Neural network architectures (part one: deep NNs & CNNs)

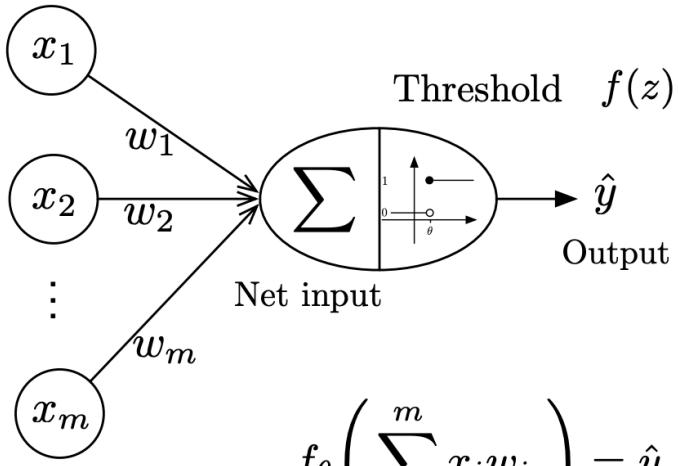
Materials for this Section:

- I. Goodfellow , Y. Bengio, and A. Courville, *Deep Learning, Chapters 6-10:* https://www.deeplearningbook.org
- https://playground.tensorflow.org

Deep neural networks

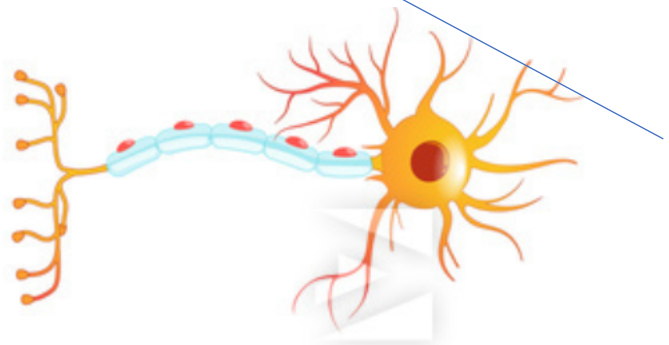
simplest neural network, *perceptron*
-- a computational model of a biological neuron

- single input *layer* with multiple input *units* and single output



Threshold $f(z) = \begin{cases} 0, & z \leq \theta \\ 1, & z > \theta \end{cases}$

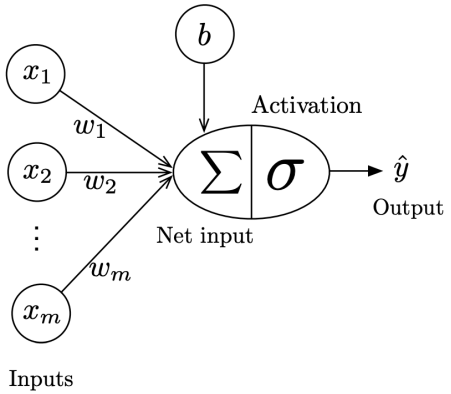
$$f_\theta\left(\sum_{i=1}^{m} x_i w_i\right) = \hat{y}$$



activation with threshold θ

- It is convenient to include threshold as additional *bias* unit

$$\hat{y} = \begin{cases} 0, & z - \theta \leq 0 \\ 1, & z - \theta > 0 \end{cases}$$

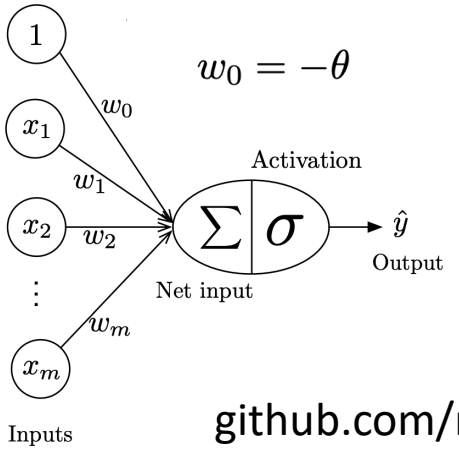


"separate" bias unit

$$\sigma\left(\sum_{i=1}^{m} x_i w_i + b\right) = \sigma\left(\mathbf{x}^T \mathbf{w} + b\right) = \hat{y}$$

$$\sigma(z) = \begin{cases} 0, & z \leq 0 \\ 1, & z > 0 \end{cases}$$

$$b = -\theta$$



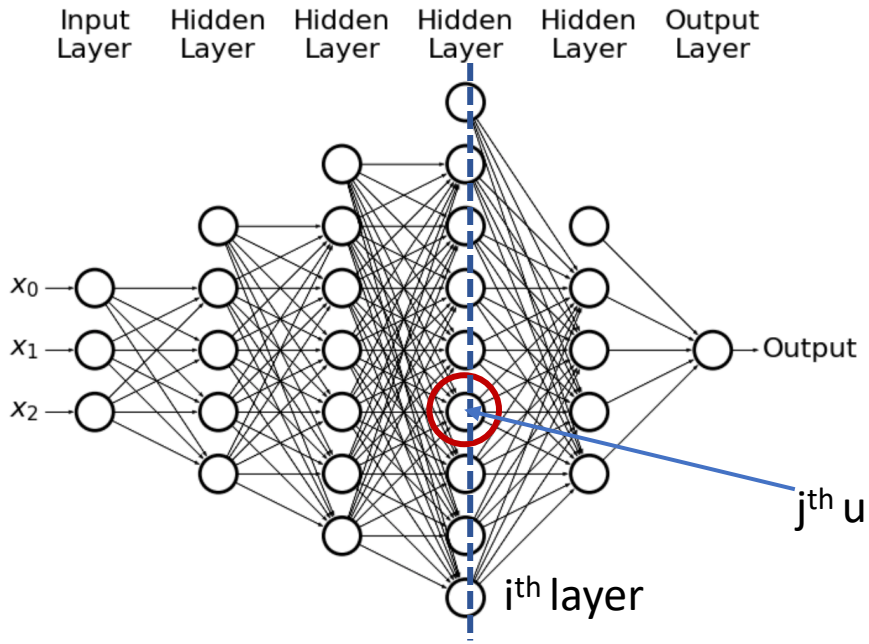
$$w_0 = -\theta$$

- or additional weight

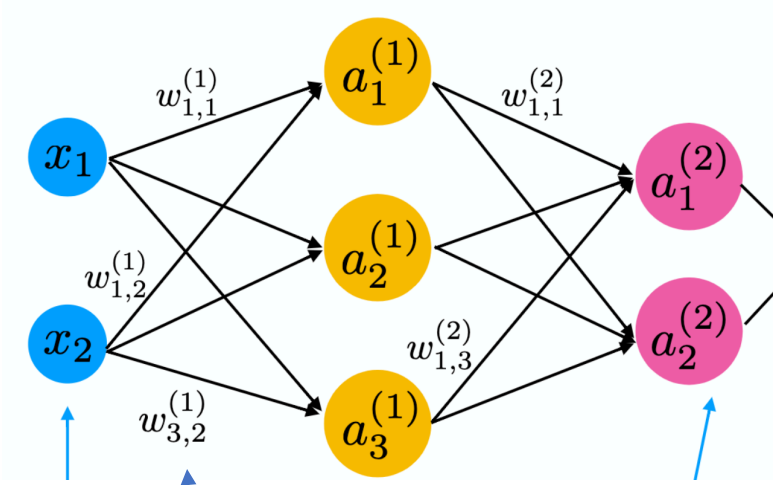
github.com/rasbt/stat453-deep-learning-ss20

Deep neural networks

- perceptron can not represent XOR function
  -- to do so we need additional *hidden* layers



- fully-connected multi-layer perceptron



- each connection is parametrized by its weight
- each neuron has its own activation function

By noting i the i^{th} layer of the network and j the j^{th} hidden unit of the layer, we have:

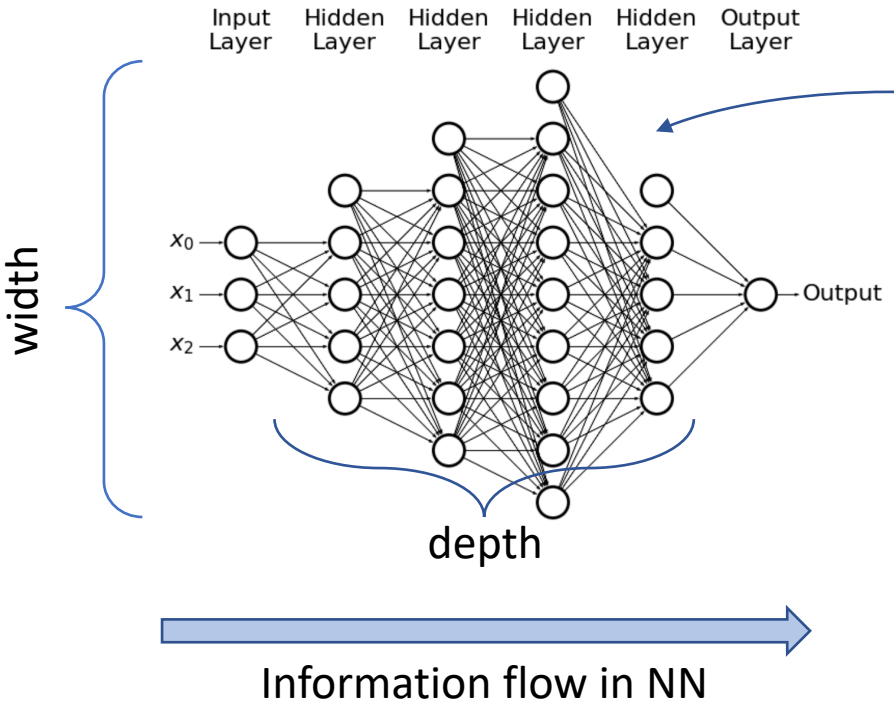
$$x_j^{(i)} = f\left(\sum_k w_k^{(i)} x_k^{(i-1)}\right)$$

where we note w, b, z the weight, bias and output respectively.

- network is called **deep** when in contains hidden layers
- all weights are fitted during *training*

github.com/rasbt/stat453-deep-learning-ss20

Deep neural networks



- fully-connected multi-layer perceptron
- however, we could have only a part of connections

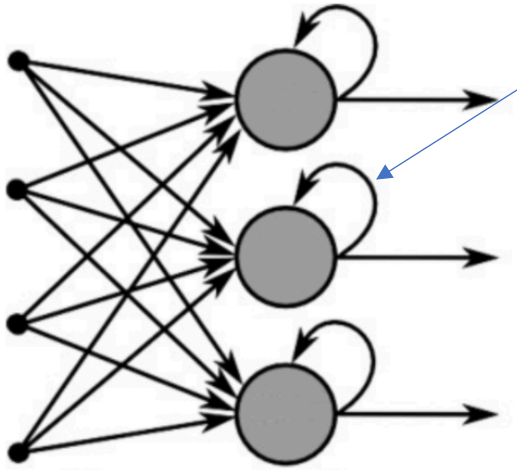
- Deep **feedforward** networks because information flows through the the subsequent layers starting from the input layer and finally to the output layers

  -> network is called deep when contains hidden layers
  -> the dimensionality of hidden layers determines the *width* of the model
  -> number of deep layers gives the *depth* of the model

Deep neural networks

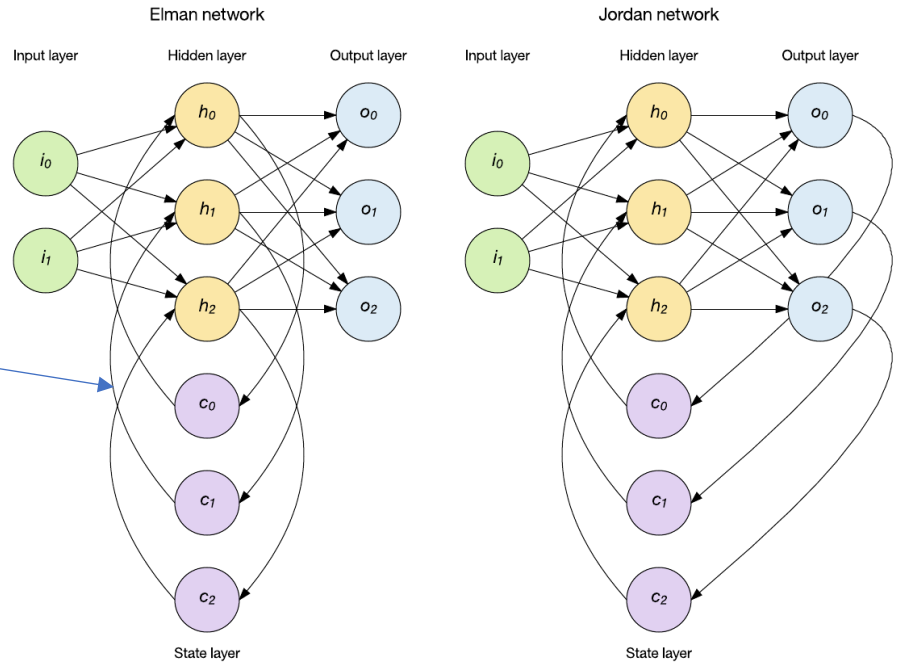
Information flow in NN

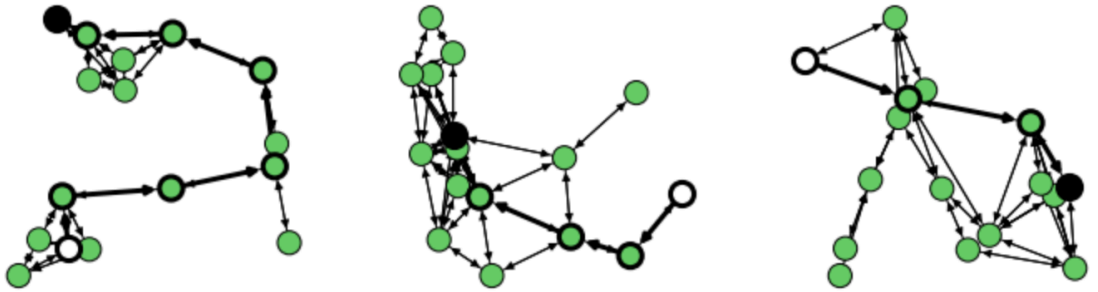
- When feedforward neural networks are extended to include feedback connections, they are called **recurrent** neural networks



Recurrent Neural Network

feedback connections enables information flow in reverse direction





- **graph** neural networks
  -> no distinguished direction of information flow through the network

developer.ibm.com/articles/cc-cognitive-recurrent-neural-networks/

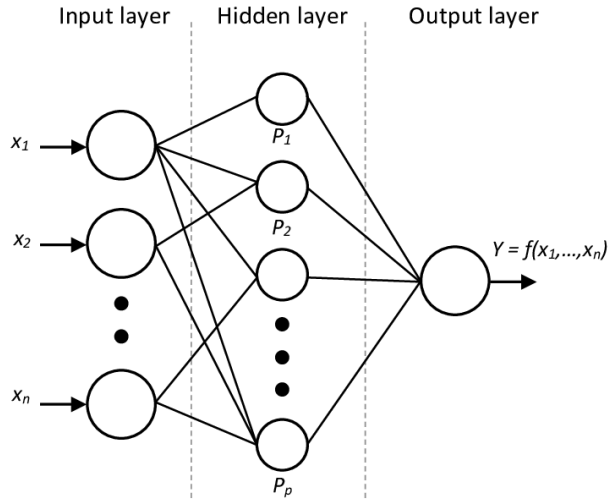
Deep neural networks: universality

Universal Approximation Theorem

$$any\ f\colon R^n \rightarrow R^m$$

- A feedforward network with a linear output layer and at least **one hidden layer** with any nonlinear bounded activation function (such as logistic activation) can **approximate any** continuous **function** (on a closed and bounded subset of $\mathbb{R}^n$), from one finite-dimensional space to another, with **any** desired **precision**
- such network should have enough hidden units (enough width)

- however the layer may be infeasibly large and may fail to generalize correctly
- using **deeper** models can reduce the number of units required to represent the desired, function and reduce the amount of generalization error



In a nutshell:

- a feedforward network with a single hidden layer is sufficient to represent any function, but may overfit,
- using deeper models can reduce the required number of hidden units and reduce the generalization error.

Deep neural networks: training

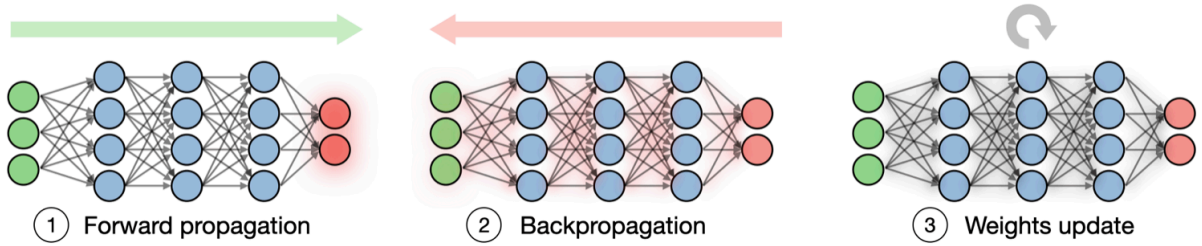
Training NN is just optimization problem:

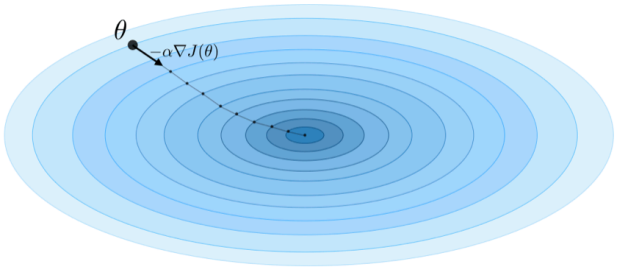
- The task is to minimize loss function in parameters (weights) space

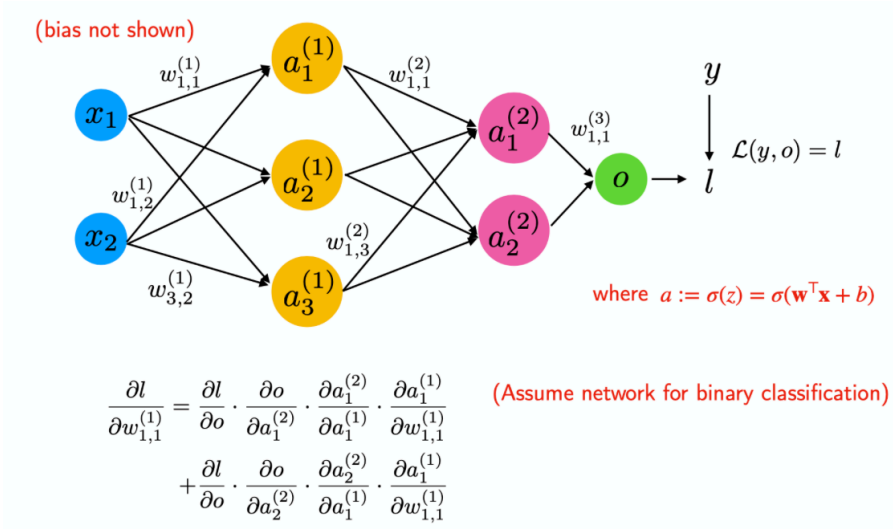
$$w \longleftarrow w - \alpha \frac{\partial L(z, y)}{\partial w}$$

□ **Updating weights** — In a neural network, weights are updated as follows:

- Step 1: Take a batch of training data and perform forward propagation to compute the loss.
- Step 2: Backpropagate the loss to get the gradient of the loss with respect to each weight.
- Step 3: Use the gradients to update the weights of the network.





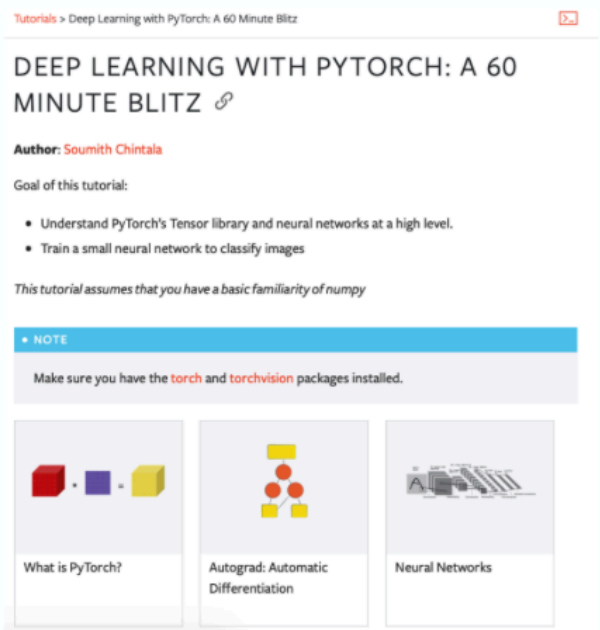


backpropagation utilizes *chain rule*, and reuses partial derivatives calculated before.

good review: brilliant.org/wiki/backpropagation/

stanford.edu/~shervine/teaching/cs-230/cheatsheet-deep-learning-tips-and-tricks

Deep neural networks: training



DEEP LEARNING WITH PYTORCH: A 60 MINUTE BLITZ

https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html

torch.autograd is PyTorch's **automatic** differentiation engine that powers neural network training

Generally speaking, `torch.autograd` is an engine for computing vector-Jacobian product. That is, given any vector $v = \begin{pmatrix} v_1 & v_2 & \cdots & v_m \end{pmatrix}^T$, compute the product $v^T \cdot J$. If v happens to be the gradient of a scalar function $l = g\left(\vec{y}\right)$, that is, $v = \begin{pmatrix} \frac{\partial l}{\partial y_1} & \cdots & \frac{\partial l}{\partial y_m} \end{pmatrix}^T$, then by the chain rule, the vector-Jacobian product would be the gradient of l with respect to $\vec{x}$:

$$J^T \cdot v = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix} \begin{pmatrix} \frac{\partial l}{\partial y_1} \\ \vdots \\ \frac{\partial l}{\partial y_m} \end{pmatrix} = \begin{pmatrix} \frac{\partial l}{\partial x_1} \\ \vdots \\ \frac{\partial l}{\partial x_n} \end{pmatrix}$$

Text source: https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html#sphx-glr-beginner-blitz-autograd-tutorial-py

Deep neural networks: training

$$\theta \longleftarrow \theta - \alpha \nabla J(\theta)$$

Gradient Descent optimization

- In the **gradient descent** (GD) method weights are updated after a full loop over training data:

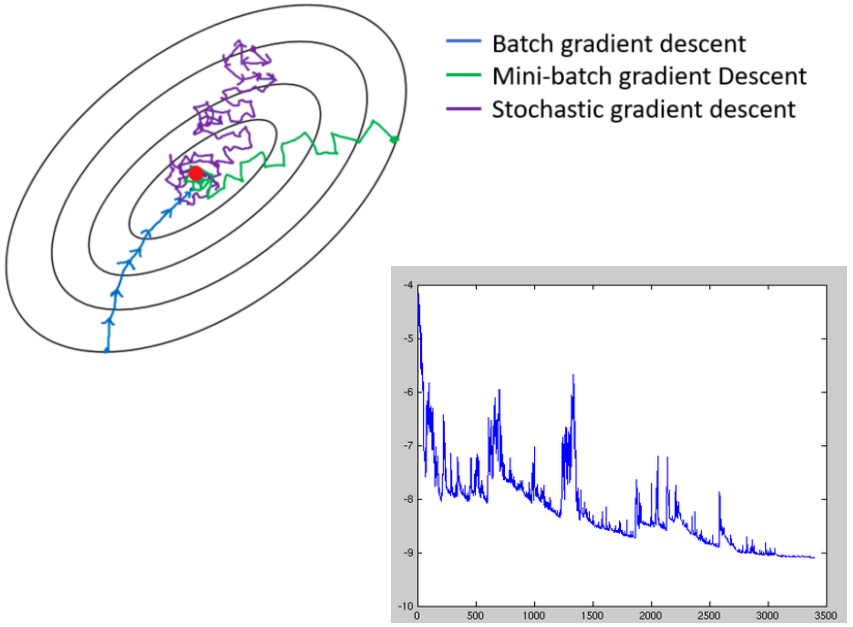
$$W \rightarrow W - \alpha \cdot \nabla_W L(W)$$

- In the **stochastic gradient descent** (SGD) method weights are updated for each training sample:

$$W \rightarrow W - \alpha \cdot \nabla_W L(W; x^{(i)}; y^{(i)})$$

- The common method, which is somehow between GD and SGD, is **mini-batch gradient descent** (MBGD) - the one we used for MNIST example:

$$W \rightarrow W - \alpha \cdot \nabla_W L(W; x^{(i;i+n)}; y^{(i;i+n)})$$



https://en.wikipedia.org/wiki/Stochastic_gradient_descent

SGD:

- fluctuations may help to escape local minima
- problem with overshooting exact minimum
- **decreasing** learning rate during optimization enables to converge

Deep neural networks: training

CLASS `torch.optim.SGD(params, lr=<required parameter>, momentum=0, dampening=0, weight_decay=0, nesterov=False)` [SOURCE]

Implements stochastic gradient descent (optionally with momentum).

The SGD optimizer in PyTorch is just gradient descent. User decide how to pass data: single sample mini-batch or batch.

Adaptive optimization

SGD: constant learning rate, which needs to be tuned by hand

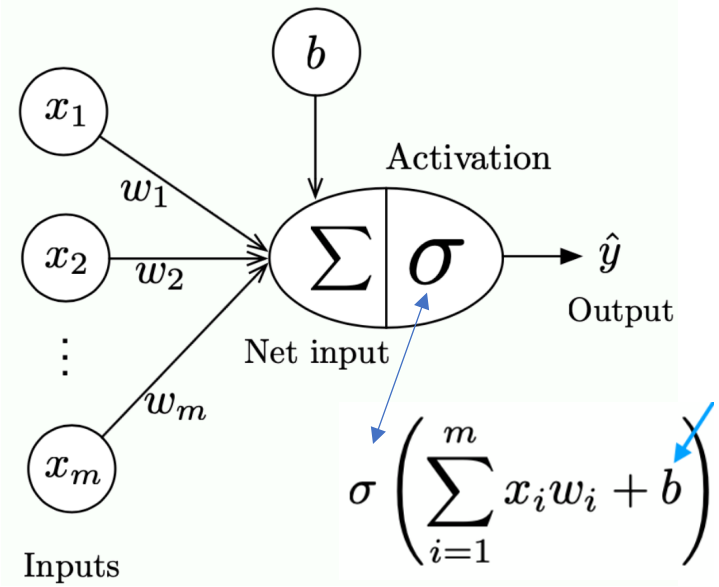
Adaptive models modifies the learning rate α using information about gradients:

- Adagrad: $W_{t+1,i} = W_{t,i} - \frac{\alpha}{\sqrt{G_{t,ii}+\varepsilon}} \nabla_W L(W_{t,i})$
- Adadelta
- Adam

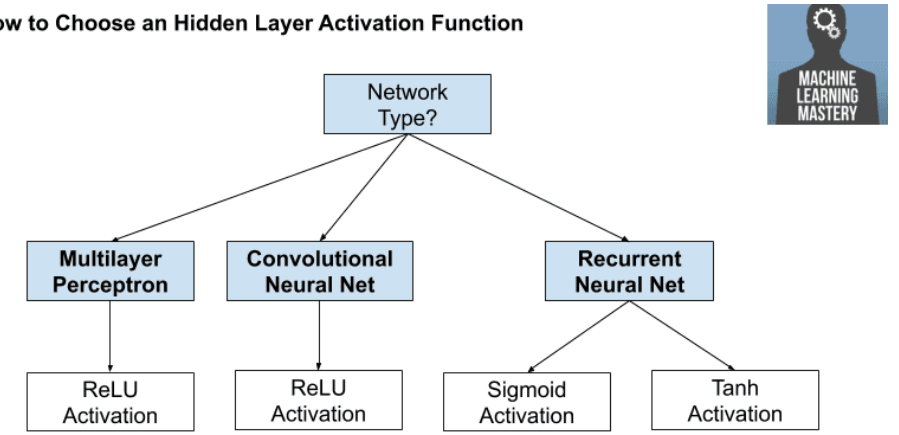
Another technique designed to accelerate learning: *momentum*

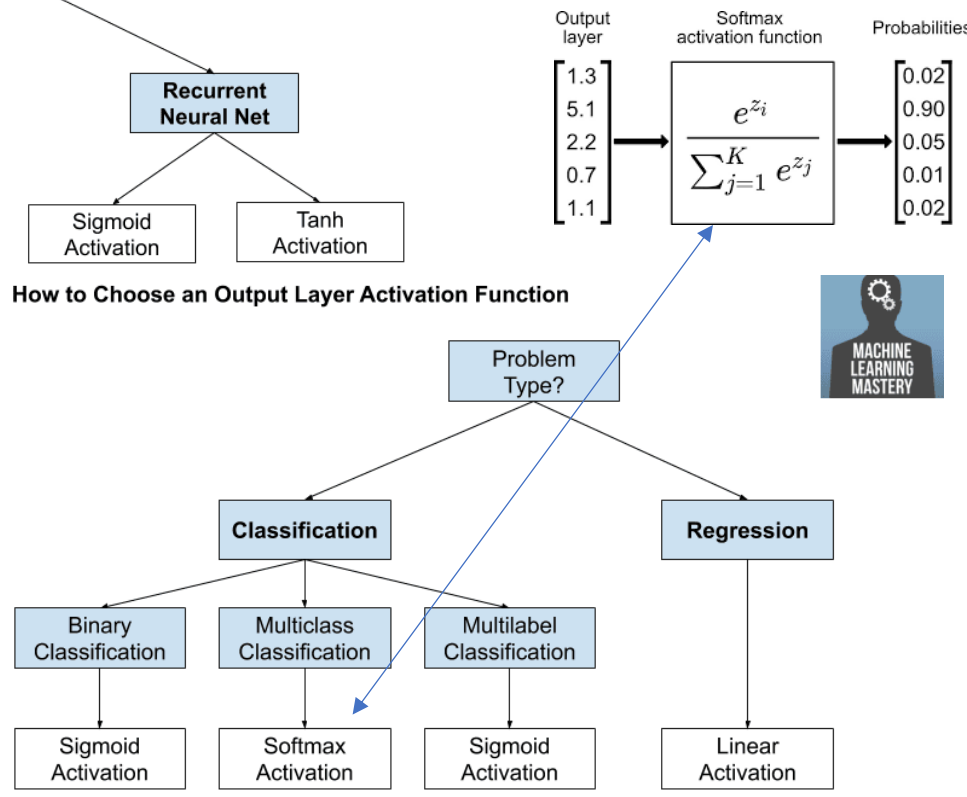
$$v_t = \gamma \cdot v_{t-1} + \alpha \cdot \nabla_W L(W; x^{(i)}; y^{(i)})$$
$$W \rightarrow W - v_t$$

Deep neural networks: activation

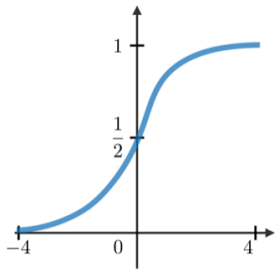
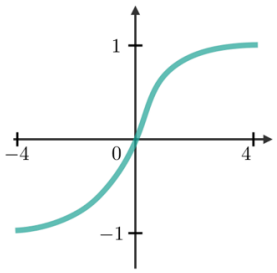
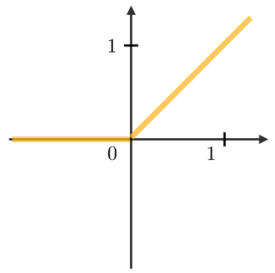
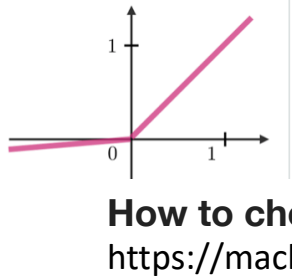


We usually chose activation for hidden units and output layer separately:





□ **Activation function** — Activation functions are used at the end of a hidden unit to introduce non-linear complexities to the model. Here are the most common ones:

Sigmoid	Tanh	ReLU	Leaky ReLU
$g(z) = \frac{1}{1+e^{-z}}$	$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	$g(z) = \max(0, z)$	$g(z) = \max(\epsilon z, z)$ with $\epsilon \ll 1$
			

How to chose Activation for Hidden and output Layers:
https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/

stanford.edu/~shervine/teaching/cs-229/cheatsheet-deep-learning

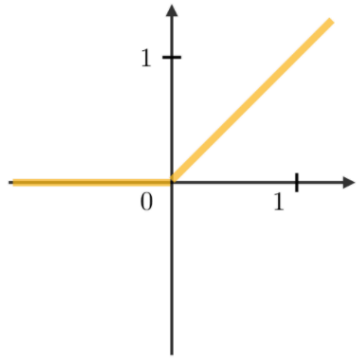
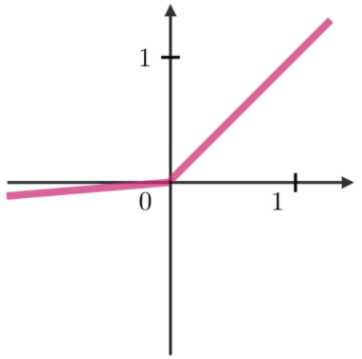
Deep neural networks: activation

ReLU is a default choice for activation of hidden units:

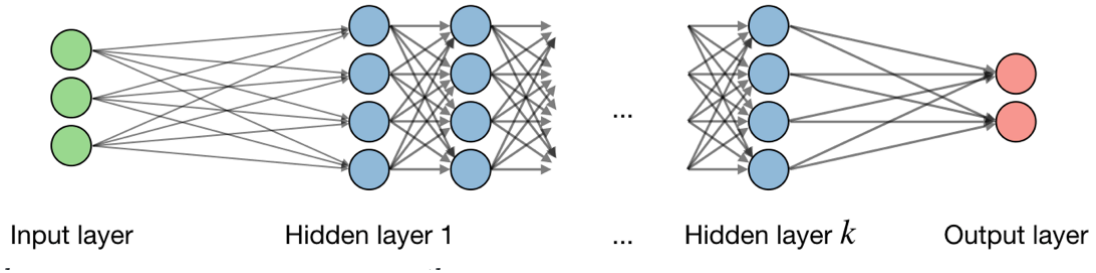
- easy to optimize because it is similar to linear unit
- reduces *vanishing gradient* problem
  - backpropagation computes gradients by the chain rule
  - to compute gradients of the early layers we multiply small numbers many *times*
  - loss signal decreases exponentially when propagating backwards therefore early layers train very slowly

some drawbacks:

- non-differentiable at zero
- *dying* ReLU problem
  - some neurons essentially ***die*** for all inputs and remain ***inactive*** no matter what input is supplied
  - corrected in *leaky* ReLU

ReLU	Leaky ReLU
$g(z) = \max(0, z)$	$g(z) = \max(\epsilon z, z)$ with $\epsilon \ll 1$
	

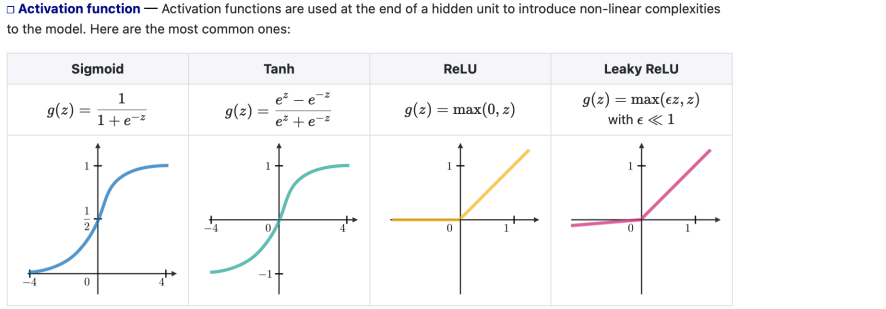
Deep neural networks: hyperparameters

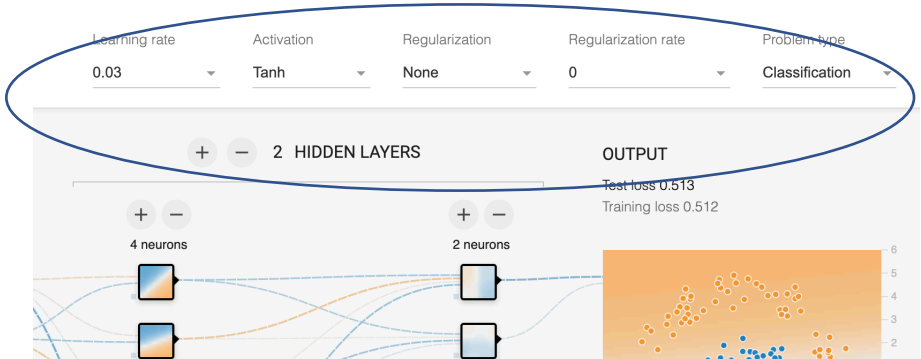


Note, that NN has much more hyperparameters than other ML models:

- number of hidden layers
- number of neurons in layer
- activation functions
- learning algorithm settings: SGD variation, learning rate, number of epochs, momentum, etc.
- batch size in mini-batch gradient descent
- regularization method and its parameters

It is crucial to use cross-validation (at least single validation subset) for hyperparameter tuning.



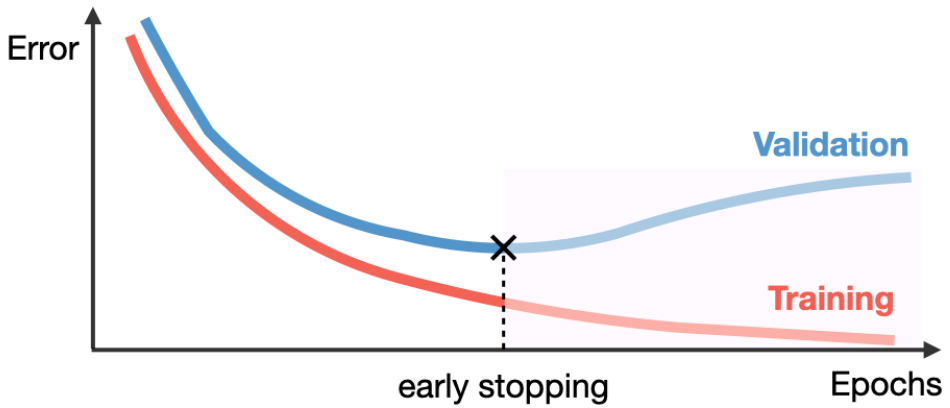


- Let's play with hyperparameters: **https://playground.tensorflow.org**

Deep neural networks: regularization

A central problem in machine learning is how to make an algorithm that generalize well

- regularization is any strategy designed to prevent overfitting
- deep networks uses three main types of regularizations:
    - loss function modification
    - dropout technique
    - early stopping:



https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-deep-learning-tips-and-tricks

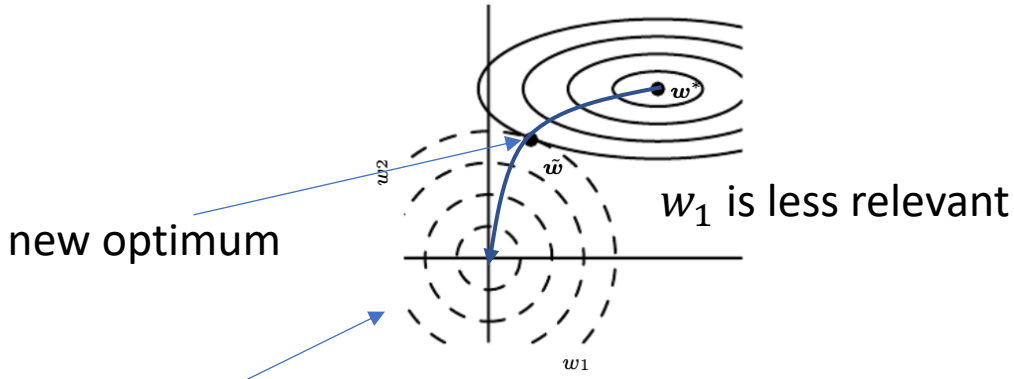
Deep neural networks: regularization

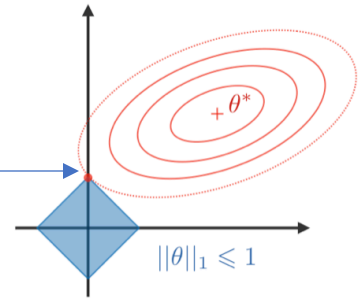
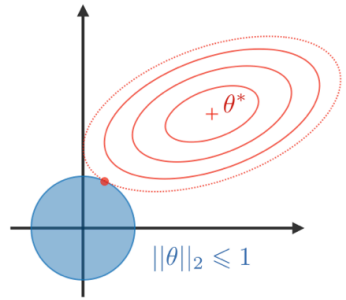
Loss function modification:
we add **regularization** term (**regularizer**) to the loss function $L(\boldsymbol{w})$:

$$L(\boldsymbol{w}, \boldsymbol{X}, \boldsymbol{y}) \longrightarrow L(\boldsymbol{w}, \boldsymbol{X}, \boldsymbol{y}) + \lambda R(\boldsymbol{w})$$

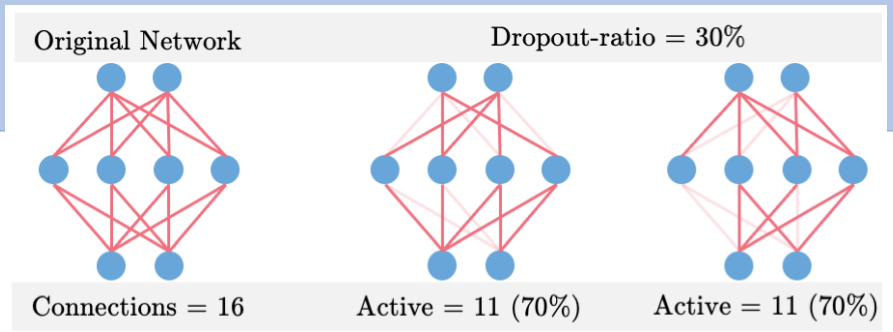
- **ridge regression**, uses L_2 norm: $R(\boldsymbol{w}) = \sum_i w_i^2$
  - we put some constraints on the magnitude of weights
  - regularizer suppress stronger less relevant weights
    -> modify loss function but not mess it up.

- **LASSO** uses L_1: $R(\boldsymbol{w}) = \sum_i |w_i|$
  - The L_1 penalty causes a subset of the weights to become zero (suppressing is stronger than L_2),
  - can be used to features selection



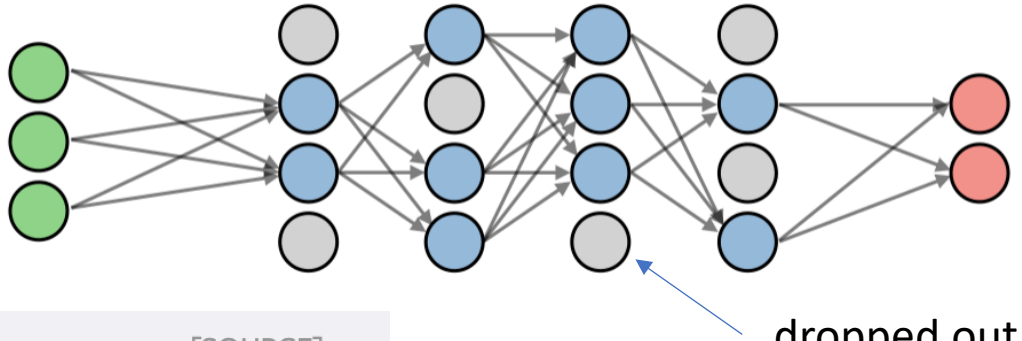
LASSO	Ridge
• Shrinks coefficients to 0 • Good for variable selection	Makes coefficients smaller
	
$\ldots + \lambda\|\|\theta\|\|_1$ $\lambda \in \mathbb{R}$	$\ldots + \lambda\|\|\theta\|\|_2^2$ $\lambda \in \mathbb{R}$

Deep neural networks: regularization



Dropout

- dropout is similar to ensemble learning:
  -> involves training multiple network models on each test example
- This improves generalization -- force different neurons to learn the same *concept*



CLASS `torch.nn.Dropout(p=0.5, inplace=False)` [SOURCE]

During training, randomly zeroes some of the elements of the input tensor with probability `p` using samples from a Bernoulli distribution. Each channel will be zeroed out independently on every forward call.

This has proven to be an effective technique for regularization and preventing the co-adaptation of neurons as described in the paper Improving neural networks by preventing co-adaptation of feature detectors .

Furthermore, the outputs are scaled by a factor of $\frac{1}{1-p}$ during training. This means that during evaluation the module simply computes an identity function.

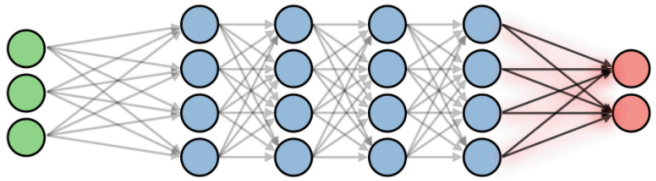
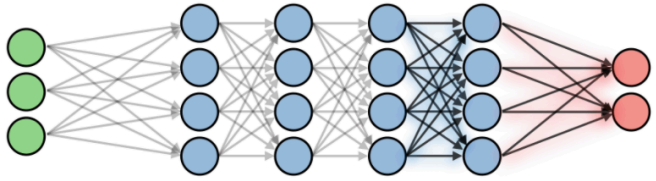
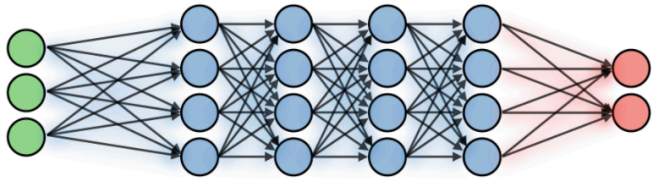
- dropout deactivates random set of neurons during training phase (usually about 50%)
- usually, dropout is applied to fully connected layers
- during inference stage dropout is turned off

pytorch.org/docs/stable/generated/torch.nn.Dropout.html

Deep neural networks: transfer learning

Transfer Learning

- It is useful to use initial weights pretrained on similar but much larger dataset
  -> it can dramatically shorten training
- it may be also useful to train only (last) part of a network
  -> especially when we have less data

Training size	Illustration	Explanation
Small		Freezes all layers, trains weights on softmax
Medium		Freezes most layers, trains weights on last layers and softmax
Large		Trains weights on layers and softmax by initializing weights on pre-trained ones

https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-deep-learning-tips-and-tricks

Idea:
to train VGG (very deep convolutional NN) from scratch we need <u>weeks</u>

-> to re-train pre-trained VGG to fit *our* data we may need <u>only hours or just days</u>