3011979 Intro to Deep Learning for Medical Imaging

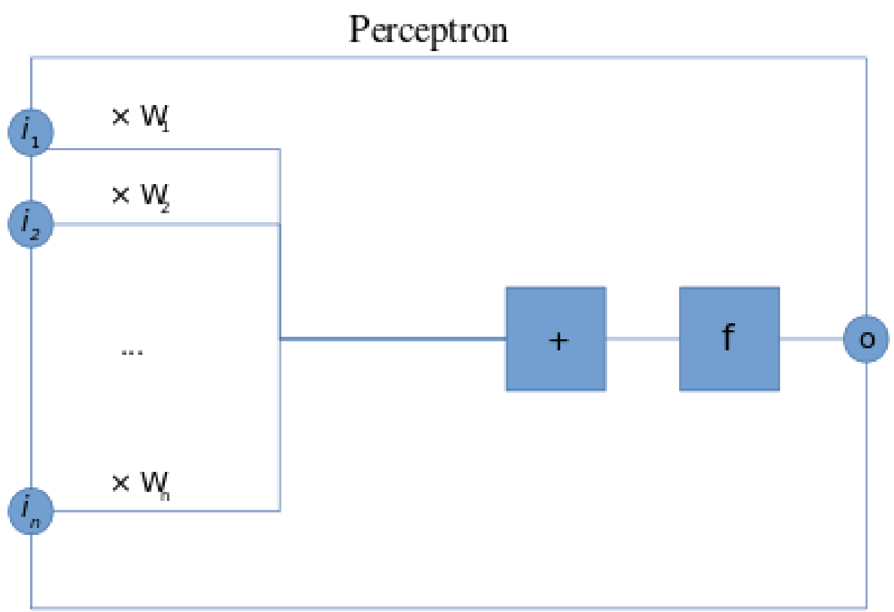
L10: Multilayer perceptron

Apr 9th, 2021

Sira Sriswasdi, Ph.D.
Research Affairs, Faculty of Medicine
Chulalongkorn University

Perceptron

(Single-layer) Perceptron



$$o = f\left(\sum_{k=1}^{n} i_k \cdot W_k\right)$$

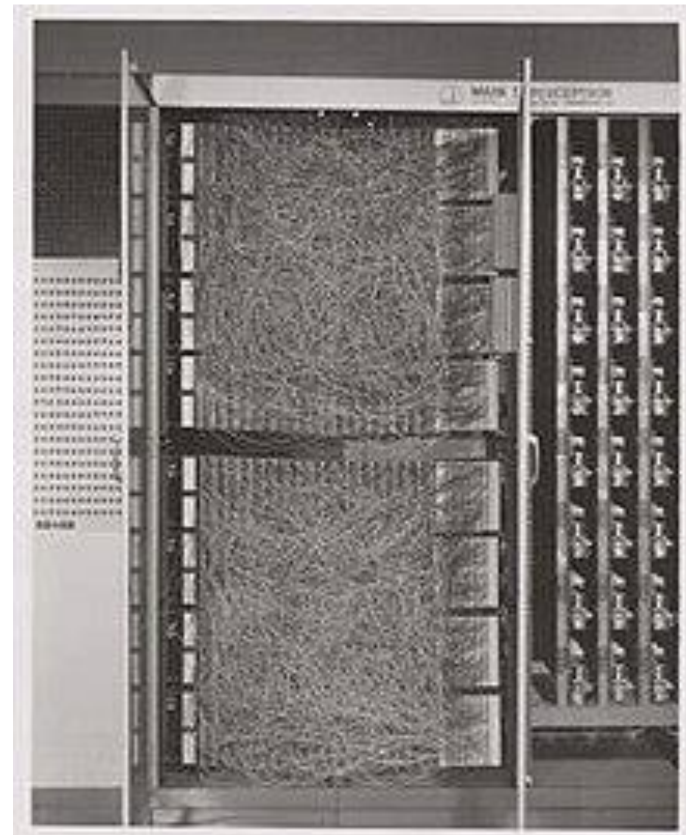


Image from Wikipedia.com

- Essentially linear regression
- But can learn from new data points on the fly

Perceptron learning algorithm

- Initialize weights $(w_1, \ldots, w_n) = 0$
- Receive new data point $(x,\ y)$
- Make prediction for the new data $y' = \sum w_i x_i$
- Update weights:
  - If the model make a mistake, set $w'_i = w_i + \mathrm{y} \cdot x_i$
  - Otherwise, the model stays the same

- Is this procedure guaranteed to converge to a good solution?

Perceptron convergence (Novikoff, 1962)

- Given data sequence $(x_1, y_1), \ldots, (x_m, y_m)$
  - $\|x_i\| \leq R$
  - There is w^* with $\|w^*\| = 1$ such that $y_i(w^* \cdot x_i) \geq \gamma$

- Then, the perceptron learning algorithm makes at most $\left(\frac{R}{\gamma}\right)^2$ mistakes through this data sequence

Proof

- Supposed w_k is the model parameter right before making the k^th mistake and that the mistake occurs on (x_i, y_i)
  - $y_i(x_i \cdot w_k) \leq 0$

- Weight update: $w_{k+1} = w_k + y_i x_i$

- $w_{k+1} \cdot w^* = w_k \cdot w^* + y_i(x_i \cdot w^*) \geq w_k \cdot w^* + \gamma$
  - $w_{k+1} \cdot w^* \geq k\gamma$

- $$\begin{aligned} \|w_{k+1}\|^2 &= \|w_k + y_i x_i\|^2 \\ &= \|w_k\|^2 + 2y_i(x_i \cdot w_k) + \|y_i x_i\|^2 \\ &\leq \|w_k\|^2 + R^2 \end{aligned}$$

Proof (continued)

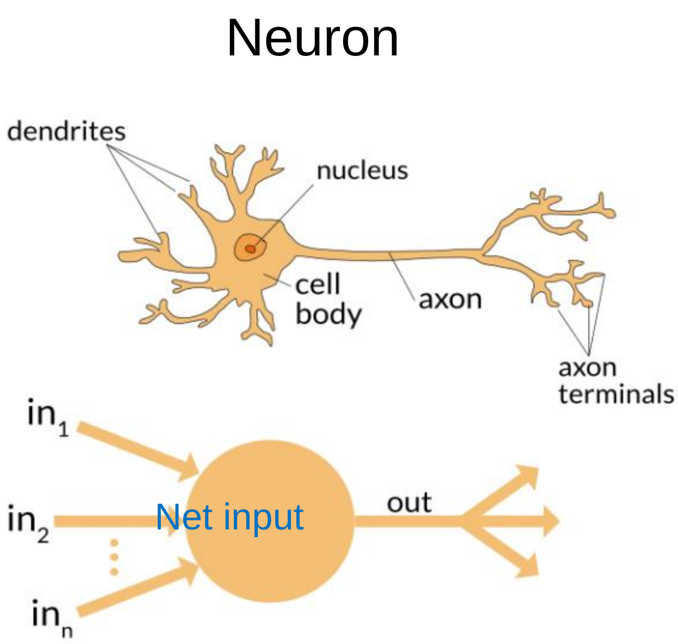
- Result 1: $\|w_{k+1}\|^2 \leq kR^2$
- Result 2: $\|w_{k+1}\|^2 \geq (w_{k+1} \cdot w^*)^2 \geq k^2\gamma^2$
- Together, number of mistake $k \leq \left(\frac{R}{\gamma}\right)^2$

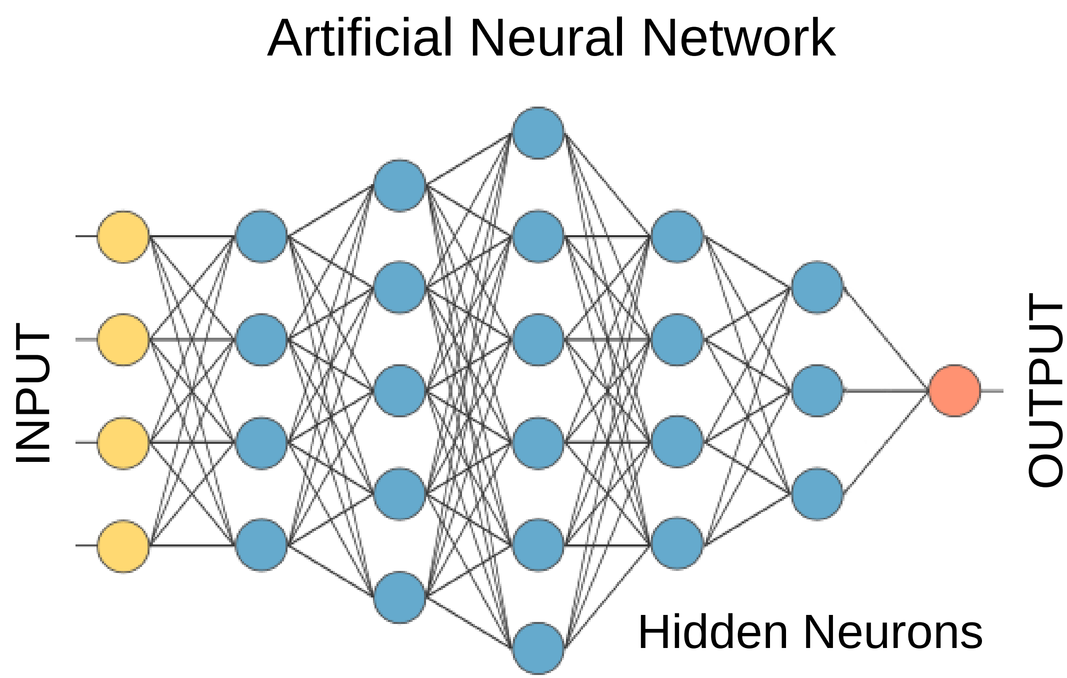
Limitation of single-layer perceptron

- Single-layer perceptron is still a linear model
  - Output is 0 or 1 based on thresholding
  - Activation function

- Simple learning algorithm
  - Cannot be extended to multi-layer architectures
  - Stochastic gradient descent and backpropagation

Multi-layer perceptron





Source: www.decom.ufop.br/imobilis/fundamentos-de-redes-neurais/

- MLP is the basic artificial neural network architecture

Artificial neuron

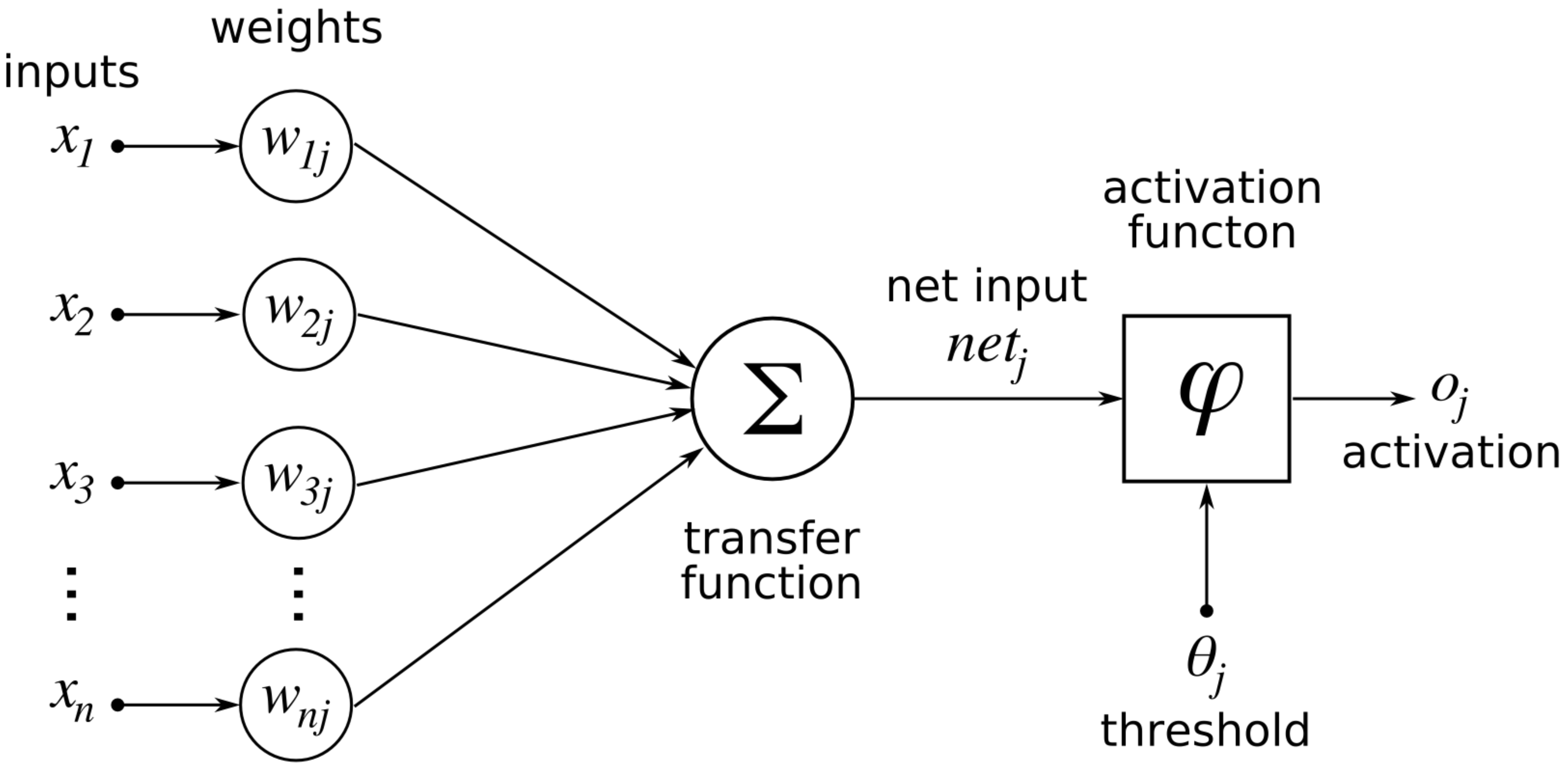
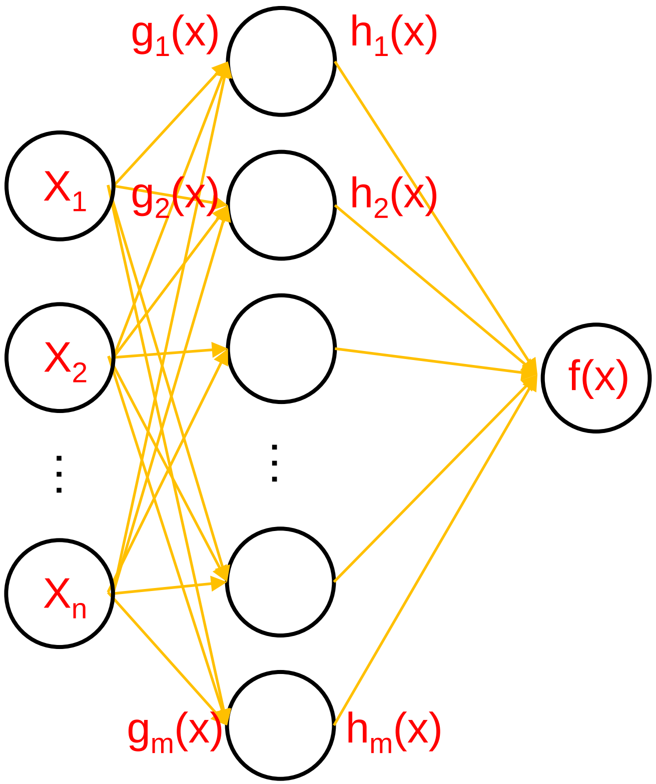


Image from Wikipedia.com

Calculation inside MLP



Neuron input = linear combination

$$g_1(x) = w_{1,1}x_1 + \cdots + w_{1,n}x_n$$

$$\ldots$$

$$g_m(x) = w_{m,1}x_1 + \cdots + w_{m,n}x_n$$

Neuron output = activation function

$$h_1(x) = \frac{1}{1 + e^{-g_1(x)}}$$

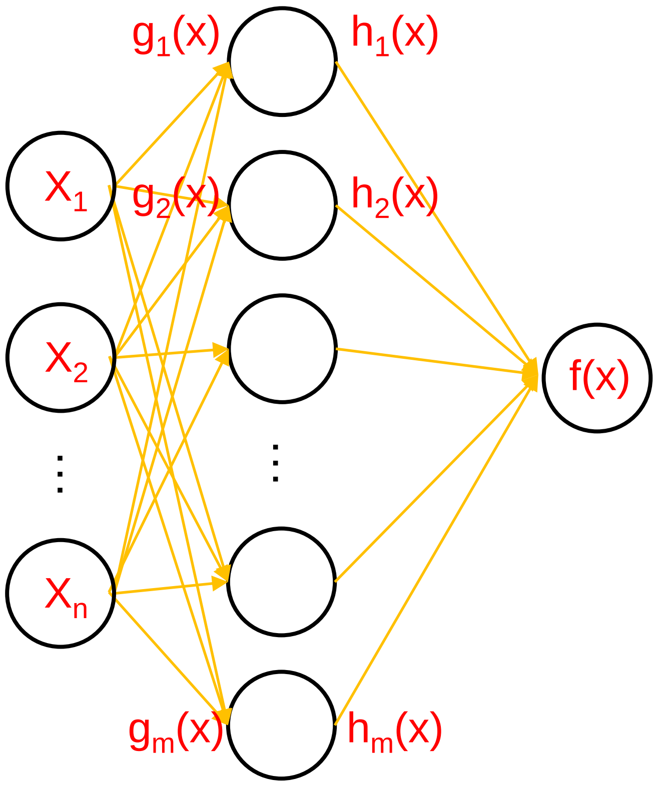
$$\ldots$$

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

Final output

$$f(x) = u_1 h_1(x) + \cdots + u_m h_m(x)$$

MLP model complexity



- How many parameters are there in an MLP with
  - Input layer of size 3
  - One hidden layer of size 5
  - One output layer of size 1

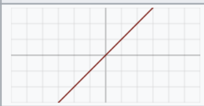
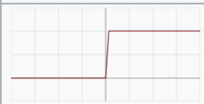
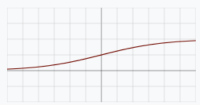
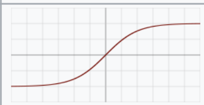
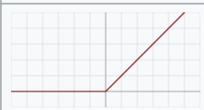
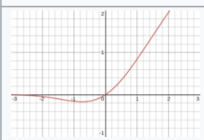
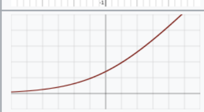
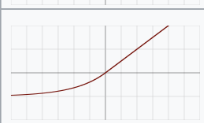
ANN model complexity

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 256, 256, 32)	896
batch_normalization (BatchNo	(None, 256, 256, 32)	128
max_pooling2d (MaxPooling2D)	(None, 85, 85, 32)	0
dropout (Dropout)	(None, 85, 85, 32)	0
conv2d_1 (Conv2D)	(None, 85, 85, 64)	18496
batch_normalization_1 (Batch	(None, 85, 85, 64)	256
conv2d_2 (Conv2D)	(None, 85, 85, 64)	36928
batch_normalization_2 (Batch	(None, 85, 85, 64)	256
max_pooling2d_1 (MaxPooling2	(None, 42, 42, 64)	0
dropout_1 (Dropout)	(None, 42, 42, 64)	0
conv2d_3 (Conv2D)	(None, 42, 42, 128)	73856
batch_normalization_3 (Batch	(None, 42, 42, 128)	512
conv2d_4 (Conv2D)	(None, 42, 42, 128)	147584
batch_normalization_4 (Batch	(None, 42, 42, 128)	512
max_pooling2d_2 (MaxPooling2	(None, 21, 21, 128)	0
dropout_2 (Dropout)	(None, 21, 21, 128)	0
flatten (Flatten)	(None, 56448)	0
dense (Dense)	(None, 1024)	57803776
batch_normalization_5 (Batch	(None, 1024)	4096
dropout_3 (Dropout)	(None, 1024)	0
dense_1 (Dense)	(None, 4)	4100

Total params: 58,091,396
Trainable params: 58,088,516
Non-trainable params: 2,880

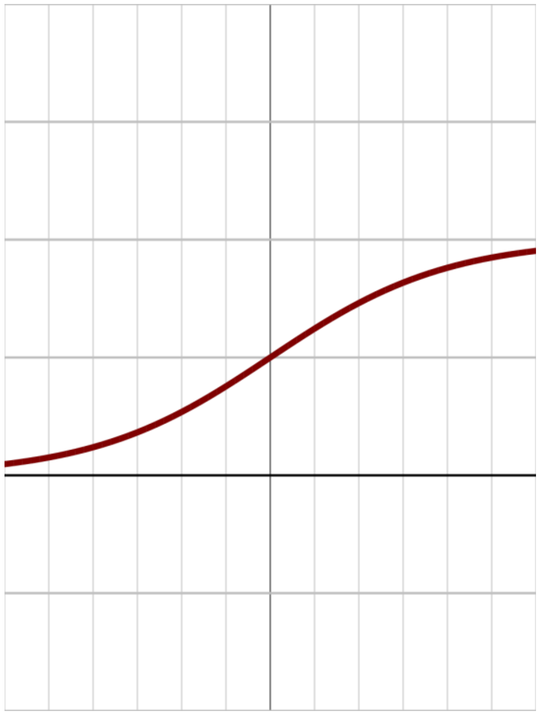
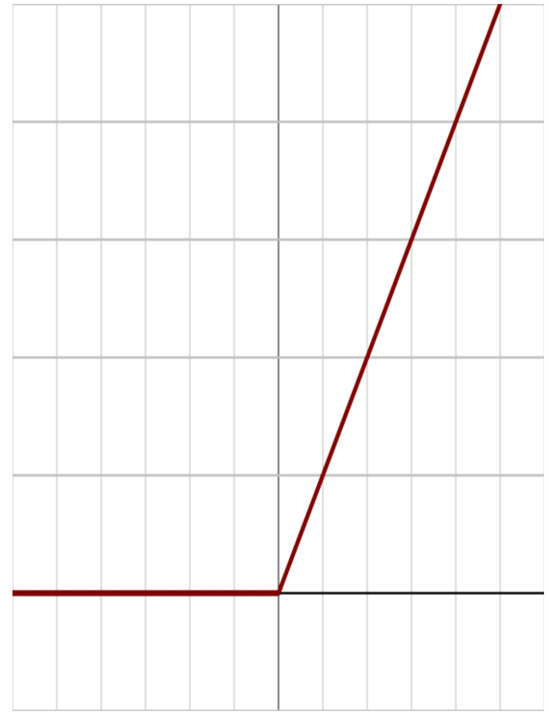
Activation function

Activation function

Name	Plot	Function, $f(x)$	Derivative of f, $f'(x)$	Range
Identity		x	1	$(-\infty, \infty)$
Binary step		$\begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	$\begin{cases} 0 & \text{if } x \neq 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$	$\{0, 1\}$
Logistic, sigmoid, or soft step		$\sigma(x) = \frac{1}{1+e^{-x}}$ [1]	$f(x)(1 - f(x))$	$(0, 1)$
tanh		$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$1 - f(x)^2$	$(-1, 1)$
Rectified linear unit (ReLU)[11]		$\begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases} = \max\{0, x\} = x\mathbf{1}_{x>0}$	$\begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$	$[0, \infty)$
Gaussian Error Linear Unit (GELU)[6]		$\frac{1}{2}x\left(1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right) = x\Phi(x)$	$\Phi(x) + x\phi(x)$	$(-0.17\ldots, \infty)$
Softplus[12]		$\ln(1 + e^x)$	$\frac{1}{1+e^{-x}}$	$(0, \infty)$
Exponential linear unit (ELU)[13]		$\begin{cases} \alpha(e^x - 1) & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$ with parameter α	$\begin{cases} \alpha e^x & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ 1 & \text{if } x = 0 \text{ and } \alpha = 1 \end{cases}$	$(-\alpha, \infty)$

- MLP with identity activation is still a linear model

Slopes of activation functions

- Sigmoid saturates at both ends (slope → 0)
  - Previous default function
- Rectified linear unit produces no response for x < 0
  - Default and popular choice nowadays

Universal approximation (Cybenko, 1989)

Universal Approximation Theorem: Fix a continuous function $\sigma : \mathbb{R} \to \mathbb{R}$ (activation function) and positive integers d, D. The function σ is not a polynomial if and only if, for every continuous function $f : \mathbb{R}^d \to \mathbb{R}^D$ (target function), every compact subset K of $\mathbb{R}^d$, and every $\epsilon > 0$ there exists a continuous function $f_\epsilon : \mathbb{R}^d \to \mathbb{R}^D$ (the layer output) with representation

$$f_\epsilon = W_2 \circ \sigma \circ W_1,$$

where W_2, W_1 are composable affine maps and $\circ$ denotes component-wise composition, such that the approximation bound

$$\sup_{x \in K} \|f(x) - f_\epsilon(x)\| < \varepsilon$$

holds for any ϵ arbitrarily small (distance from f to f_ϵ can be infinitely small).

- MLP with just one hidden layer and non-polynomial activation function can approximate any continuous function with arbitrarily high precision

The search for new activation function

Computer Science > Neural and Evolutionary Computing

[Submitted on 16 Oct 2017 (v1), last revised 27 Oct 2017 (this version, v2)]

Searching for Activation Functions

Prajit Ramachandran, Barret Zoph, Quoc V. Le

The choice of activation functions in deep networks has a significant effect on the training dynamics and task performance. Currently, the most successful and widely-used activation function is the Rectified Linear Unit (ReLU). Although various hand-designed alternatives to ReLU have been proposed, none have managed to replace it due to inconsistent gains. In this work, we propose to leverage automatic search techniques to discover new activation functions. Using a combination of exhaustive and reinforcement learning-based search, we discover multiple novel activation functions. We verify the effectiveness of the searches by conducting an empirical evaluation with the best discovered activation function. Our experiments show that the best discovered activation function, $f(x) = x \cdot \text{sigmoid}(\beta x)$, which we name Swish, tends to work better than ReLU on deeper models across a number of challenging datasets. For example, simply replacing ReLUs with Swish units improves top-1 classification accuracy on ImageNet by 0.9\% for Mobile NASNet-A and 0.6\% for Inception-ResNet-v2. The simplicity of Swish and its similarity to ReLU make it easy for practitioners to replace ReLUs with Swish units in any neural network.

- ReLU is simple yet powerful

Learning algorithm

MLP learning algorithm

- Define loss function $L(y', y)$
  - Cross-entropy for classification problem
  - MSE for regression problem

- Stochastic gradient descent
  - Stochastic = gradient calculated from subset of data

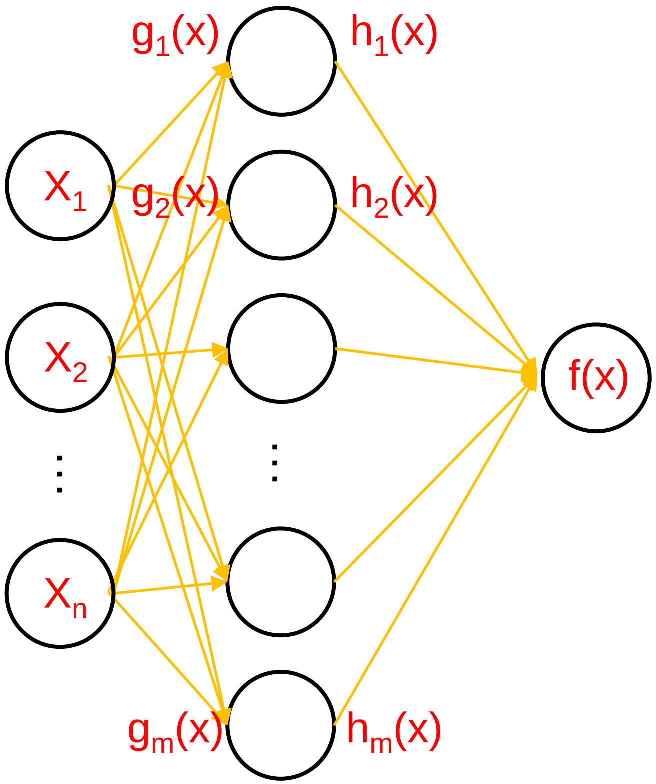
- Example:
  - Final output: $f(x) = u_1 h_1(x) + \cdots + u_m h_m(x)$
  - Sigmoid activation: $h_i(x) = \frac{1}{1+e^{-g_i(x)}}$
  - $g_i(x) = w_{i,1} x_1 + \cdots + w_{i,n} x_n$
  - MSE loss: $L(f(x), y) = \frac{1}{2} \|f(x) - y\|^2$

Gradient calculation with chain rule

- Setup:
  - Final output: $f(x) = u_1 h_1(x) + \cdots + u_m h_m(x)$
  - Sigmoid activation: $h_i(x) = \frac{1}{1+e^{-g_i(x)}}$
  - $g_i(x) = w_{i,1}x_1 + \cdots + w_{i,n}x_n$
  - MSE loss: $L(f(x), y) = \frac{1}{2}\|f(x) - y\|^2$

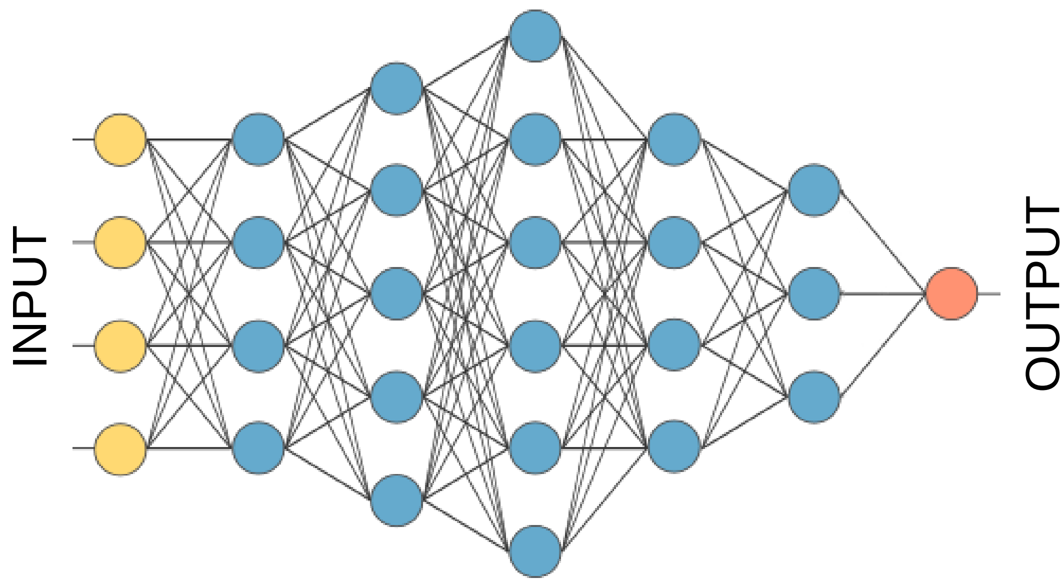
- Gradients:
  - $\frac{\delta L}{\delta u_i} = \frac{\delta L}{\delta f}\frac{\delta f}{\delta u_i} = (f(x) - y) \cdot h_i(x)$
  - $\frac{\delta L}{\delta w_{i,j}} = \frac{\delta L}{\delta f}\frac{\delta f}{\delta h_i}\frac{\delta h_i}{\delta w_{i,j}} = (f(x) - y) \cdot u_i \cdot \frac{\delta h_i}{\delta w_{i,j}}$
  - $\frac{\delta h_i}{\delta w_{i,j}} = \frac{\delta h_i}{\delta g_i}\frac{\delta g_i}{\delta w_{i,j}} = g_i(\mathrm{x})(1 - g_i(\mathrm{x}))\ x_j$

Backpropagation



- Work backward from output layer to input layer
- Memorize all parameters and input/output values into every neuron
- Compute gradient for each parameter using the chain rule
- How many terms are there in a chain rule going back through one hidden layer?
  - What would happen if each term is much smaller than 1?

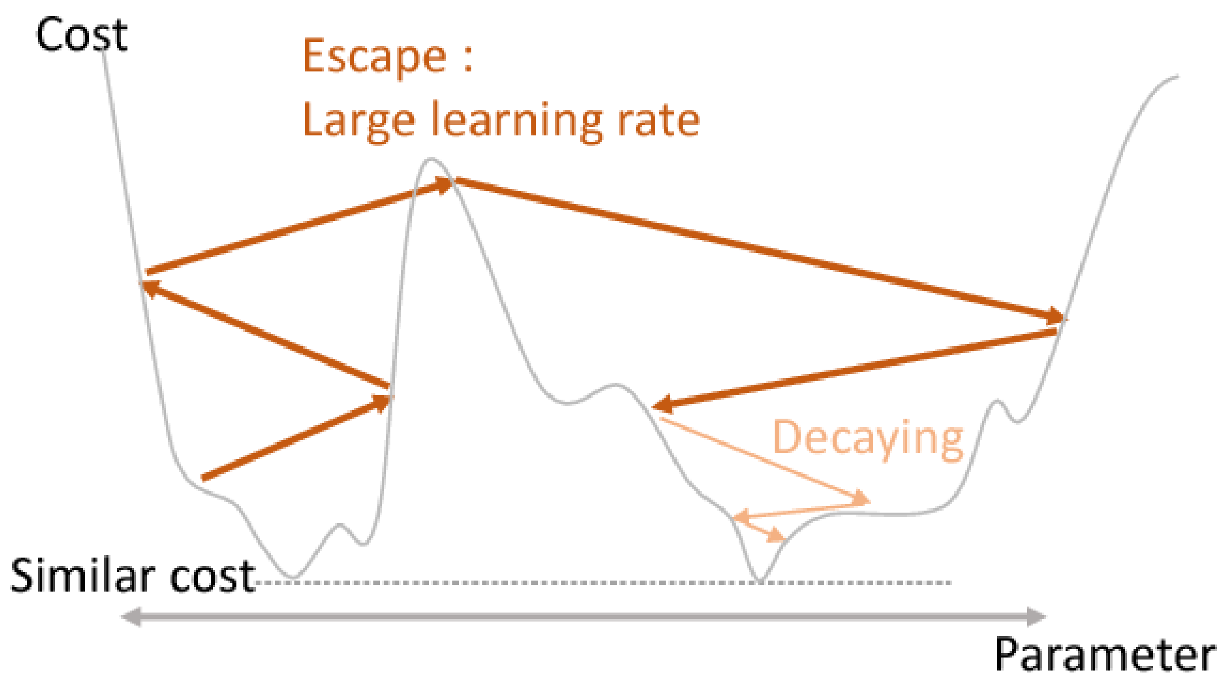
Vanishing gradient problem



- For very deep neural network, the magnitude of gradient can approach zero for parameters of the early layer of the network
- Depend on activation function
- Can be solved via architecture (next week)

Optimization of ANN

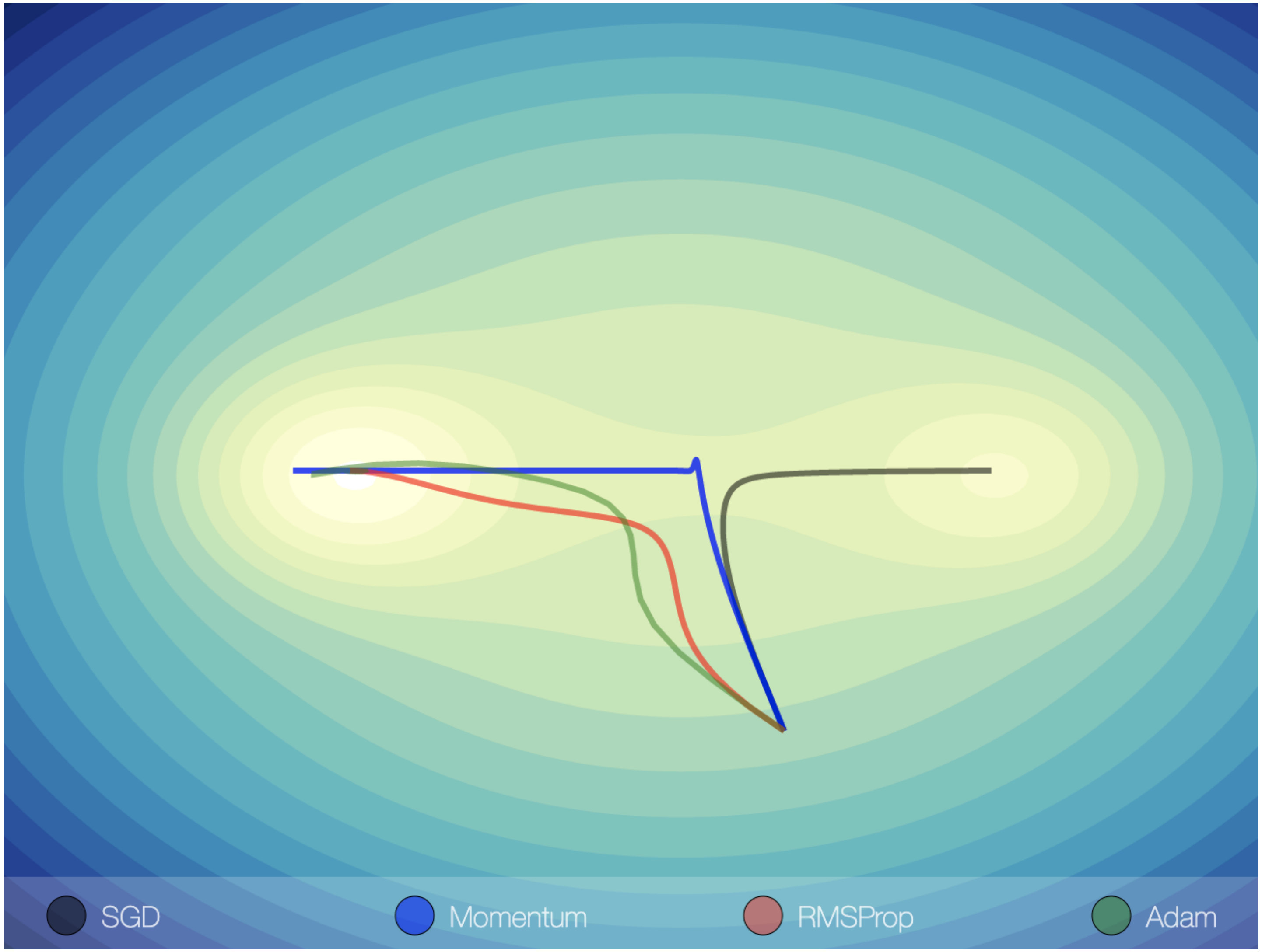
Gradient descent for ANN



Seong et al. “Toward flatter loss surface via nonmonotonic learning rate scheduling” UAI 2018

- Neural network has >1,000,000 parameters
- Many local optima can trap gradient descent
- Start from multiple initial model parameters
- Allow learning rate to switch

Various optimizers



Seong et al. “Toward flatter loss surface via nonmonotonic learning rate scheduling” UAI 2018

Effect of sample batches on gradient descent

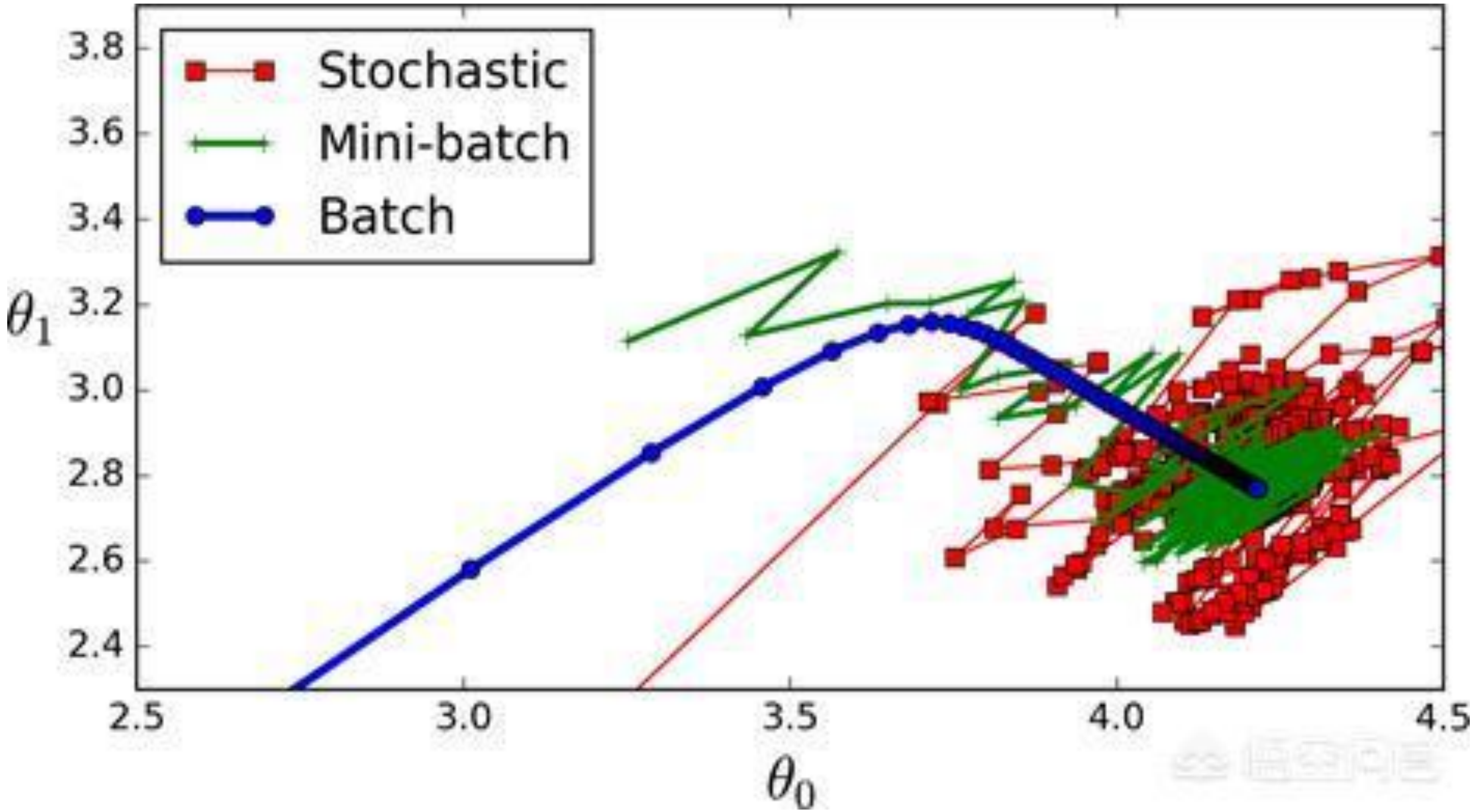
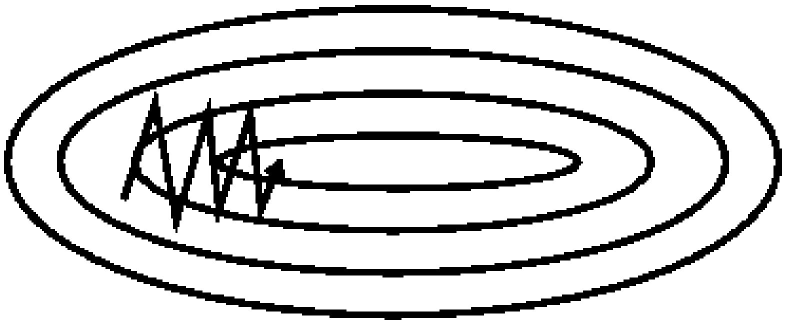


Image from programmersought.com

Gradient descent with momentum

Without momentum



With momentum

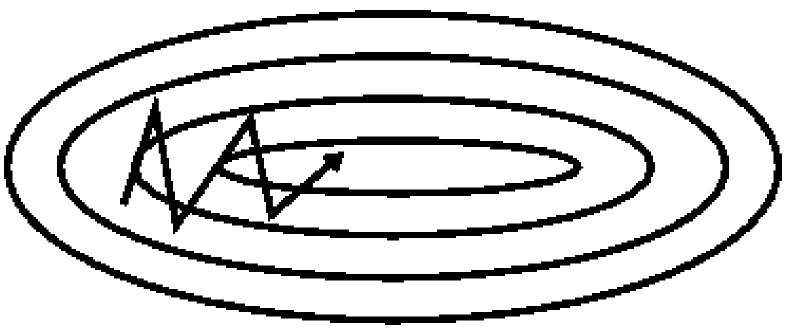


Image from ruder.io

- Gradient descent
  - $w^{t+1} = w^t - \eta \nabla_w(w^t; \text{data})$
  - No memory in update direction from $\nabla_w(w^t; \text{data})$ to $\nabla_w(w^{t+1}; \text{data})$
- Momentum
  - Previous update (velocity) influence the next update (velocity)
  - $v^{t+1} = \gamma v^t + \eta \nabla_w(w^t; \text{data})$
  - $w^{t+1} = w^t - v^{t+1}$

Nesterov momentum

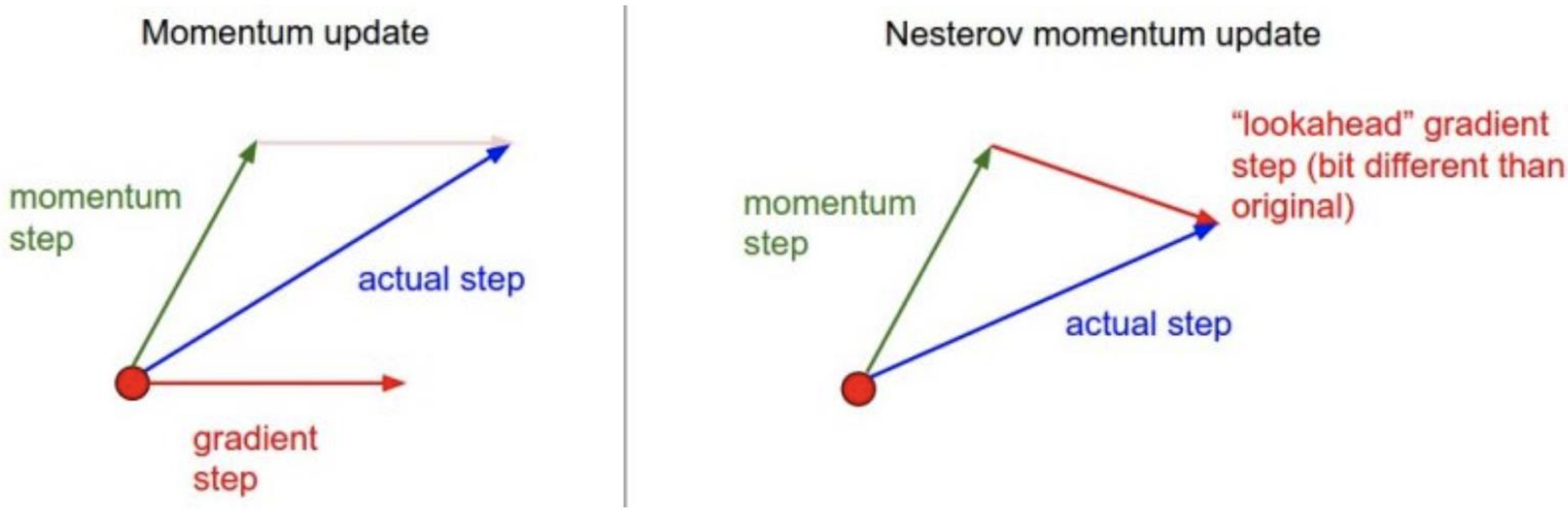


Image from cs231n.github.io

- Compute gradient using the expected future parameter instead of current parameter
  - $v^{t+1} = \gamma v^t + \eta \nabla_w (w^t - \gamma v^t; \mathrm{data})$
  - $w^{t+1} = w^t - v^{t+1}$
- Perform better than regular momentum in practice

Per-parameter learning rate

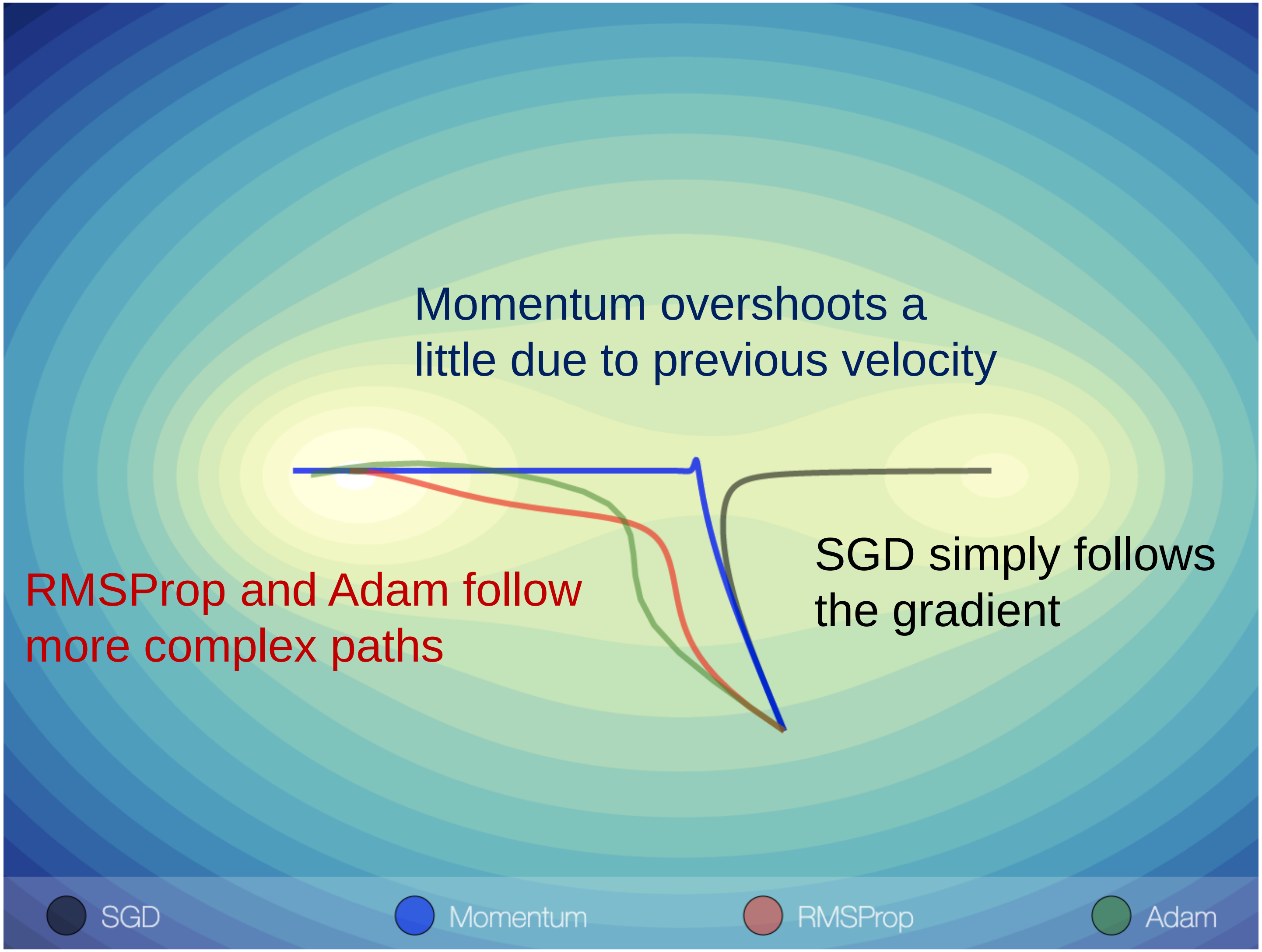
- Adagrad (Adaptive Gradient Descent)
  - Keep track of total update size history for each parameter
    - $s^{t+1} = s^t + (\nabla_w(w^t; \text{data}))^2$
  - Parameter with small updates are given more priority
    - $w^{t+1} = \frac{w^t - \eta \nabla_w(w^t; \text{data})}{\sqrt{s^{t+1}} + \varepsilon}$
    - As s^{t+1} gets larger, the update sizes get smaller

- RMSProp
  - Keep track of update size history, giving more weight to recent ones
    - $s^{t+1} = \alpha \cdot s^t + (1 - \alpha)(\nabla_w(w^t; \text{data}))^2$
    - α is the decay rate
    - Address the problem of small update sizes in Adagrad

Adam optimizer

- Combine momentum and per-parameter learning rate together
- $m^{t+1} = \beta_1 m^t + (1 - \beta_1) \cdot \nabla_w$
- $s^{t+1} = \beta_2 s^t + (1 - \beta_2)(\nabla_w)^2$
- $w^{t+1} = \frac{w^t - \eta \nabla_w (w^t;\text{data})}{\sqrt{s^{t+1}} + \varepsilon}$

Optimizer behaviors



https://emiliendupont.github.io/2018/01/24/optimization-visualization/

ANN in scikit-learn

sklearn.neural_network.MLPClassifier

class `sklearn.neural_network.`**`MLPClassifier`**(*hidden_layer_sizes=100, activation='relu', *, solver='adam', alpha=0.0001, batch_size='auto', learning_rate='constant', learning_rate_init=0.001, power_t=0.5, max_iter=200, shuffle=True, random_state=None, tol=0.0001, verbose=False, warm_start=False, momentum=0.9, nesterovs_momentum=True, early_stopping=False, validation_fraction=0.1, beta_1=0.9, beta_2=0.999, epsilon=1e-08, n_iter_no_change=10, max_fun=15000*) [source]

Multi-layer Perceptron classifier.

This model optimizes the log-loss function using LBFGS or stochastic gradient descent.

New in version 0.18.

Parameters:

hidden_layer_sizes : *tuple, length = n_layers - 2, default=(100,)*

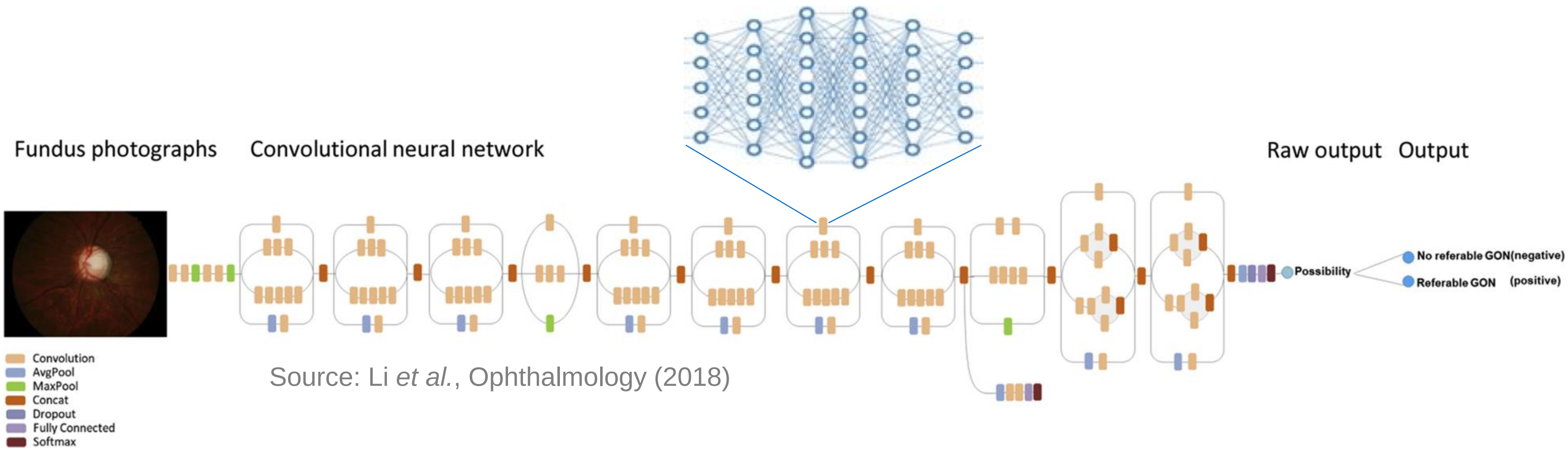
The ith element represents the number of neurons in the ith hidden layer.

activation : *{'identity', 'logistic', 'tanh', 'relu'}, default='relu'*

Activation function for the hidden layer.

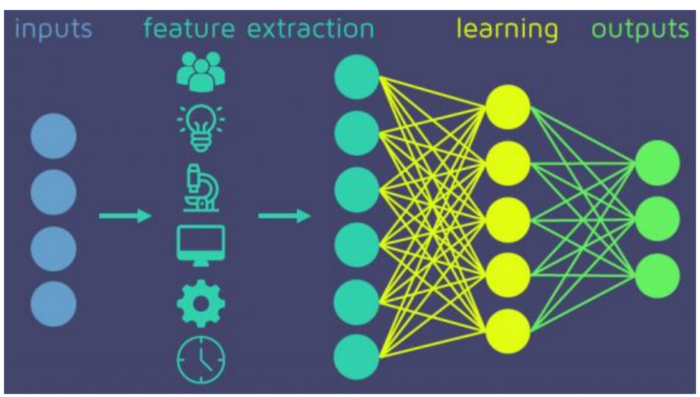
Deep learning

Deep learning is deep ANN

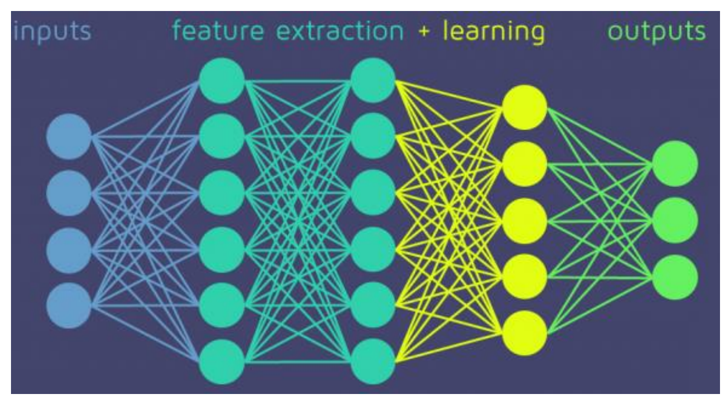


Source: Li *et al.*, Ophthalmology (2018)

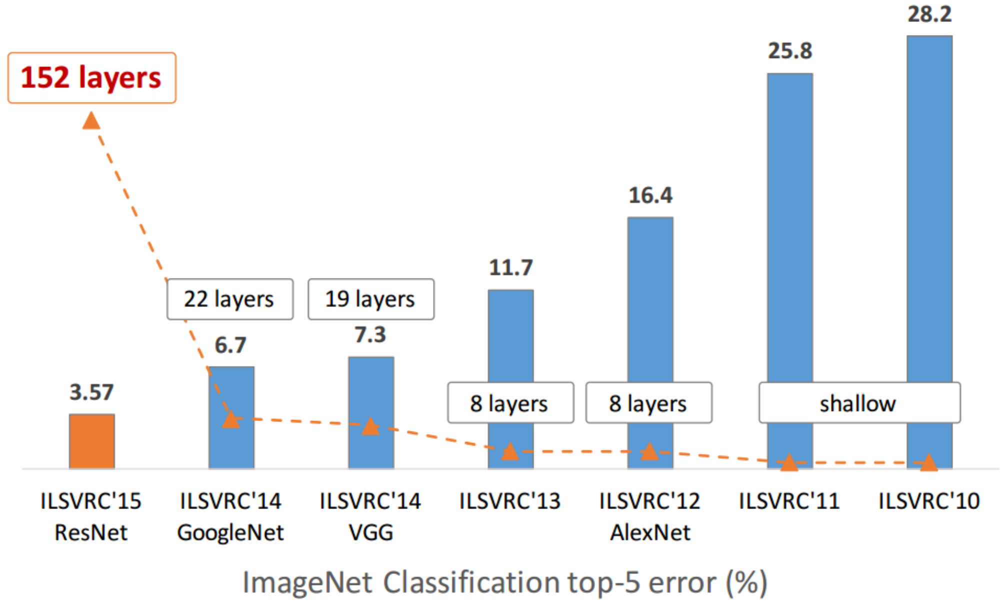
Traditional

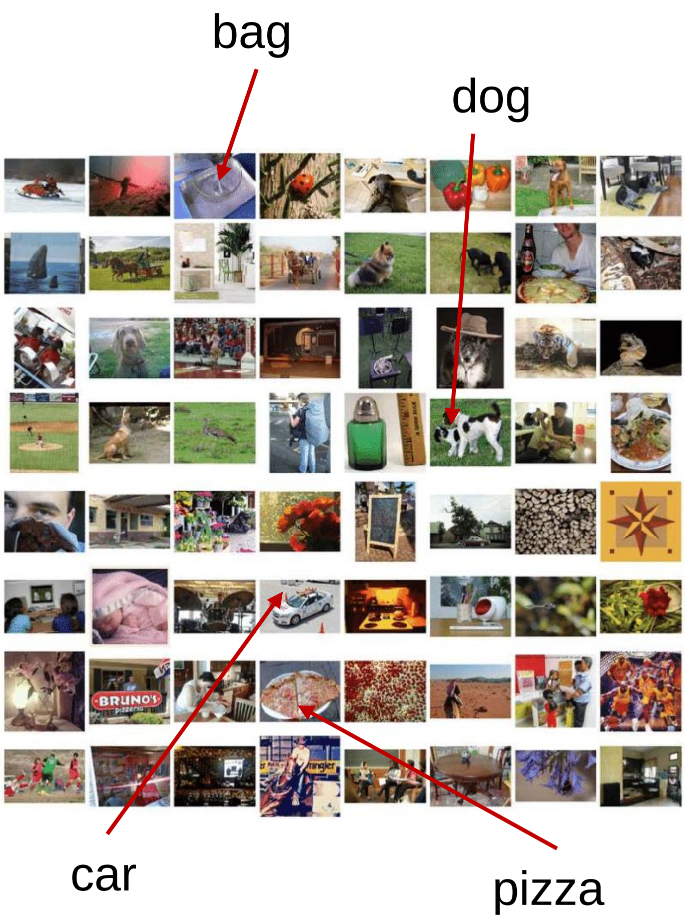


End-to-end



ImageNet





- Superhuman performance with deep ANN and millions of data points

Graphical processing unit (GPU)

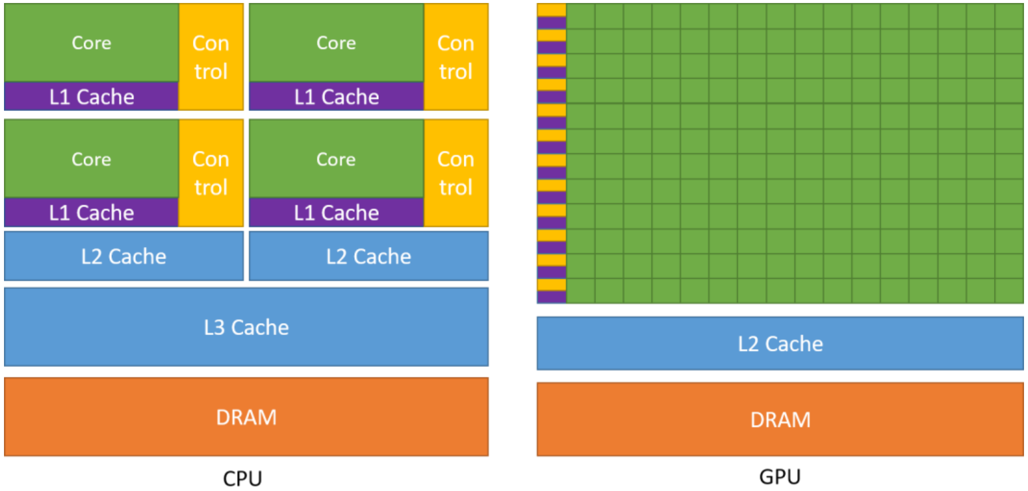


Image from analyticsvidhya.com

- CPU = few high capability compute unit
- GPU = swarm of low capability compute unit

Any question?