Feedforward Neural Networks

Elements of Machine Learning

Jens Petersen

with illustrations from Deep Learning, by Goodfellow et al.
Assistant Professor, Department of Computer Science
University of Copenhagen

About this lecture

Feedforward Neural Networks

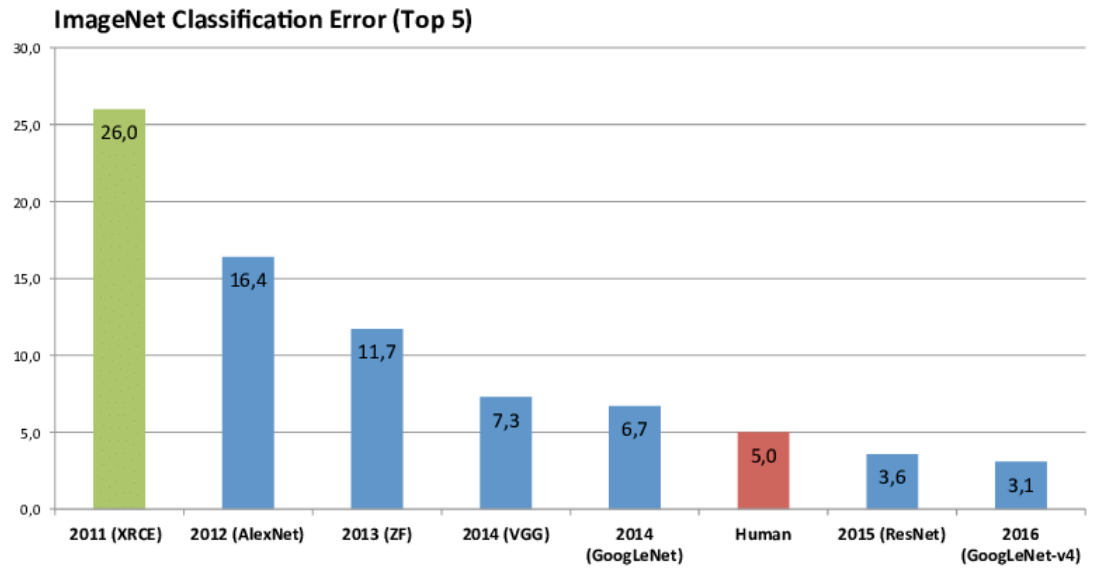
- Introduction to neural networks
- Feedforward as opposed to networks with cycles
- Deep learning

Whats the point?

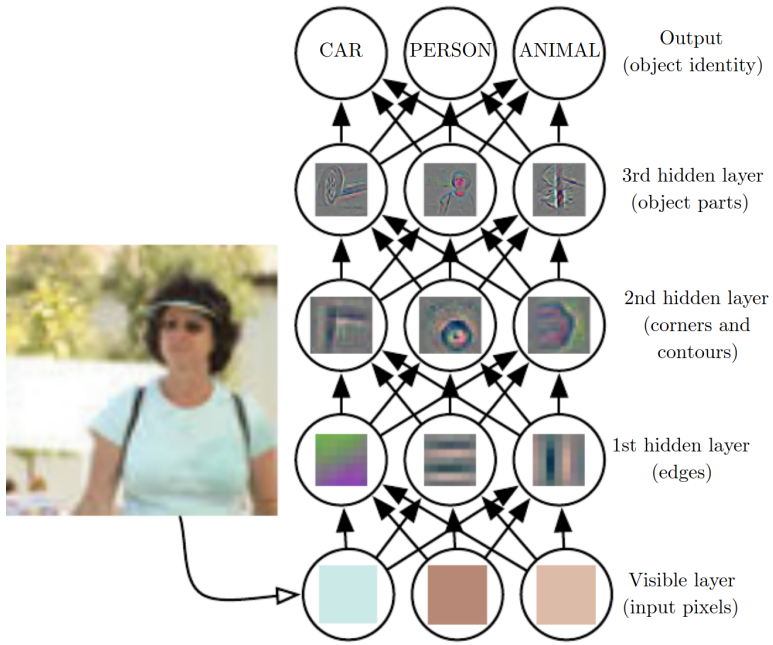
- Deep learning methods are currently changing machine learning and influencing almost every field of research. Changes that are increasingly moving into our daily lifes.
- Neural networks is the underlying machinery behind the recent success and proliferation of deep learning methods
- Feedforward neural networks is the first and simplest neural network type.

Whats the point?

- Deep learning methods are currently changing machine learning and influencing almost every field of research. Changes that are increasingly moving into our daily lifes.
- Neural networks is the underlying machinery behind the recent success and proliferation of deep learning methods
- Feedforward neural networks is the first and simplest neural network type.



Feedforward neural networks



Feedforward neural network example predicting image content

- Feedforward, because information moves in one direction, from input to output (no cycles)
- Also called **multilayer perceptrons** (MLPs)

Feedforward neural networks

The goals of a feedforward neural network is to approximate some function f^*, mapping input $\mathbf{x}$ to output y, $y = f^*(\mathbf{x})$.

$$\mathbf{y} = f(\mathbf{x}, \theta)$$

The parameters θ are ideally chosen such that the function f best approximates f^*.

Feedforward neural networks

The goals of a feedforward neural network is to approximate some function f^*, mapping input $\mathbf{x}$ to output y, $y = f^*(\mathbf{x})$.

$$\mathbf{y} = f(\mathbf{x}, \theta)$$

The parameters θ are ideally chosen such that the function f best approximates f^*.

They are called networks because they are composed of many different functions, such as[1]

$$f(\mathbf{x}) = f^{(3)}\left(f^{(2)}\left(f^{(1)}(\mathbf{x})\right)\right)$$

[1] The parameters θ have been left out for clarity

Feedforward neural networks

The goals of a feedforward neural network is to approximate some function f^*, mapping input $\mathbf{x}$ to output y, $y = f^*(\mathbf{x})$.

$$\mathbf{y} = f(\mathbf{x}, \theta)$$

The parameters θ are ideally chosen such that the function f best approximates f^*.

They are called networks because they are composed of many different functions, such as[1]

$$f(\mathbf{x}) = f^{(3)}\Big(f^{(2)}\big(f^{(1)}(\mathbf{x})\big)\Big)$$

- $f^{(1)}$ is called the **first layer**, $f^{(2)}$ is called the **second layer**, etc.
- The first layer, $f^{(1)}$ in this case, is called the **input layer**, and the final layer, $f^{(3)}$ in this case, is called the **output layer**
- Intermediate layers, $f^{(2)}$ in this case, are called **hidden layers**.

[1] The parameters θ have been left out for clarity

Feedforward neural networks

The goals of a feedforward neural network is to approximate some function f^*, mapping input $\mathbf{x}$ to output y, $y = f^*(\mathbf{x})$.

$$\mathbf{y} = f(\mathbf{x}, \theta)$$

The parameters θ are ideally chosen such that the function f best approximates f^*.

They are called networks because they are composed of many different functions, such as[1]

$$f(\mathbf{x}) = f^{(3)}\Big(f^{(2)}\big(f^{(1)}(\mathbf{x})\big)\Big)$$

- The overall length of the chain is called the **depth**.
- The name deep learning arose from this terminology. That is, it simply refers to many layers of function compositions.
- The dimensionality of the hidden layers is the network **width**

[1] The parameters θ have been left out for clarity

Feedforward neural networks

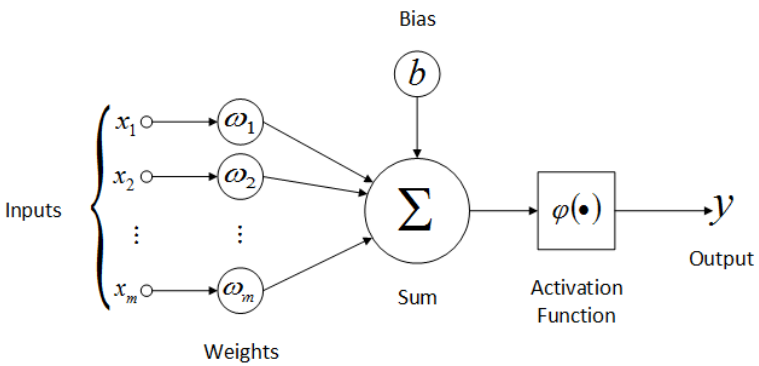
Training

- involves changing θ, such that $f(\mathbf{x}, \theta)$ match $f^*(\mathbf{x})$
- using samples of $\mathbf{x}$ and $\mathbf{y} = f^*(\mathbf{x})$

Note we do not specify the behaviour of the input or hidden layers, only what the output layer should do.The training algorithm will have to learn the behaviour of these input and hidden layers on its own.

Artificial Neurons[1]

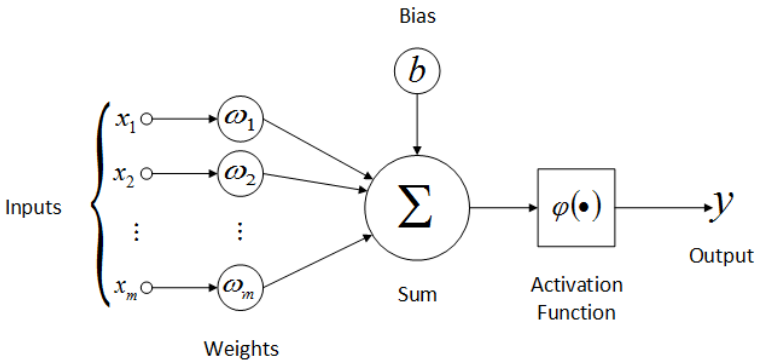
- Note: the previous example was a chain, but feedforward neural networks in general take the form of a directed acyclic graph.
- A layer receives input from many other layers and returns a multidimensional output. Each of the dimensions in this output is formed by the action of a neuron.



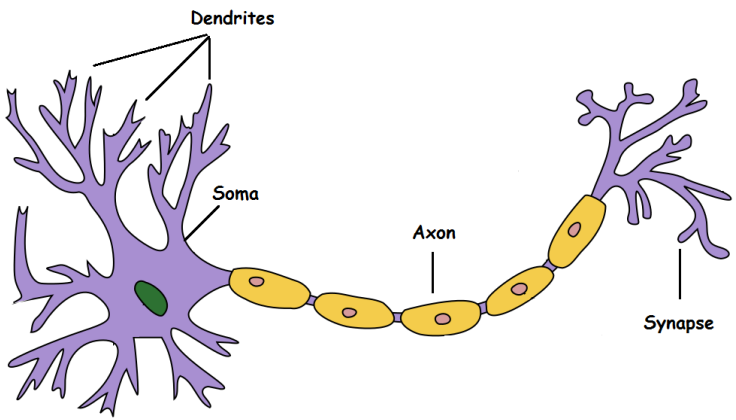
Artificial neuron, vector-valued input is multiplied by a weight-vector, summed including **a bias** and passed through an **activation function** to the scalar-output.

[1] Illustrations from `https://cinnamon.agency/blog/post/artificial_neural_networks`

Artificial Neurons[1]



Artificial neuron, vector-valued input is multiplied by a weight-vector, summed including **a bias** and passed through an **activation function** to the scalar-output.



Biological neuron, basis of the brains communication and information processing network and the loose inspiration for the artificial neuron.

[1] Illustrations from `https://cinnamon.agency/blog/post/artificial_neural_networks`

Feedforward neural networks

To construct a neural network, we need to choose

- the overall directed acyclic graph, the **architecture** that defines how input flows through hidden layers to the output, this includes
  - the number of layers
  - how layers are connected
  - the width of each layer (the number of artificial neural network units in each layer)
  - the activation functions to use
  - etc.

Feedforward neural networks

To construct a neural network, we need to choose

- the overall directed acyclic graph, the **architecture** that defines how input flows through hidden layers to the output, this includes
  - the number of layers
  - how layers are connected
  - the width of each layer (the number of artificial neural network units in each layer)
  - the activation functions to use
  - etc.

All of these choices are hyper parameters of the feedforward neural network model because they affect model complexity. So we cannot simply optimize them on the training set.

Feedforward neural networks

To construct a neural network, we need to choose

- the overall directed acyclic graph, the **architecture** that defines how input flows through hidden layers to the output, this includes
  - the number of layers
  - how layers are connected
  - the width of each layer (the number of artificial neural network units in each layer)
  - the activation functions to use
  - etc.

All of these choices are hyper parameters of the feedforward neural network model because they affect model complexity. So we cannot simply optimize them on the training set.

What we do optimize, is the weights and biases of each neuron, we do this through gradient descent using a procedure called **back-propagation** for computing gradients.

Feedforward neural networks - example

Learning XOR, $x, y \in \{0, 1\}$

$$f^*(x_0, x_1) = \begin{cases} 0 & \text{if } x_0 = x_1 \\ 1 & \text{else} \end{cases}$$

Training set $\mathbf{X} = \{(0, 0), (0, 1), (1, 0), (1, 1)\}$

Feedforward neural networks - example

Learning XOR, $x, y \in \{0, 1\}$

$$f^*(x_0, x_1) = \begin{cases} 0 & \text{if } x_0 = x_1 \\ 1 & \text{else} \end{cases}$$

Training set $\mathbf{X} = \{(0,0), (0,1), (1,0), (1,1)\}$ Using a mean squared error (MSE) loss

$$L(\theta) = \frac{1}{4} \sum_{\mathbf{x} \in \mathbf{X}} \left(f^*(\mathbf{x}) - f(\mathbf{x}; \theta) \right)^2$$

Feedforward neural networks - example

We choose a linear model

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

This can be done in closed form, to obtain $\mathbf{w} = \mathbf{0}$ and $b = \frac{1}{2}$

Feedforward neural networks - example

We choose a linear model

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

This can be done in closed form, to obtain $\mathbf{w} = \mathbf{0}$ and $b = \frac{1}{2}$ So a linear model, perhaps not surprisingly cannot represent the XOR function well.

Feedforward neural networks - example

We could use a two layer network, but another linear layer will not help:

$$f^{(2)}\left(f^{(1)}(\mathbf{x})\right) = w(\mathbf{u}^T\mathbf{x} + b^{(1)}) + b^{(2)} \quad (1)$$

$$= w\mathbf{u}^T\mathbf{x} + wb^{(1)} + b^{(2)} \quad (2)$$

Feedforward neural networks - example

We could use a two layer network, but another linear layer will not help:

$$f^{(2)}\left(f^{(1)}(\mathbf{x})\right) = w(\mathbf{u}^T\mathbf{x} + b^{(1)}) + b^{(2)} \quad (1)$$
$$= w\mathbf{u}^T\mathbf{x} + wb^{(1)} + b^{(2)} \quad (2)$$
$$= \mathbf{v}^T\mathbf{x} + c \quad (3)$$

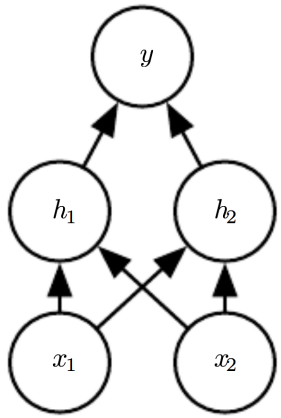
- So stacking multiple linear layers, does not increase model complexity, it is the same as a single linear layer.
- Note in the above, the output of the first layer is a scalar. We could have chosen a different width. A width of two would keep the input dimensions until layer two. In that case we would need the weights of $f^{(1)}$ to be a 2×2 matrix and the bias $b^{(1)}$ a vector of length two.

Feedforward neural networks - example

If we want multiple layers, we should use non-linear activation functions

$$\mathbf{h_i} = g(\mathbf{w_i}^T \mathbf{x} + c)$$

where g is a non-linear activation function.

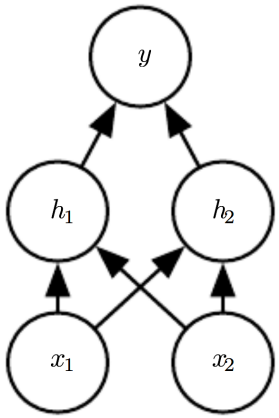


Feedforward neural networks - example

If we want multiple layers, we should use non-linear activation functions

$$\mathbf{h_i} = g(\mathbf{w_i}^T\mathbf{x} + c)$$

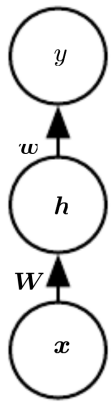
where g is a non-linear activation function.



Note we could have written (like the DL book)

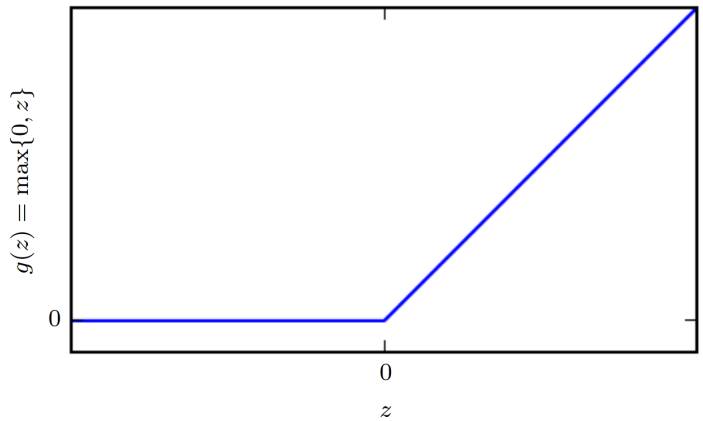
$$\mathbf{h} = g(\mathbf{x}^T\mathbf{W} + \mathbf{c})$$

to indicate that width might be larger than one. Here $\mathbf{W}$ is a matrix and $\mathbf{c}$ is a vector with length corresponding to the width of the layer.



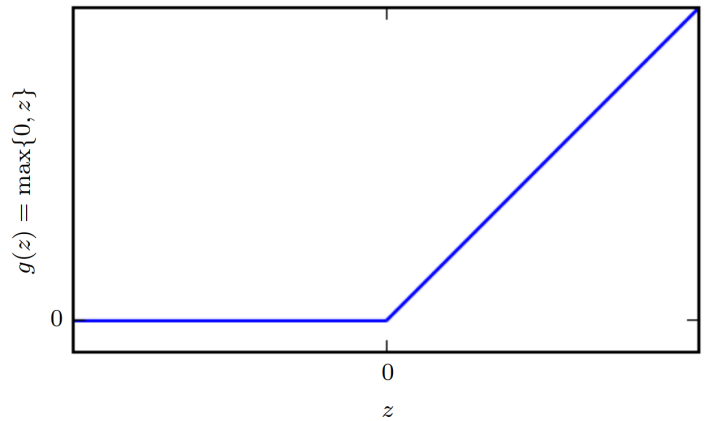
Feedforward neural networks - example

Today, the most often used activation function is the **rectified linear unit**, or ReLU, given by $g(z) = \max\{0, z\}$.



Feedforward neural networks - example

Today, the most often used activation function is the **rectified linear unit**, or ReLU, given by $g(z) = \max\{0, z\}$.



- Close to linear, which makes it easy to optimize
- More on activation function later

Feedforward neural networks - example

Lets write up a two layer model with a ReLU.

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

The solution to the XOR problem is

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

Feedforward neural networks - example

Lets write up a two layer model with a ReLU.

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

The solution to the XOR problem is

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

So given a full design matrix $\mathbf{X}$, we multiply with the weights $\mathbf{W}$

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

Feedforward neural networks - example

Lets write up a two layer model with a ReLU.

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

The solution to the XOR problem is

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

add the bias $\mathbf{c}$

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

Feedforward neural networks - example

Lets write up a two layer model with a ReLU.

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

The solution to the XOR problem is

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

run it through the ReLU

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

Feedforward neural networks - example

Lets write up a two layer model with a ReLU.

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

The solution to the XOR problem is

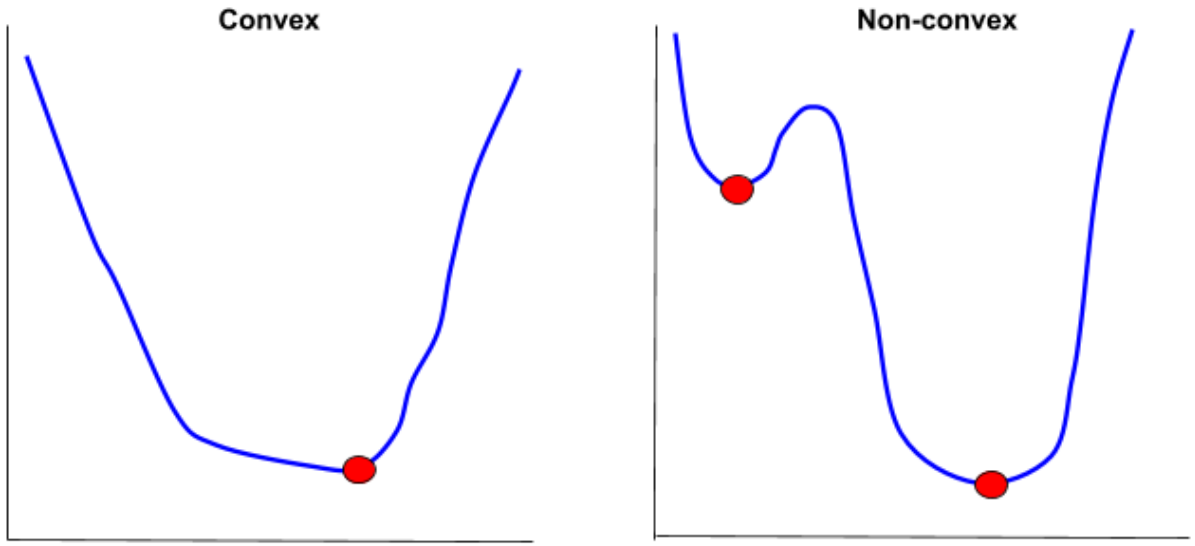
$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, \text{ and } b = 0$$

and multiply by the weight $\mathbf{w}$

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

Gradient-based learning[1]

- minimizing average loss with gradient descent
- non-linearity leads to non-convex optimization problems.
  - we thus cannot in general find the globally optimal solution
  - methods are sensitive to initial parameters and initialization strategies are active research areas (we will cover this later in the course)



[1] Figure from https://automaticaddison.com/how-to-choose-an-optimal-learning-rate-for-gradient-descent/

Loss functions and conditional distributions

Often we are interested in maximizing likelihood, or equivalently minimize the expected negative log-likehood of observing a sample $\mathbf{x}$, $\mathbf{y}$ drawn from our data distribution p_{data} under the model $p_{\text{model}}(\mathbf{y}|\mathbf{x})$.

$$J(\theta) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim p_{\text{data}}} \log p_{\text{model}}(\mathbf{y}|\mathbf{x})$$

Here θ is the parameters of the model.

Loss functions and conditional distributions

Often we are interested in maximizing likelihood, or equivalently minimize the expected negative log-likehood of observing a sample $\mathbf{x}$, $\mathbf{y}$ drawn from our data distribution p_{data} under the model $p_{\text{model}}(\mathbf{y}|\mathbf{x})$.

$$J(\theta) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim p_{\text{data}}} \log p_{\text{model}}(\mathbf{y}|\mathbf{x})$$

Here θ is the parameters of the model.

For classification, where the output can be described as probabilities of each class, that is, a vector of scalar values in $[0, 1]$ that sum to 1, optimizing maximum likelihood is the same as minimizing cross-entropy

$$L(\mathbf{x}, \mathbf{y}) = -\mathbf{y} \cdot \log\left(f(\mathbf{x}; \theta)\right)$$

leading to

$$J(\theta) = \frac{1}{2}\mathbb{E}_{\mathbf{x},\mathbf{y}\sim p_{\text{data}}} L(\mathbf{x}, \mathbf{y})$$

Loss functions and conditional distributions

Often we are interested in maximizing likelihood, or equivalently minimize the expected negative log-likehood of observing a sample $\mathbf{x}$, $\mathbf{y}$ drawn from our data distribution p_{data} under the model $p_{\text{model}}(\mathbf{y}|\mathbf{x})$.

$$J(\theta) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim p_{\text{data}}} \log p_{\text{model}}(\mathbf{y}|\mathbf{x})$$

Here θ is the parameters of the model.

For regression, if we assume the probability of observing $\mathbf{y}$ given $\mathbf{x}$ under our model, that is, $p_{\text{model}}(\mathbf{y}|\mathbf{x})$ is $\mathcal{N}(\mathbf{y}; f(\mathbf{x};\theta), \mathbf{I})$, optimizing maximum likelihood is the same as minimizing mean squared error (MSE)

$$L(\mathbf{x},\mathbf{y}) = \frac{1}{2}||\mathbf{y} - f(\mathbf{x};\theta)||^2$$

Loss functions and conditional statistics

Often we are not interested in probabilities, but conditional statistics of $\mathbf{y}$ given $\mathbf{x}$.

Loss functions and conditional statistics

Often we are not interested in probabilities, but conditional statistics of $\mathbf{y}$ given $\mathbf{x}$.

Instead of $p(\mathbf{y}|\mathbf{x};\theta)$ we may just want to estimate the mean $\mathbf{y}$ given a sample $\mathbf{x}$.

One can show using calculus of variations that minimizing MSE on an infinite large sample will lead to a model that predicts the mean of $\mathbf{y}$ for each value of $\mathbf{x}$.

Loss functions and conditional statistics

Often we are not interested in probabilities, but conditional statistics of $\mathbf{y}$ given $\mathbf{x}$.

Instead of $p(\mathbf{y}|\mathbf{x};\theta)$ we may just want to estimate the mean $\mathbf{y}$ given a sample $\mathbf{x}$.

One can show using calculus of variations that minimizing MSE on an infinite large sample will lead to a model that predicts the mean of $\mathbf{y}$ for each value of $\mathbf{x}$.

If we on the other hand minimize mean absolute error

$$L(\mathbf{x},\mathbf{y}) = ||\mathbf{y} - f(\mathbf{x};\theta)||_1$$

One can show that this leads to a model that predicts the median $\mathbf{y}$ for each value of $\mathbf{x}$.

Output units

The role of the output layer is to transform features to complete the task that the network must perform.

In regression linear output units are often used

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

Output units

The role of the output layer is to transform features to complete the task that the network must perform.

In regression linear output units are often used

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

Linear units do not saturate (become flat) which is good for gradient based optimization. They can also map to all of $\mathbb{R}$.

Output units

In classification of binary variables we may want the output to be the probability of each value, that is, it should be constrained to $[0, 1]$

Output units

In classification of binary variables we may want the output to be the probability of each value, that is, it should be constrained to $[0, 1]$

This could be accomplished with a thresholded linear unit

$$P(y = 1|\mathbf{x}) = \max\left\{0, \min\{1, \mathbf{w}^T\mathbf{h} + b\}\right\}$$

However, if $\mathbf{w}^T\mathbf{h} + b$ ended up outside the unit interval, then the gradient would be $\mathbf{0}$, which would stop the optimization process.

Output units

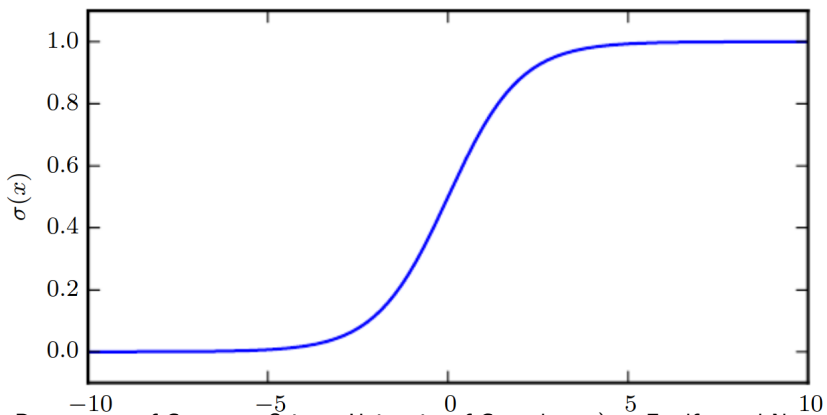
In classification of binary variables we may want the output to be the probability of each value, that is, it should be constrained to $[0, 1]$

Instead one can use a sigmoid output unit

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

where σ is the logistic sigmoid function

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



Output units

If the output is a discrete variable with n possible values. We may want an output that gives us the probability of each value.

That is we want the output to be a vector of n values between 0 and 1 that all sum to one. The softmax function can be used to do this.

A linear layer predicts unnormalized log probabilities

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

The softmax function normalizes the output of the linear layer

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

Hidden units

We mentioned ReLUs. The ReLU is useful because the derivatives are large and consistent across half its domain.

- Drawback is that in the other half of the domain the gradient is $\mathbf{0}$
- Initialization is important because we do not want to end up with a zero gradient

Other units have been created to improve on this

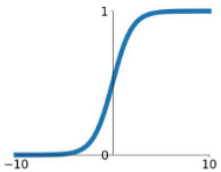
- absolute value rectification $g(z) = |z|$
- leaky ReLU uses a small constant gradient of α for $z < 0$.
- parametric ReLU or PReLU learns this gradient

Hidden units

Prior to ReLUs most neural networks used logistic sigmoid and hyperbolic tangent activation functions

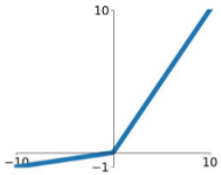
Sigmoid

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



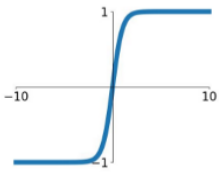
Leaky ReLU

$\max(0.1x, x)$



tanh

$\tanh(x)$

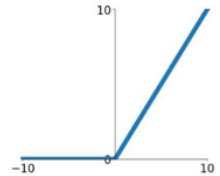


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

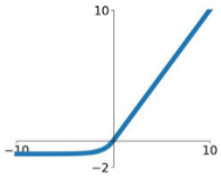
ReLU

$\max(0, x)$



ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



[1]

however, both saturate, which can make training deep neural networks hard.

[1] Figure from `https://towardsdatascience.com/complete-guide-of-activation-functions-34076e95d044`

Architecture Design

The word **architecture** refers to the overall structure of the network

- Number of units
- How units should be connected

Units are typically arranged in layers, with output of one layer going into the next. First layer

$$\mathbf{h}^{(1)} = g^{(1)}(\mathbf{W}^{(1)T}\mathbf{x} + \mathbf{b}^{(1)})$$

and second layer:

$$\mathbf{h}^{(2)} = g^{(2)}(\mathbf{W}^{(2)T}\mathbf{h}^{(1)} + \mathbf{b}^{(2)})$$

etc. Architecture decisions might then be

- Depth of the network
- Width of a layer

Universal approximation properties and depth

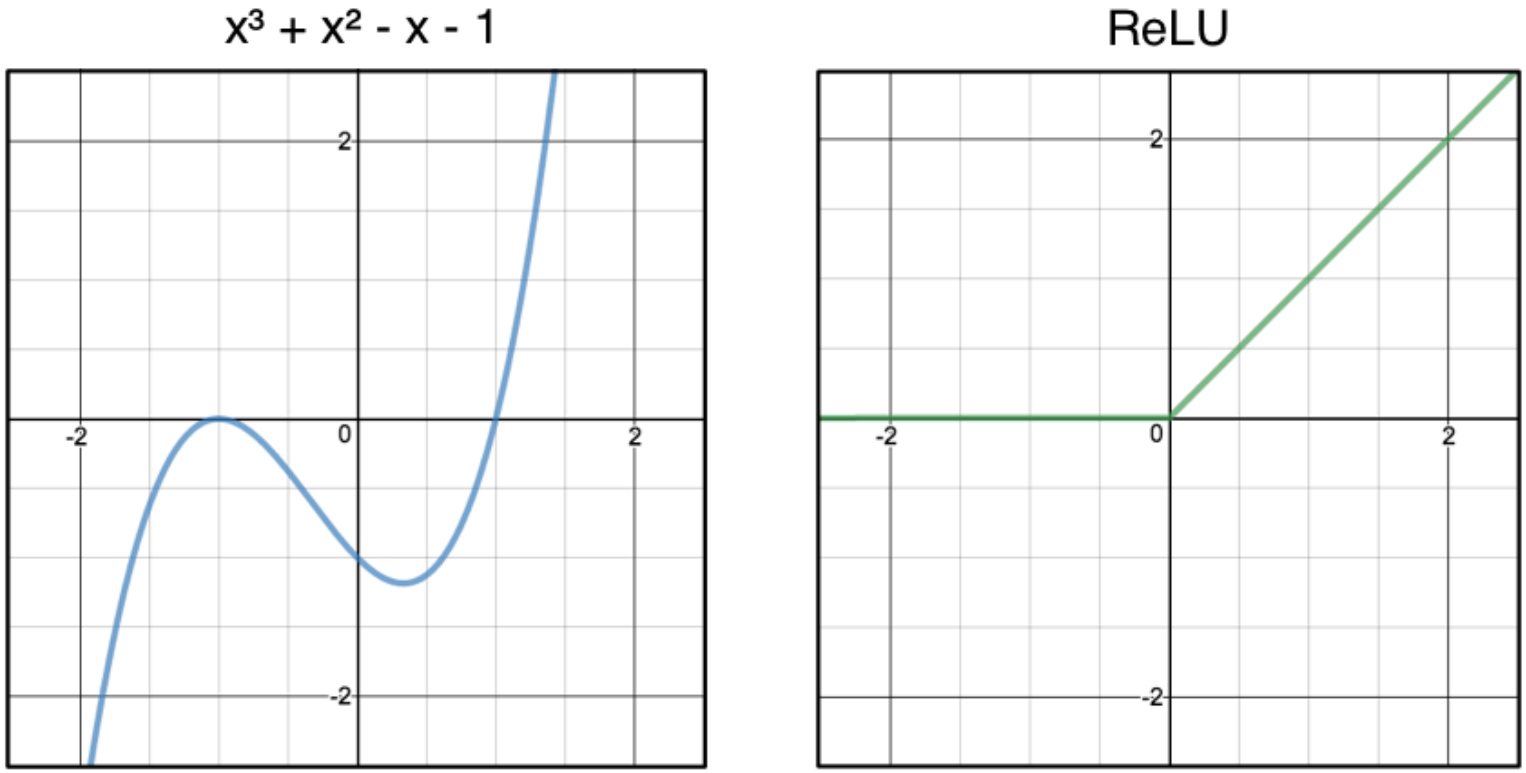
> ***Universal approximation theorem*** *a feedforward network with a single hidden layer with any "squashing" activation function can approximate any Borel measurable function from one finite-dimensional space to another with any desired nonzero amount of error, provided that the network can also approximate the derivatives of the function arbitrarily well.*

Deep Learning, Goodfellow et al. (2016), theorem due to Hornik et al., 1989/1890, Cybenko, 1989

any continuous function on a closed and bounded subset of $\mathbb{R}^n$ is Borel measurable.

The logistic sigmoid activation function is an example of a "squashing" activation function. The theorem has now been extended to include the ReLU activation function.

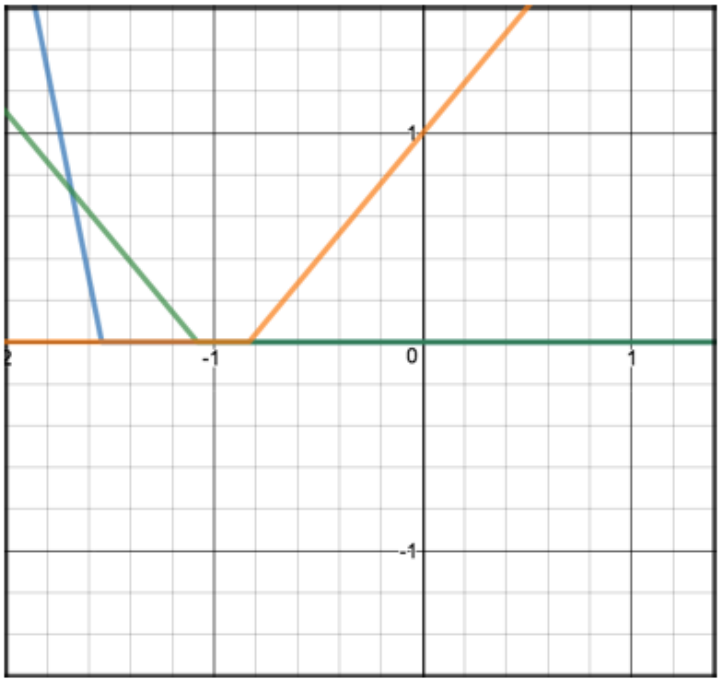
Function approximation with ReLUs[1]



Suppose we want to approximate the polynomial (left) with a weighted sum of ReLUs (right), corresponding to a single neural network layer.

[1] example from `https://towardsdatascience.com/can-neural-networks-really-learn-any-function-65e106617fc6`

Function approximation with ReLUs[1]



One can shift the ReLU function left and right by changing the bias and the ReLU's slope can be adjusted with the weight. Left shows the ReLUs whose sum partly approximates the polynomial.

[1] example from `https://towardsdatascience.com/can-neural-networks-really-learn-any-function-65e106617fc6`

Function approximation with ReLUs[1]

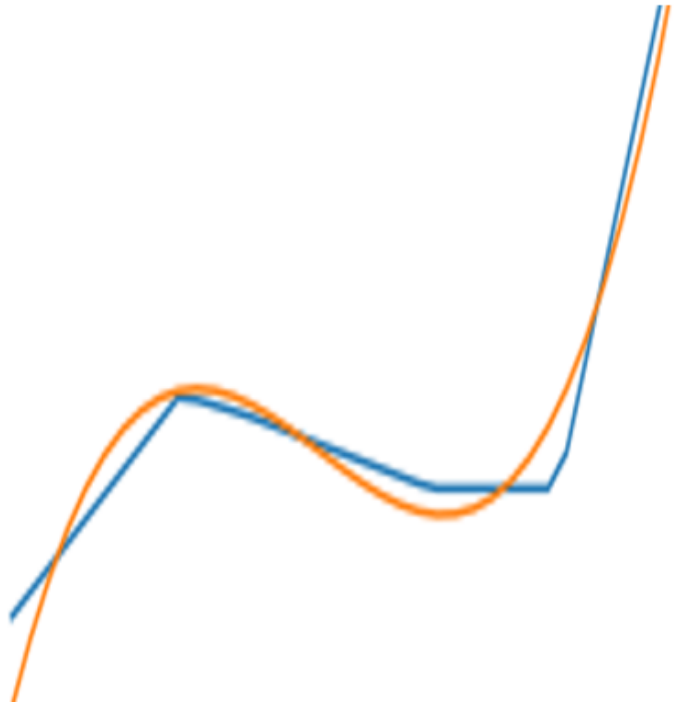
$$n_1(x) = Relu(-5x - 7.7)$$
$$n_2(x) = Relu(-1.2x - 1.3)$$
$$n_3(x) = Relu(1.2x + 1)$$
$$n_4(x) = Relu(1.2x - .2)$$
$$n_5(x) = Relu(2x - 1.1)$$
$$n_6(x) = Relu(5x - 5)$$

$$Z(x) = -n_1(x) - n_2(x) - n_3(x) + n_4(x) + n_5(x) + n_6(x)$$

Using 6 ReLUs gets pretty close.

[1] example from `https://towardsdatascience.com/can-neural-networks-really-learn-any-function-65e106617fc6`

Function approximation with ReLUs[1]



Learning using gradient descent leads to a different approximation (right), and perhaps slightly worse fit, but works.

[1] example from `https://towardsdatascience.com/can-neural-networks-really-learn-any-function-65e106617fc6`

Universal approximation properties and depth

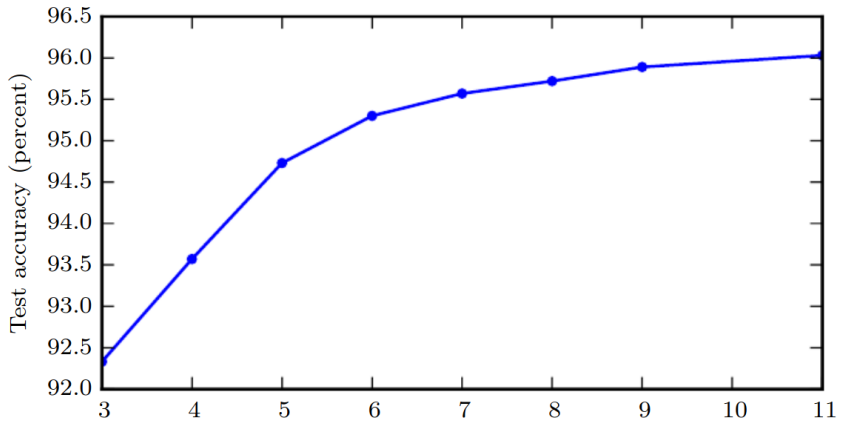
So approximation is possible in theory, but in practice we are limited by what the training algorithm will be able to learn.

- The optimization algorithm may fail to find the best parameters
- It may overfit

Worst case these networks may also be infeasibly wide.

Depth vs width

In general, approximation accuracy increases with width, but depth is as important and often you will find that it is advantageous to trade width for depth.



Depth vs width

In general, approximation accuracy increases with width, but depth is as important and often you will find that it is advantageous to trade width for depth. Trading width for depth

- Computation time, whether in training or testing, is often just a matter of network size (whether deep or wide)
  - Some caveats to this - wider networks may offer better parallelization than deeper networks, as each unit in a layer can be compute in parallel, but depend on the output of the previous layers.
- Deeper networks can more efficiently learn higher order relationships. Example: if you need to classify something as a tree, you may benefit from knowing what is a leaf and a branch and use that as input to the hypothetical tree classifier.

Other architectural considerations

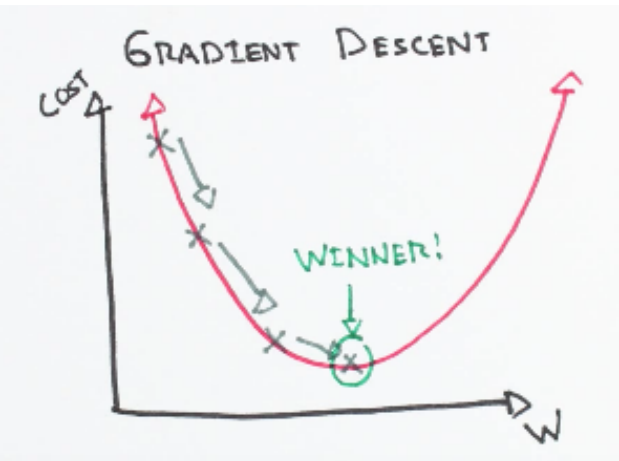
- Specialized architectures
  - Convolutional neural networks (computer vision)
  - Recurrent neural networks (sequence processing)
- Skip connections (connections from layer i to layer $i + 2$ for instance)
- Networks that are not fully connected

Optimizing neural network parameters

Goal: we want to find the weights that minimize the expected loss (risk) of our neural network.

The typical way to approximate this is to look for the weights that minimize the average loss on a sample of the data.

Gradient descent is the typical approach to this, which can be done using a technique known as **back-propagation**.



[1]

[1] Figure from `https://ml-cheatsheet.readthedocs.io/en/latest/gradient_descent.html`

Back-propagation

The processing of input through each layer to output is called **forward propagation**.

The **back-propagation** algorithm or **backprop**, allows information to flow backward through the network in order to compute gradients.

Back-propagation

Thinking of neural network layers as composition of functions, we simplify things a bit and just look at composition of three functions g, h, and i

$$f(x, y, z) = g(h(i(x), y), z)$$

If this was a neural network, x, y, and z might be the neural network weights, and the outer most function g could be the average loss across a sample.

Back-propagation

Thinking of neural network layers as composition of functions, we simplify things a bit and just look at composition of three functions g, h, and i

$$f(x, y, z) = g(h(i(x), y), z)$$

If this was a neural network, x, y, and z might be the neural network weights, and the outer most function g could be the average loss across a sample.

We want to minimize f and therefore seek the gradient of f with respect to x, y, and z.

$$\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \text{ and } \frac{\partial f}{\partial z}$$

Back-propagation

Thinking of neural network layers as composition of functions, we simplify things a bit and just look at composition of three functions g, h, and i

$$f(x, y, z) = g(h(i(x), y), z)$$

If this was a neural network, x, y, and z might be the neural network weights, and the outer most function g could be the average loss across a sample.

Lets only look at y. We can simplify $f(i, y, z) = g(h(i, y), z)$, with $i = i(x)$. We can then arrive at

$$\frac{\partial f}{\partial y}(i, y, z) = \frac{\partial h}{\partial y}(i, y)\frac{\partial f}{\partial h}(h, z)$$

using the chain rule, where $h = h(i, y)$.

Back-propagation

Lets look at this a little closer

$$\frac{\partial f}{\partial y}(i, y, z) = \frac{\partial h}{\partial y}(i, y)\frac{\partial f}{\partial h}(h, z)$$

The local gradient of h with respect to y. The gradient of f with respect to h's output.

Back-propagation

Lets look at this a little closer

$$\frac{\partial f}{\partial y}(i, y, z) = \frac{\partial h}{\partial y}(i, y)\frac{\partial f}{\partial h}(h, z)$$

The local gradient of h with respect to y. The gradient of f with respect to h's output.

Notice that to compute the particular gradient, we need

- the values of the inputs and outputs (i and h in this case) to the function (layer) where the parameter is used, which can be computed in a forward pass ($i = i(x) \rightarrow h = h(i, y) \rightarrow g(h, z)$).
- the local gradient of the layer with respect to its parameter ($\frac{\partial h}{\partial y}(i, y)$ in this case), which typically can be pre-computed analytically, and then evaluated using the i arrivated at in the forward pass.
- plus the gradient with respect to the particular layers output $\frac{\partial f}{\partial h}(h, z)$.

Back-propagation

Lets look at this a little closer

$$\frac{\partial f}{\partial y}(i, y, z) = \frac{\partial h}{\partial y}(i, y)\frac{\partial f}{\partial h}(h, z)$$

The local gradient of h with respect to y. The gradient of f with respect to h's output.

- plus the gradient with respect to the particular layers output $\frac{\partial f}{\partial h}(h, z)$.

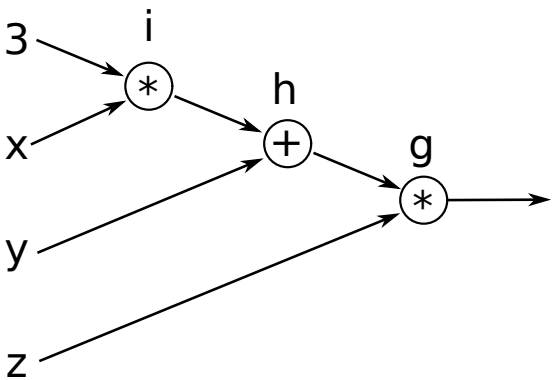
To get this last part, the realization we need to make is that this is simply the gradient from 'further along' in the function composition (or in the neural network). So if we start gradient computation from the end and propagate them back in a backward pass, using values computed in the forward pass, we already have this. This is the **back-propagation** algorithm.

Back-propagation

Lets exemplify this further...[1]

$$
\begin{aligned}
f(x, y, z) &= g(h(i(x), y), z) \\
i(x) &= 3x \\
h(i, y) &= i + y \\
g(h, z) &= hz
\end{aligned}
$$

Enter the computational graph



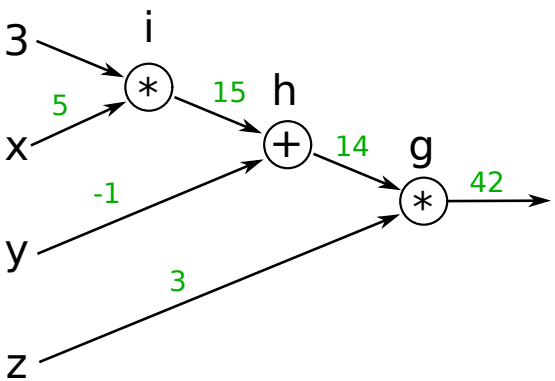
[1] Figure inspired by
https://www.youtube.com/watch?v=d14TUNcbn1k&index=4&list=PLC1qU-LWwrF64f4QKQT-Vg5Wr4qEE1Zxk

Back-propagation

Lets exemplify this further...[1]

$$
\begin{aligned}
f(x, y, z) &= g(h(i(x), y), z) \\
i(x) &= 3x \\
h(i, y) &= i + y \\
g(h, z) &= hz
\end{aligned}
$$

Enter the computational graph



[1] Figure inspired by
https://www.youtube.com/watch?v=d14TUNcbn1k&index=4&list=PLC1qU-LWwrF64f4QKQT-Vg5Wr4qEE1Zxk

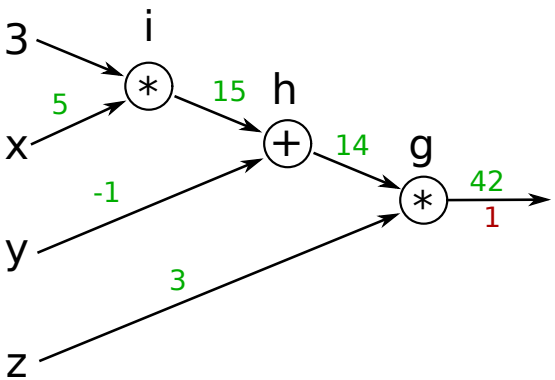
Back-propagation

Lets exemplify this further...[1]

$$
\begin{aligned}
f(x, y, z) &= g(h(i(x), y), z) \\
i(x) &= 3x \\
h(i, y) &= i + y \\
g(h, z) &= hz
\end{aligned}
$$

Enter the computational graph

$$\frac{\partial f}{\partial g} = 1$$



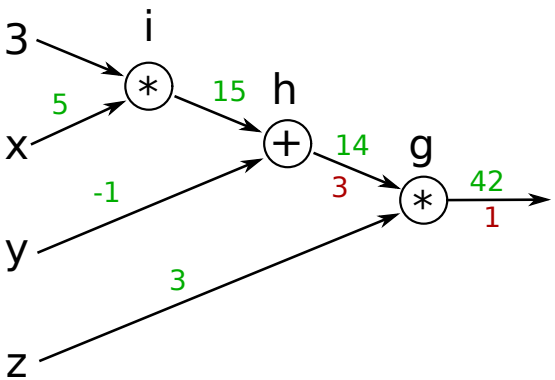
[1] Figure inspired by
https://www.youtube.com/watch?v=d14TUNcbn1k&index=4&list=PLC1qU-LWwrF64f4QKQT-Vg5Wr4qEE1Zxk

Back-propagation

Lets exemplify this further...[1]

$$
\begin{aligned}
f(x, y, z) &= g(h(i(x), y), z) \\
i(x) &= 3x \\
h(i, y) &= i + y \\
g(h, z) &= hz
\end{aligned}
$$

Enter the computational graph



$$
\begin{aligned}
\frac{\partial f}{\partial g} &= 1 \\
\frac{\partial f}{\partial h} &= \frac{\partial g}{\partial h} \cdot \frac{\partial f}{\partial g} = 3 \cdot 1 = 3
\end{aligned}
$$

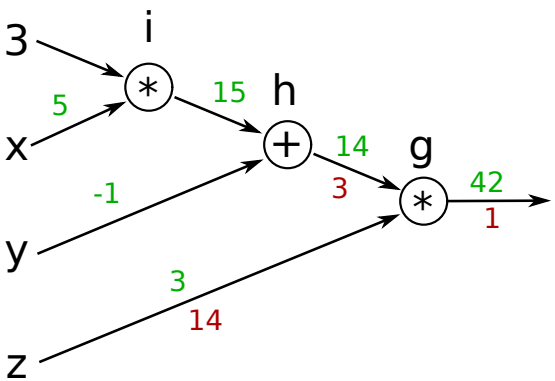
[1] Figure inspired by
https://www.youtube.com/watch?v=d14TUNcbn1k&index=4&list=PLC1qU-LWwrF64f4QKQT-Vg5Wr4qEE1Zxk

Back-propagation

Lets exemplify this further...[1]

$$
\begin{aligned}
f(x, y, z) &= g(h(i(x), y), z) \\
i(x) &= 3x \\
h(i, y) &= i + y \\
g(h, z) &= hz
\end{aligned}
$$

Enter the computational graph



$$
\begin{aligned}
\frac{\partial f}{\partial g} &= 1 \\
\frac{\partial f}{\partial h} &= \frac{\partial g}{\partial h} \cdot \frac{\partial f}{\partial g} = 3 \cdot 1 = 3 \\
\frac{\partial f}{\partial z} &= \frac{\partial g}{\partial z} \cdot \frac{\partial f}{\partial g} = 14 \cdot 1 = 14
\end{aligned}
$$

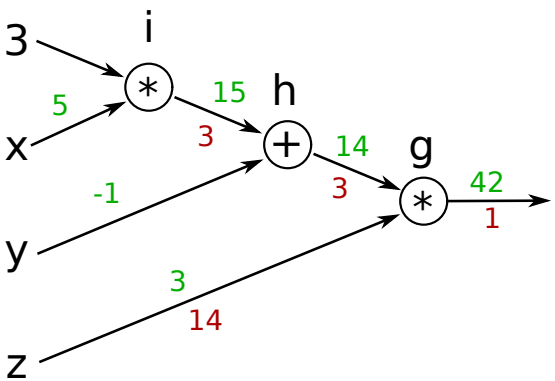
[1] Figure inspired by
https://www.youtube.com/watch?v=d14TUNcbn1k&index=4&list=PLC1qU-LWwrF64f4QKQT-Vg5Wr4qEE1Zxk

Back-propagation

Lets exemplify this further...[1]

$$
\begin{aligned}
f(x, y, z) &= g(h(i(x), y), z) \\
i(x) &= 3x \\
h(i, y) &= i + y \\
g(h, z) &= hz
\end{aligned}
$$

Enter the computational graph



$$
\begin{aligned}
\frac{\partial f}{\partial g} &= 1 \\
\frac{\partial f}{\partial h} &= \frac{\partial g}{\partial h} \cdot \frac{\partial f}{\partial g} = 3 \cdot 1 = 3 \\
\frac{\partial f}{\partial z} &= \frac{\partial g}{\partial z} \cdot \frac{\partial f}{\partial g} = 14 \cdot 1 = 14 \\
\frac{\partial f}{\partial i} &= \frac{\partial h}{\partial i} \cdot \frac{\partial f}{\partial h} = 1 \cdot 3 = 3
\end{aligned}
$$

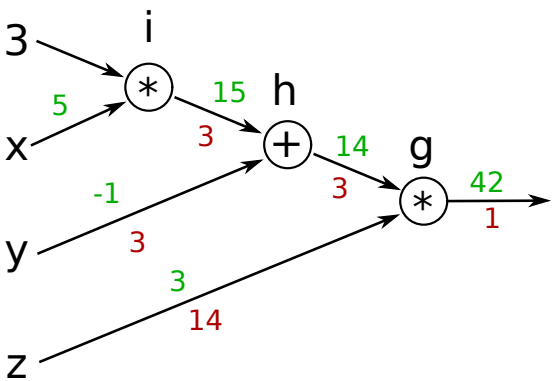
[1] Figure inspired by
https://www.youtube.com/watch?v=d14TUNcbn1k&index=4&list=PLC1qU-LWwrF64f4QKQT-Vg5Wr4qEE1Zxk

Back-propagation

Lets exemplify this further...[1]

$$
\begin{aligned}
f(x, y, z) &= g(h(i(x), y), z) \\
i(x) &= 3x \\
h(i, y) &= i + y \\
g(h, z) &= hz
\end{aligned}
$$

Enter the computational graph



$$
\begin{aligned}
\frac{\partial f}{\partial g} &= 1 \\
\frac{\partial f}{\partial h} &= \frac{\partial g}{\partial h} \cdot \frac{\partial f}{\partial g} = 3 \cdot 1 = 3 \\
\frac{\partial f}{\partial z} &= \frac{\partial g}{\partial z} \cdot \frac{\partial f}{\partial g} = 14 \cdot 1 = 14 \\
\frac{\partial f}{\partial i} &= \frac{\partial h}{\partial i} \cdot \frac{\partial f}{\partial h} = 1 \cdot 3 = 3 \\
\frac{\partial f}{\partial y} &= \frac{\partial h}{\partial y} \cdot \frac{\partial f}{\partial h} = 1 \cdot 3 = 3
\end{aligned}
$$

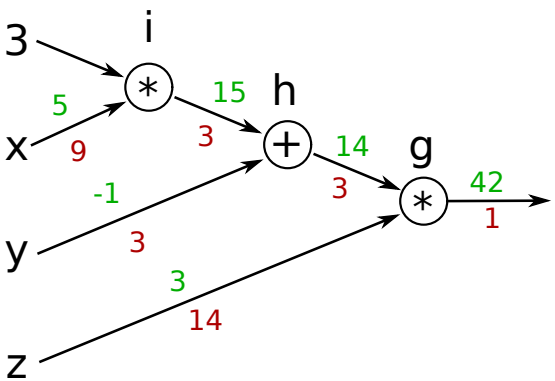
[1] Figure inspired by
https://www.youtube.com/watch?v=d14TUNcbn1k&index=4&list=PLC1qU-LWwrF64f4QKQT-Vg5Wr4qEE1Zxk

Back-propagation

Lets exemplify this further...[1]

$$
\begin{aligned}
f(x, y, z) &= g(h(i(x), y), z) \\
i(x) &= 3x \\
h(i, y) &= i + y \\
g(h, z) &= hz
\end{aligned}
$$

Enter the computational graph



$$
\begin{aligned}
\frac{\partial f}{\partial g} &= 1 \\
\frac{\partial f}{\partial h} &= \frac{\partial g}{\partial h} \cdot \frac{\partial f}{\partial g} = 3 \cdot 1 = 3 \\
\frac{\partial f}{\partial z} &= \frac{\partial g}{\partial z} \cdot \frac{\partial f}{\partial g} = 14 \cdot 1 = 14 \\
\frac{\partial f}{\partial i} &= \frac{\partial h}{\partial i} \cdot \frac{\partial f}{\partial h} = 1 \cdot 3 = 3 \\
\frac{\partial f}{\partial y} &= \frac{\partial h}{\partial y} \cdot \frac{\partial f}{\partial h} = 1 \cdot 3 = 3 \\
\frac{\partial f}{\partial x} &= \frac{\partial i}{\partial x} \cdot \frac{\partial f}{\partial i} = 3 \cdot 3 = 9
\end{aligned}
$$

[1] Figure inspired by
https://www.youtube.com/watch?v=d14TUNcbn1k&index=4&list=PLC1qU-LWwrF64f4QKQT-Vg5Wr4qEE1Zxk

Back-propagation beyond scalars

Suppose $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{y} \in \mathbb{R}^n$, g maps from $\mathbb{R}^m$ to $\mathbb{R}^n$ and f maps from $\mathbb{R}^n$ to $\mathbb{R}$. If $\mathbf{y} = g(\mathbf{x})$ and $z = f(\mathbf{y})$, then

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

In vector notation

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

where $\frac{\partial y}{\partial x}$ is the $n \times m$ Jacobian matrix of g and $\nabla_{\mathbf{y}} z$ the gradient.

- Back-propagation consists of performing such jacobian-gradient products for each operation in the computational graph.
- We may have an arbitrary number of dimensions and thus have tensors of arbitrary dimensionality to deal with, but conceptually this does not change things, only how data is arranged.

Alternatives to back-propagation

Back-propagation is just one technique for automatic differentiation

- It is a special case of techniques known as **reverse mode accumulation**
- Other techniques process the subexpressions of the chain rule in different order. The order has relevance for how much information needs to be stored and how many subexpressions needs to be computed multiple times.
- One may alternatively use **forward mode accumulation** when the number of outputs of the graph is larger than the inputs. This also avoids having to store forward results, which saves memory.
- Higher-order derivatives can be beneficial for optimization, but Hessian matrices can become so large they may be infeasible to represent.

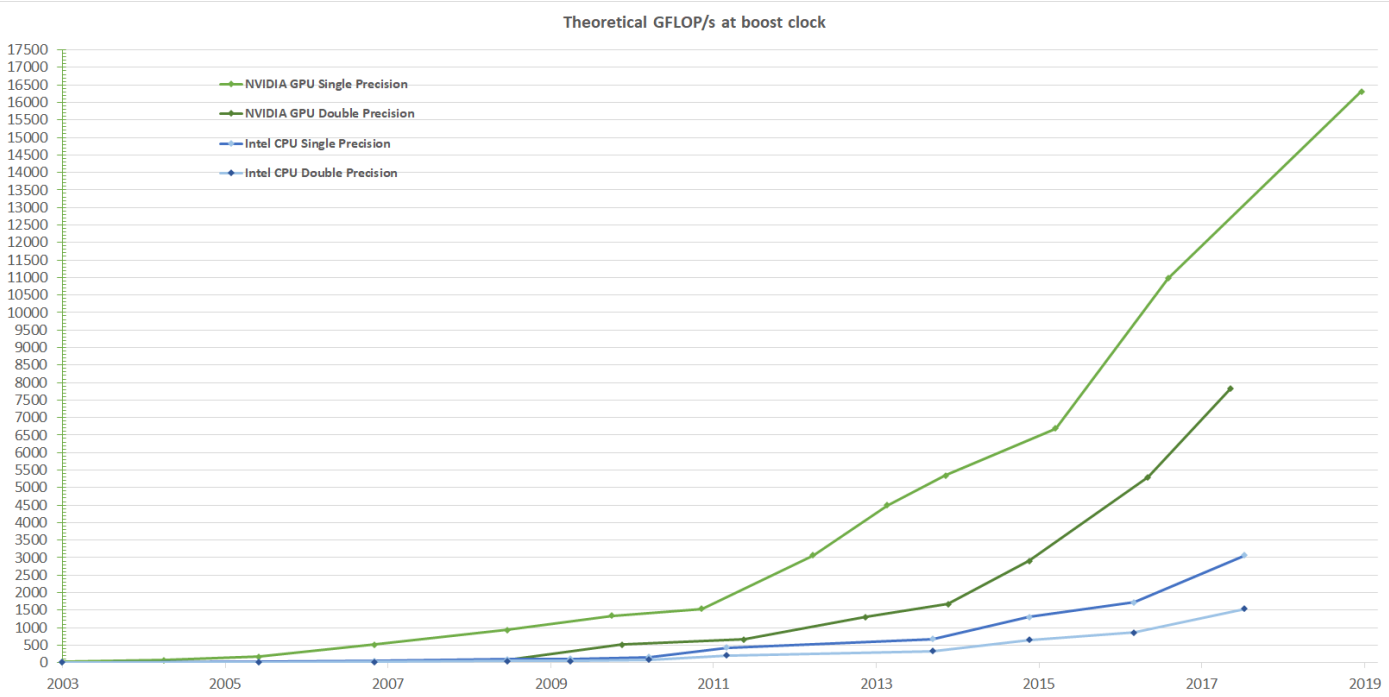
Historical notes

The recent popularity and growth of neural networks, the DL book argues,

- is not due to substantial changes in techniques (since the 1980's)
  - back-propagation has not changed
  - the same gradient descent techniques are used
- is due to
  - larger datasets
  - better software infrastructure and hardware (Graphical Processing Units (GPUs))
  - replacing sigmoid hidden units with piecewise linear hidden units
    - however, ReLUs are not new, but they perform worse than sigmoids in smaller networks and therefore did not find widespread use until networks became larger

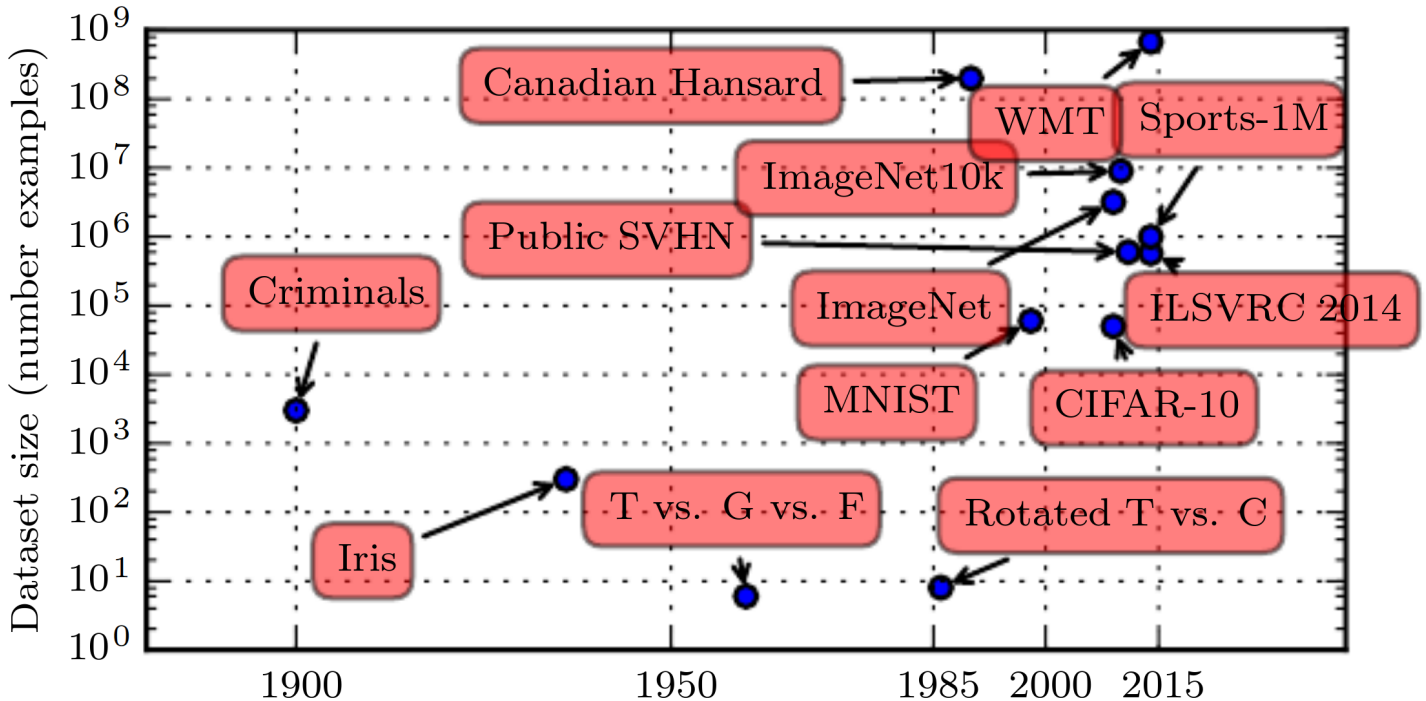
CPU vs GPU performance [1]

- Neural networks are highly parallellizable, meaning training and inference can easily exploit multiple processing units (which GPUs have).
- GPUs lead CPUs in the amount of floating point operations they can perform per second.

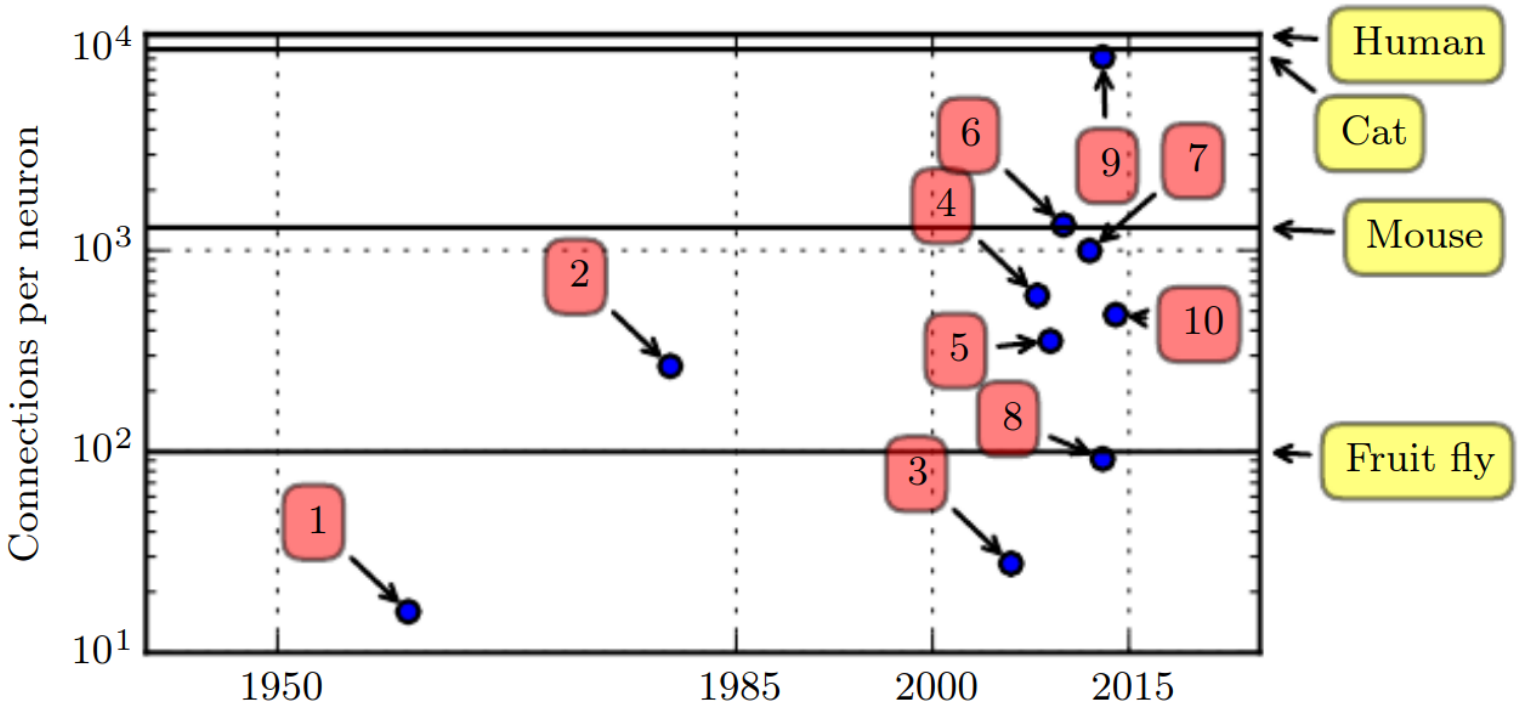


[1] Figure from `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

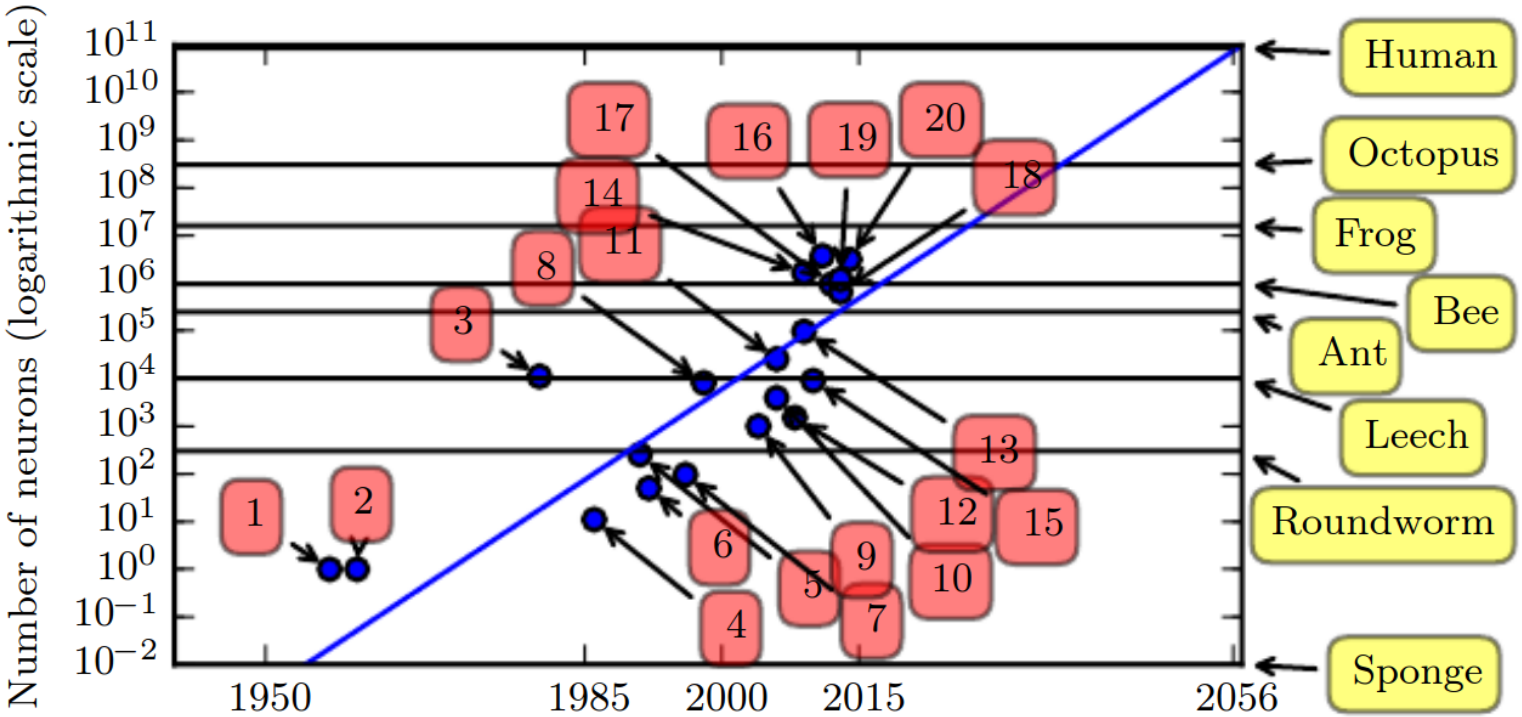
Historical notes - dataset sizes



Historical notes - connections per neuron



Historical notes - neural network size over time



Questions?