cross-entropy** between model pdf and data pdf)

$$\min_{\theta} -E_{x,y\sim\hat{p}_{\text{data}}}[\log p_{\text{model}}(y|x;\theta)]$$

Let's start with the choice of cost function.
One option is to recover the whole probability density function of y|x.
This can be achieved with the cross-entropy minimization, which is equivalent to using maximum likelihood.

Saturation

- Functions that saturate (have flat regions) have a very small gradient and slow down gradient descent
- We choose loss functions that have a non flat region when the answer is incorrect (it might be flat otherwise)
- E.g., exponential functions saturate in the negative domain; with a binary variable $y \in \{0, 1\}$ map errors to the nonflat region and then minimize



- The logarithm also helps with saturation (see next slides)

In practice the cross entropy is preferred over other methods (e.g., based on statistics) because numerically is it more stable.
In the example above, we want z to have the same sign as 2y-1 where y=0 or y=1.
If we minimize the exponential form given above, we achieve this purpose.
The gradient will be nonzero when there is a mismatch (so the gradient will not stop).
When there is a match, the gradient can become zero.

Cost Function

- Based on **conditional statistics** $f(x;\theta)$ of $y|x$
- For example

$$f^* = \arg\min_f E_{x,y\sim\hat{p}_{\text{data}}}|y - f(x;\theta)|^2$$

gives the conditional mean

$$f^* = E_{y\sim\hat{p}_{\text{data}}(y|x)}[y]$$

Another option is to directly estimate some statistics of y|x. The L2 loss yields the conditional mean. Other well-known options are the L1 loss yields the conditional median.

Output Units

- The choice of the output representation (e.g., a probability vector or the mean estimate) determines the cost function
- Let us denote with

$$h = f(x; \theta)$$

the output of the layer before the output unit

Linear Units

- With a little abuse of terminology, linear units include **affine transformations**

$$\hat{y} = W^\top h + b$$

  can be seen as the mean of the conditional Gaussian distribution (in the Maximum Likelihood loss)

$$p(y|x) = \mathcal{N}(y; \hat{y}, I)$$

- The Maximum Likelihood loss becomes

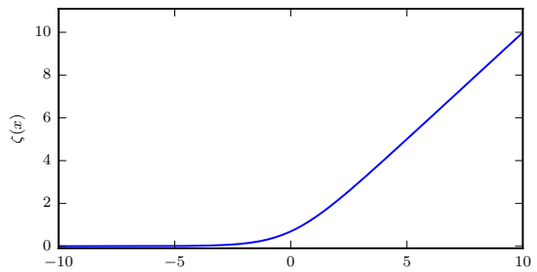
$$-\log p(y|\hat{y}) = |y - \hat{y}|^2 + \text{const}$$

Linear units are easy to work with (during training the gradients are simple and stable)

Softplus

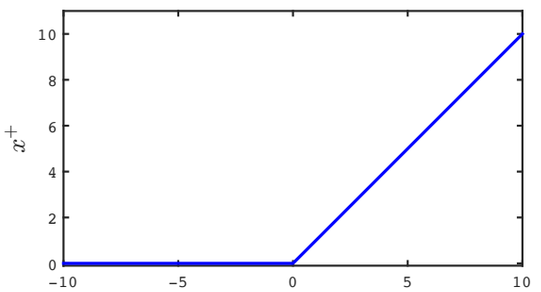
- The **softplus** function is defined as

$$\zeta(x) = \log(1 + \exp(x))$$



and it is a smooth approximation of the Rectified Linear Unit (ReLU)

$$x^+ = \max(0, x)$$



Sigmoid Units

- Use to predict binary variables or to predict the probability of binary variables

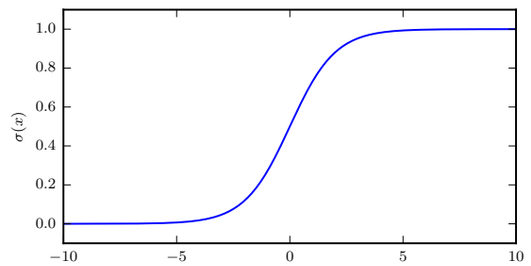
$$p(y = 0|x) \in [0, 1]$$

- The sigmoid unit defines a suitable mapping and has no flat regions (useful in gradient descent)

$$\hat{y} = \sigma(w^\top h + b)$$

where we have used the **logistic sigmoid** function

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



In practice, the sigmoid unit is a combination of a linear unit and the logistic sigmoid function.
The lack of (strongly) flat regions helps gradient descent. When regions are flat the gradients become zero and there is no useful update to the parameters (an extremum has been reached).
Notice that the logistic sigmoid function is the derivative of the softplus function.

Bernoulli Parametrization

- Let $z = w^\top h + b$. Then, we can define the Bernoulli distribution

$$p(y|z) = \sigma((2y-1)z)$$

- The loss function with Maximum Likelihood is then

$$-\log p(y|z) = \zeta((1-2y)z) \simeq \max(0, (1-2y)z)$$

and saturation occurs only when the output is correct (y=0 and z<0 or y=1 and z>0)

When the output is incorrect the gradient changes linearly with z.
This is a desirable stable behaviour in the algorithm.
This behaviour is not guaranteed when we use other loss functions (e.g., the least squares).
Thus, ML is recommended when using the sigmoid.

Smoothed Max

- An extension to the softplus function is the smoothed max

$$\log \sum_j \exp(z_j)$$

  which gives a smooth approximation to $\max_j z_j$

- If we rewrite the softplus function as

$$\log(1 + \exp(z)) = \log(\exp(0) + \exp(z))$$

  we can see that it is the case with $z_1 = 0, z_2 = z$

This is a generalisation of softplus that is used in the Softmax output layer.

Softmax Units

- An extension of the logistic sigmoid to multiple variables
- Used as the output of a multi-class classifier
- The **Softmax** function is defined as

$$\mathrm{softmax}(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

- Shift-invariance: $\mathrm{softmax}(z + \mathbf{1}c) = \mathrm{softmax}(z)$

  gives numerically stable implementation

$$\mathrm{softmax}(z - \max_j z_j) = \mathrm{softmax}(z)$$

The softmax gives a probability over multiple classes.
It is automatically normalized.

Softmax Units

- In Maximum Likelihood we have

$$\log \mathrm{softmax}(z)_i = z_i - \log \sum_j \exp(z_j)$$

- Recall the smoothed max, then we can write

$$\log \mathrm{softmax}(z)_i \simeq z_i - \max_j z_j$$

- Maximization, with $i = \arg\max_j z_j$, yields

$$\mathrm{softmax}(z)_i = 1 \quad \text{and} \quad \mathrm{softmax}(z)_{j \neq i} = 0$$

ML leads softmax to a vector with one 1 and all the rest at 0.

Softmax Units

- Softmax is an extension to the logistic sigmoid where we have 2 variables and $z_1 = 0, z_2 = z$

$$p(y = 1|x) = \text{softmax}(z)_1 = \sigma(z_2)$$

- Softmax is a winner-take-all formulation

- Softmax is more related to the arg max function than the max function

Because probabilities must add up to 1, we can parametrize the probability on n variables with an n-1 dimensional vector. In the case of binary variables this is convenient because we only need to care about 1 probability and the logistic sigmoid is a practical parametrization choice. However, with multi-dimensional variables it is often simpler to implement the full n-dimensional probability vector.

General Output Units

- A neural network can be written as a function $f(x;\theta)$
- This function could output the value of y or parameters ω of the pdf of y such that $f(x;\theta) = \omega$
- In this case the loss function (with ML) is

$$-\log p(y;\omega(x))$$

- For example, the parameters could represent the mean and precision of the Gaussian distribution of y

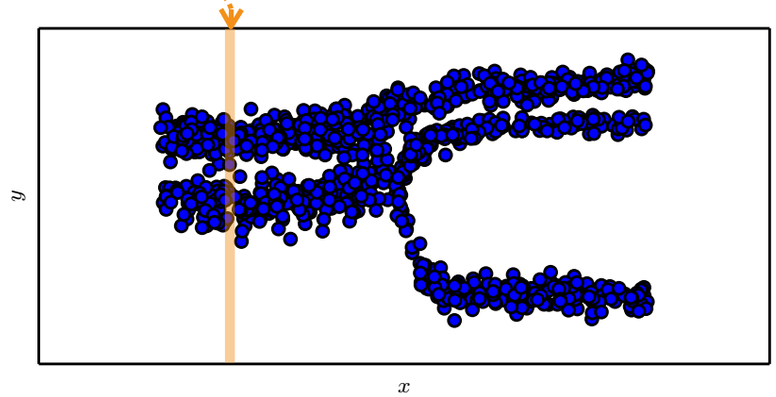
General Output Units

- Mixture density models are used for multimodal probability densities (i.e., multi-peaked outputs)

$$p(y|x) = \sum_i p(c = i|x)\mathcal{N}(y; \mu^i(x), \Sigma^i(x))$$

- The parameters include

$p(c = i|x)$
$\mu^i(x)$
$\Sigma^i(x)$



In the example above we sample x uniformly and then we sample
p(y|x). As we can see, depending on x we have different multi-modal distributions. The concentration of y samples takes place at different locations and these locations change with x.

Hidden Units

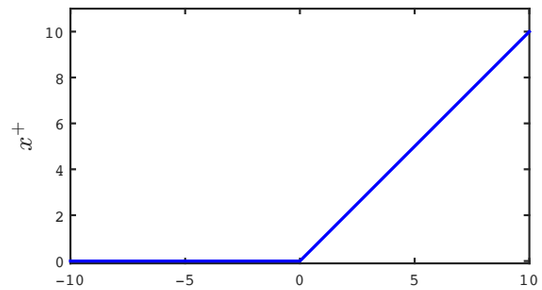
- The design of a neural network is so far still an art
- The basic principle is the **trial and error** process:
  1. Start from a known model
  2. Modify
  3. Implement and test (go back to 2. if needed)
- A good choice is to always use ReLUs
- In general the hidden unit picks a g for

$$h(x) = g(W^\top x + b)$$

Rectified Linear Units

- ReLUs typically use also an affine transformation

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



- Good initialization is b = 0.1 (initially, a linear layer)
- Negative axis cannot learn due to null gradient
- Generalizations help avoid the null gradient

Leaky ReLUs and More

- A generalization of ReLU is

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

- To avoid a null gradient the following are in use

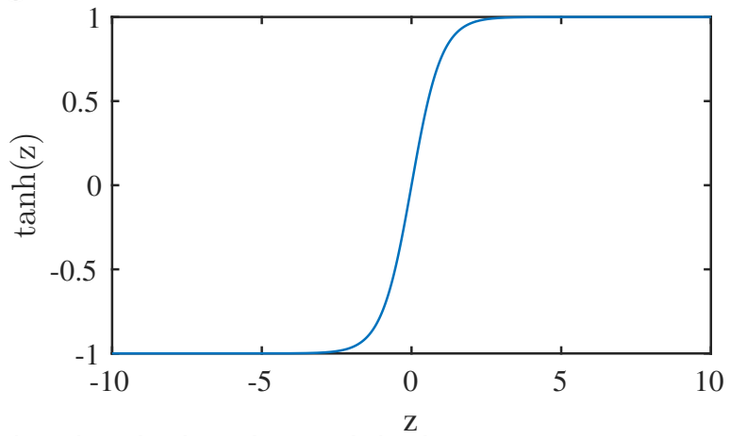
1. Absolute value rectification	$\alpha = -1$
2. Leaky ReLU	$\alpha = 0.01$
3. Parametric ReLU	α learnable
4. Maxout Units	$g(z)_i = \max_{j \in S_i} z_j$
	$\cup_i S_i = [1, \ldots, m]$
	$S_i \cap S_j = \varnothing \quad i \neq j$

Maxout units generalize all the other units. However, they may be more difficult to train.
Their redundancy in the representation also allows them to be more robust to catastrophic forgetting (the network forgets how to perform a task that was previously learned — transfer learning).

Hyperbolic Tangent

- The hyperbolic tangent is defined as

$$g(z) = \tanh(z)$$



and it is related to the logistic sigmoid via

$$\tanh(z) = 2\sigma(2z) - 1$$

The hyperbolic tangent and the logistic sigmoid were used quite widely as the nonlinear components in neural networks. However, their saturation properties can make gradient-based learning very difficult. Thus, they are now mostly substituted by ReLUs. These networks can be used as output units especially when the loss function "undoes" the saturation.

They are used often in recurrent neural networks, probabilistic models and autoencoders.

Other Units

- Linear projection $W = VU$
- Radial Basis Functions $h_i(x) = \exp\left(-\frac{1}{\sigma_i^2}\left|x - W_i\right|^2\right)$
- Softplus $g(z) = \zeta(z) = \log(1 + \exp(z))$
- Hard tanh $g(z) = \max\{-1, \min\{+1, z\}\}$

Many other hidden units have been tested and published. When they perform as the existing ones, they are not deemed useful or interesting.
The linear projection allows factorizations.
Instead of using an affine transformation W we can have first a projection U (possibly to a lower dimensional space) and then a projection V. The (low-rank) factorization VU of W uses fewer parameters than in the case of a general W matrix.
The skip connection is a special case that implements the identity function.
The RBF saturates for most x values so it is also difficult to optimize.
The softplus seems to be a better option than ReLU because it is always differentiable. However, experimentally it performs worse than ReLU.

Network Design

- The **network architecture** is the overall structure of the network: number of units and their connectivity

- Today, the design for a task must be found experimentally via a careful analysis of the training and validation error

Universal Approximation

- **Theorem**
  A feedforward network with a linear output layer and enough (but at least one) hidden nonlinear layers (e.g., the logistic sigmoid unit) can approximate up to any desired precision any (Borel measurable) function between two finite-dimensional spaces

- This means that neural networks provide a **universal representation**/approximation

An example of Borel measurable function is any continuous function on a closed and bounded subset of R^n.

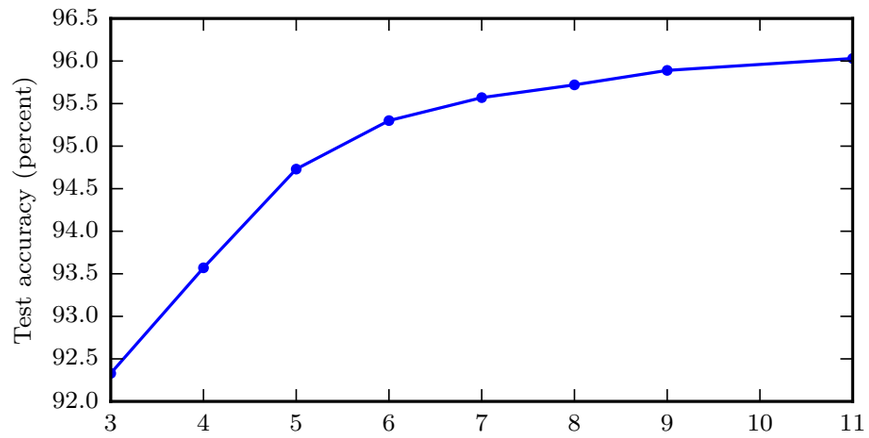
Universal Approximation

- However, we are not guaranteed that the learning algorithm will be able to build that representation
- Learning might fail to find some good parameters
- Learning might fail due to overfitting (see “No Free Lunch” theorem)

An example of Borel measurable function is any continuous function on a closed and bounded subset of R^n.

Depth

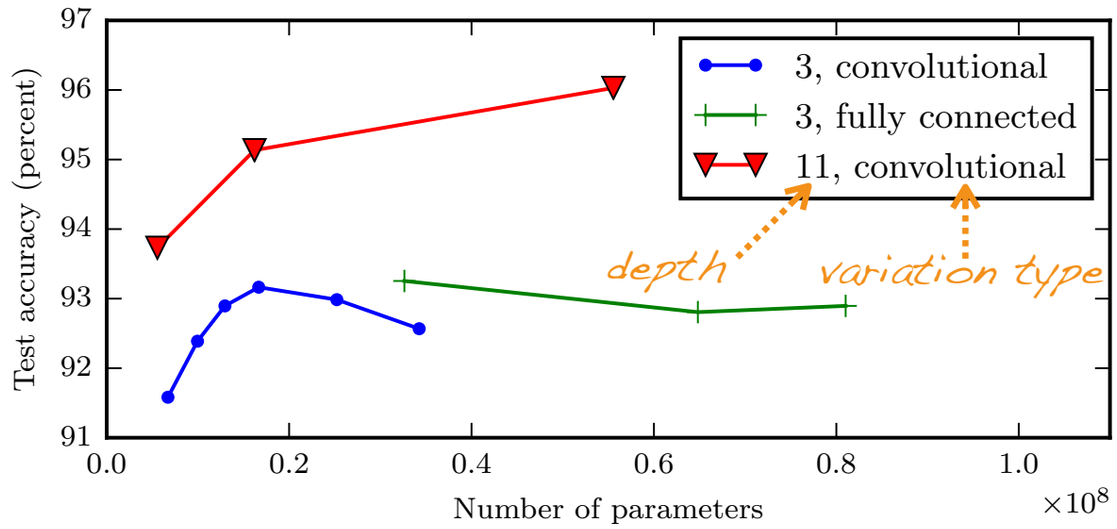
- A general rule is that depth helps generalization
- It is better to have many simple layers than few highly complex ones



The figure shows that deeper networks to transcribe multi-digit numbers from images generalize better than shorter ones.

Depth

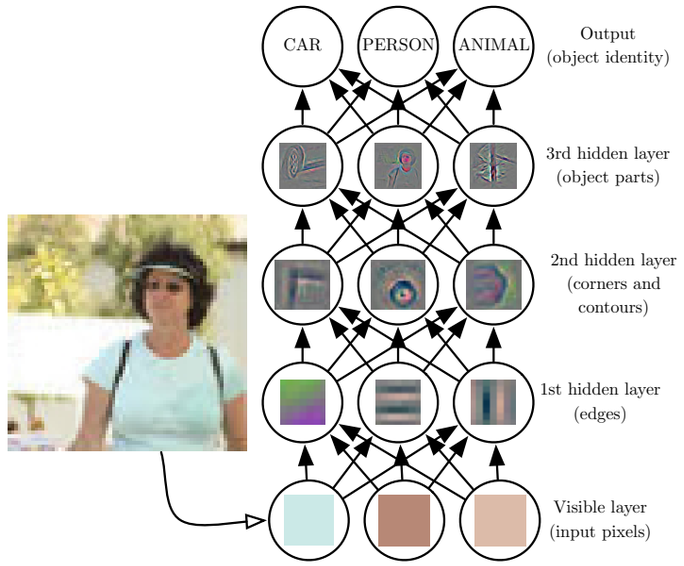
- Other network modifications do not have the same effect



Depth increases the number of parameters of the network (and thus its capacity). Increasing the parameters by working on other factors does not have the same impact on the performance. Moreover, all shallow models overfit.

Depth

- Another interpretation is that depth allows a more gradual abstraction



A deep network might give a useful representation, where concepts are gradually more and more abstract. A learning problem is solved via the detection of simple underlying factors of variation that can also be described in a similar way via the detection of more abstract factors of variation.

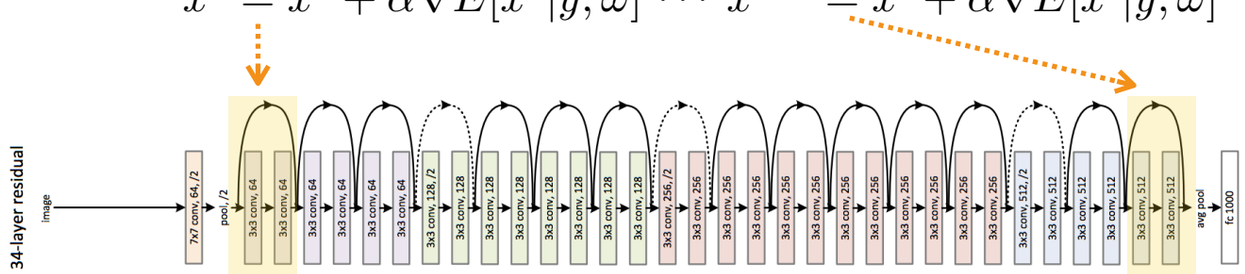
Depth

- Another interpretation is that the network implements a gradient descent algorithm for

$$\tilde{x} = \arg\min_x E[x|y,\omega]$$

where the network represents $f(x^0;\theta) = x^{t+1}$ and

$$x^1 = x^0 + \alpha\nabla E[x^0|y,\omega] \cdots x^{t+1} = x^t + \alpha\nabla E[x^t|y,\omega]$$



This interpretation defines explicitly the network and the depth corresponds to the number of gradient descent iterations (the more iterations and the better the final accuracy of the function); each step performs a local linearized operation (that is, relatively simple).

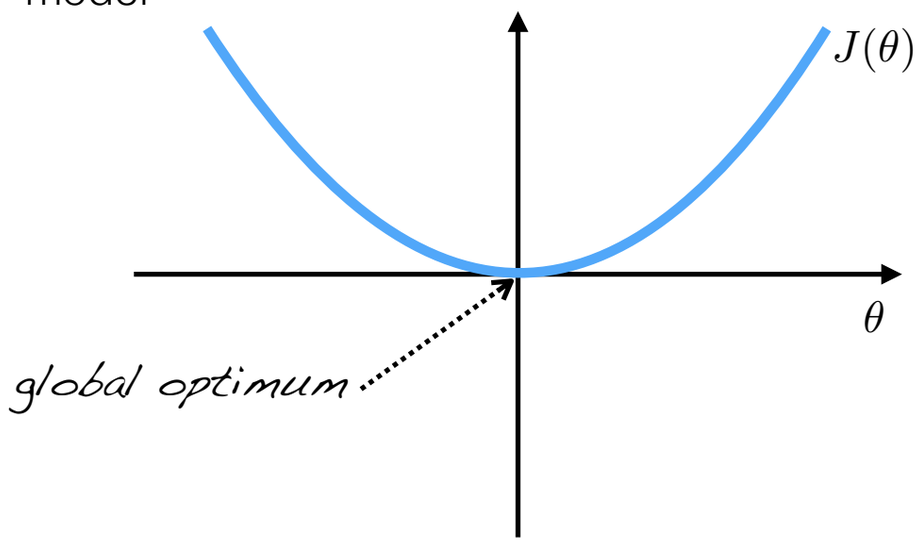
Optimization

- Given a task we define
  - The training data $\{x^i, y^i\}_{i=1,\ldots,m}$
  - A network design $f(x;\theta)$
  - The loss function $J(\theta) = \sum_{i=1}^{m} \text{loss}\left(y^i, f(x^i;\theta)\right)$
- Next, we **optimize** the network parameters θ
- This operation is called **training**

We now look more in detail at how we can improve the network parameters so that the network can make better predictions on the training set.

Optimization

- The MSE cost function $J(\theta)$ is **convex** with a linear model



We used the MSE as our loss function in the initial (XOR) example. With a linear model the optimization problem is convex and has a unique global optimum.
While we could get a closed form solution in this case, the solution was not good.
Then, we considered a neural network which made the model non linear and thus the loss function non convex.

Optimization

- However, since the cost function $J(\theta)$ is typically **non convex** in the parameters, we use an iterative solution

- We consider the **gradient descent** method

$$\theta_{t+1} = \theta_t - \alpha \nabla J(\theta_t)$$

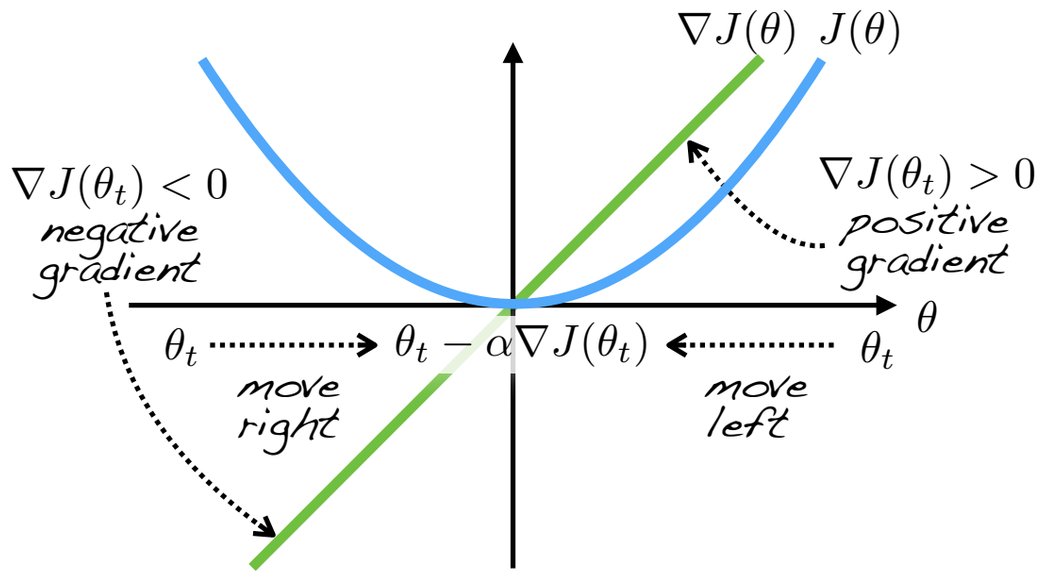
where $\alpha > 0$ is the learning rate

In the more general case, we need to resort to iterative solutions such as gradient descent. This method takes a scaled version of the negative of the gradient as the update for the previous set of parameters.

When we want to reach the valley and we start from a random position we could follow the slope at our current location. This is the principle of gradient descent.

Optimization

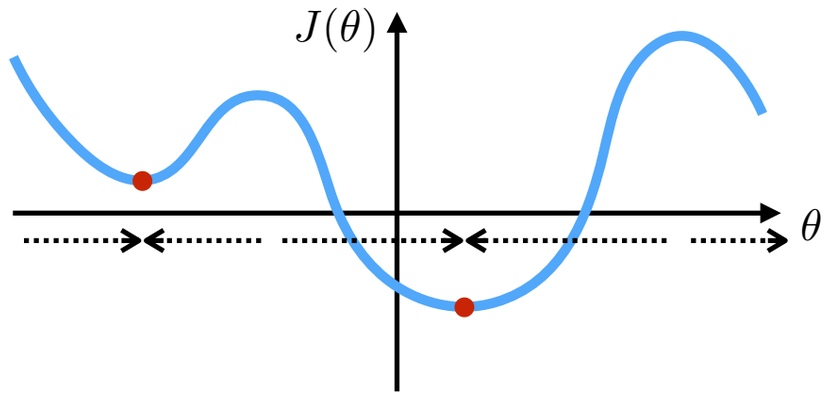
gradient descent $\theta_{t+1} = \theta_t - \alpha \nabla J(\theta_t)$



Let us illustrate the way gradient descent works when the loss is convex and in 1D. In this case it is immediate to see how the negative of the gradient points always towards the global minimum.

Local Minima

- Does gradient descent reach a (local) minimum even with a non convex function?

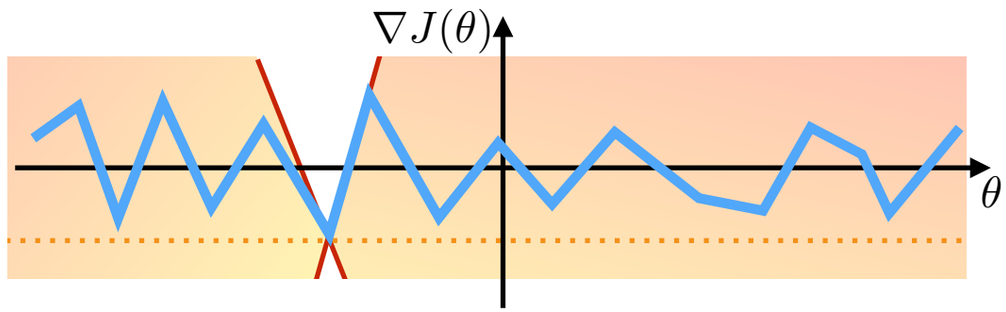


- How do we show that?

How do we ensure that, regardless of where we are in the domain (x-axis in the figure), we make gradient descent move towards the local minimum?

Assumptions

- Gradient descent will reach a local minimum under some **constraints** on both the cost function and the learning rate
- We show that by using a smoothness assumption called **Lipschitz continuity** on $\nabla J(\theta)$



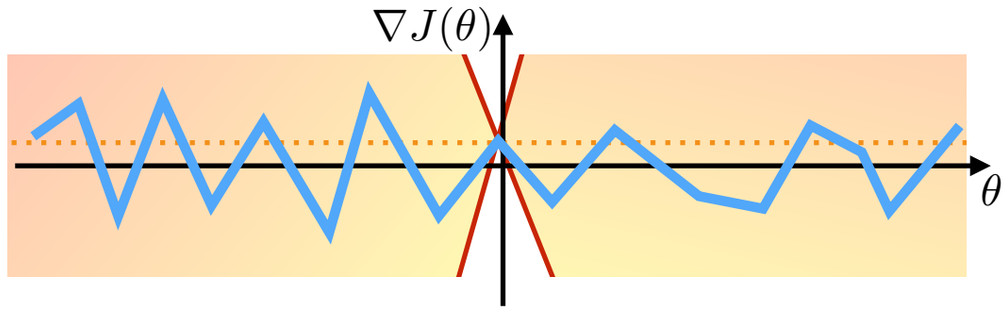
Gradient descent requires some additional constraints to converge to a local minimum.
Understanding these constraints is key to know what to do when gradient descent does not work.

One such constraint is Lipschitz continuity of the gradient of the cost function (notice that it applies to the gradient, not to the cost function itself).

Lipschitz continuity defines the maximum slope in the whole domain. In the illustration Lipschitz continuity states that there exists a cone (red lines) that can be translated at any point of the function and such that the function will not enter it (the function will always be in the red-shaded region).

Assumptions

- Gradient descent will reach a local minimum under some **constraints** on both the cost function and the learning rate
- We show that by using a smoothness assumption called **Lipschitz continuity** on $\nabla J(\theta)$



Convergence

If we assume (Lipschitz continuity)

$$\exists L \geq 0: \quad \left|\nabla J(\theta) - \nabla J(\bar{\theta})\right| \leq L|\theta - \bar{\theta}|, \qquad \forall \theta, \bar{\theta}$$

then for a **small enough learning rate** α the gradient descent iteration will generate a sequence $\theta_1, \ldots, \theta_T$ such that*

$$J(\theta_{t+1}) < J(\theta_t)$$

decreasing

if $\nabla J(\theta_t) \neq 0$; i.e., it will converge to a local minimum.

***See Tutorial 2 of the Machine Learning Course**

Under mild conditions, we can show that there is a suitable small learning rate such that gradient descent is guaranteed to converge to a local minimum.

Diagnosing GD

- In practice, what do we do when gradient descent does not work?
- Since it must work when the assumptions are true, it must be that the assumptions are violated
- The next step is to determine which assumptions are violated
- There are two: Lipschitzianity and small learning rate

Diagnosis 1/2

- **Case 1: Lipschitzianity**

Diagnosis: this is a purely analytical diagnosis. One can simply look at the cost function formula and determine if the functions in it may not satisfy Lipschitz continuity (of its gradient).
This step should be done first.

Diagnosis 1/2

- **Case 1: Lipschitzianity**

- The cost function does not satisfy the Lipschitz condition for any L

Diagnosis 1/2

- **Case 1: Lipschitzianity**
- The cost function does not satisfy the Lipschitz condition for any L
- **Solution**: Smooth the cost function until an L exists

Diagnosis 2/2

- **Case 2: Learning rate**

Diagnosis here is more experimental.
One can also compute the magnitude of the gradient of the cost function and compare it to the magnitude of the current parameters. The learning rate should scale the update (the gradient of the cost function) to a small percentage (e.g., 10^-4) of the current parameters magnitude.

Diagnosis 2/2

- **Case 2: Learning rate**

- The learning rate is not small enough

Diagnosis 2/2

- **Case 2: Learning rate**
- The learning rate is not small enough
- **Solution**: Make it smaller until gradient descent starts to work

Optimization

- For more efficiency, we use the **stochastic gradient descent** method
- The gradient of the loss function is computed on a small set of samples from the training set

$$\tilde{J}(\theta) = \sum_{i \in B \subset [1,\ldots,m]} \text{loss}\left(y^i, f(x^i;\theta)\right)$$

and the iteration is as before

$$\theta_{t+1} = \theta_t - \alpha \nabla \tilde{J}(\theta_t)$$

Under some mild requirements (some smoothness of the loss function and a decreasing learning rate) also Stochastic GD (and other similar variations) can be shown to converge to a local minimum.
Because SGD requires a small subset of all the samples, it can be computed very efficiently with a positive impact on the overall training time.

Before we discussed how to set the learning rate. Now we look at the other remaining component
in the update equation: the gradient of the loss. How do we obtain it?

Back-Propagation

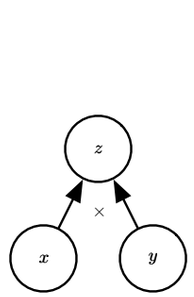
- (Stochastic) gradient descent boils down to the calculation of the loss gradient with respect to the parameters
- Due to the compositional structure of the network, the gradient can be computed in many ways
- **Back-propagation** (or simply backprop) refers to a particularly computationally efficient procedure for computing the gradient

Note that backprop does not refer to the whole training algorithm, but just to the specific way used to compute the gradient

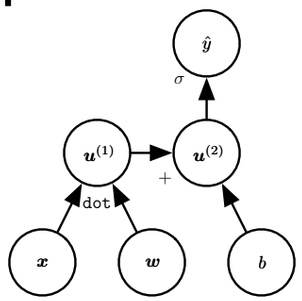
Computational Graphs

- Because neural networks can quickly become very complex, a clear representation is needed
- We use a **computational graph** to formalize the computations
- Each **node** is assigned to a variable (e.g., a scalar, a vector, a matrix, a tensor)
- An **operation** transforms one or more variables into another variable

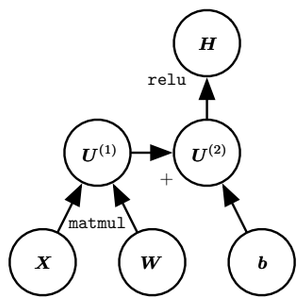
Examples



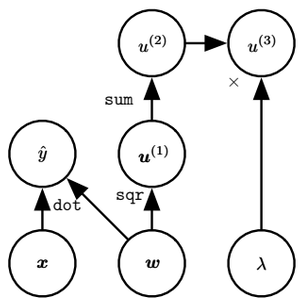
(a)



(b)



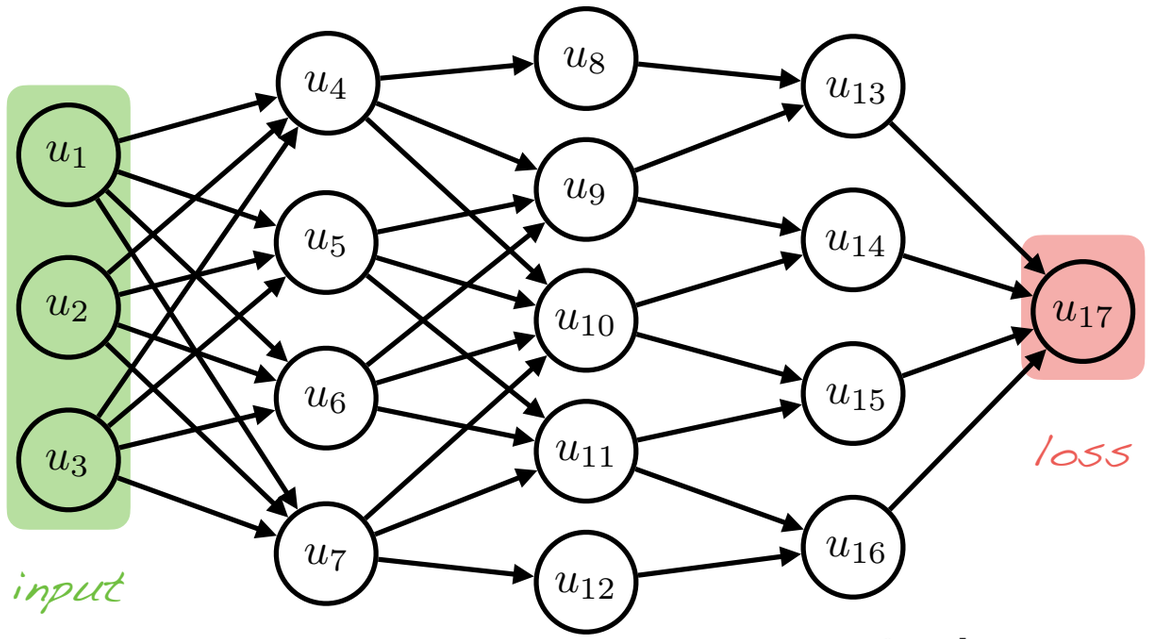
(c)



(d)

Examples of computational graphs.
(a) z = xy
(b) \hat y = \sigma(x^T w + b) [log. reg.]
(c) H = max{0,XW+b}
(d) \hat y = x^T w and u3 = \lambda \sum w^2; w is used to perform multiple operations

The Computational Graph



$\theta = [u_4, \ldots, u_{16}]$

We define each variable in the network with a node u_i. The nodes in the first layer denote the input variables and the last node denotes the loss function. The input could be just 1 minibatch. Each link defines the inputs to a node, and each node is associated to a function f_i of the inputs.

Computing Gradients

- The main objective is to compute the derivatives of the loss node with respect to the input nodes

$$\frac{\partial u_{17}}{\partial u_i} \qquad i = 1, 2, 3$$

- The loss depends on the input nodes through functional composition

$$u_{17} = f_{17}(u_{13}, u_{14}, u_{15}, u_{16})$$

Because of the compositional structure of the network (and therefore of the computational graph), the natural tool to compute derivatives is the chain rule. Moreover, to compute the gradients of the loss with respect to the inputs in a computationally and memory efficient way we need to implement the chain rule carefully.

Chain Rule

- The derivatives of a function composition can be computed via the chain rule
- Consider $y = g(x)$ and $z = f(g(x)) = f(y)$

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$

- In the multivariate case we have $\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$

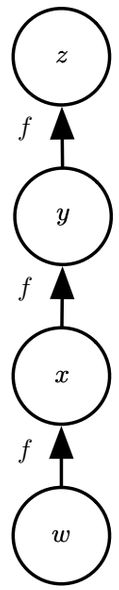
  or, more compactly $\nabla_x z = (\nabla_x y)^\top \nabla_y z$

The same rule can be applied to matrices and tensors simply by vectorizing them, then by applying the above equations and finally by reshaping them to the original shape. Indeed we can think of a tensor as a vector whose indices have multiple coordinates.

Computational Challenges

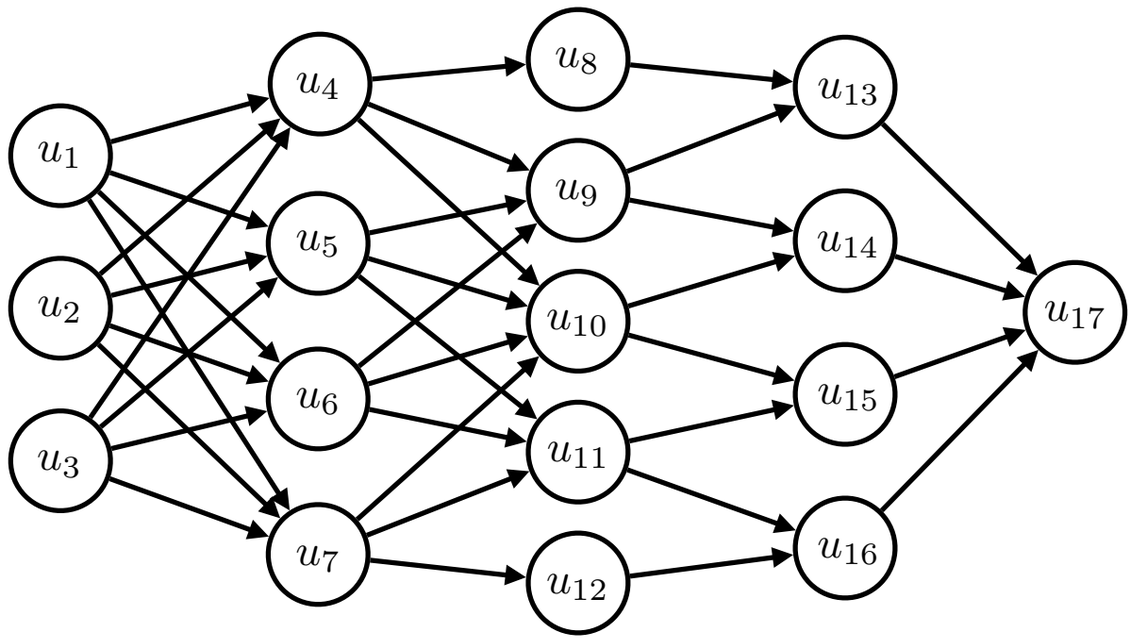
- In the chain rule, subexpressions may repeat
- Example

$$
\begin{aligned}
&\frac{\partial z}{\partial w} \\
=&\frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \\
=&f'(y)f'(x)f'(w) \\
=&f'(f(f(w)))f'(f(w))f'(w)
\end{aligned}
$$



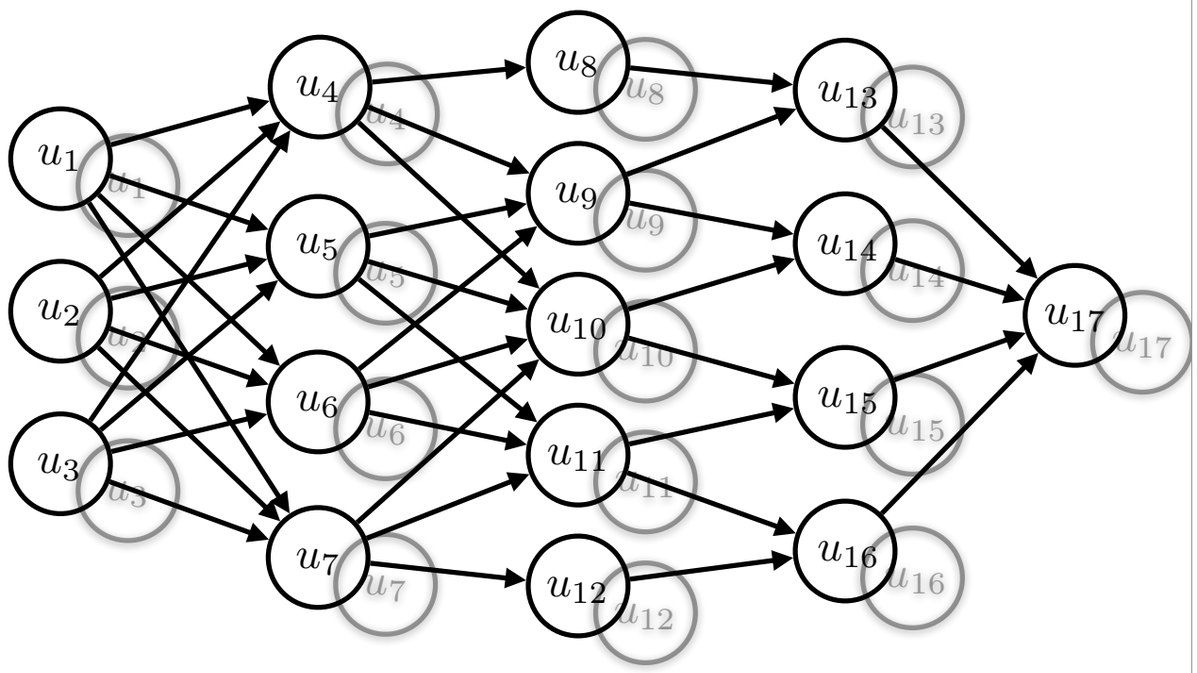
Repeating the same calculation multiple times is wasteful. Also, keeping these calculations in memory (to avoid repeated calculations) could become unmanageable. Keeping all the intermediate Jacobians may require too many memory resources.

The Computational Graph



The original graph can be used to compute the loss given the input. To compute the gradient we need to duplicate each node (and allocate the associated memory).

The Computational Graph



Forward Propagation

The forward propagation can be computed by propagating the inputs to the nodes to the right until the loss node.

Forward Propagation

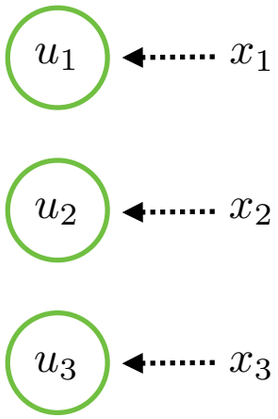
u_1 x_1

u_2 x_2

u_3 x_3

inputs

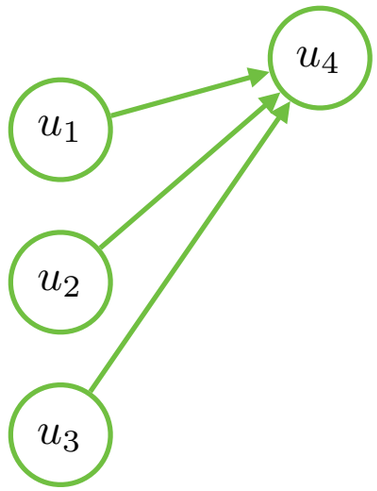
Forward Propagation



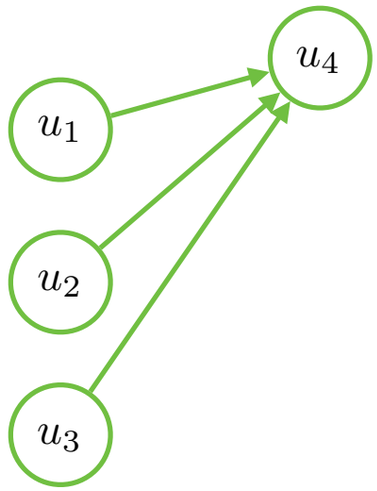
Forward Propagation

The forward propagation can be computed by propagating the inputs to the nodes to the right until the loss node.

Forward Propagation



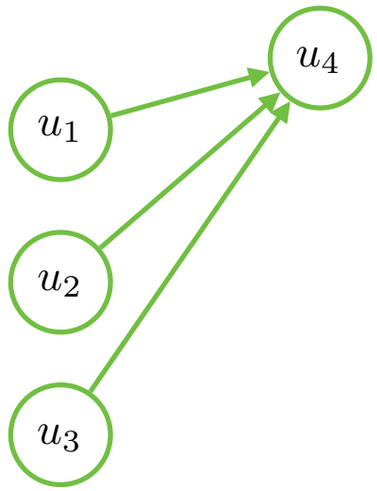
Forward Propagation



$$u_i = f_i\left(\mathbb{A}^i\right)$$

$$\mathbb{A}^i = \{u_j | j \in \text{Parents}(u_i)\}$$

Forward Propagation

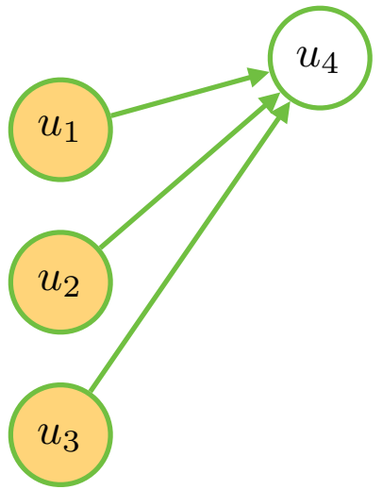


$$u_4 = f_4(u_1, u_2, u_3)$$

$$u_i = f_i\left(\mathbb{A}^i\right)$$

$$\mathbb{A}^i = \{u_j | j \in \text{Parents}(u_i)\}$$

Forward Propagation



$$u_4 = f_4(u_1, u_2, u_3)$$

$$u_i = f_i\left(\mathbb{A}^i\right)$$

$$\mathbb{A}^i = \{u_j | j \in \text{Parents}(u_i)\}$$

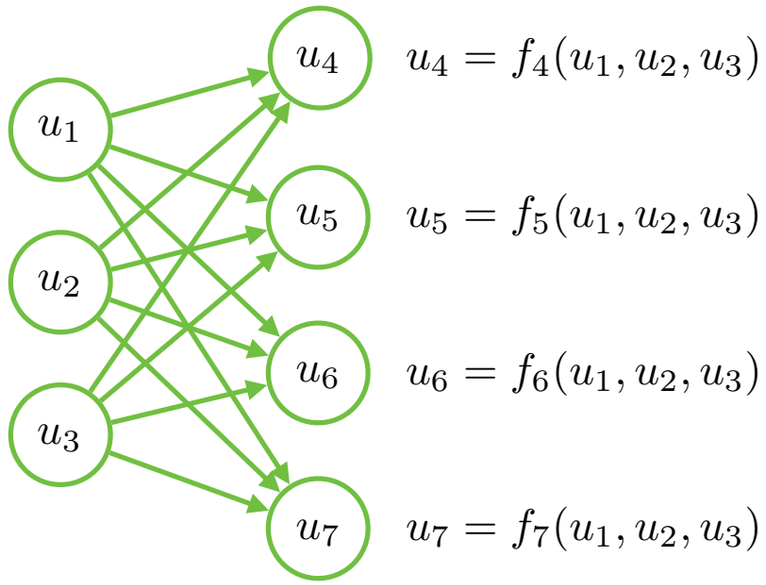
parents of u_4

The parents of a node are the arguments of the function that computes the value of that node.

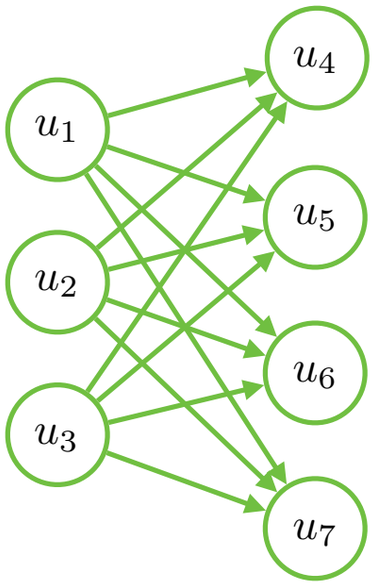
Forward Propagation

The forward propagation can be computed by propagating the inputs to the nodes to the right until the loss node.

Forward Propagation

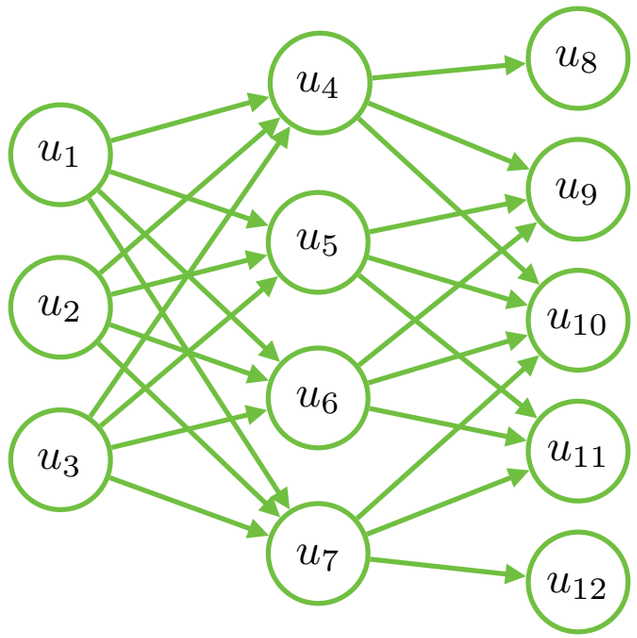


Forward Propagation



The forward propagation can be computed by propagating the inputs to the nodes to the right until the loss node.

Forward Propagation



$$u_8 = f_8(u_4)$$

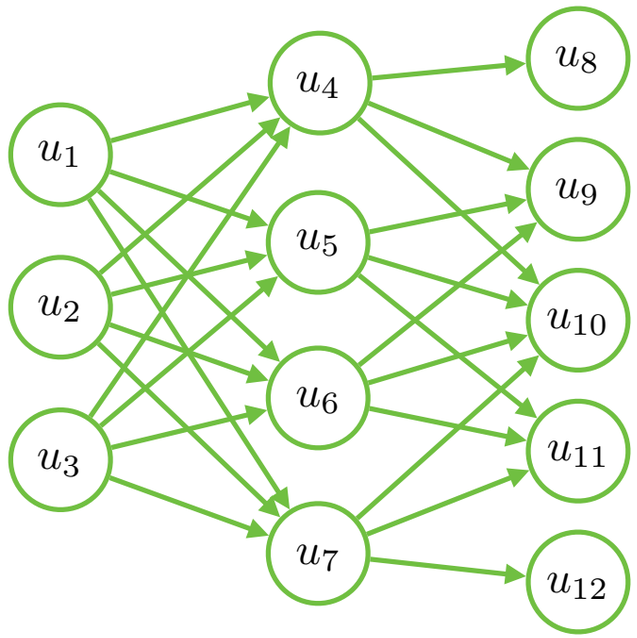
$$u_9 = f_9(u_4, u_5, u_6)$$

$$u_{10} = f_{10}(u_4, u_5, u_6, u_7)$$

$$u_{11} = f_{11}(u_5, u_6, u_7)$$

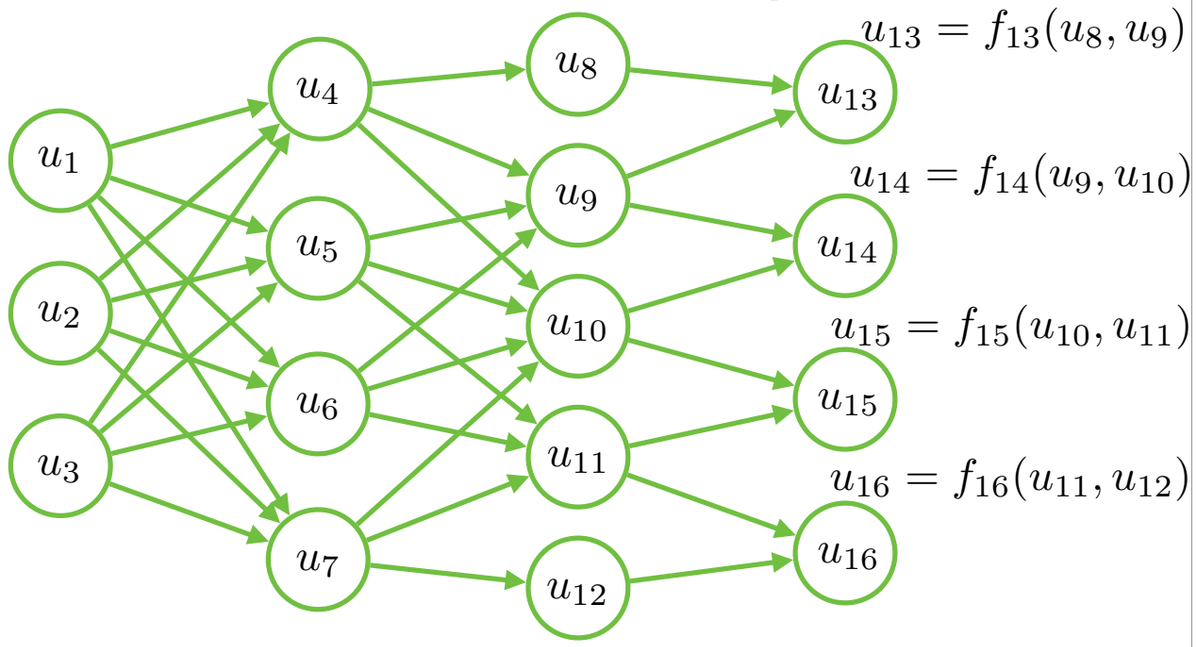
$$u_{12} = f_{12}(u_7)$$

Forward Propagation

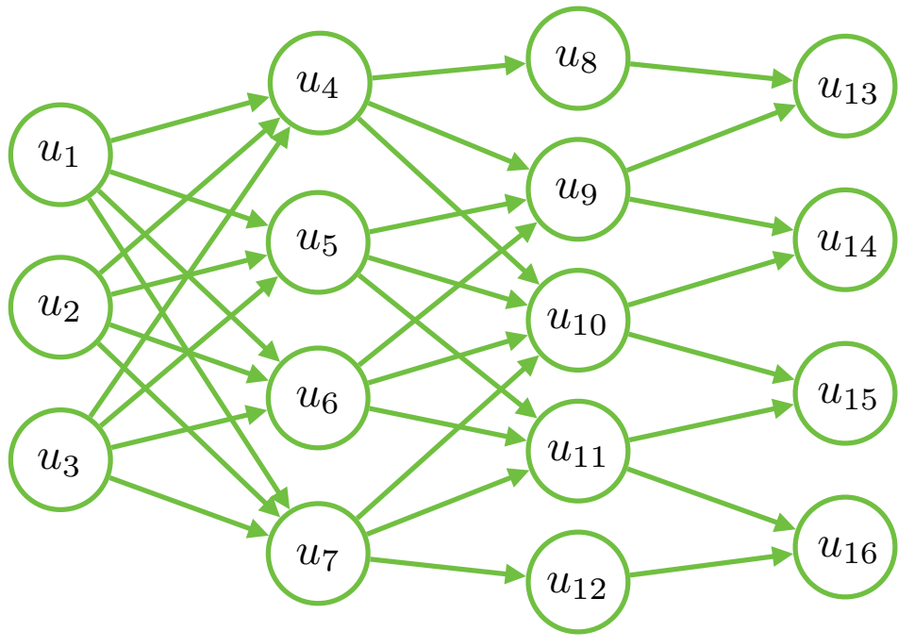


The forward propagation can be computed by propagating the inputs to the nodes to the right until the loss node.

Forward Propagation

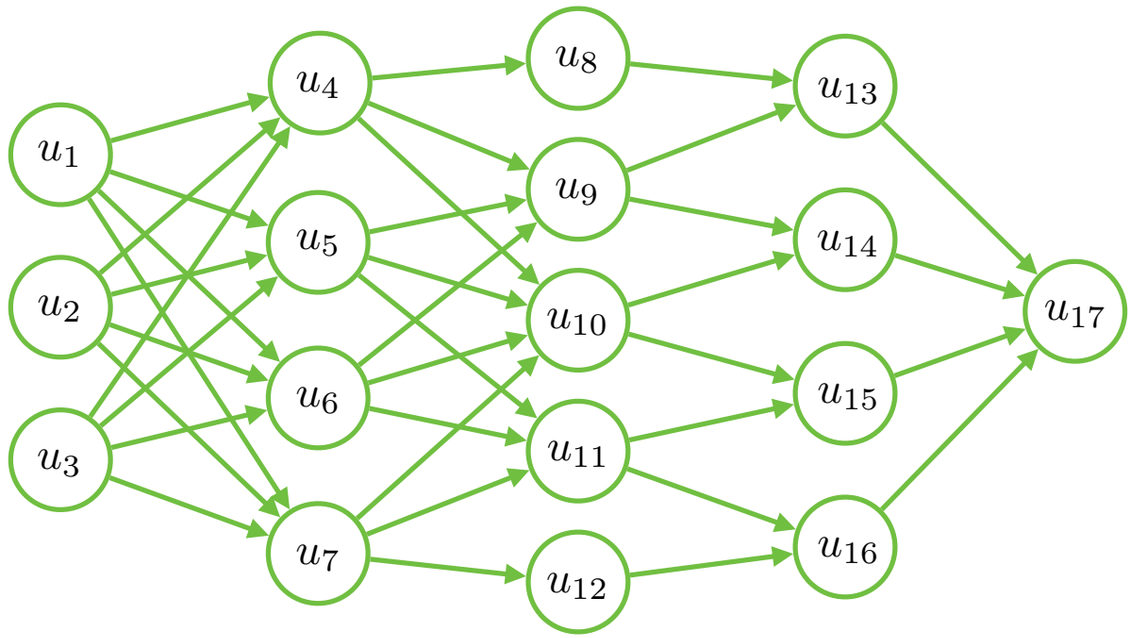


Forward Propagation

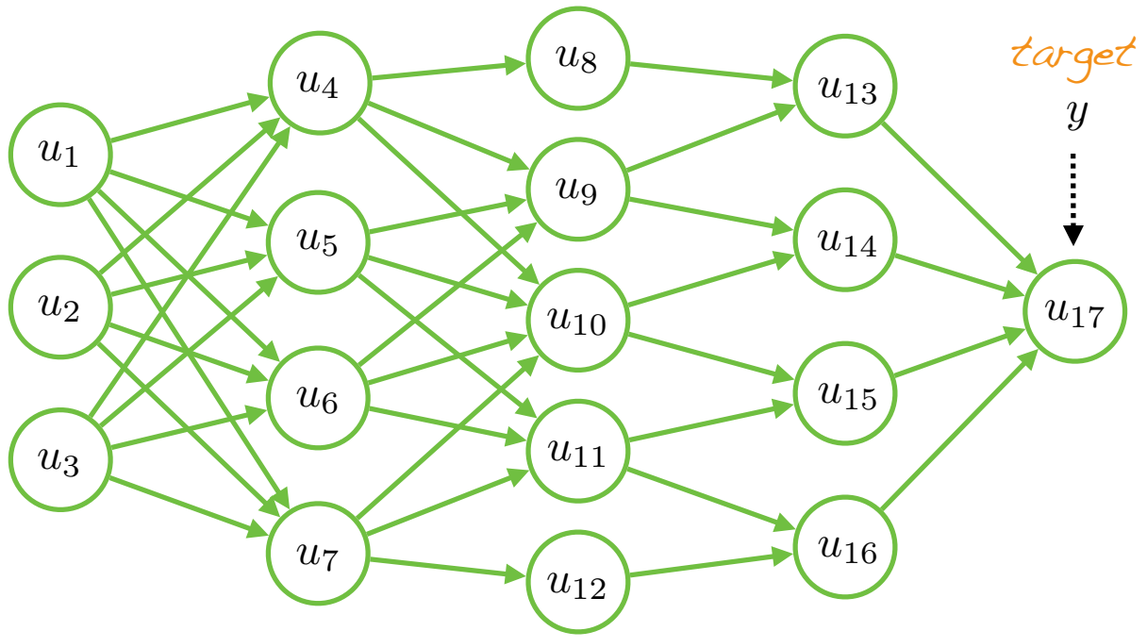


The forward propagation can be computed by propagating the inputs to the nodes to the right until the loss node.

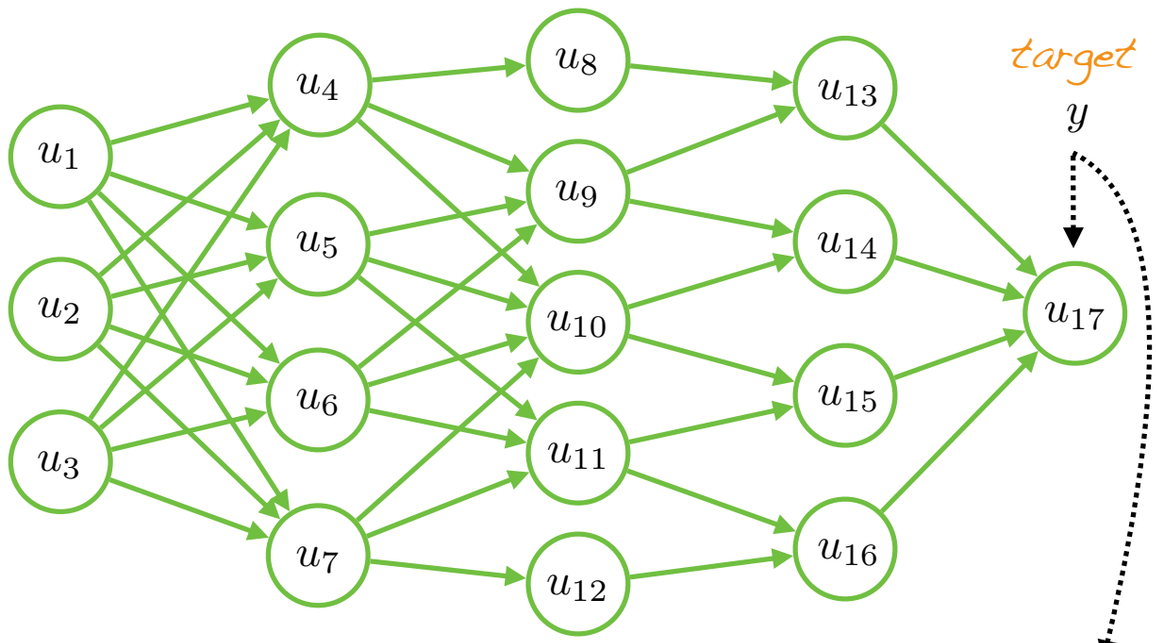
Forward Propagation



Forward Propagation

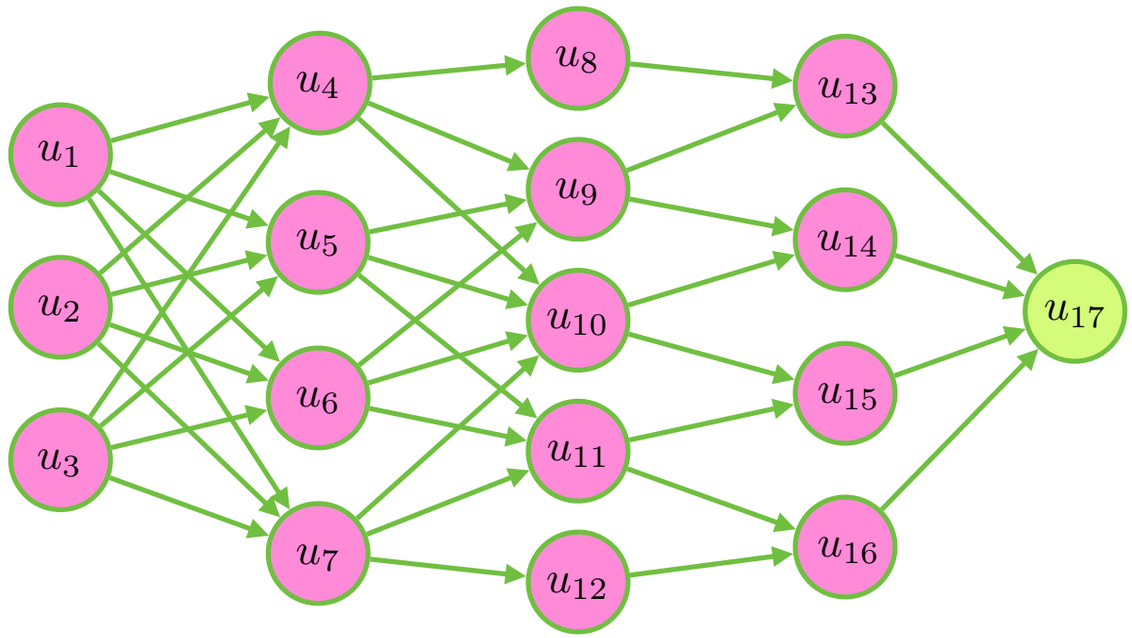


Forward Propagation



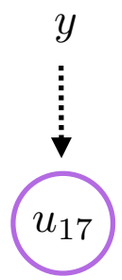
loss function $u_{17} = f_{17}(u_{13}, u_{14}, u_{15}, u_{16}, y)$

Backward Propagation



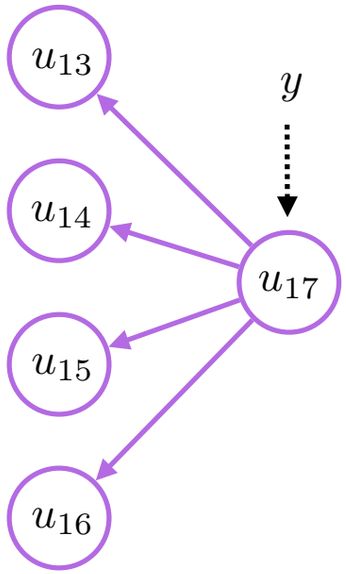
The backward propagation is an efficient way to compute the derivatives of u17 (in green) with respect to all the other variables (in red).

Backward Propagation



The backward propagation can be computed by propagating information from the loss node on the right to the parent nodes on the left up to the input nodes. In this way we avoid the computation of repeated gradients.

Backward Propagation



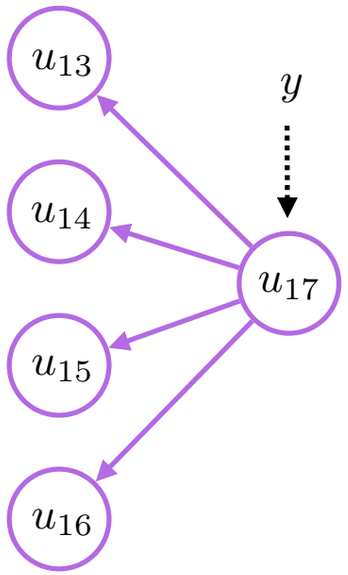
Backward Propagation

$$\frac{\partial u_{17}}{\partial u_{13}} = \frac{\partial f_{17}(u_{13}, u_{14}, u_{15}, u_{16}, y)}{\partial u_{13}}$$

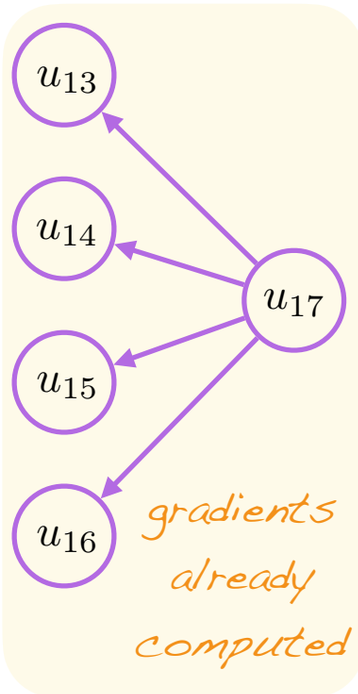
$$\frac{\partial u_{17}}{\partial u_{14}} = \frac{\partial f_{17}(u_{13}, u_{14}, u_{15}, u_{16}, y)}{\partial u_{14}}$$

$$\frac{\partial u_{17}}{\partial u_{15}} = \frac{\partial f_{17}(u_{13}, u_{14}, u_{15}, u_{16}, y)}{\partial u_{15}}$$

$$\frac{\partial u_{17}}{\partial u_{16}} = \frac{\partial f_{17}(u_{13}, u_{14}, u_{15}, u_{16}, y)}{\partial u_{16}}$$

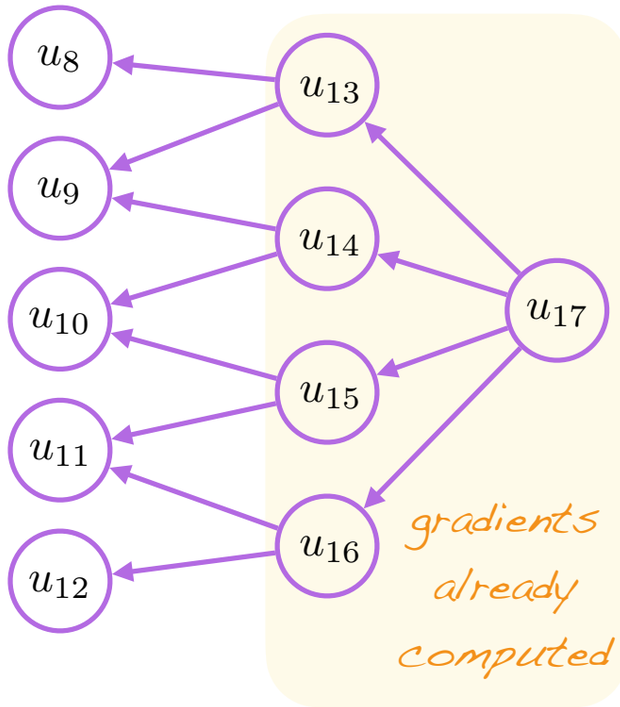


Backward Propagation



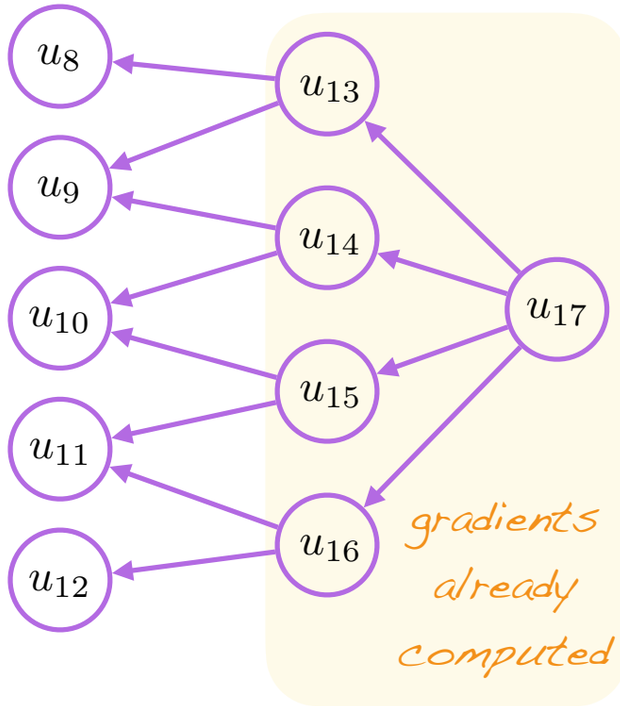
More in general, we can use the chain rule formula and exploit the previous calculations.

Backward Propagation

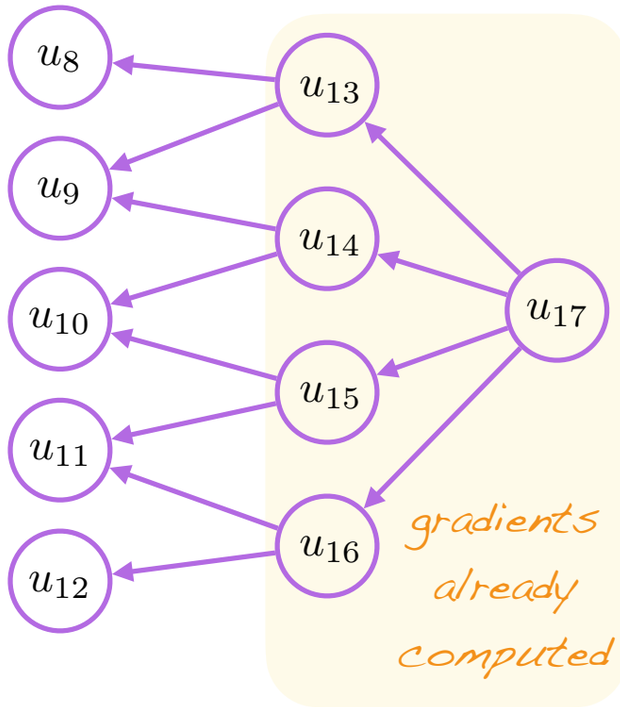


Backward Propagation

$$\frac{\partial u_n}{\partial u_j} = \sum_{i:j \in \mathrm{Pa}(u_i)} \frac{\partial u_n}{\partial u_i} \frac{\partial u_i}{\partial u_j}$$

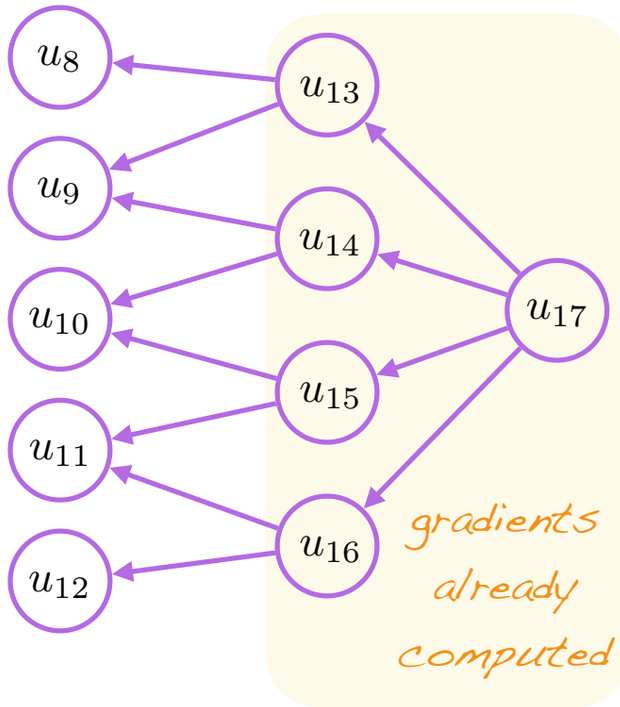


Backward Propagation

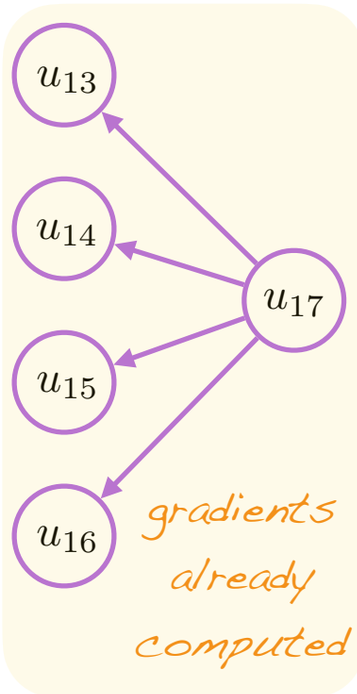


Backward Propagation

$$\frac{\partial u_{17}}{\partial u_9} = \sum_{i:9 \in \mathrm{Pa}(u_i)} \frac{\partial u_{17}}{\partial u_i} \frac{\partial u_i}{\partial u_9}$$

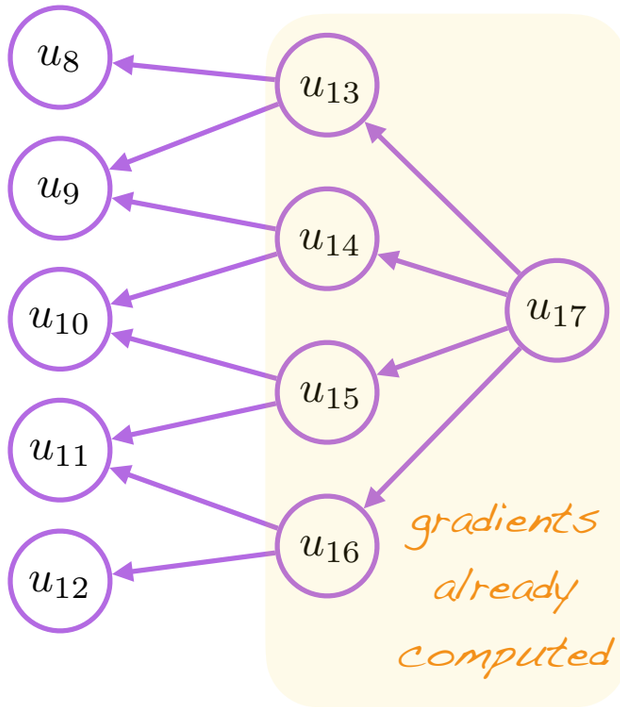


Backward Propagation



These terms (in orange) have already been computed at the previous iteration (right-hand-side).

Backward Propagation



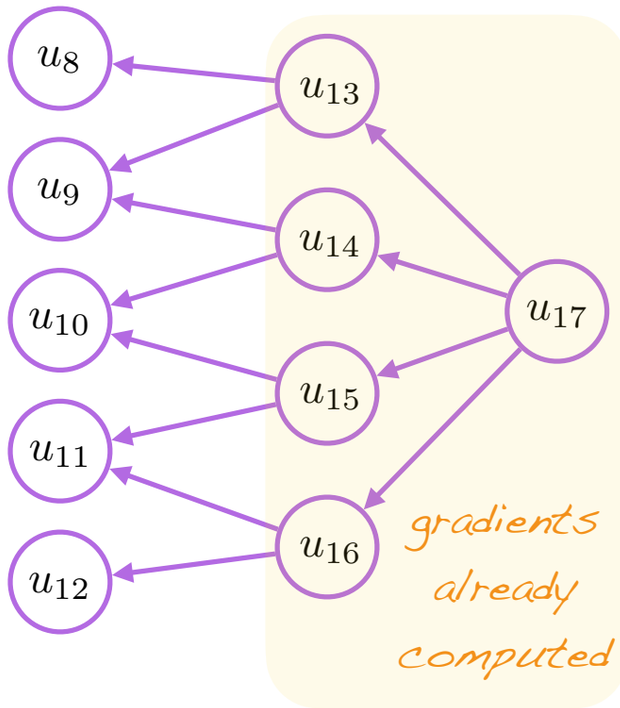
Backward Propagation

$$\frac{\partial u_{17}}{\partial u_8} = \frac{\partial u_{17}}{\partial u_3}\frac{\partial u_{13}}{\partial u_8}$$

$$\frac{\partial u_{17}}{\partial u_9} = \frac{\partial u_{17}}{\partial u_{13}}\frac{\partial u_{13}}{\partial u_9} + \frac{\partial u_{17}}{\partial u_{14}}\frac{\partial u_{14}}{\partial u_9}$$

$$\vdots$$

$$\frac{\partial u_{17}}{\partial u_{12}} = \frac{\partial u_{17}}{\partial u_{16}}\frac{\partial u_{16}}{\partial u_{12}}$$



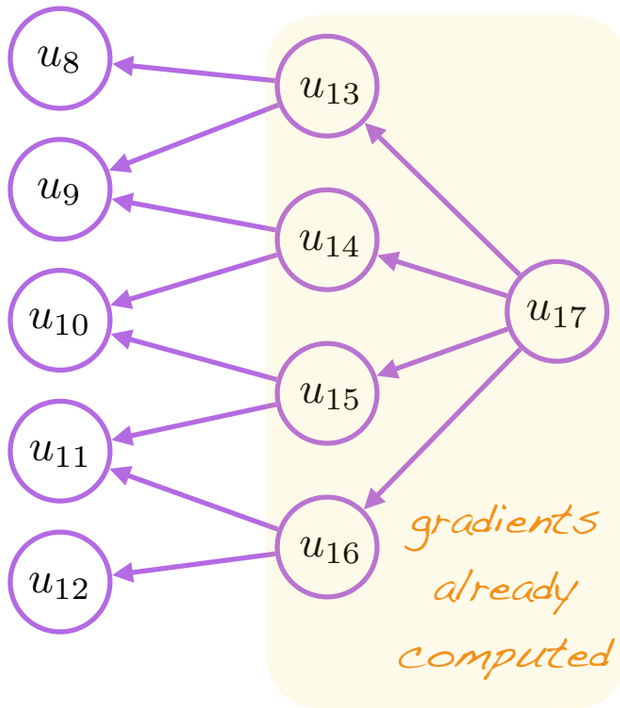
Backward Propagation

$$\frac{\partial u_{17}}{\partial u_8} = \frac{\partial u_{17}}{\partial u_3}\frac{\partial u_{13}}{\partial u_8}$$

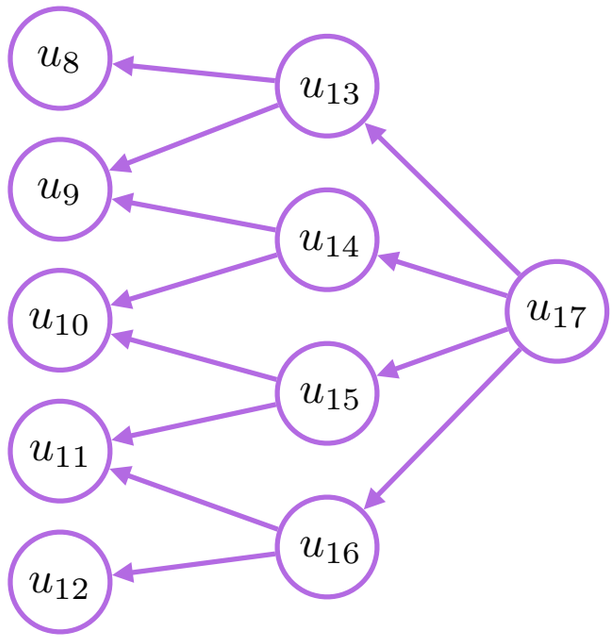
$$\frac{\partial u_{17}}{\partial u_9} = \frac{\partial u_{17}}{\partial u_{13}}\frac{\partial u_{13}}{\partial u_9} + \frac{\partial u_{17}}{\partial u_{14}}\frac{\partial u_{14}}{\partial u_9}$$

$$\vdots$$

$$\frac{\partial u_{17}}{\partial u_{12}} = \frac{\partial u_{17}}{\partial u_{16}}\frac{\partial u_{16}}{\partial u_{12}}$$

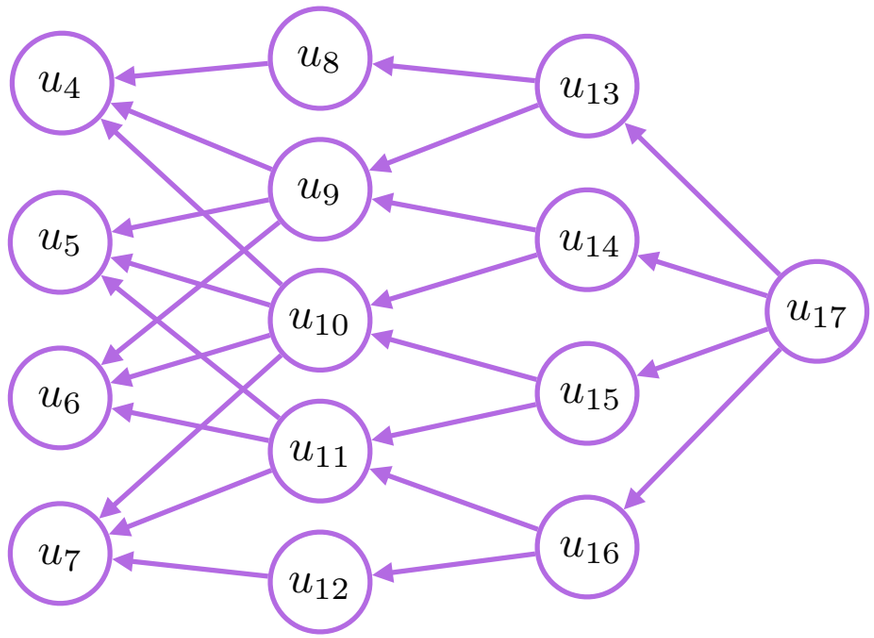


Backward Propagation

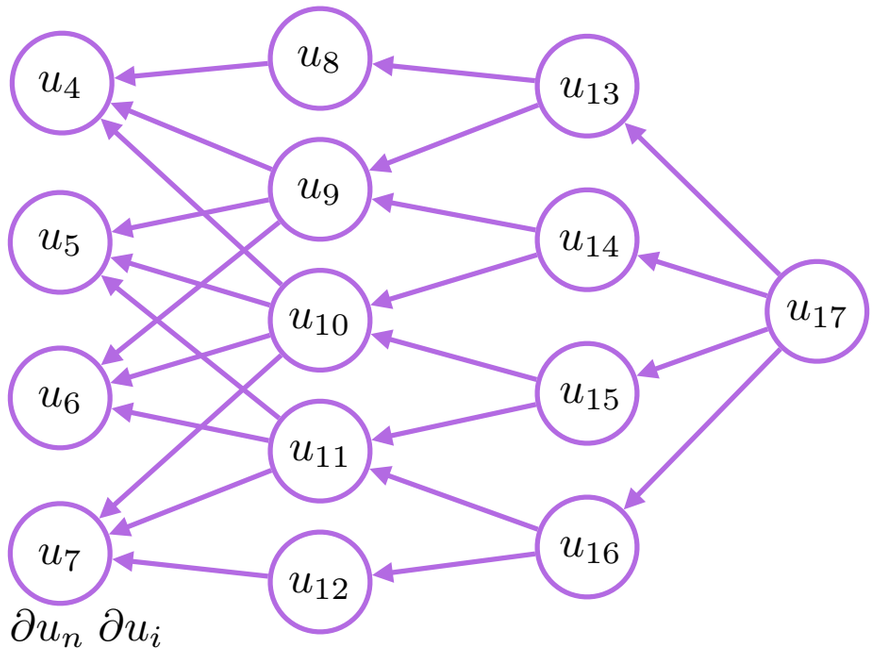


Note

Backward Propagation

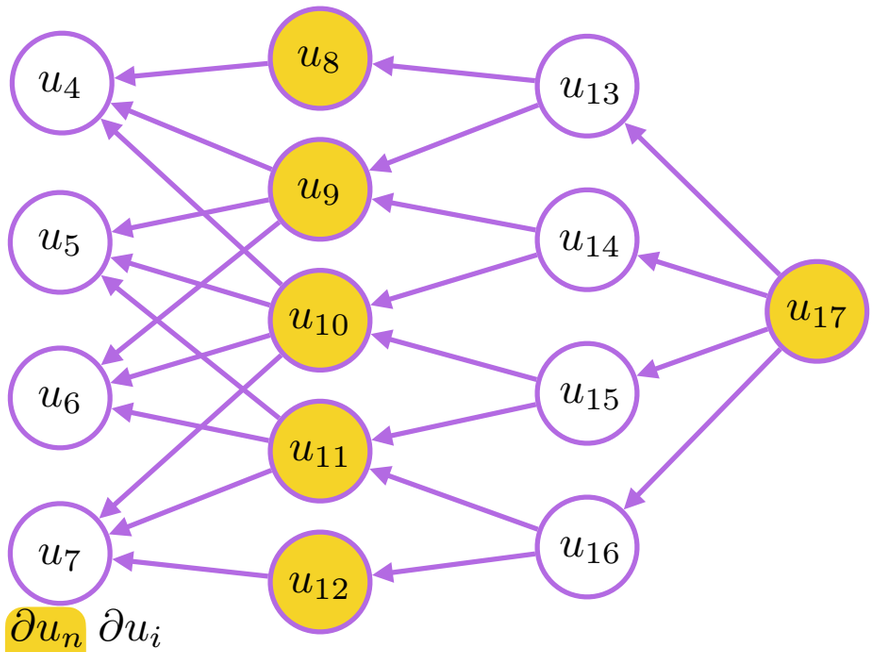


Backward Propagation



$$\frac{\partial u_n}{\partial u_j} = \sum_{i:j \in \mathrm{Pa}(u_i)} \frac{\partial u_n}{\partial u_i} \frac{\partial u_i}{\partial u_j}$$

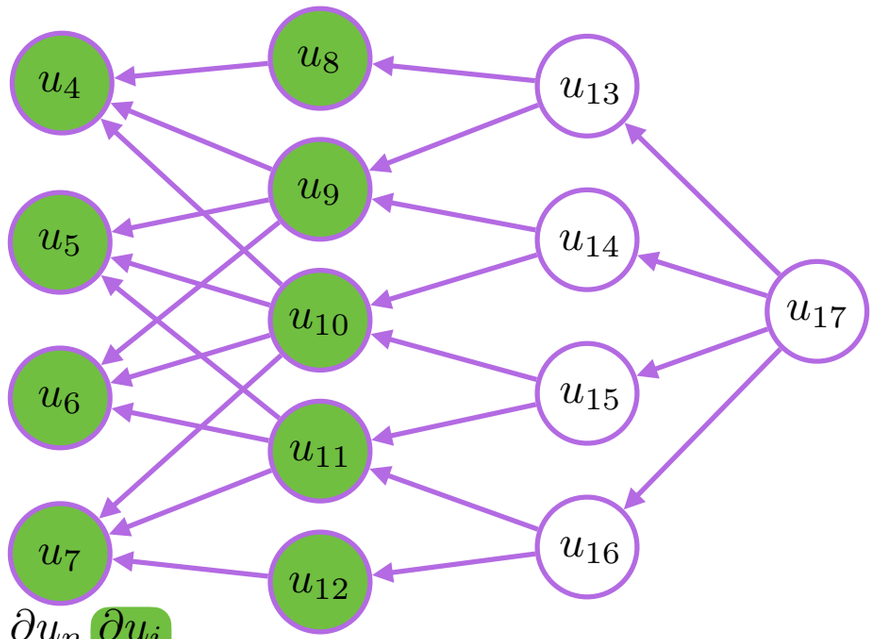
Backward Propagation



$$\frac{\partial u_n}{\partial u_j} = \sum_{i:j \in \mathrm{Pa}(u_i)} \frac{\partial u_n}{\partial u_i} \frac{\partial u_i}{\partial u_j}$$

These are nodes where the gradients have already been computed (and stored) that we can reuse in the chain rule to compute the new gradients.

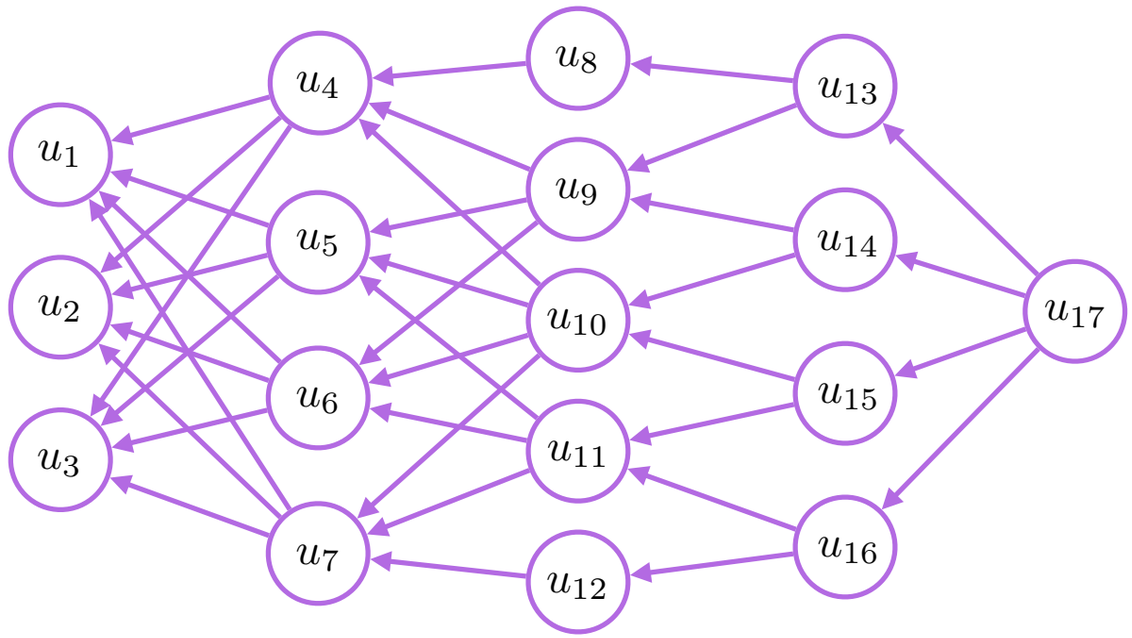
Backward Propagation



$$\frac{\partial u_n}{\partial u_j} = \sum_{i:j \in \mathrm{Pa}(u_i)} \frac{\partial u_n}{\partial u_i} \frac{\partial u_i}{\partial u_j}$$

These are the new gradients between directly connected nodes (these gradients are defined when the function that implements the forward pass from one layer to the one on the immediate right is defined).

Backward Propagation



The Backpropagation algorithm thus defines an efficient method to compute the gradients while sacrificing memory. Derivative computations grow linearly with the number of edges.

Back-Propagation

- The approach seen so far is called **symbol-to-value** differentiation
- This is used by libraries such as Caffe and Torch

Note

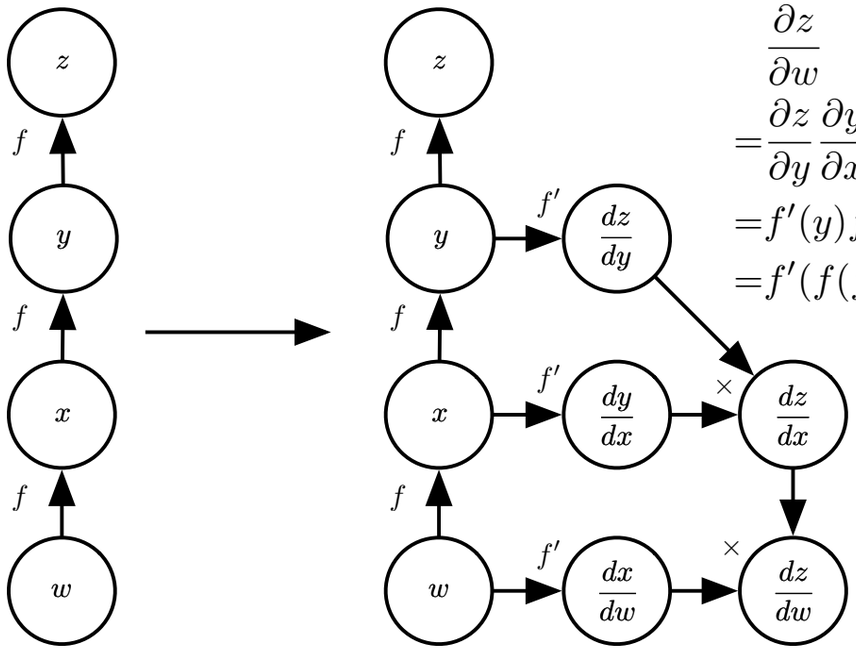
Back-Propagation

- An alternative approach is the **symbol-to-symbol** differentiation

- This is used by libraries such as Theano and TensorFlow

Instead of assigning values to the functional expressions one obtains the functional expression of the derivatives.

Symbol-to-Symbol



$$\frac{\partial z}{\partial w}$$
$$= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w}$$
$$= f'(y)f'(x)f'(w)$$
$$= f'(f(f(w)))f'(f(w))f'(w)$$

The idea is to convert the network into another network with the corresponding derivatives layers.

Symbol-to-Symbol

- Advantages
  - Derivatives are computed as a forward propagation in another graph
  - Higher order derivatives can be easily computed

Higher order derivatives can be computed by building the computational graph of the previous (gradient) computational graph.
However, the dimensionality of higher order derivatives makes this option not very useful. Other practical solutions are possible (e.g. Krylov methods).

Back-Propagation Forms

- The back-propagation algorithm exploits a special case of the chain rule, written in recursive form

$$\frac{\partial u_n}{\partial u_j} = \sum_{i:j \in \mathrm{Pa}(u_i)} \frac{\partial u_n}{\partial u_i} \frac{\partial u_i}{\partial u_j}$$

- In alternative one could use the sequential form

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

The number of paths might grow exponentially with the length of the paths and this would lead to a computational explosion in the non-recursive formula.
The recursive formula can also be seen as a way to compute the gradient via a dynamic programming approach (split original problem in repeated subproblems and solution of original problem is defined as a composition of solutions of subproblems).

Further Issues

- Returning more that one output (1 tensor) might be more efficient
- Memory consumption
- Data types
- Undefined gradients (e.g. L1 norm)

When is more efficient to return more than 1 output? For example, if we need both the maximum and the argument of the maximum of a tensor it is better to implement a single function that can do both at once (rather than as two separate nodes).
Memory might be challenged by the temporary memorization of several intermediate tensors (e.g. Gi in the previous function); one workaround is to keep a separate buffer where the tensors are added as they are computed.
Keeping track of undefined gradients.

Extensions

- Automatic simplification of the derivatives (or the computational graph) — Theano, TensorFlow
- Reverse mode accumulation (what we have seen so far; efficient with a single output)
- Forward mode accumulation (efficient when outputs are more than the inputs)

Theano and TensorFlow use known rules to try and simplify the computational graph.
When there are k outputs it is not efficient to repeat k times the reverse mode accumulation.

Forward vs Reverse Mode

- The computation of the gradients involves the products (and sums) of Jacobians
- The order of these products determines the forward (left to right) or reverse (right to left) mode
- Suppose that there is one output $D \in \mathbb{R}^{m \times 1}$

$$ABCD$$

$\longleftarrow$

reverse more efficient

In this case it is more efficient to multiply from right to left so that the products are always between a matrix and a vector (as they result in a vector).

Forward vs Reverse Mode

- The computation of the gradients involves the products (and sums) of Jacobians
- The order of these products determines the forward (left to right) or reverse (right to left) mode
- With multiple outputs $D \in \mathbb{R}^{m \times n}$, $A \in \mathbb{R}^{p \times q}$ and $p < n$

$$ABCD$$

$\longrightarrow$

forward more efficient

In this case it is more efficient to multiply from left to right so that the products are always between small matrices.