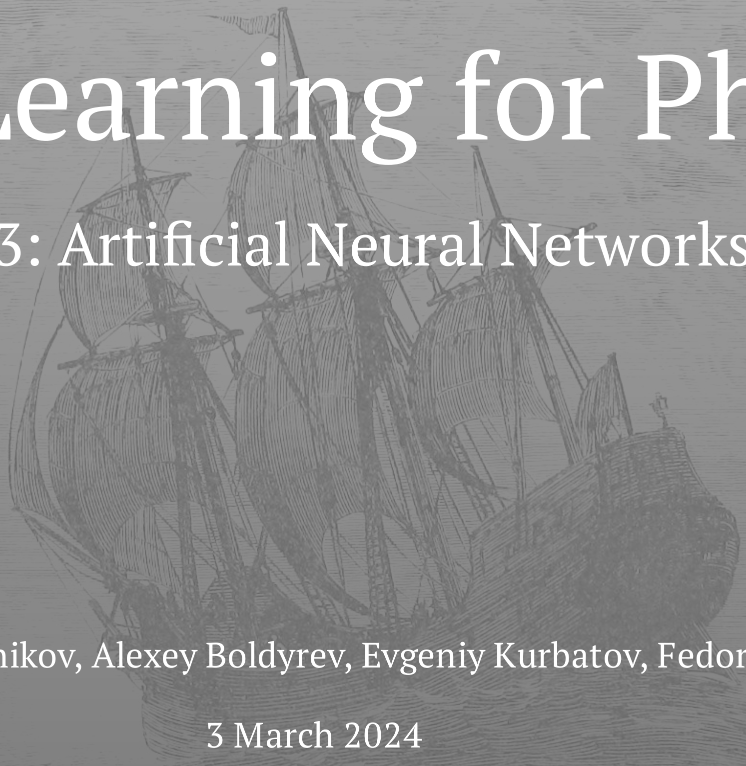


Machine Learning for Physicists

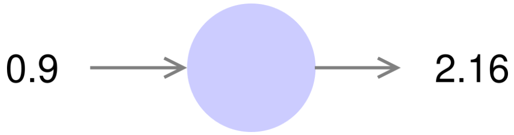
Day 3: Artificial Neural Networks

Vladimir Bocharnikov, Alexey Boldyrev, Evgeniy Kurbatov, Fedor Ratnikov

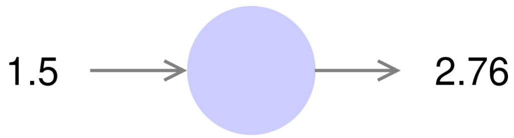
3 March 2024

Image credit: ‘The Mayflower at Sea’ by Granville Perkins, 1876.
Wallach Division Picture Collection, The New York Public Library.

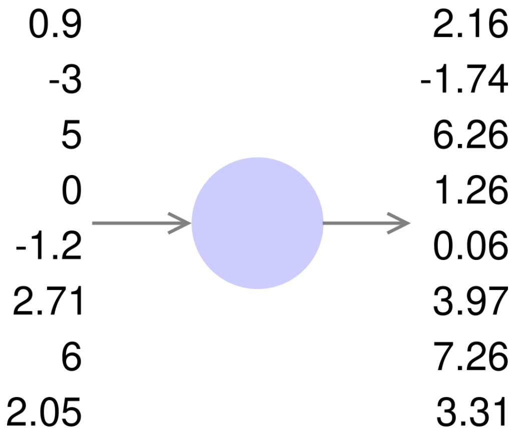
A Neuron



A Neuron



A Neuron



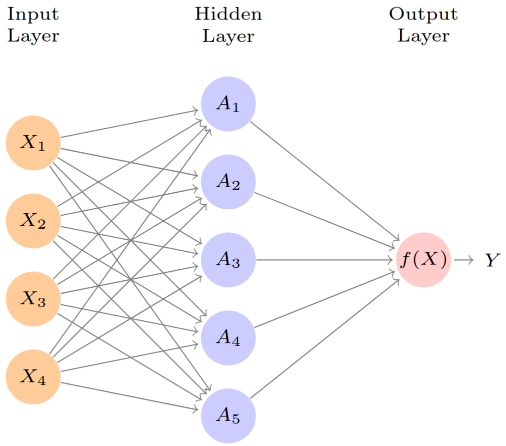
Single Layer Neural Networks

A neural network takes an input vector of p variables $X = (X_1, X_2, ..., X_p)$ and builds a nonlinear function $f(X)$ to predict the response Y:

- $f(X) = \beta_0 + \sum_{k=1}^{K} \beta_k A_k$
- Features $X_1, X_2, ..., X_p$ make up the units in the **input layer**
- K **hidden units** form **hidden layer**
- A_k are **activations** computed as functions of the input features:

  $$A_k = h_k(X) = g(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j),$$

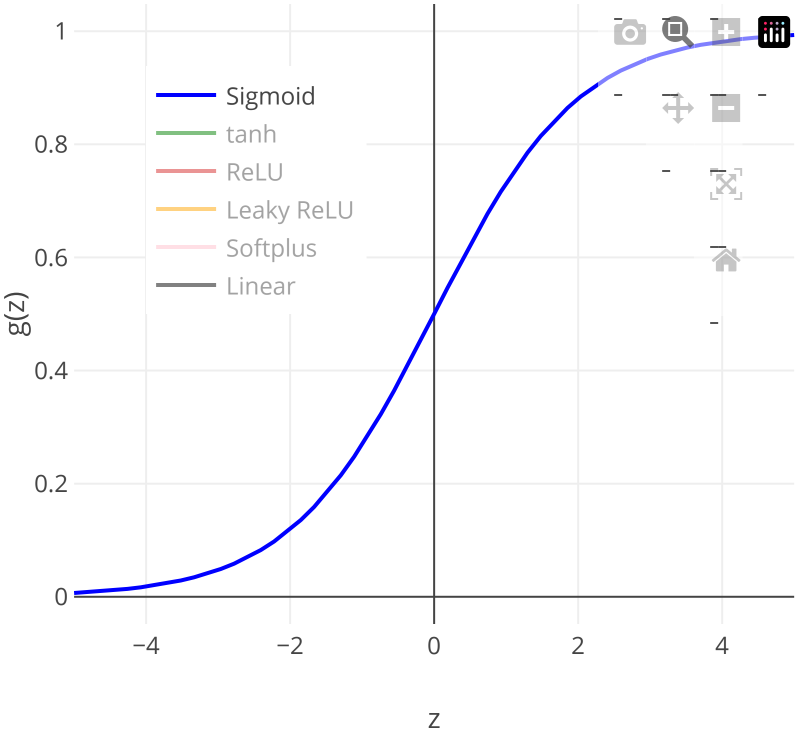
  - $h_k(X)$ is a transformation of original features
  - $g(z)$ is a nonlinear **activation function** specified in advance
- Finally, $f(X) = \beta_0 + \sum_{k=1}^{K} \beta_k g(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j)$



Feed-forward NN. Image source: ISLR Fig. 10.1

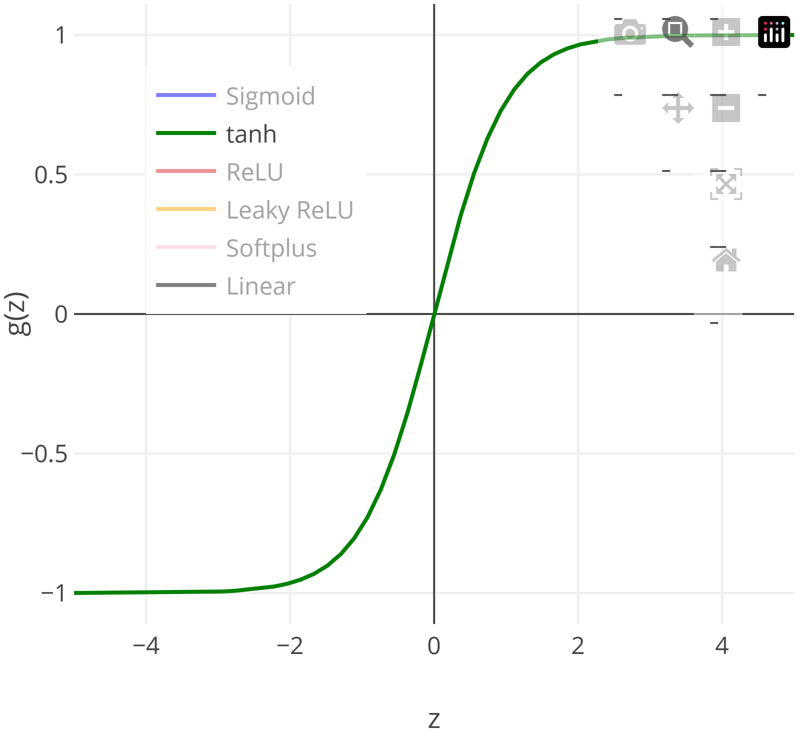
Popular activation functions

- Sigmoid (Logistic): $\sigma(z) := \frac{1}{1+e^{-z}}$
  - Differentiable on $\mathbb{R}$ (just compute a derivative) $\implies$ continuous on $\mathbb{R}$
  - $\sigma(-\infty) = 0, \sigma(0) = 0.5, \sigma(\infty) = 1$
  - 😞 Poorly distinguishes values "far from zero", but almost **linear** near zero
    - $\sigma'(z)|_0 = \sigma(z)(1-\sigma(z))|_0 = \frac{1}{2} \cdot (1-\frac{1}{2}) = \frac{1}{4}$ is a rate of change of σ at zero
    - $\sigma''(z)|_0 = \sigma(z)(1-\sigma(z))(1-2\sigma(z))|_0 = \frac{1}{4} \cdot 0 = 0$ is a rate of change of derivative at zero
  - Sigmoid is a CDF with the corresponding bell-shaped derivative, i.e. PDF



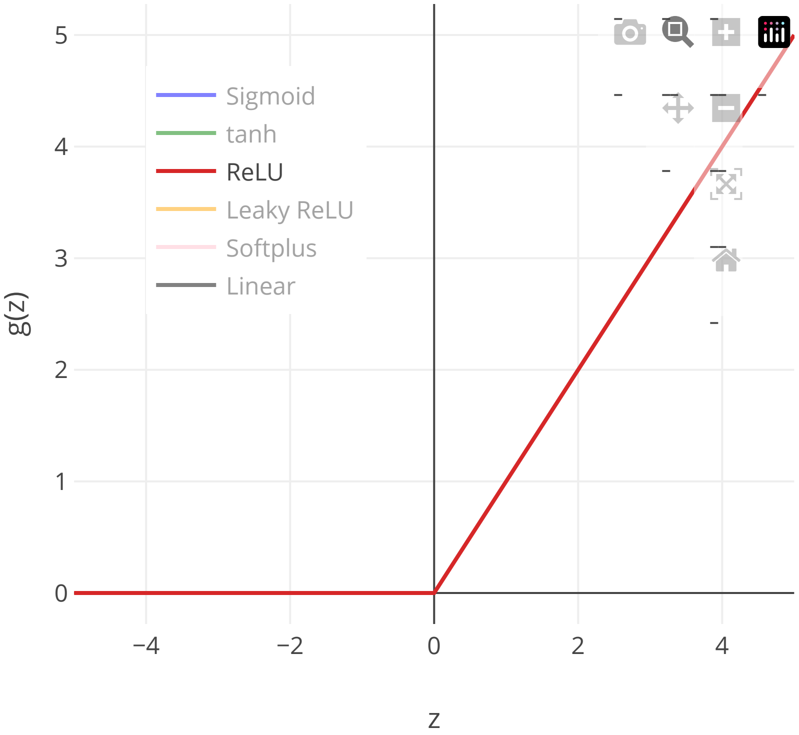
Popular activation functions

- Sigmoid (Logistic): $\sigma(z) := \frac{1}{1+e^{-z}}$
- tanh (Hyperbolic tangent): $\tanh(z) := \frac{e^z - e^{-z}}{e^z + e^{-z}}$
  - Differentiable everywhere with output in $[-1, 1]$
  - Similar shape to sigmoid, also linear (and an **identity**) near zero



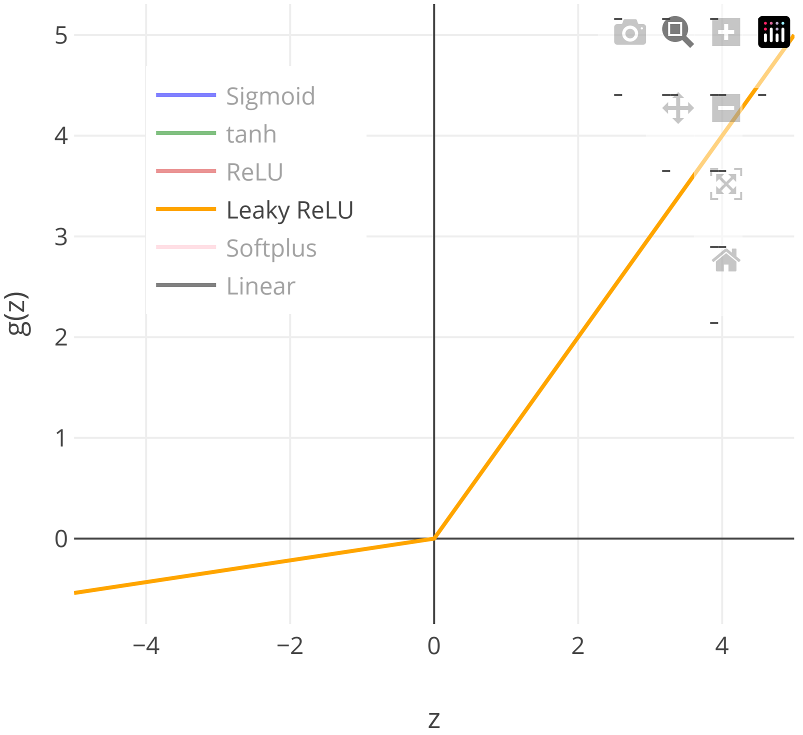
Popular activation functions

- Sigmoid (Logistic): $\sigma(z) := \frac{1}{1+e^{-z}}$
- tanh (Hyperbolic tangent): $\tanh(z) := \frac{e^z - e^{-z}}{e^z + e^{-z}}$
- ReLU (Rectified linear unit): $\mathrm{ReLU}(z) := \max(0, z)$
  - Differentiable everywhere, except $z = 0$ (where it has a "corner")
    - However, it is still continuous everywhere
  - Works well in practice, "*fast*" to compute



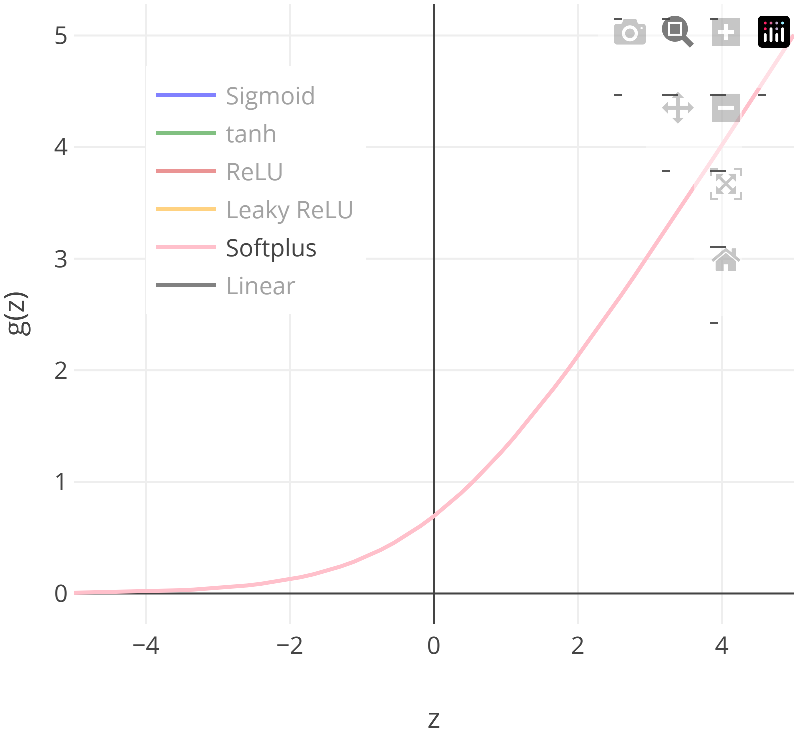
Popular activation functions

- Sigmoid (Logistic): $\sigma(z) := \frac{1}{1+e^{-z}}$
- tanh (Hyperbolic tangent): $\tanh(z) := \frac{e^z - e^{-z}}{e^z + e^{-z}}$
- ReLU (Rectified linear unit): $\text{ReLU}(z) := \max(0, z)$
- Leaky ReLU (and Parametric ReLU)
  - Leaky ReLUs allow a small, positive gradient when the unit is not active
  - PReLUs take this idea further by making the coefficient of leakage into a parameter that is learned along with the other neural-network parameters
  - $a \leq 1$: "*maxout*" networks



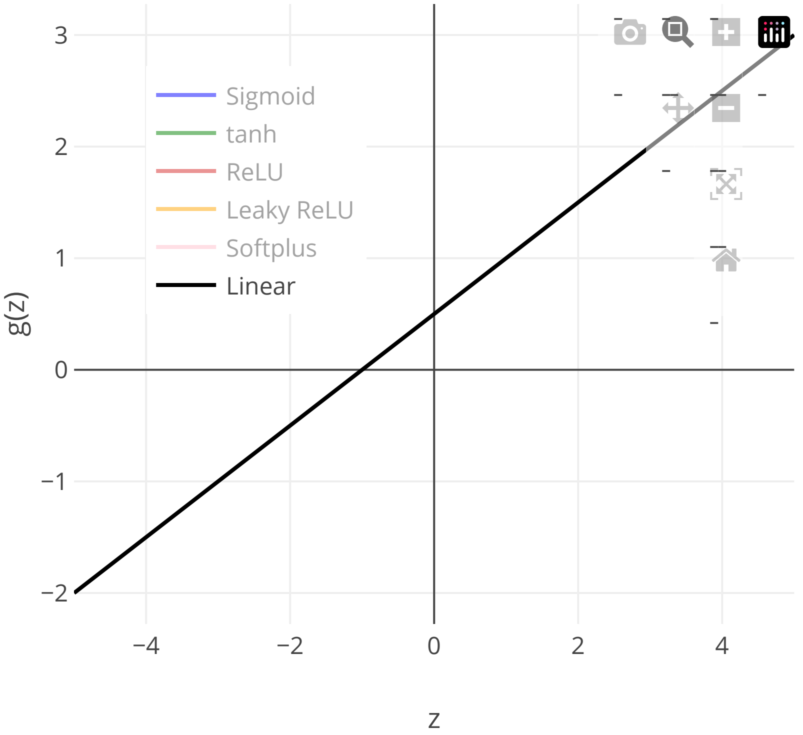
Popular activation functions

- Sigmoid (Logistic): $\sigma(z) := \frac{1}{1+e^{-z}}$
- tanh (Hyperbolic tangent): $\tanh(z) := \frac{e^z - e^{-z}}{e^z + e^{-z}}$
- ReLU (Rectified linear unit): $\mathrm{ReLU}(z) := \max(0, z)$
- Leaky ReLU (and Parametric ReLU)
- Softplus: $\mathrm{softplus}(z) := \log(1 + \exp(z)) : \mathbb{R} \to [0, \infty)$
  - Smooth version of ReLU (no corner), a.k.a SmoothReLU
  - $\mathrm{softplus}(-\infty) = 0$, $\mathrm{softplus}(z) \approx z$ for $z \gg 1$



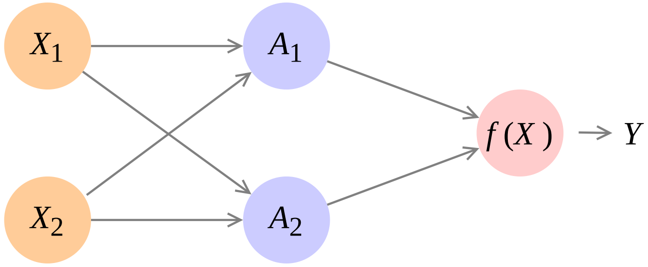
Popular activation functions

- Sigmoid (Logistic): $\sigma(z) := \frac{1}{1+e^{-z}}$
- tanh (Hyperbolic tangent): $\tanh(z) := \frac{e^z - e^{-z}}{e^z + e^{-z}}$
- ReLU (Rectified linear unit): $\mathrm{ReLU}(z) := \max(0, z)$
- Leaky ReLU (and Parametric ReLU)
- Softplus: $\mathrm{softplus}(z) := \log(1 + \exp(z))$
- Linear: $\phi(x) := a + b'x$
  - Differentiable everywhere, "*fast*" to compute
  - 😞 Does not introduce non-linearity. A linear combination of linear combinations is just a linear combination. Stacked layers with ϕ activation can be collapsed to a single linear model.



Example

Ex. $p = 2$ input variables $X = (X_1, X_2)$. $K = 2$ hidden units.



Find $f(X)$ if:

$g(z) = z^2$ and

$$\beta_0 = 0, \beta_1 = \frac{1}{4}, \beta_2 = -\frac{1}{4},$$
$$w_{10} = 0, w_{11} = 1, w_{12} = 1,$$
$$w_{20} = 0, w_{21} = 1, w_{22} = -1.$$

Squared-error loss: $\sum_{i=1}^{n}(y_i - f(x_i))^2$

$$A_k = g(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j)$$
$$A_1 = (0 + X_1 + X_2)^2; \quad A_2 = (0 + X_1 - X_2)^2$$

$$f(X) = \beta_0 + \sum_{k=1}^{K} \beta_k A_k = 0 + \frac{1}{4} \cdot (0 + X_1 + X_2)^2 - \frac{1}{4} \cdot (0 + X_1 - X_2)^2 =$$
$$= \frac{1}{4}[(X_1 + X_2)^2 - (X_1 - X_2)^2] = X_1 X_2$$

Multilayer Neural Networks

Consider a neural network with 2 hidden layers:

- The first hidden layer is as in single-layer NN:

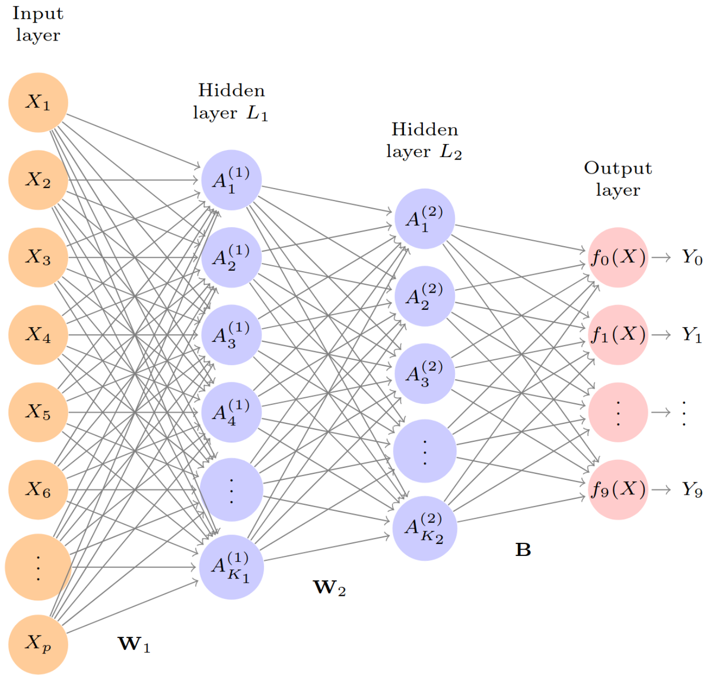
$$A_k^{(1)} = g(w_{k0}^{(1)} + \sum_{j=1}^{p} w_{kj}^{(1)} X_j)$$

- The second hidden layer treats the activations from the first hidden layer:

$$A_l^{(2)} = g(w_{l0}^{(2)} + \sum_{k=1}^{K_1} w_{lk}^{(2)} A_k^{(1)})$$

- Output layer. For $m = 0, 1, ..., 9$ we need to build 10 different linear models: $Z_m = \beta_{m0} + \sum_{l=1}^{K_2} \beta_{ml} A_l^{(2)}$
- Class probability:

$$f_m(X) = \Pr(Y = m|X) = \frac{e^{Z_m}}{\sum_{l=0}^{9} e^{Z_l}} \text{ (softmax)}$$



NN with 2 hidden layers. Image source: ISLR Fig. 10.4

Notation:
W_i - **weights** (coefficients), B - **bias** (intercept)

Fitting a Neural Network

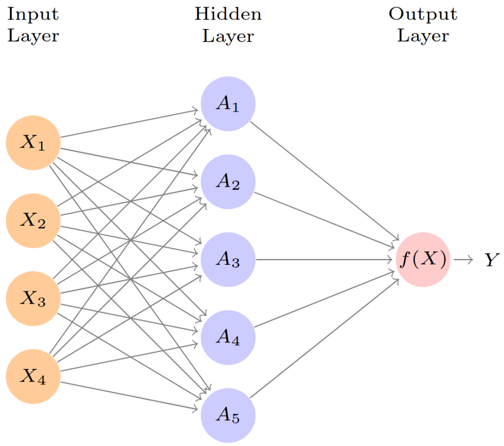
- The model parameters θ are:

  $\beta = (\beta_0, \beta_1, \ldots, \beta_K)$ and $w_k = (w_{k0}, w_{k1}, \ldots, w_{kp})$
- We need to solve a nonlinear least squares problem:

  $$\underset{\{w_k\}_1^K, \beta}{\text{minimize}} \frac{1}{2} \sum_{i=1}^{n} (y_i - f(x_i))^2,$$

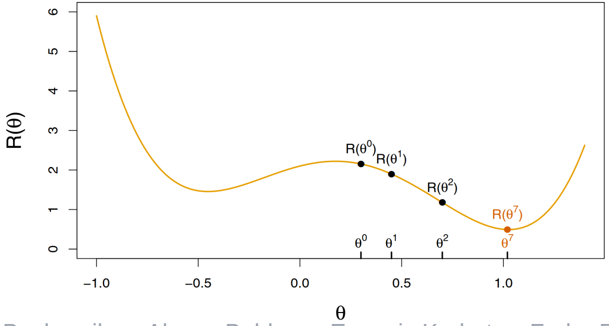
where $f(x_i) = \beta_0 + \sum_{k=1}^{K} \beta_k g(w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij})$

The problem is **nonconvex** in the parameters $\rightsquigarrow$ multiple solutions.



Feed-forward NN. Image source: ISLR Fig. 10.1

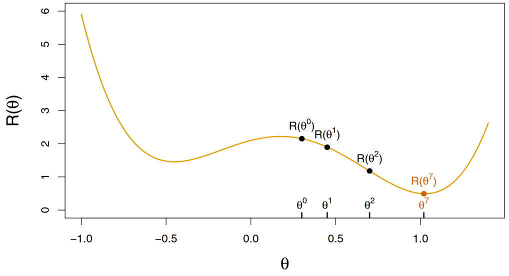
Left image: Gradient descent for 1D θ. Source: ISLR Fig. 10.17



To overcome some of these issues we can use:

- **Slow (or adaptive) learning**
  - **Gradient descent**
- **Regularization** (Ridge/Lasso/Dropout)

Fitting a Neural Network: Gradient Descent



Gradient descent for 1D θ. Image source: ISLR Fig. 10.17

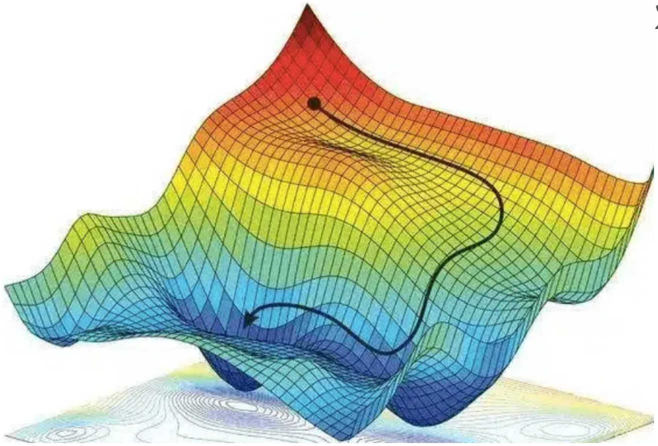


Image source: https://easyai.tech/en/ai-definition/gradient-descent

Rewriting the least squares problem as:

$$R(\theta) = \frac{1}{2}\sum_{i=1}^{n}(y_i - f_\theta(x_i))^2$$

We can formulate the general **gradient descent algorithm**:

1. Start with a guess θ^0 for all the parameters in θ, and set $t = 0$
2. Iterate until the objective $R(\theta)$ fails to decrease:
   1. Find a vector δ that reflects a small change in θ, such that $\theta^{t+1} = \theta^t + \delta$ reduces the objective; i.e. such that $R(\theta^{t+1}) < R(\theta^t)$
   2. Set $t \leftarrow t + 1$

Towards Backpropagation

How do we find the directions to move θ so as to decrease the objective $R(\theta)$?

One need to calculate **gradient** of $R(\theta)$ evaluated at some current value $\theta = \theta^m$:

$$\nabla R(\theta^m) = \frac{\partial R(\theta)}{\partial \theta}\Big|_{\theta=\theta^m}$$

The idea of gradient descent is to move θ a little in the opposite direction:

$$\theta^{m+1} \leftarrow \theta^m - \rho \nabla R(\theta^m),$$

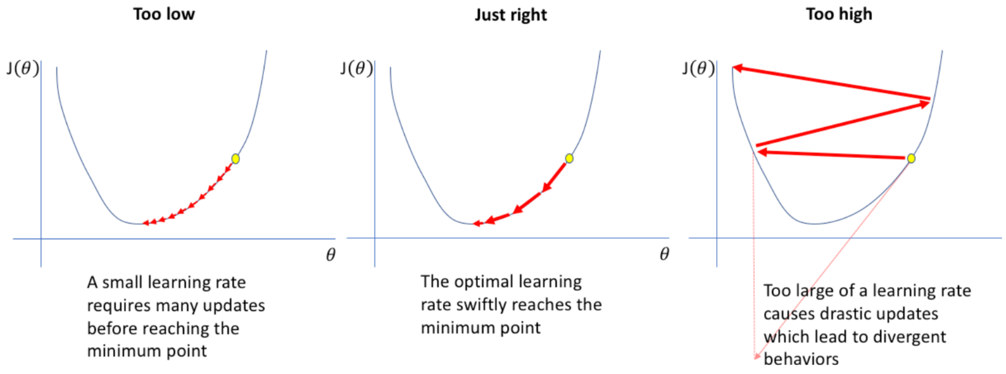
where ρ is the **learning rate**.

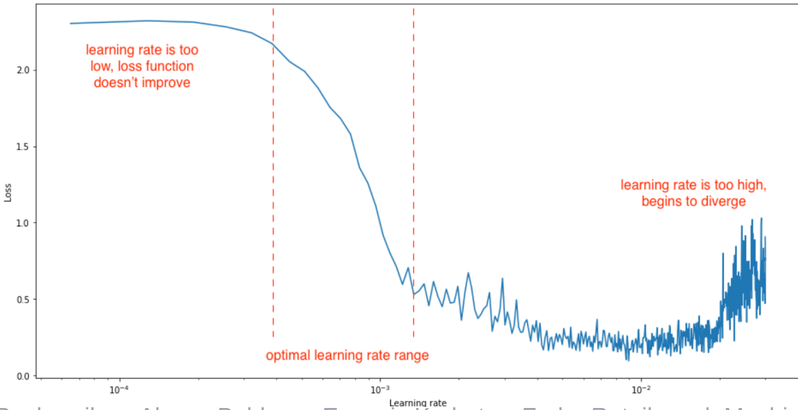
If the gradient vector is zero, then we may have arrived at a minimum of the objective.

Backpropagation

- Compute gradients algorithmically
- Used by deep learning frameworks (PyTorch, TensorFlow, JAX, etc.)

Note on Learning Rate



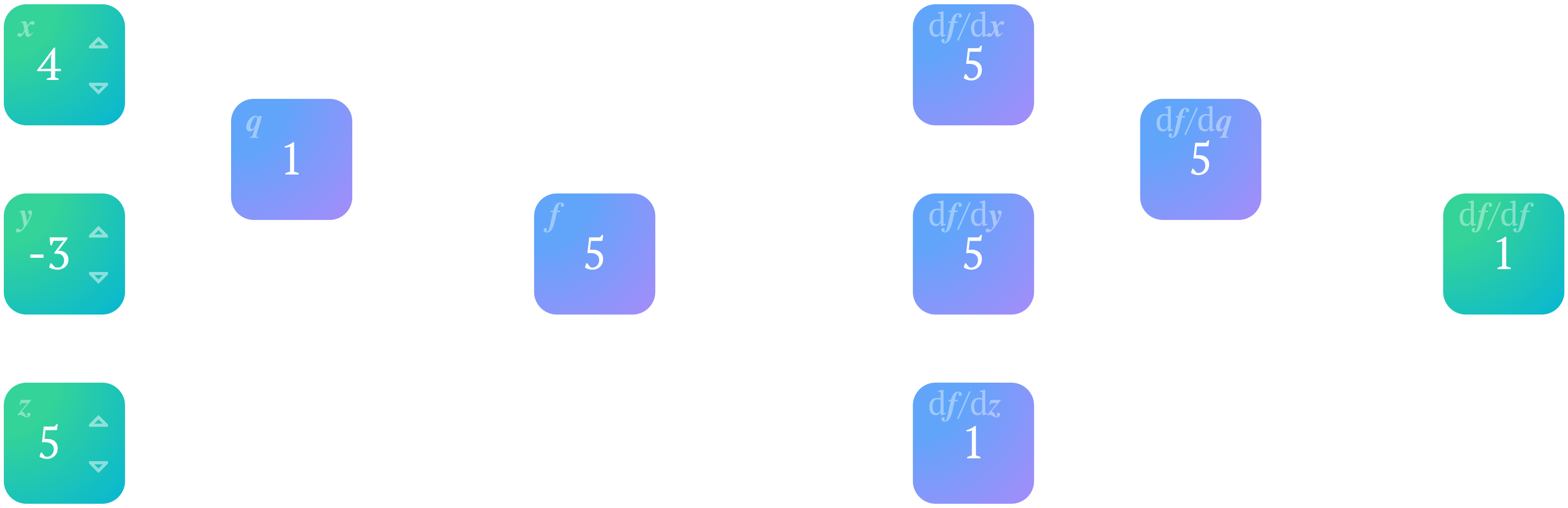


Images source: https://www.jeremyjordan.me/nn-learning-rate

Backpropagation: Example 1

$q = x + y = 4 + -3 = 1$

$f = q * z = 1 * 5 = 5$



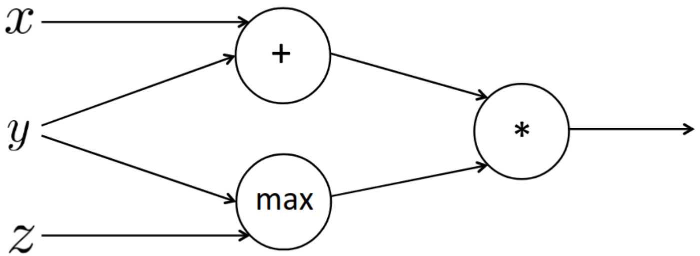
Backpropagation in PyTorch

```
# Initialize x, y and z to values 4, -3 and 5
x = torch.tensor(4., requires_grad=True)
y = torch.tensor(-3., requires_grad=True)
z = torch.tensor(5., requires_grad=True)

# Set q to sum of x and y, set f to product of q with z
q = x + y
f = q * z

# Compute the derivatives
f.backward()

# Print the gradients
print("Gradient of x is: " + str(x.grad))
print("Gradient of y is: " + str(y.grad))
print("Gradient of z is: " + str(z.grad))
```

The code above produces:

```
Gradient of x is: tensor(5.)
Gradient of y is: tensor(5.)
Gradient of z is: tensor(1.)
```

Backpropagation: Example 2



Ex. credits: https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1184

$$f(x, y, z) = (x + y)\max(y, z)$$
$$x = 1,\ y = 2,\ z = 0$$

Find $\frac{\partial f}{\partial y}$

Forward propagation steps:

$$a = x + y$$

$$b = \max(y, z)$$

$$f = a \cdot b$$

Local gradients:

$$\frac{\partial a}{\partial x} = 1 \quad \frac{\partial a}{\partial x} = 1$$

$$\frac{\partial b}{\partial y} = \mathbf{1}(y > z) = 1 \quad \frac{\partial b}{\partial z} = \mathbf{1}(y < z) = 0$$

$$\frac{\partial f}{\partial a} = b = 2 \quad \frac{\partial f}{\partial b} = a = 3$$

$$\frac{\partial f}{\partial x} = 2 \quad \frac{\partial f}{\partial y} = 3 + 2 = 5 \quad \frac{\partial f}{\partial z} = 0$$

Essentials of Artificial Neural Networks

Building blocks:

- Neuron
- Fully-connected (a.k.a. *Linear*) layer
- Activation function
- Loss function (see backup slides)
- Convolution layer
- Pooling layer
- Recurrent layer

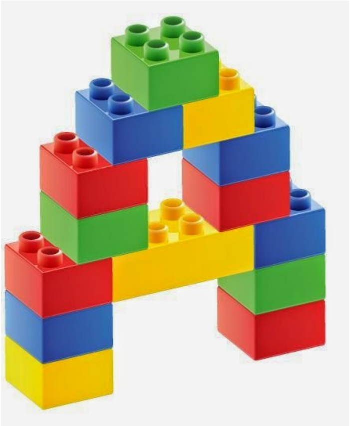


Image source: http://sgaguilarmjargueso.blogspot.com

Concepts:

- Weights & Biases
- Backpropagation
- Gradient descent
- Learning rate
- Batch
- Regularization

Artificial Neural Networks: Overview

- ANN is a flexible class of models, which can find highly non-linear relations in I/O
- ANN is an old technology, revived with recent boom in GPU/TPU, novel algorithms, revenue generation and big investors
- ANN is built from neurons, basic building blocks
- ANN encompasses many infrastructures
- ANN help focus efforts on engineering infrastructure, rather than engineering input features
- ANN are more effective with tasks on unstructured data: text, audio, images, video, video-captioning, …

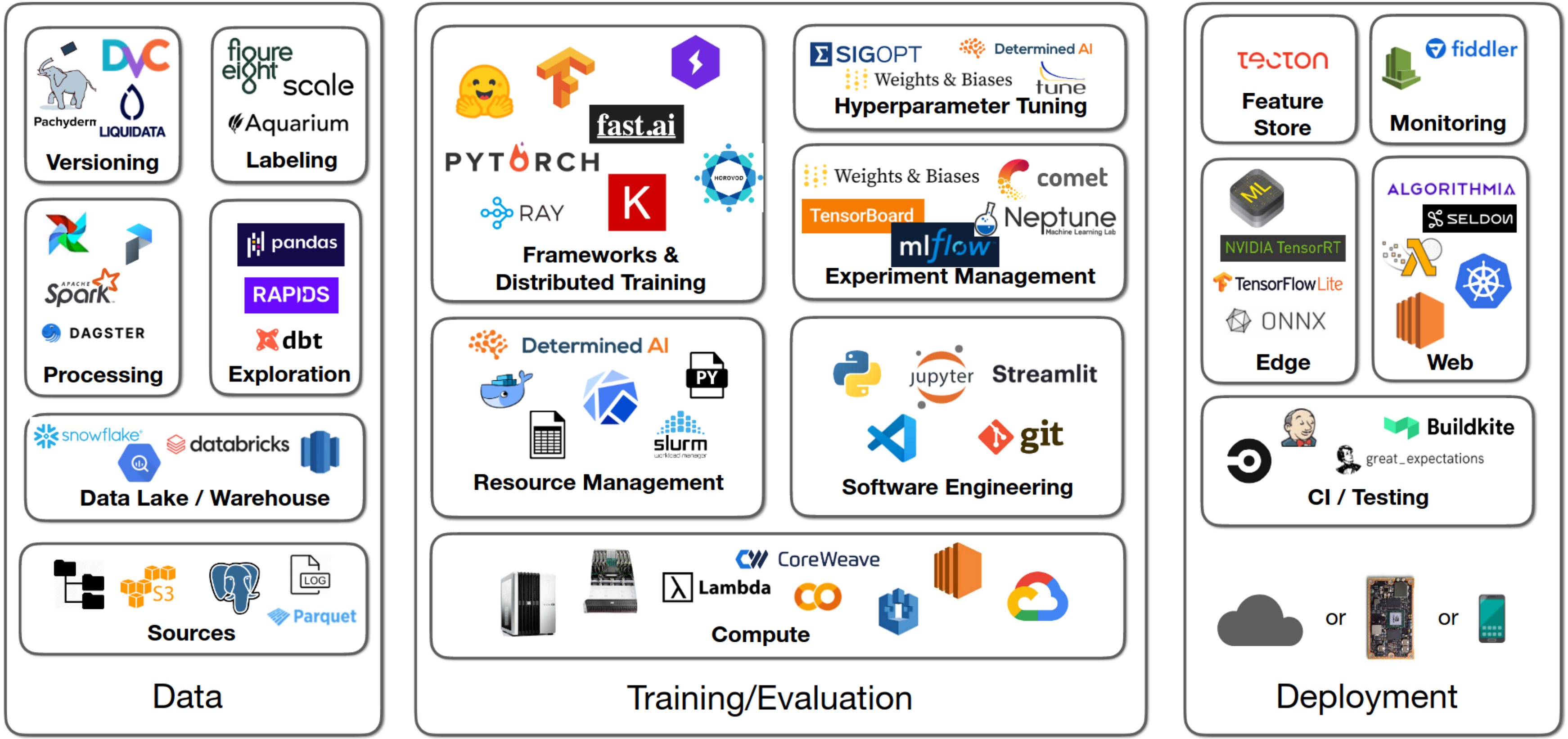
Artificial Neural Networks: Examples

ANN encompasses many infrastructures:

1. RNN for sequence data with short dependencies
2. LSTM for sequence data with short and long dependencies
3. CNN for image-like data with spatial dependencies
4. U-Net is an improved CNN
5. VAE to compress representation of images, audio, ...
6. GAN to generate new observations (e.g. faces, voices) from the training distribution
7. Transformers builds "*attention*" to the "*important*" input data (e.g. lesser value of common stop words in text)
8. DRL trains an agent to take max-reward actions based on current state and past history (e.g. gaming, robotics)
9. GNN for graph-based data (e.g. social network, street maps, citation network)
10. RBM to learn the distribution of input for generative tasks
11. SOM for dimension reduction with maintaining the topological structure

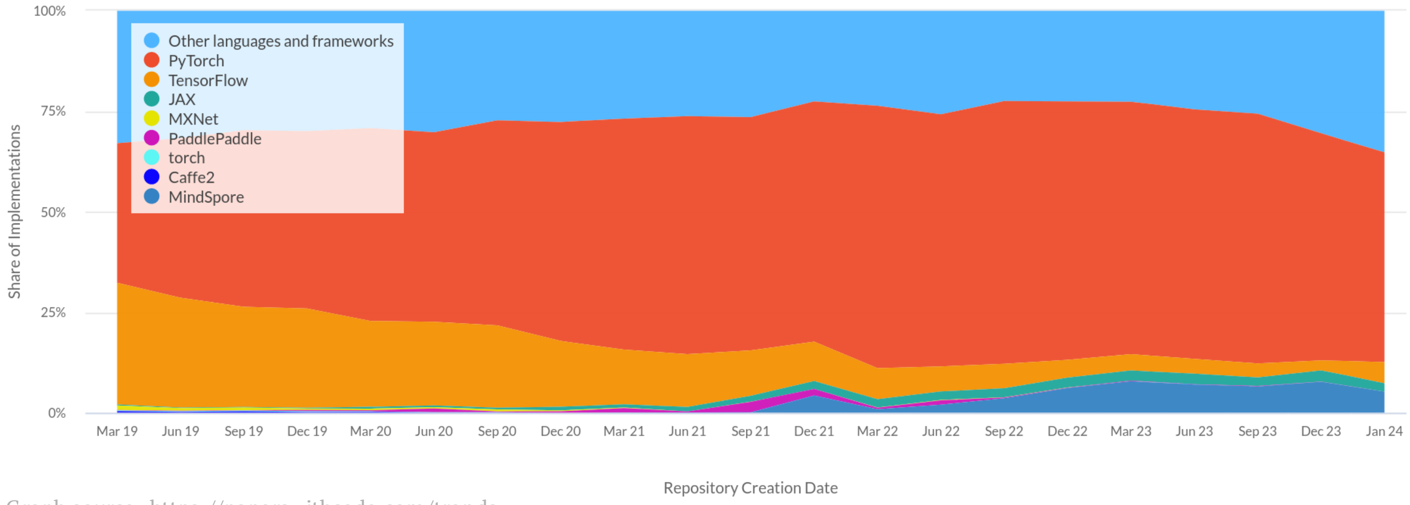
Backup slides

Deep Learning tools



Slide by Sergey Karayev: https://fullstackdeeplearning.com/spring2021/lecture-6/

Deep Learning tools



Graph source: https://paperswithcode.com/trends

Recommended Python-based frameworks for common Deep Learning problem solving:

PyTorch TensorFlow JAX

Deep Learning tools: Tensorflow Playground [link]

Deep Learning tools: Netron [link]

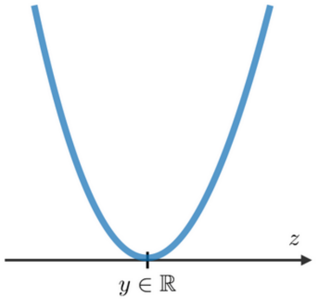
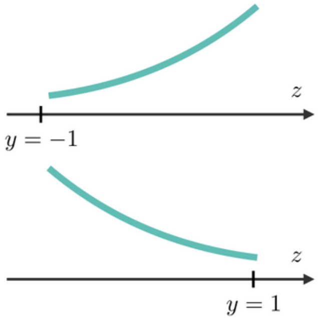
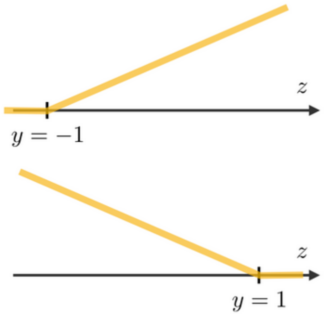
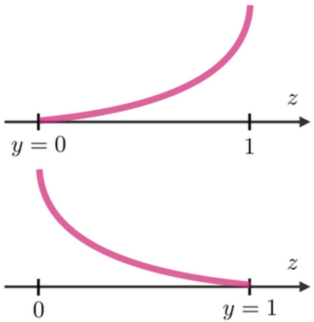
Deep Learning tools: Comet ML [link]

Loss Functions

Loss Function

Image source: by Shervine Amidi

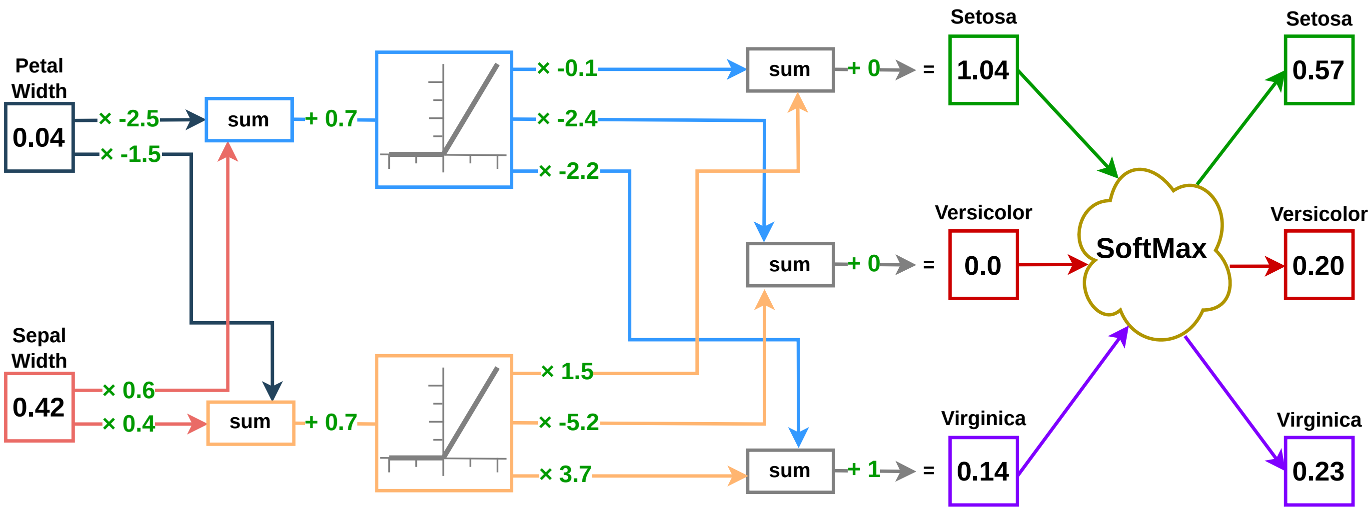
A loss function is a function $L : (z, y) \in \mathbb{R} \times Y \to L(z, y) \in R$ that takes as inputs the predicted value z corresponding to the real data value y and outputs how different they are.

Least squared error	Logistic loss	Hinge loss	Cross-entropy
$\frac{1}{2}(y-z)^2$	$\log(1+\exp(-yz))$	$\max(0, 1-yz)$	$-\Big[y\log(z)+(1-y)\log(1-z)\Big]$
			
Linear regression	Logistic regression	SVM	Neural Network

Cross Entropy

Example inspired by: Josh Starmer's video

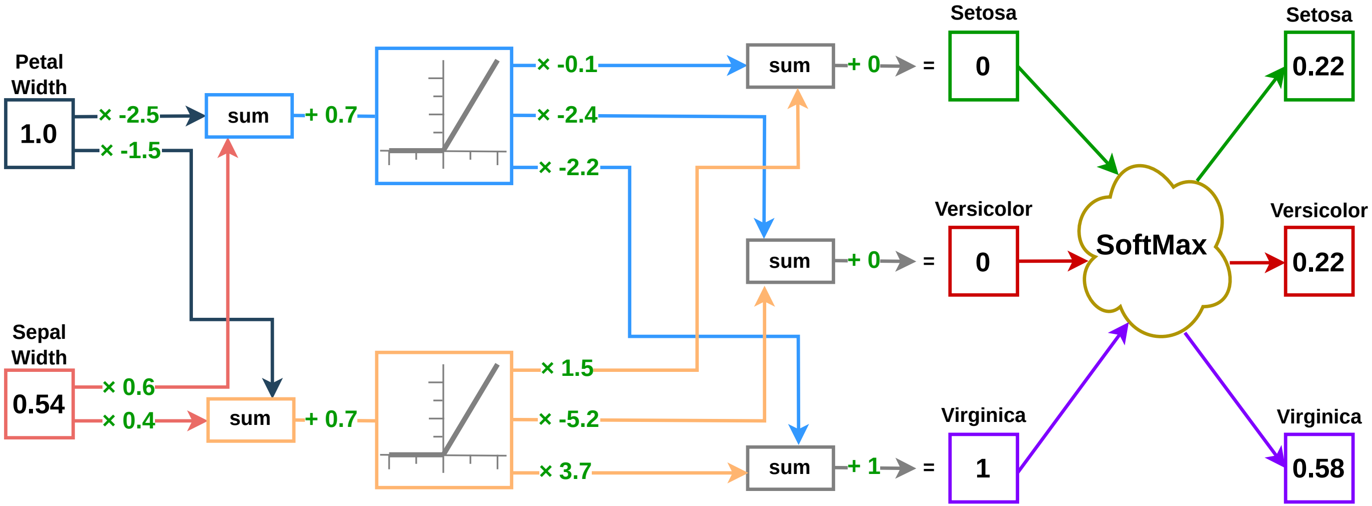
Petal Width	Sepal Width	Species	$"p"$	Cross Entropy
0.04	**0.42**	**Setosa**	**0.57**	$-\log("p")$ = **0.56**
1.0	0.54	Virginica	0.58	$-\log("p")$ = 0.54
0.50	0.37	Versicolor	0.52	$-\log("p")$ = 0.65



Cross Entropy

Example inspired by: Josh Starmer's video

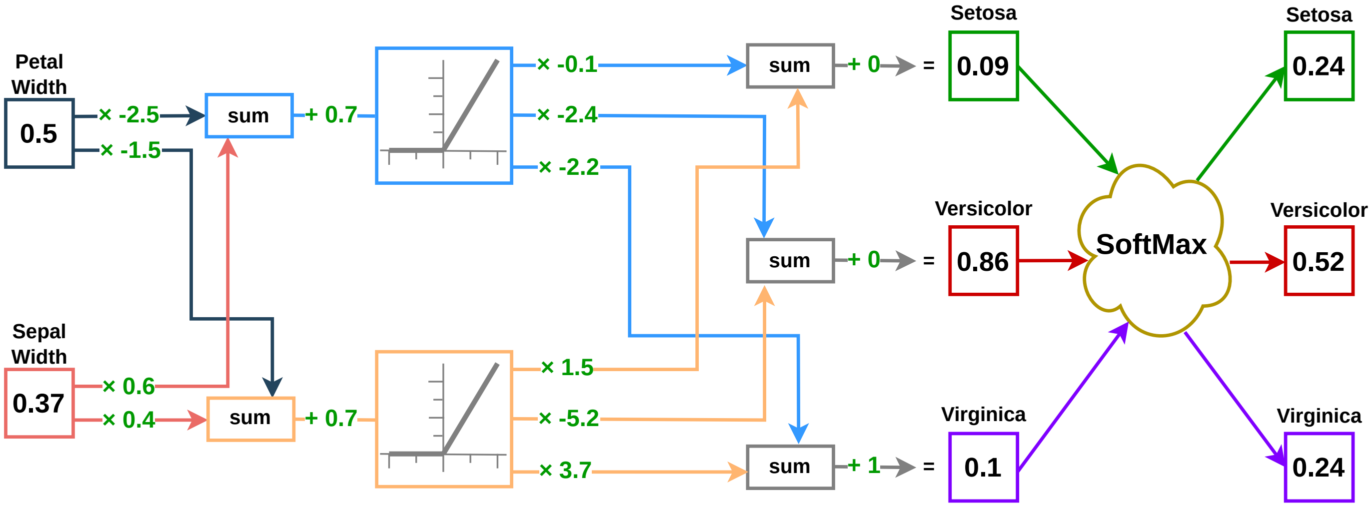
Petal Width	Sepal Width	Species	$"p"$	Cross Entropy
0.04	0.42	Setosa	0.57	$-\log("p")$ = 0.56
1.0	**0.54**	**Virginica**	**0.58**	$-\log("p")$ = **0.54**
0.50	0.37	Versicolor	0.52	$-\log("p")$ = 0.65



Cross Entropy

Example inspired by: Josh Starmer's video

Petal Width	Sepal Width	Species	$"p"$	Cross Entropy
0.04	0.42	Setosa	0.57	$-\log("p")$ = 0.56
1.0	0.54	Virginica	0.58	$-\log("p")$ = 0.54
0.50	**0.37**	**Versicolor**	**0.52**	$-\log("p")$ **= 0.65**



Cross Entropy

Petal Width	Sepal Width	Species	$"p"$	Cross Entropy
0.04	0.42	Setosa	0.57	$-\log("p")$ = **0.56**
1.0	0.54	Virginica	0.58	$-\log("p")$ = **0.54**
0.50	0.37	Versicolor	0.52	$-\log("p")$ = **0.65**

Total Cross Entropy = 0.56 + 0.54 + 0.65 = 1.75

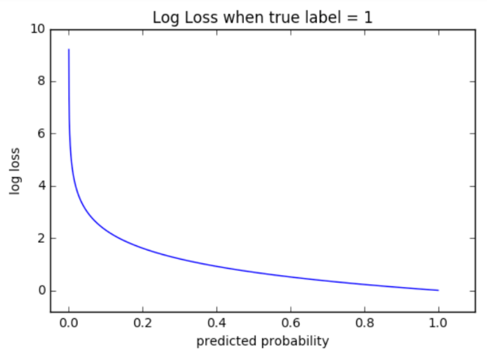


Image source: ml-cheatsheet.readthedocs.io

Usage of common Loss Functions

Mean Absolute Error (MAE) Loss: $L(x, y) = |x - y|$

```
# MAE Loss
import torch
import torch.nn as nn

input = torch.randn(3, 5, requires_grad=True)
target = torch.randn(3, 5)
mae_loss = nn.L1Loss()
output = mae_loss(input, target)
output.backward()

print('output: ', output)
```

```
output:  tensor(1.2850, grad_fn=<L1LossBackward>)
```

When could it be used?

- Regression problems. It is considered to be more robust to outliers.

Slides 36-42 are based on the PyTorch documentation and on the neptune.ai guide.

Usage of common Loss Functions

Mean Squared Error (MSE) Loss: $L(x, y) = (x - y)^2$

```
# MSE Loss
import torch
import torch.nn as nn

input = torch.randn(3, 5, requires_grad=True)
target = torch.randn(3, 5)
mse_loss = nn.MSELoss()
output = mse_loss(input, target)
output.backward()

print('output: ', output)
```

```
output:  tensor(2.3280, grad_fn=<MseLossBackward>)
```

When could it be used?

- Regression problems. MSE is the default loss function for most Pytorch regression problems

Usage of common Loss Functions

Negative Log-Likelihood (NLL) Loss: $L(x, y) = \{l_1, ..., l_N\}^T$, where $l_N = -w_{y_n} x_{n,y_n}$. Softmax required!

```
# NLL Loss
import torch
import torch.nn as nn

# size of input (N x C) is = 3 x 5
input = torch.randn(3, 5, requires_grad=True)
# every element in target should have 0 <= value < C
target = torch.tensor([1, 0, 4])
m = nn.LogSoftmax(dim=1)
nll_loss = nn.NLLLoss()
output = nll_loss(m(input), target)
output.backward()

print('output: ', output)
```

```
output:  tensor(2.9472, grad_fn=<NllLossBackward>)
```

When could it be used?

- Multi-class classification problems

Usage of common Loss Functions

Cross Entropy Loss: $L(x, y) = -[y \cdot \log(x) + (1-y) \cdot \log(1-x)]$

```
# Cross Entropy Loss
import torch
import torch.nn as nn

input = torch.randn(3, 5, requires_grad=True)
target = torch.empty(3, dtype=torch.long).random_(5)
cross_entropy_loss = nn.CrossEntropyLoss()
output = cross_entropy_loss(input, target)
output.backward()

print('output: ', output)
```

```
output:  tensor(1.0393, grad_fn=<NllLossBackward>)
```

When could it be used?

- Binary classification tasks (default loss for classification in PyTorch)

Usage of common Loss Functions

Hinge Embedding Loss: $L(x, y) = \begin{cases} x, & \text{if } y = 1 \\ \max\{0, \Delta - x\}, & \text{if } y = -1 \end{cases}$

```
# Hinge Embedding Loss
import torch
import torch.nn as nn

input = torch.randn(3, 5, requires_grad=True)
target = torch.randn(3, 5)
hinge_loss = nn.HingeEmbeddingLoss()
output = hinge_loss(input, target)
output.backward()

print('output: ', output)
```

```
output:  tensor(1.2183, grad_fn=<MeanBackward0>)
```

When could it be used?

- Classification problems, especially when determining if two inputs are dissimilar or similar
- Learning nonlinear embeddings or semi-supervised learning tasks

Usage of common Loss Functions

Margin Ranking Loss: $L(x_1, x_2, y) = \max(0, -y \cdot (x_1 - x_2) + \text{margin})$

```
# Margin Ranking Loss
import torch
import torch.nn as nn

input_one = torch.randn(3, requires_grad=True)
input_two = torch.randn(3, requires_grad=True)
target = torch.randn(3).sign()

ranking_loss = nn.MarginRankingLoss()
output = ranking_loss(input_one, input_two, target)
output.backward()

print('output: ', output)
```

```
output:  tensor(1.3324, grad_fn=<MeanBackward0>)
```

When could it be used?

- Ranking problems

Usage of common Loss Functions

Kullback-Leibler Divergence (KLD) Loss: $L(x, y) = y \cdot (\log y - x)$

```
# Kullback-Leibler Divergence Loss
import torch
import torch.nn as nn

input = torch.randn(2, 3, requires_grad=True)
target = torch.randn(2, 3)
kl_loss = nn.KLDivLoss(reduction = 'batchmean')
output = kl_loss(input, target)
output.backward()

print('output: ', output)
```

```
output:  tensor(0.8774, grad_fn=<DivBackward0>)
```

When could it be used?

- Approximating complex functions
- Multi-class classification tasks
- If you want to make sure that the distribution of predictions is similar to that of training data

Best practices for Loss Functions

Limitations of loss functions:

A loss function, more or less, cannot totally reflect the our objectives when training a model, in essence. In fact, we have some prior knowledge about "What we want to optimize" and we try to model our prior knowledge by designing some loss function by hand.

Practical uses of loss functions:

- Use a composite loss function, i.e. a composition of many different loss functions, to train your model.

Designing new loss functions:

- What is the aspect you want the model to learn to optimize, e.g. to address the problem of class imbalance, etc.
- Try to mathematically model your objective by a function, whose inputs are the predicted segmentation mask and the corresponding ground-truth segmentation mask.

Based on the CoTAI lecture

Biological NNs and Artificial NNs

Biological NNs and Artificial NNs

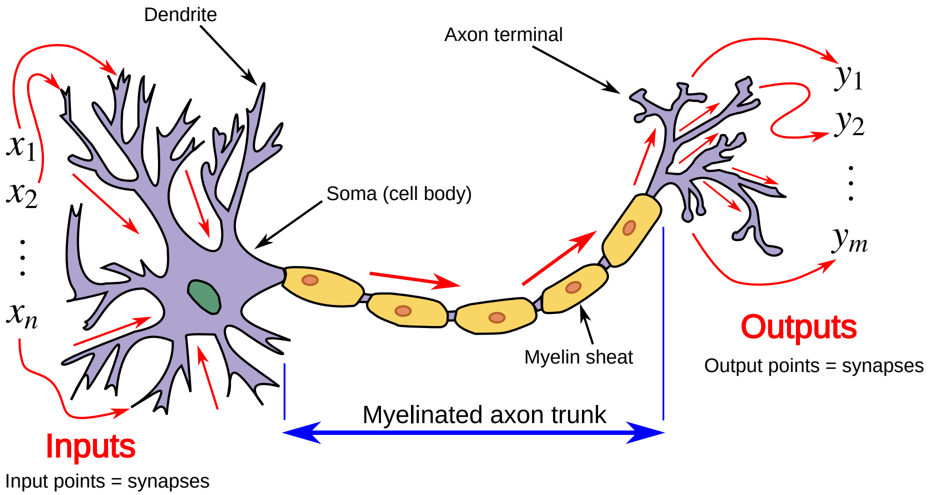
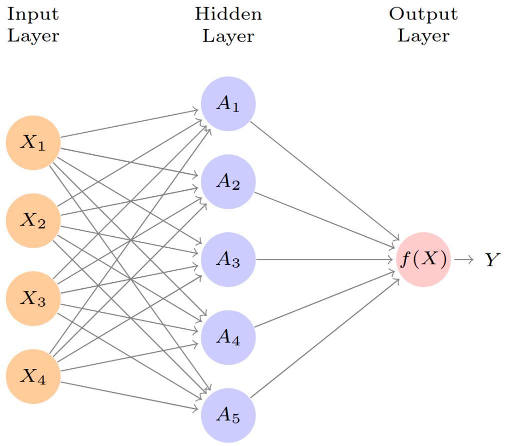


Image source: https://commons.wikimedia.org/wiki/File:Neuron3.svg



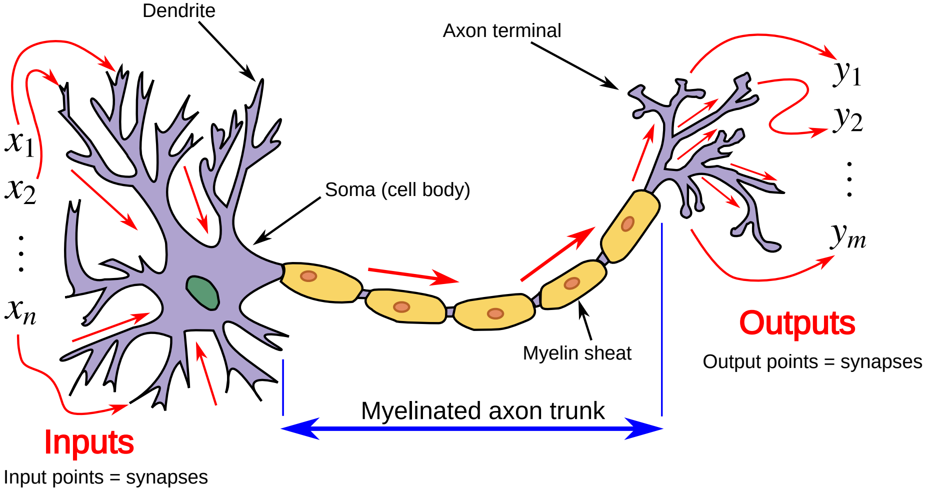
Feed-forward NN. Image source: ISLR Fig. 10.1

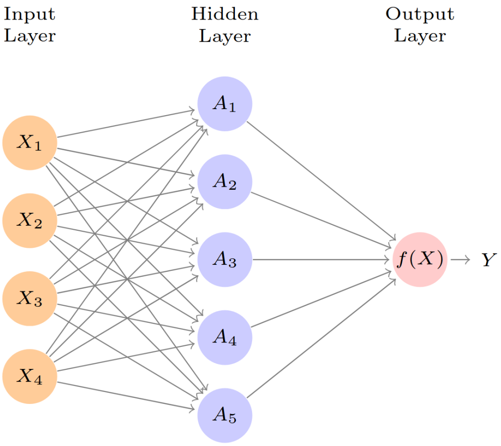
- Both have *multiple* **inputs** from and **outputs** to other neurons
- Both use **activation** of the neurons
- Both are **designed to learn** an optimal behavior

In ANN:

- "**dendrites**" are connections, which carry information (learnt coefficients)
- "**synapses**" are activation functions, which augment or filter information flow; and "**soma**" acts as the summation function

Biological NNs and Artificial NNs





Feed-forward NN. Image source: ISLR Fig. 10.1

Image source: https://commons.wikimedia.org/wiki/File:Neuron3.svg

Further reading on biological NNs. Christof Koch:

- Biophysics of Computation: Information Processing in Single Neurons
- A model of saliency-based visual attention for rapid scene analysis
- Consciousness & Reality Colloquium Series: Inaugural Lecture

- Neuroscience by Dale Purves et al. (6th ed., 2018)

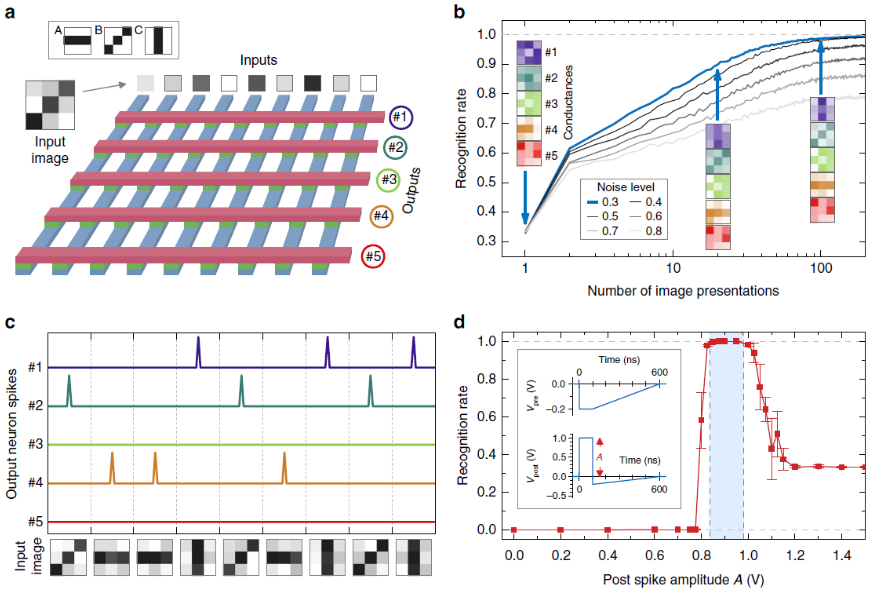
Vyacheslav Dubynin (in russian):

- Мозг и его потребности: От питания до признания (2021)
- Lectures on the YouTube

Biological NNs and Artificial NNs

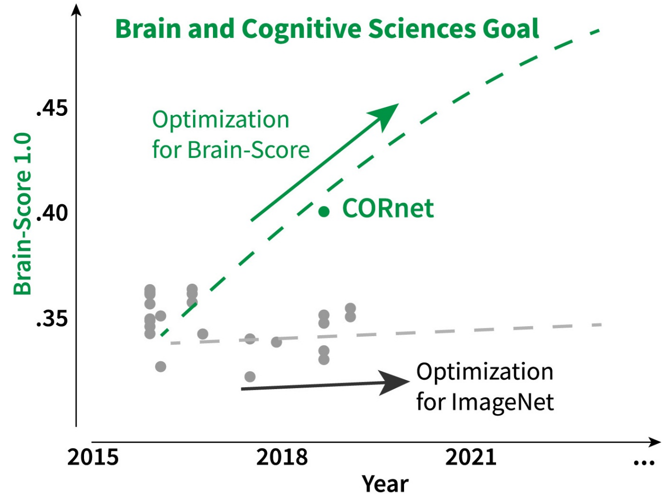
Artificial neural networks are **inspired by** the biological neural networks (BNNs) but most of them are only **loosely based on** the BNNs.

Spiking neural networks are ANNs that more closely mimic natural neural networks



Unsupervised learning with ferroelectric synapses. Image source: *Nature Communications* 8, 14736 (2017)

Brain Score



Integrative Benchmarking to Advance Neurally Mechanistic Models of Human Intelligence. Image source: *Neuron* 108.3 (2020)