More Supervised Learning

- **Week 10**

- **Perceptron**
- **Deep Learning**

This Lecture is about …

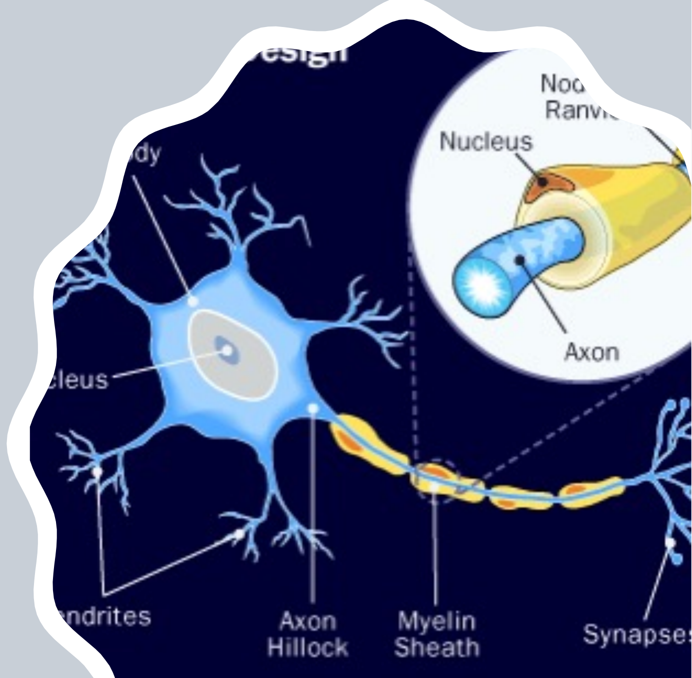
- Motivation to Neural Networks
- Perceptron
- Multi-layer Perceptron (MLP)
- Connections to Deep Learning

What is different about Neural Networks?

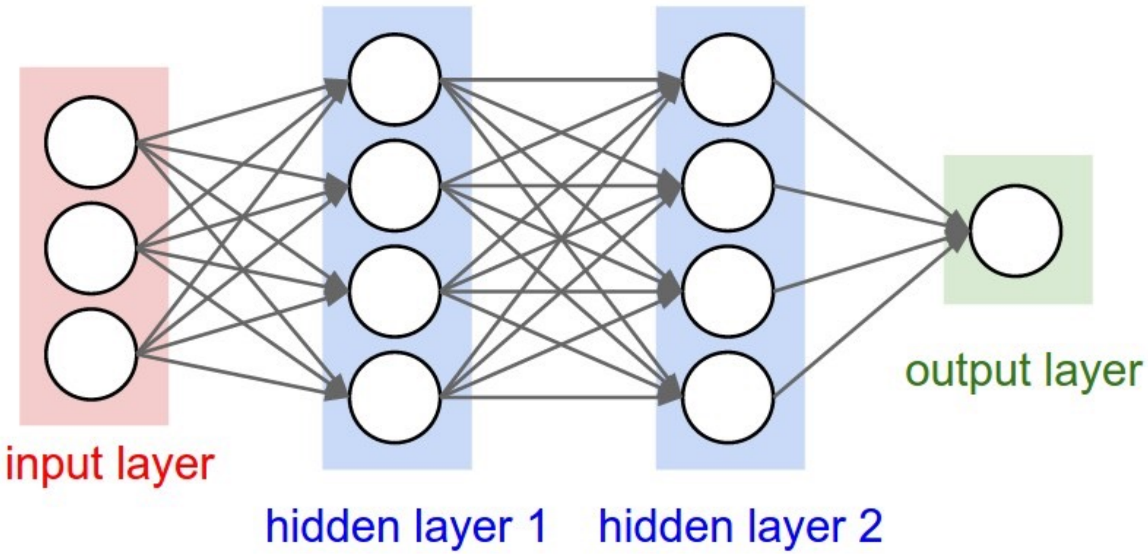
- Linear models may not be sufficient when the underlying functions/decision boundaries are extremely nonlinear.
- Support vector machines can construct nonlinear functions but use fixed feature transformations, which depends on the kernel function.
- Neural Networks allow the feature transformations to be learnt from data.

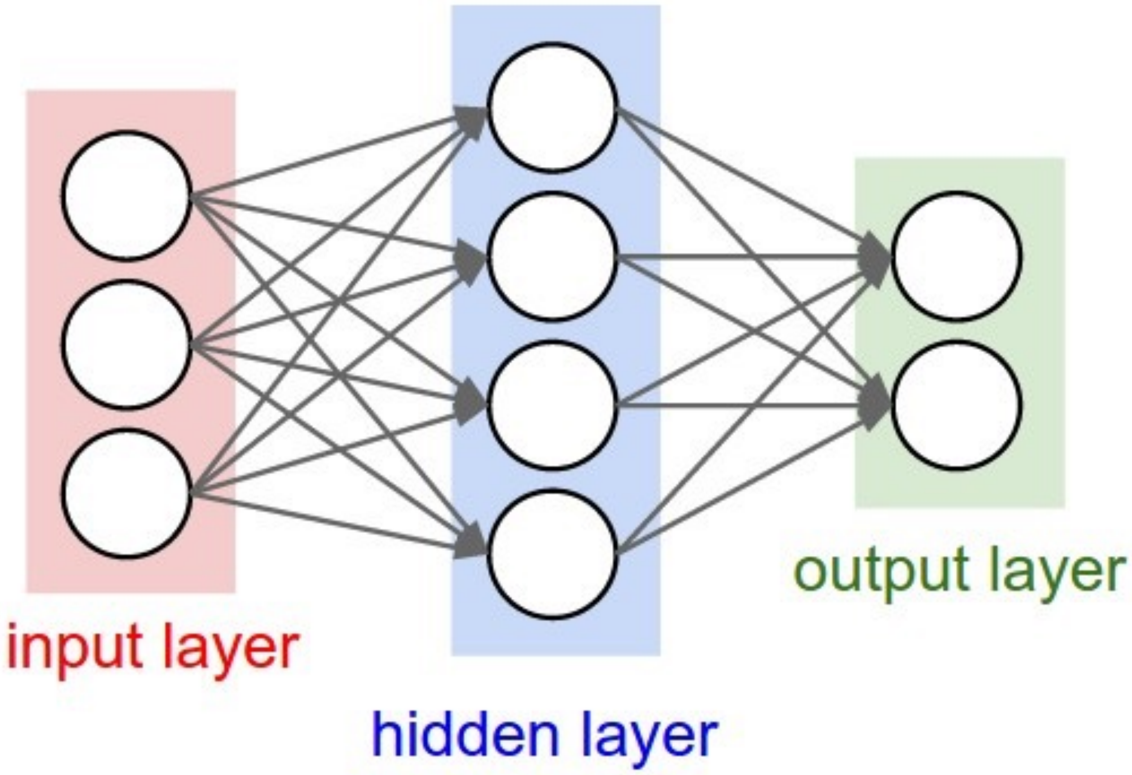
Historical Motivation

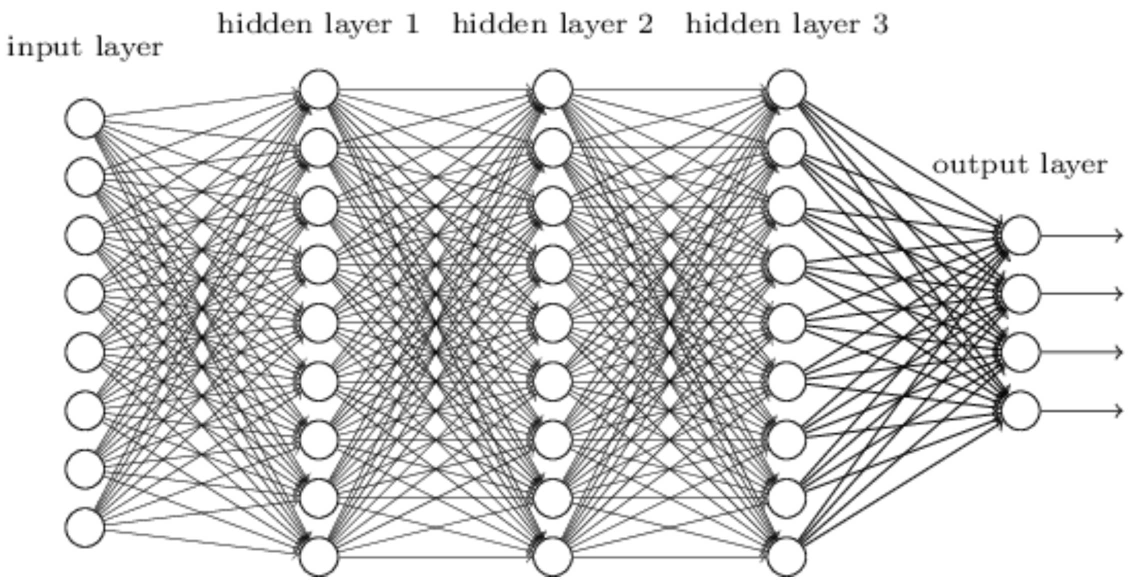
- Our brain has networks of inter-connected neurons and has highly-parallel architecture.
- ANN (artificial neural networks) are motivated by biological neural systems.
- Two groups of ANN researchers:
  1. Group that uses ANN to study/model the brain
  2. Group that uses the brain as the motivation to design ANN as an effective learning machine, which might not be the true model of the brain.
- We are following the second group's approach.



Some Examples of ANN







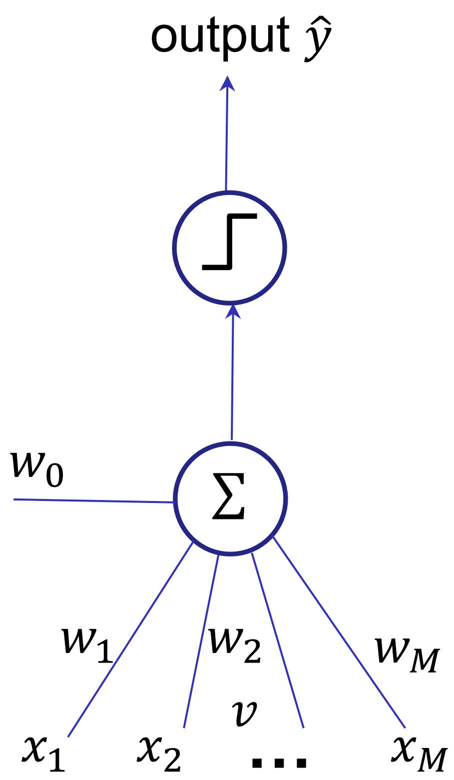
Perceptron

Motivation & Algorithm

Multi-layer Perceptron

Back-propagation

Perceptron



- Perceptron is a simple neural network used for binary classification.
- It has only one layer with single node.
  - Given: input vector $\boldsymbol{x} = (x_1, x_2, \dots, x_M)$ of M dimensions and weight vector $\boldsymbol{w} = (w_0, w_1, \dots, w_M)$
  - The perceptron produces output: $\hat{y} = \text{sign}[v(\boldsymbol{x}, \boldsymbol{w})]$ where $v(\boldsymbol{x}, \boldsymbol{w})$ is the linear combiner:

$$v(\boldsymbol{x}, \boldsymbol{w}) = \sum_{i=1}^{M} w_i \, x_i + w_0$$

  - Better notation, let $x_0 = 1$ and $\boldsymbol{x} = (x_0, x_1, \dots, x_M)$

    then $v(\boldsymbol{x}, \boldsymbol{w}) = \boldsymbol{w}^T\boldsymbol{x} = \boldsymbol{x}^T\boldsymbol{w}$

    and $\hat{y}(\boldsymbol{x}, \boldsymbol{w}) = \text{sign}[\boldsymbol{w}^T\boldsymbol{x}]$

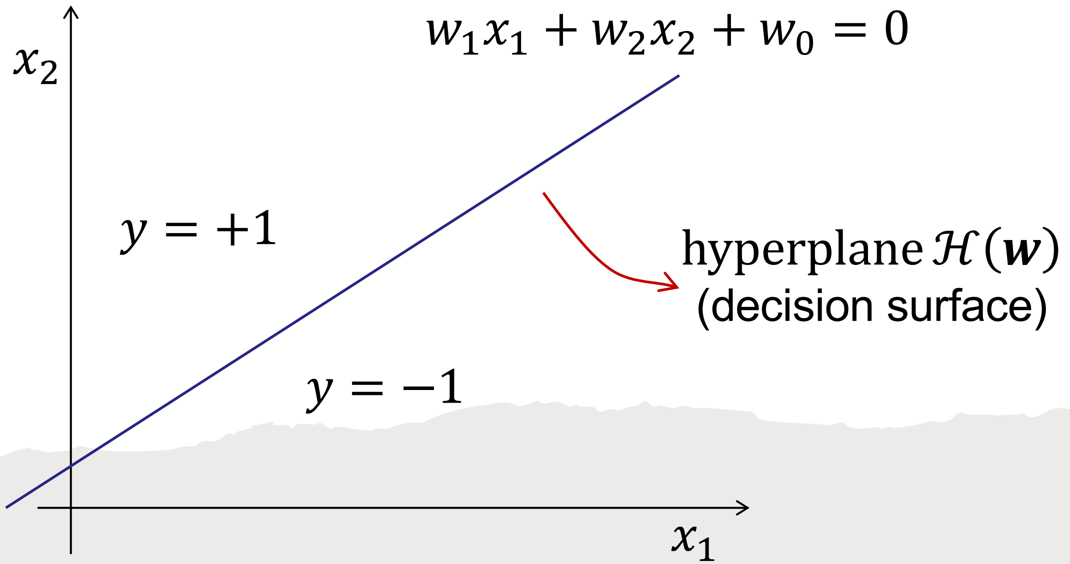
What Perceptron is doing

- Given weight $\boldsymbol{w}$, the perceptron linearly divides input space into two regions:
  - All $\boldsymbol{x}$'s such that $\hat{y}(\boldsymbol{x}, \boldsymbol{w}) = 1$, or $v(\boldsymbol{x}, \boldsymbol{w}) \geq 0$
  - All $\boldsymbol{x}$'s such that $\hat{y}(\boldsymbol{x}, \boldsymbol{w}) = -1$, or $v(\boldsymbol{x}, \boldsymbol{w}) < 0$
- This corresponds to the two sides of the hyperplane defined by the equation:

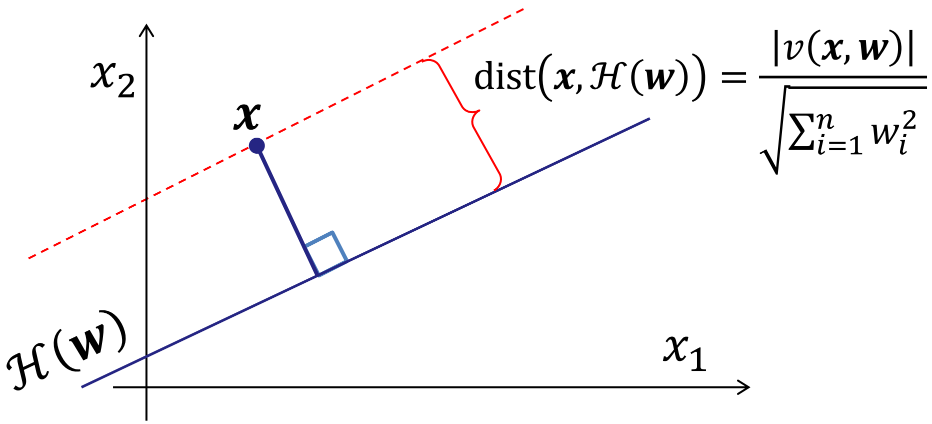
$$v(\boldsymbol{x}, \boldsymbol{w}) = \sum_{i=1}^{M} w_i x_i + w_0 = 0$$

- Note that $|v(\boldsymbol{x}, \boldsymbol{w})|$ is proportional to the distance from x to the hyperplane.

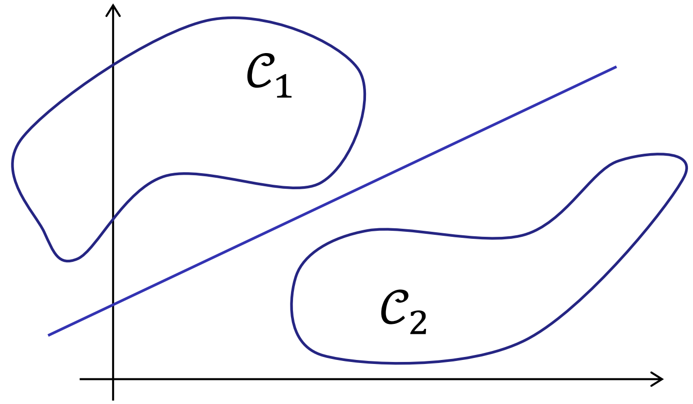
$$\mathrm{dist}\big(x, \mathcal{H}(w)\big) = \frac{\left|\sum_{i=1}^{M} w_i x_i + w_0\right|}{\sqrt{\sum_{i=1}^{M} w_i^2}} = \frac{|v(\boldsymbol{x}, \boldsymbol{w})|}{\sqrt{\sum_{i=1}^{M} w_i^2}}$$



What Perceptron is doing



- The sign of $v(\boldsymbol{x}, \boldsymbol{w})$ indicates on which side of hyperplane $\mathcal{H}(\boldsymbol{w})$ is $\boldsymbol{x}$.
- While the magnitude $|v(\boldsymbol{x}, \boldsymbol{w})|$ indicates how far away $\boldsymbol{x}$ is from $\mathcal{H}(\boldsymbol{w})$.

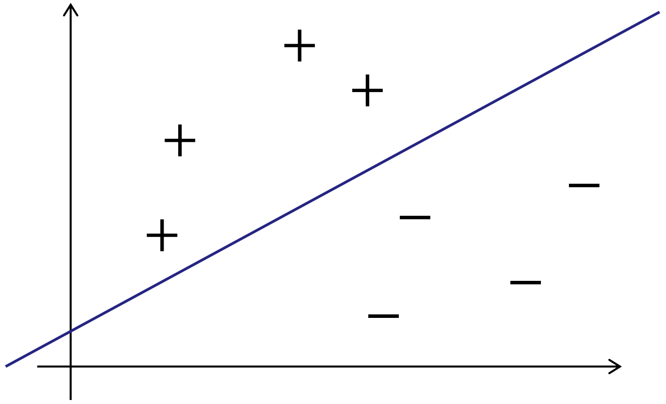


Linearly separable

- Two sets $\mathcal{C}_1$ and $\mathcal{C}_2$ are called linearly separable if there exists a hyperplane $\mathcal{H}(\boldsymbol{w})$ that separates them.

Training (or Learning) Perceptron

- Find the weight vector $\boldsymbol{w}$ so that the perceptron correctly classify 2 classes, given sample training data



- Training data $D = \{(\boldsymbol{x}_t, y_t)\}, t = 1, \ldots, n$

  where $\boldsymbol{x}_t = (x_{t1}, \ldots, x_{tM})$ is the input vector at time t

  $y_t = \pm 1$ is the desired output

Learning Perceptron

Perceptron Learning Algorithm

1. Initialize $\boldsymbol{w} = \mathbf{0}$
2. Retrieve next input $\boldsymbol{x}_t$ and desired output y_t

   Compute actual output $\hat{y}_t = \text{sign}[\boldsymbol{x}_t \cdot \boldsymbol{w}]$

   Compute output error $e_t = y_t - \hat{y}_t$

   Update weight, for all i:

   $w_i \leftarrow w_i + \Delta w_i$ with $\Delta w_i = \eta e_t x_{ti}$

   where $0 < \eta \leq 1$ is learning rate
3. Repeat from step 2 until convergence

$\boldsymbol{x}_t = (x_{t1}, \ldots, x_{tM})$

Remark: no update if $\hat{y}_t = y_t$, e.g., the current weight $\boldsymbol{w}$ already correctly classify current sample $\boldsymbol{x}(t)$

Learning Perceptron

Example:

Current weight $\boldsymbol{w} = (-1, 2, 1)$

Current training sample $\boldsymbol{x} = \left(\frac{1}{2}, 1\right), y = -1$

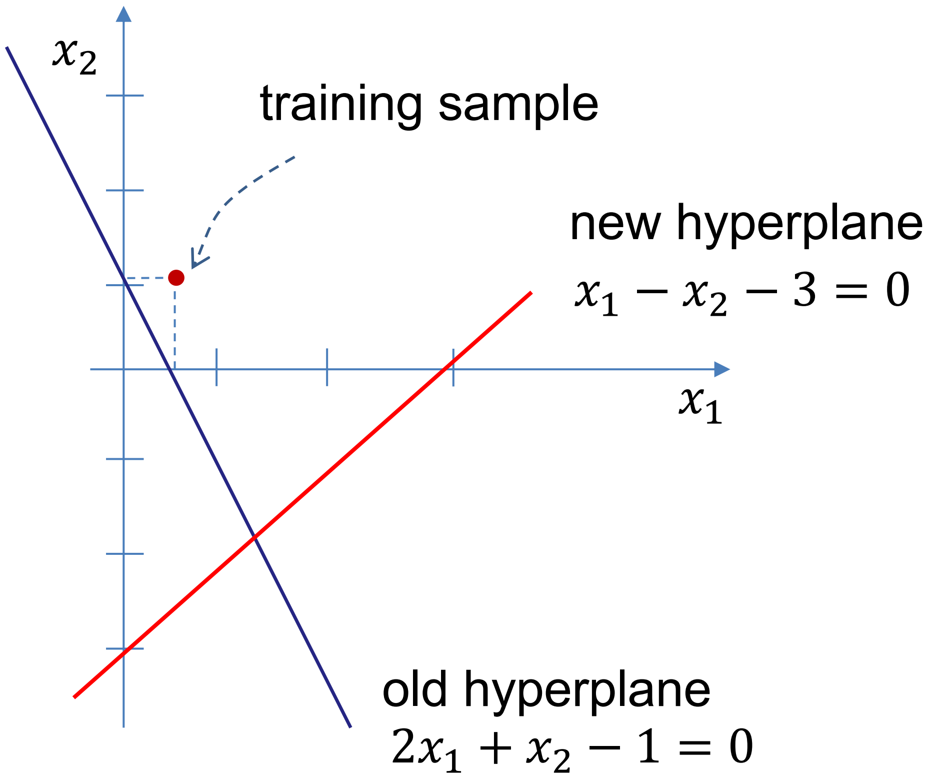
$$v(\boldsymbol{x}, \boldsymbol{w}) = 2\times\frac{1}{2} + 1\times1 \; - 1 = 1$$

$$\hat{y}(\boldsymbol{x}, w) = \text{sign}(1) = 1$$

$$e = y - \hat{y} = -2$$

New $\quad w_i = w_i - 2\eta x_i = w_i - 2x_i \text{ (let } \eta = 1)$

$$w_0 = -1 \; - 2\, x_0 = -3$$
$$w_1 = 2 - 2x_1 = 1$$
$$w_2 = 1 \; - 2x_2 = -1$$



Learning Perceptron

- Effect of the update rule

$$\boldsymbol{w}' \leftarrow \boldsymbol{w} + \eta e \boldsymbol{x} = \boldsymbol{w} + \eta(y - \hat{y})\boldsymbol{x}$$

- Inspect how w' classify x:

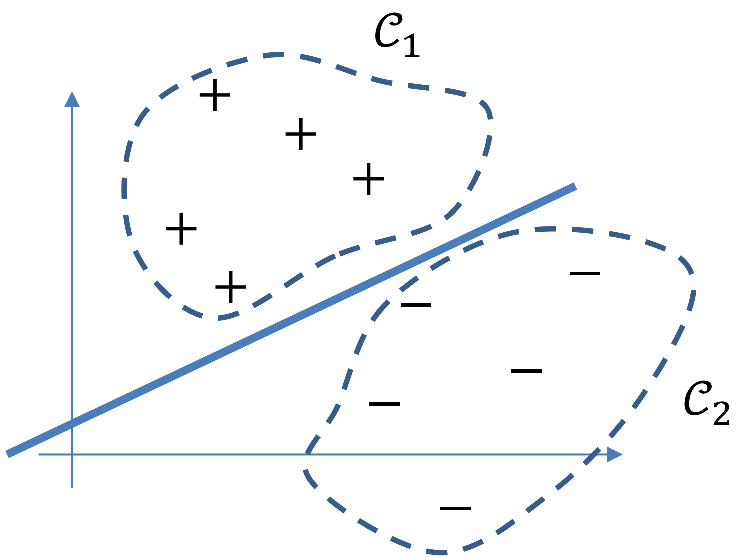
$$\begin{aligned} v' = \boldsymbol{w}' \cdot \boldsymbol{x} &= [\boldsymbol{w} + \eta(y - \hat{y})\boldsymbol{x}] \cdot \boldsymbol{x} \\ &= \boldsymbol{w} \cdot \boldsymbol{x} + \eta(y - \hat{y})\boldsymbol{x}^2 \\ &= v + \eta(y - \hat{y})\boldsymbol{x}^2 \end{aligned}$$

  - Thus, when $\hat{y} = -1$ but $y = 1$, i.e., $v < 0$ and we want $v > 0$ and indeed $v' > v$ (why?)
  - When $\hat{y} = 1$ but $y = -1$, i.e. $v \geq 0$ but we want $v < 0$, and indeed $v' < v$

The update rule creates new $\boldsymbol{w}$' that is better than $\boldsymbol{w}$ in classifying $\boldsymbol{x}$. That's what we want!

Perceptron Convergence Theorem

- If training instances are drawn from two linearly separate sets $\mathcal{C}_1$ and $\mathcal{C}_2$, then the perceptron learning rule will converge after finite iterations.

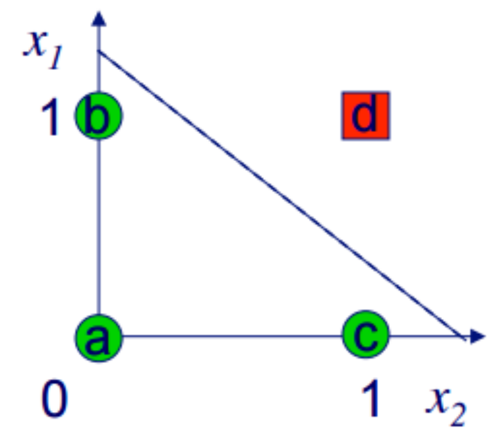


- However, no guarantee for convergence if $\mathcal{C}_1$ and $\mathcal{C}_2$ are not linearly separable!

Some Linearly Separable Problems

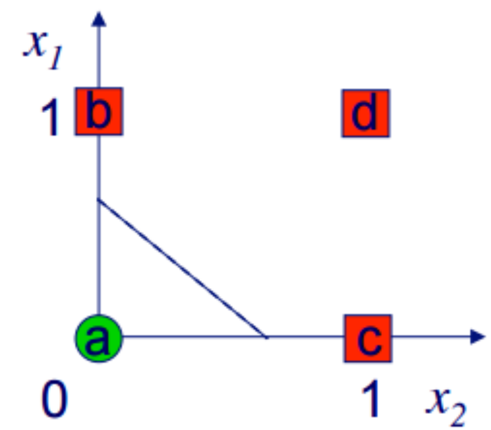
AND

	x_1	x_2	y
a	0	0	0
b	0	1	0
c	1	0	0
d	1	1	1



OR

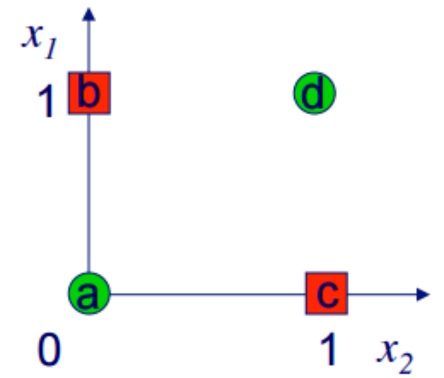
	x_1	x_2	y
a	0	0	0
b	0	1	1
c	1	0	1
d	1	1	1



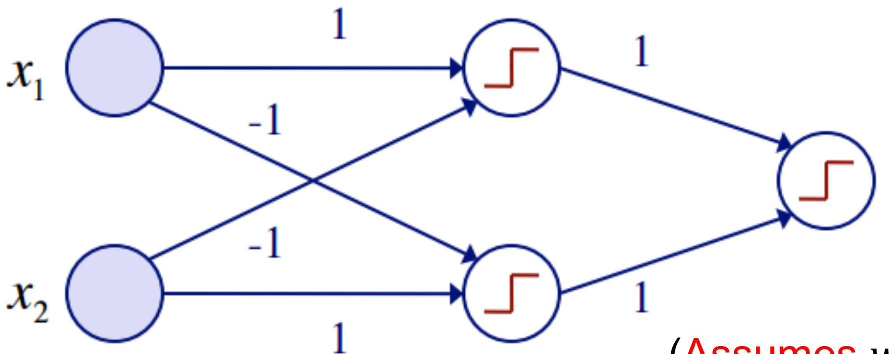
XOR Problem

Perceptron cannot deal with problems such as XOR!

	x_1	x_2	y
a	0	0	0
b	0	1	1
c	1	0	1
d	1	1	0



A multilayer Perceptron (MLP) can represent XOR problem



(Assumes $w_0 = 0$)

Learning Perceptron in Python

`sklearn.linear_model`.Perceptron

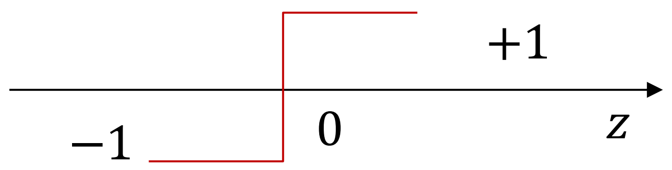
class `sklearn.linear_model.` **Perceptron** (*penalty=None, alpha=0.0001, fit_intercept=True, n_iter=5, shuffle=True, verbose=0, eta0=1.0, n_jobs=1, random_state=0, class_weight=None, warm_start=False*) [source]

Methods

`decision_function` (X)	Predict confidence scores for samples.
`densify` ()	Convert coefficient matrix to dense array format.
`fit` (X, y[, coef_init, intercept_init, ...])	Fit linear model with Stochastic Gradient Descent.
`fit_transform` (X[, y])	Fit to data, then transform it.
`get_params` ([deep])	Get parameters for this estimator.
`partial_fit` (X, y[, classes, sample_weight])	Fit linear model with Stochastic Gradient Descent.
`predict` (X)	Predict class labels for samples in X.
`score` (X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
`set_params` (*args, **kwargs)	
`sparsify` ()	Convert coefficient matrix to sparse format.
`transform` (*args, **kwargs)	DEPRECATED: Support to use estimators as feature selectors will be removed in version 0.19.

Multi-layer Feed-forward NN

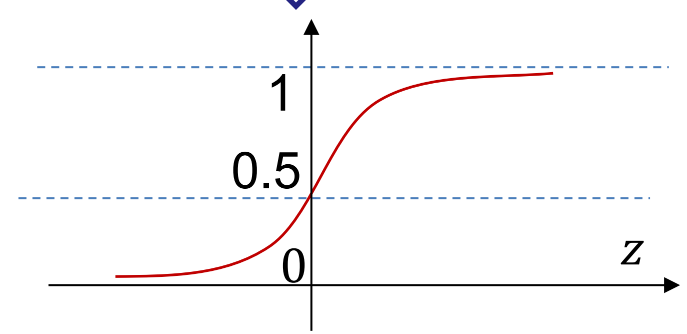
- Perceptron is quite weak in what it can represent.



sign(z)

- For complex, non-linear decision surfaces, we need multi-layer network.

big leap in modelling



- Choice of node in multi-layer network
  - Perceptron: discontinuity
  - Answer: sigmoid function!

- Sigmoid node: like a perceptron, but with the sigmoid function $\sigma(z) = (1 + e^{-z})^{-1}$ instead of the sign function, i.e., $y = \sigma(\boldsymbol{w}^T \boldsymbol{x})$.

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

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1 - \sigma(z))$$

Multi-layer Perceptron

- A feedforward neural network is an ANN wherein connections between units do not form a cycle.

- Multi-layer feed-forward NN is also known as Multi-layer Perceptron (MLP).

- The term “MLP” is really a misnomer.
  - Why? Because the model comprises multiple layers of logistic regression like models (with continuous nonlinearities) rather than multiple Perceptrons (with discontinuous nonlinearities).

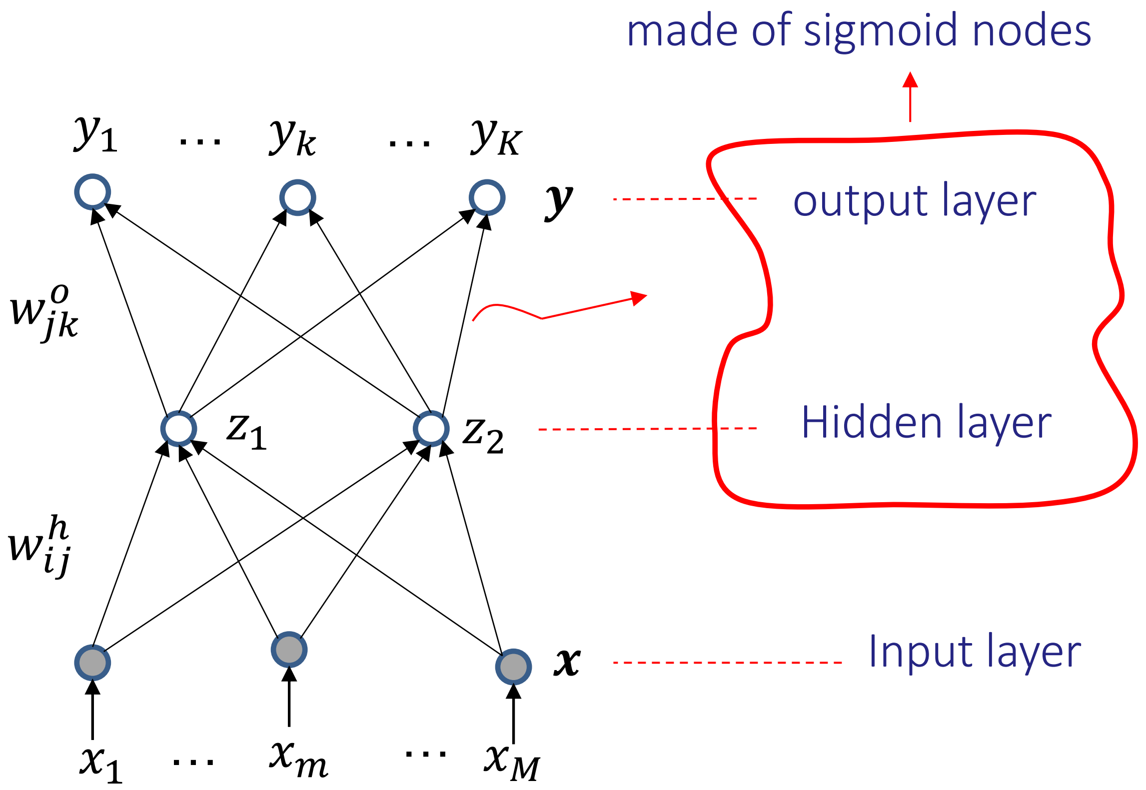
- Although a misnomer, we will continue using MLP term.

Structure of Multi-layer Perceptron

Consider a two-layer network: input layer, hidden layer and output layer

Remarks:

- Output now is a vector
- Two kinds of weights:
    - input → hidden
    - hidden → output
- w_{ij}^h: from i^{th} input → j^{th} hidden
- w_{jk}^o: from j^{th} hidden → k^{th} output
- Input layer does no computation, only to relay input vector.
- Can have more than one hidden layers.
- Doesn't have to be fully connected.



MLP Formulation

- Given input $\boldsymbol{x}_t$ and desired output $\boldsymbol{y}_t$, $t = 1, \dots, n$, find the network weights $\boldsymbol{w}$ such that

$$\hat{\boldsymbol{y}}_t \approx \boldsymbol{y}_t, \forall t$$

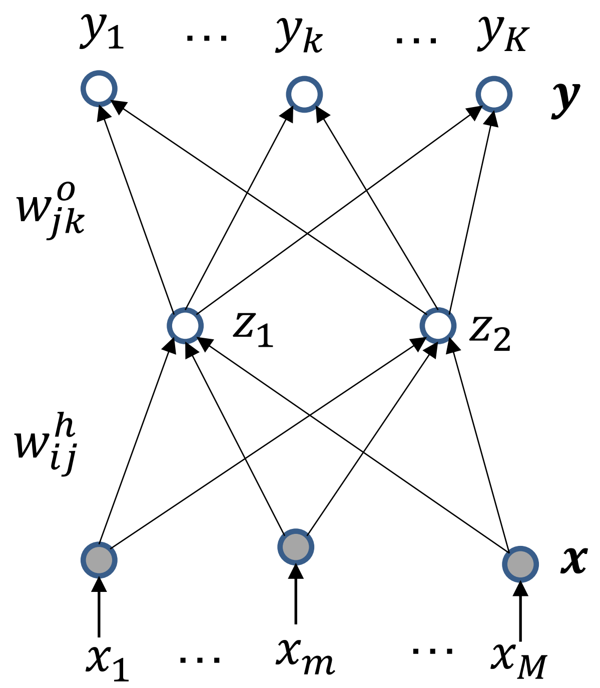
- Stating above as an optimization problem: find w to minimize the error function

error in the k^{th} output

$$E(\boldsymbol{w}) = \frac{1}{2} \sum_{t=1}^{n} \sum_{k=1}^{K} (y_{tk} - \hat{y}_{tk})^2$$

sum over all outputs

sum over all training samples



- We will use gradient-descent for minimization.
- $E(\boldsymbol{w})$ is not convex, but a complex function with possibly many local minima.
- We will use an algorithm called Backpropagation.

Detour: Gradient-based Optimization

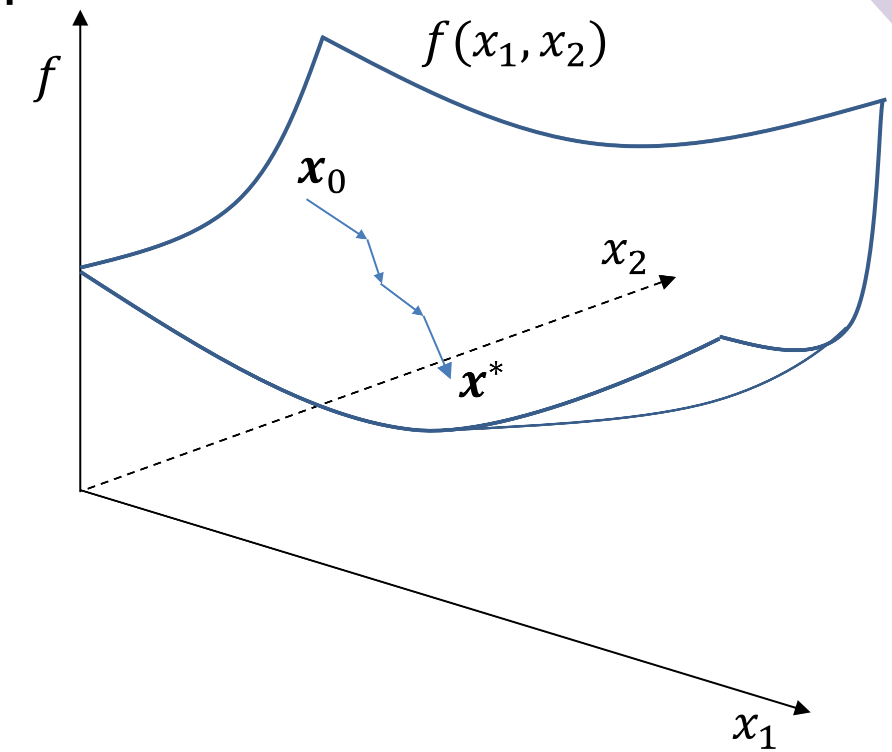
- To minimize a functional $f(\boldsymbol{x})$, use gradient-descent:
  - Initialize random $\boldsymbol{x}_0$
  - Slide down the surface of f in the direction of steepest decrease:

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t - \eta \times \nabla f(\boldsymbol{x}_t)$$

learning rate

- Similarly, to maximize $f(\boldsymbol{x})$, use gradient-ascent.

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t + \eta \times \nabla f(\boldsymbol{x}_t)$$



Detour: Stochastic Gradient Descent (SGD)

- Instead of minimizing $E(\boldsymbol{w})$, SGD minimizes the instantaneous approximation of $E(\boldsymbol{w})$ using only t-th instance, i.e.,

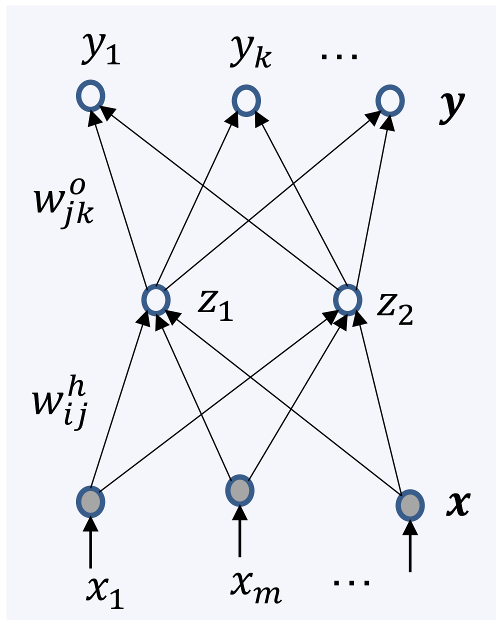
$$E_t(\boldsymbol{w}) = \frac{1}{2}\sum_{k=1}^{K}(y_{tk} - \hat{y}_{tk})^{2}$$

- Update rule (where t denotes the current training sample) is

$$w_i \leftarrow w_i - \eta \frac{\partial E_t(\boldsymbol{w})}{\partial w_i}$$

- SGD is cheap to perform and guaranteed to reach a local minimum in a stochastic sense.

Training MLP: Backpropagation



- It is in fact a stochastic gradient-descent rule!
- Minimizing instantaneous approximation for current training sample $(\boldsymbol{x}_t, \boldsymbol{y}_t)$

$$E_t(\boldsymbol{w}) = \frac{1}{2}\sum_{k=1}^{K}(\hat{y}_k - y_k)^2$$

- Gradient-descent rule: $w^o_{jk} \leftarrow w^o_{jk} - \eta \frac{\partial E_t(w)}{\partial w^o_{jk}}$
- Let $\bar{y}_k$ be the (unsigmoided) argument value at output node $\hat{y}_k$, i.e., $\hat{y}_k = \sigma(\bar{y}_k)$, we have:

$$\frac{\partial E_t}{\partial w^o_{jk}} = \boxed{\frac{\partial E_t}{\partial \bar{y}_k}} \times \frac{\partial \bar{y}_k}{\partial w^o_{jk}} \qquad = z_j \quad \text{since } \bar{y}_k = \sum w^0_{jk} z_j$$

$$\boxed{\frac{\partial E_t}{\partial \bar{y}_k}} = \underbrace{\frac{\partial E_t}{\partial \hat{y}_k}}_{-(y_k - \hat{y}_k)} \times \underbrace{\frac{\partial \hat{y}_k}{\partial \bar{y}_k}}_{(1-\hat{y}_k)\hat{y}_k}$$

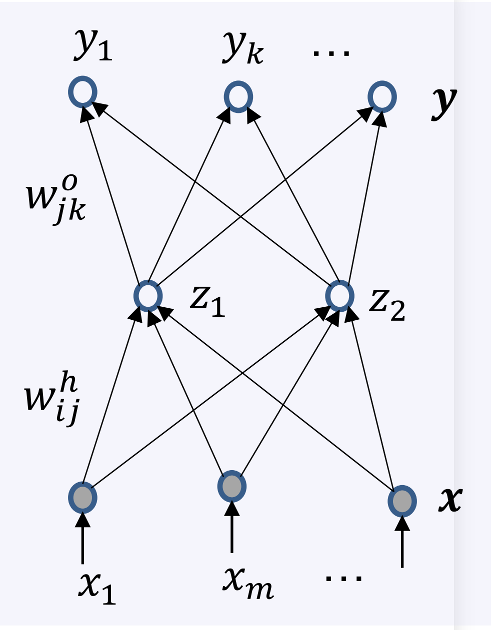
$$-\delta^o_k = -(y_k - \hat{y}_k)(1-\hat{y}_k)\hat{y}_k$$

- This gradient rule implies: $\boxed{w^o_{jk} \leftarrow w^o_{jk} + \eta \delta^o_k z_j}$

Training MLP: Backpropagation

- Similarly for input → hidden weights

$$E_t(\mathbf{w}) = \frac{1}{2}\sum_{k=1}^{K}(\hat{y}_k - y_k)^2 \qquad z_j = \sigma(\bar{z}_j) \text{ where } \bar{z}_j = \sum_{i=1}^{M} x_i w_{ij}^h$$



$$w_{ij}^h \leftarrow w_{ij}^h - \eta \frac{\partial E_t(\mathbf{w})}{\partial w_{ij}^h}$$

$= x_i$

$$\frac{\partial E_t(\mathbf{w})}{\partial w_{ij}^h} = \boxed{\frac{\partial E_t}{\partial \bar{z}_j}} \times \frac{\partial \bar{z}_j}{\partial w_{ij}^h}$$

$= z_j\ (1 - z_j)$

$$\boxed{\frac{\partial E_t}{\partial \bar{z}_j}} = \underbrace{\frac{\partial E_t}{\partial z_j}} \times \frac{\partial z_j}{\partial \bar{z}_j}$$

chain rule

$= w_{jk}^0$ since $\bar{y}_k = \sum w_{jk}^0 z_j$

$$\sum_{k=1}^{K} \boxed{\frac{\partial E_t}{\partial \bar{y}_k}} \times \frac{\partial \bar{y}_k}{\partial z_j}$$

$-\delta_k^o$

$$\frac{\partial E_t}{\partial \bar{z}_j} = -\delta_j^h = -z_j(1 - z_j)\sum_{k=1}^{K} w_{jk}^o \delta_k^o$$

- Gradient descent update

$$\boxed{w_{ij}^h \leftarrow w_{ij}^h + \eta \delta_j^h x_i}$$

Backpropagation (SGD)

Algorithm

Input: training data
Initialize weights // use small random numbers, between -0.5 and 0.5
Until stopping criteria is met **do**
    For each training sample **do**
        // propagate input forward
        Compute values of hidden nodes $\boldsymbol{z}$ and output nodes $\hat{\boldsymbol{y}}$
        // propagate error backward
        Compute output error term: $\delta_k^o = \hat{y}_k(1 - \hat{y}_k)(y_k - \hat{y}_k)$
        Compute hidden error term: $\delta_j^h = z_j(1 - z_j)\sum_{k=1}^{K} w_{jk}^o \delta_k^o$
        Update weights:
            Hidden → output: $w_{jk}^o \leftarrow w_{jk}^o + \eta\delta_k^o h_j$
            Input → hidden: $w_{ij}^h \leftarrow w_{ij}^h + \eta\delta_j^h x_i$
Output: trained weights

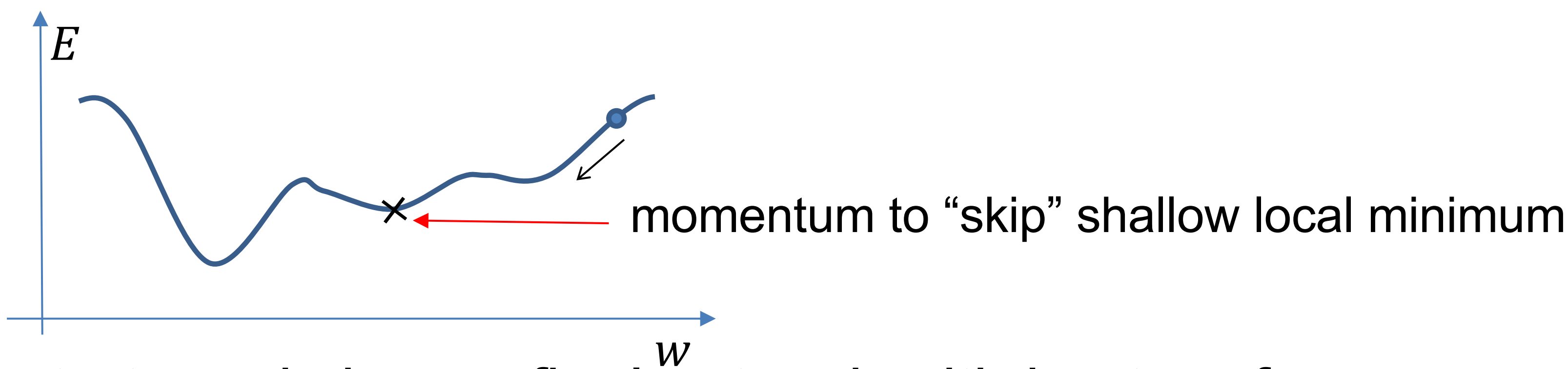
Issues with Backpropagation

Local minima.

- Possible fixes:
  - Add a momentum term in the update rule, e.g.,

$$w_{jk}^{o} \leftarrow w_{jk}^{o} + \eta \delta_k^o z_j + \overset{\text{momentum constant >0}}{\alpha} \underbrace{\Delta^{t-1} w_{jk}^{o}}_{\text{last amount of update of } w_{jk}^{o}}$$

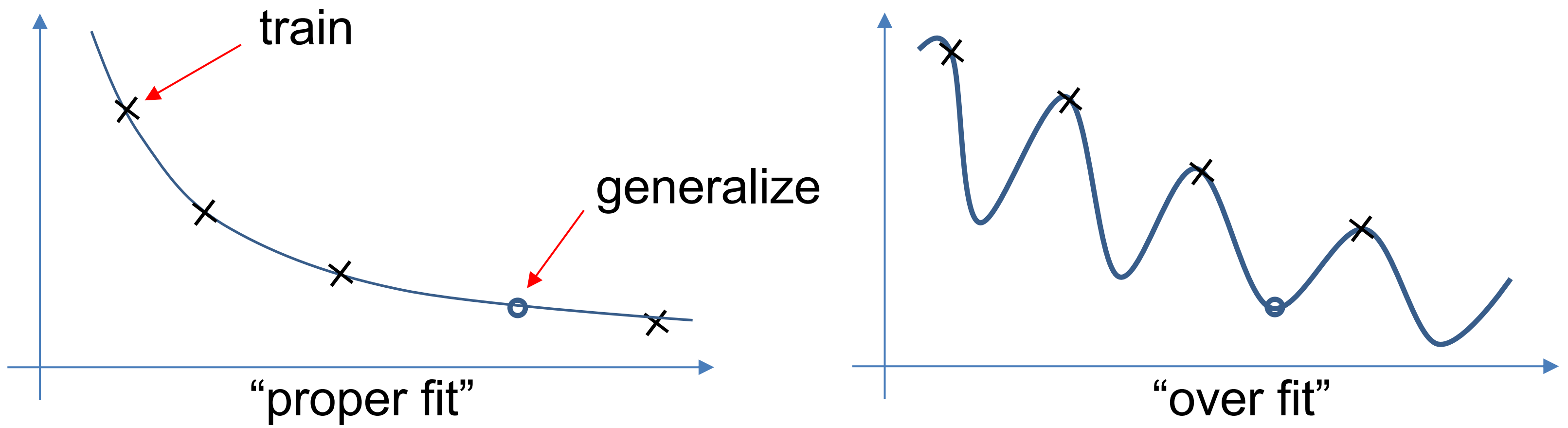
  - Can prevent getting stuck in shallow local minimum



- Multiple restarts and choose final network with best performance.

Issues with Backpropagation

Overfitting



- The tendency of the network to "memorize" all training samples, leading to poor generalization
- Usually happens with network of too many hidden nodes and overtrained.
- Possible fixes:
  - Use cross validation, e.g., stop training when validation error starts to grow.
  - Weight decaying: minimize also the magnitude of weights, keeping weights small (since the sigmoid function is almost linear near 0, if weights are small, decision surfaces are less non-linear and smoother)
  - Keep small number of hidden nodes!

Using MLP in Scikit Learn Python

Methods

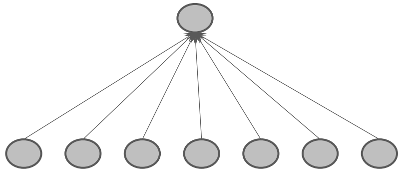
`fit` (X, y)	Fit the model to data matrix X and target y.
`get_params` ([deep])	Get parameters for this estimator.
`predict` (X)	Predict using the multi-layer perceptron classifier
`predict_log_proba` (X)	Return the log of probability estimates.
`predict_proba` (X)	Probability estimates.
`score` (X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
`set_params` (**params)	Set the parameters of this estimator.

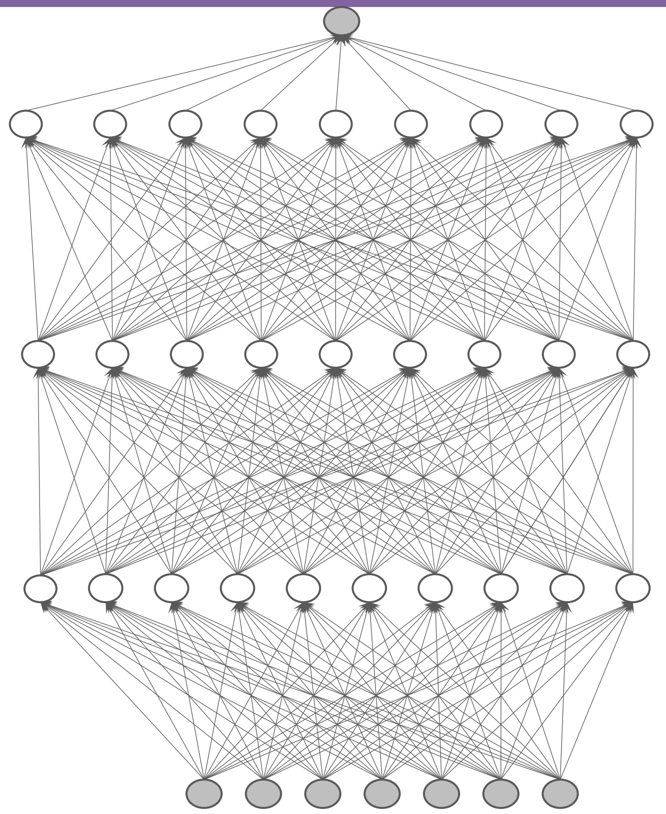
sklearn.neural_network.MLPClassifier

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*) [source]

Multi-layer Perceptron classifier.

Where to go from here?





?

Deep Learning

Deep Learning

- Deep Learning methods are advanced neural networks.

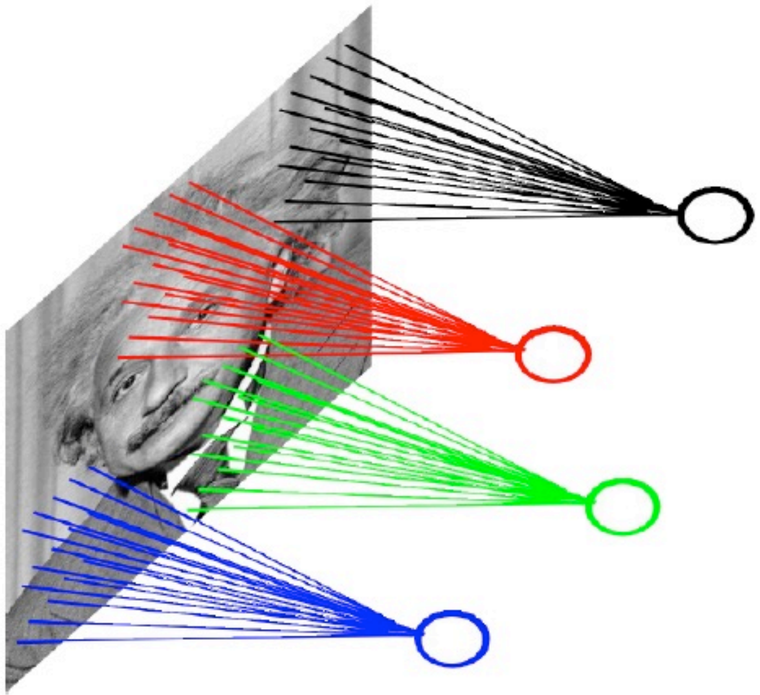
- They have been successful in learning many real world tasks e.g. handwritten digit recognition, image recognition!

- Some of the common Deep Learning architectures are:

  - **Convolutional Networks** (Due to Le Cun *et al.*)
  - **Autoencoders** (Due to Yoshua Bengio *et al.*)
  - Deep Belief Networks (due to Geoff Hinton *et al.*)
  - Boltzmann Machines
  - Restricted Boltzmann Machines
  - Deep Boltzmann Machines
  - Deep Neural Networks

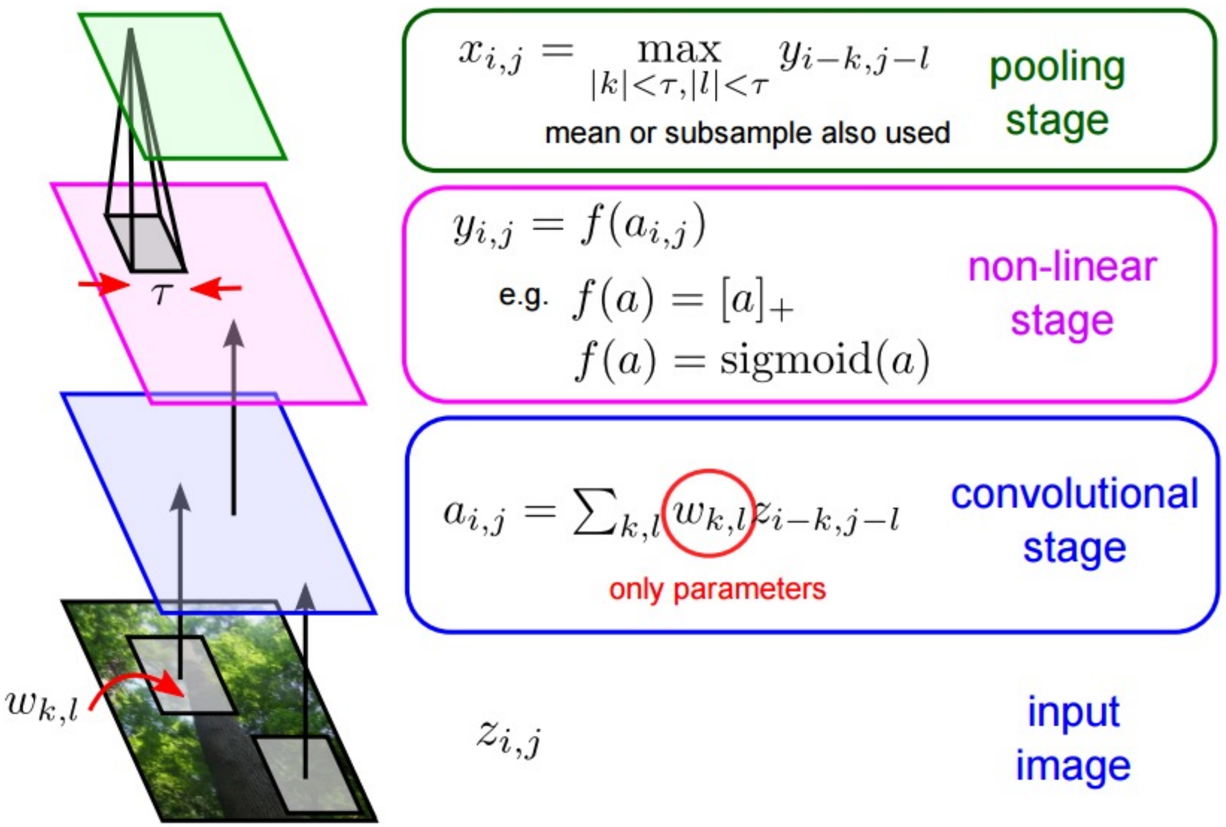
Convolutional Neural Networks

- Also called CNN or ConvNets.
- Motivation:
  - vision processing in our brain is fast
  - Also (Hubel & Wiesel, 62'):
    - Simple cells detect local features
    - Complex cells pool local features

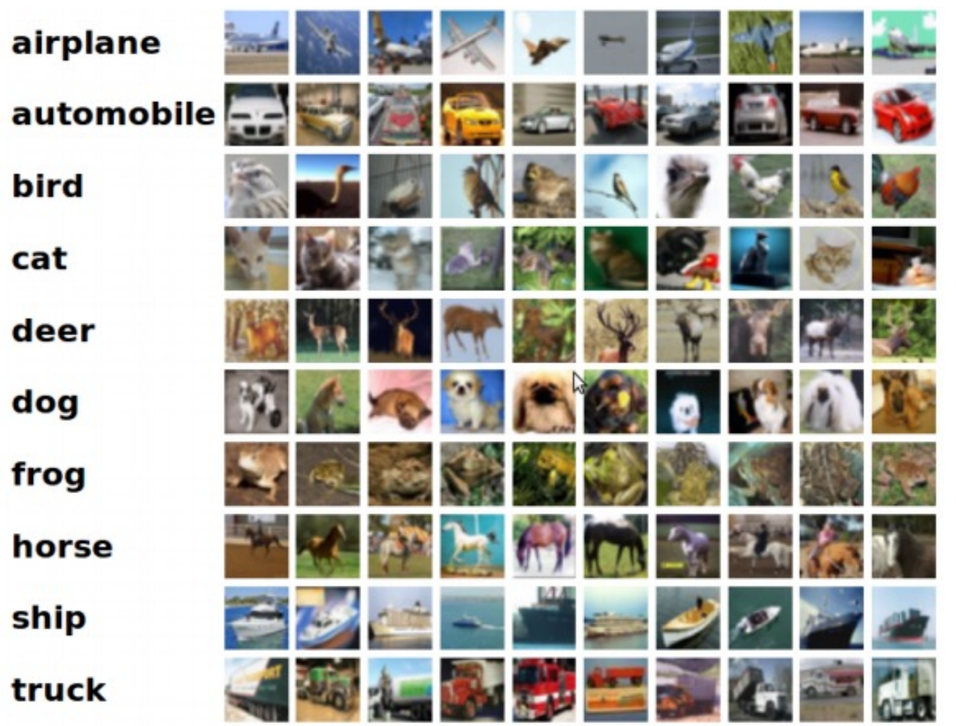


- Translated in technical terms:
  - Sparse interactions: sparse weights within a smaller kernel (e.g., 3x3, 5x5) instead of the whole input. This helps reduce #params.
  - Parameter sharing: a kernel use the same set of weights while applying onto different location (sliding windows).
  - Translation invariance.

Convolutional Neural Networks



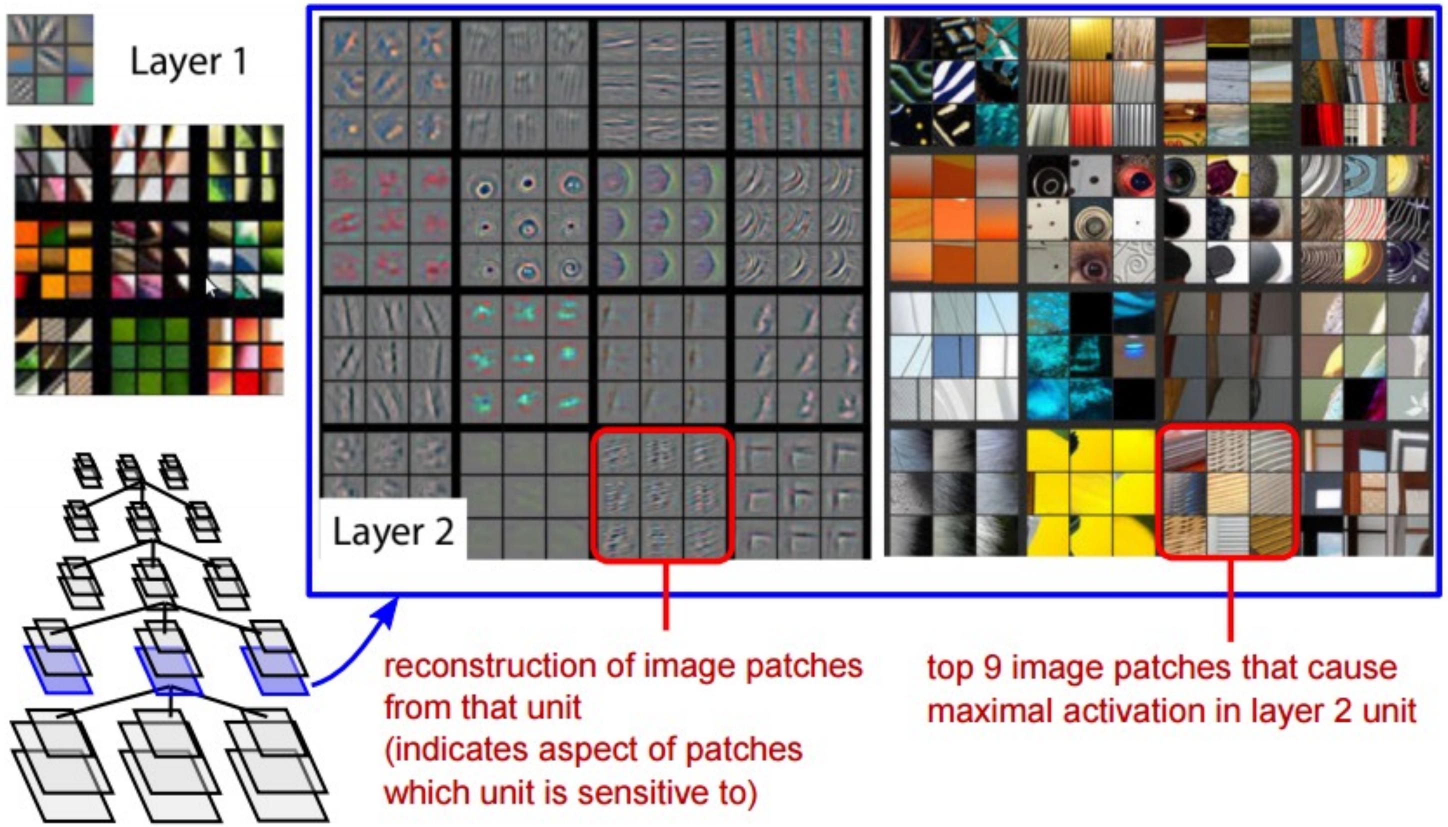
Application of CNN



CIFAR 10 dataset: 50,000 training images, 10,000 test images

`http://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html`

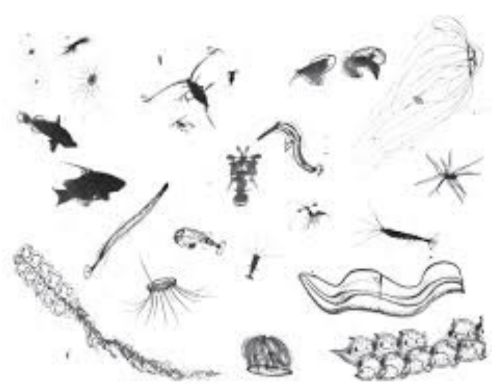
Peeping into CNN's Brain



ConvNets won all recent Computer Vision Challenges

- Galaxy

- Plankton



121 classes

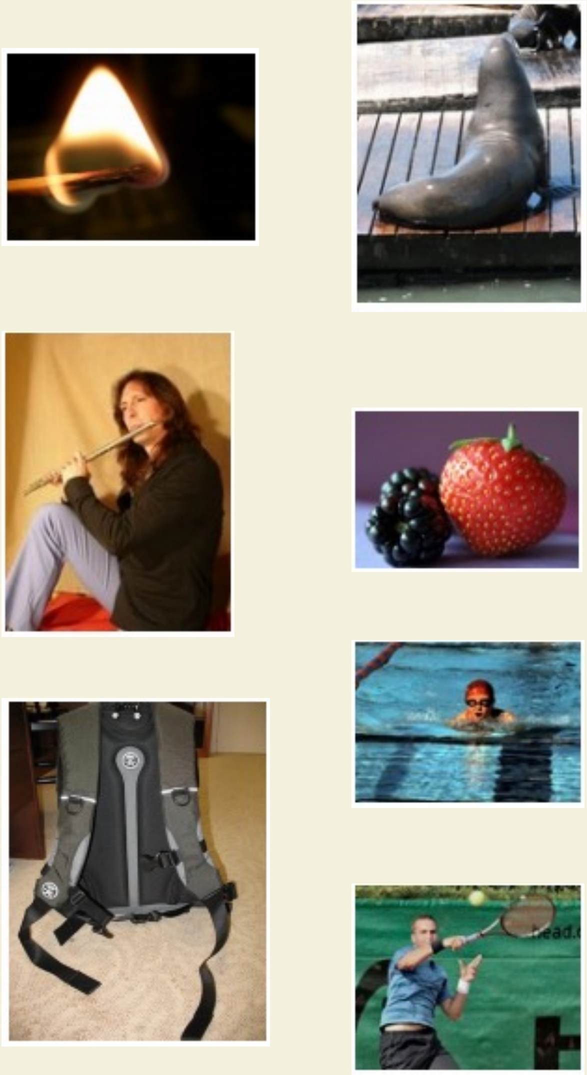
- Retina (high-res images)
  - Diabetes recognition from retina image

Read here:
http://benanne.github.io/2014/04/05/galaxy-zoo.html

http://benanne.github.io/2015/03/17/plankton.html

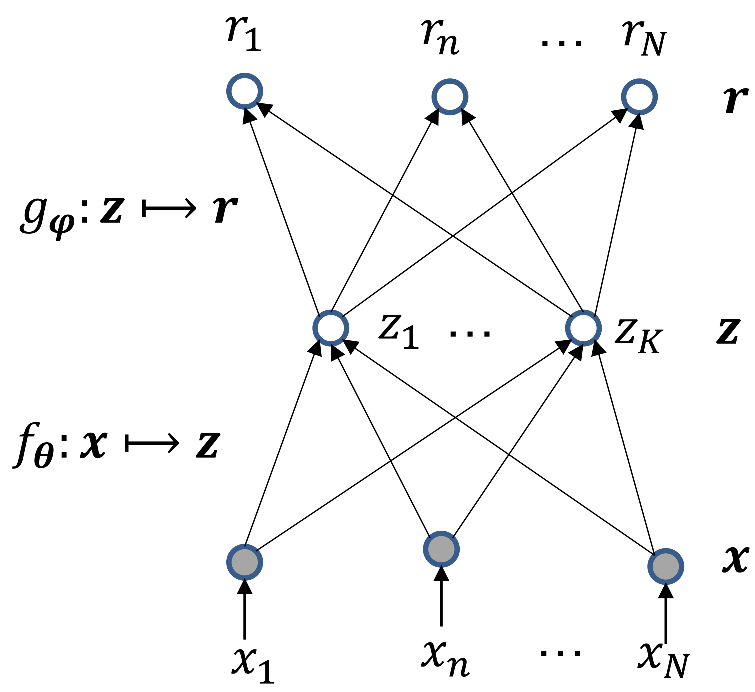
What helped Deep Learning?

- Larger models with new training techniques:
  - Dropout, Maxout, Maxnorm, ReLU,…
- Large ImageNet dataset [Fei-Fei et al. 2012]
  - 1.2 million training samples
  - 1000 categories
- Fast graphical processing units (GPU)
  - Capable of 1 trillion operations per second



Deep Autoencoder

- Simply a neural network that tries to copy its input to its output.
  - Input $\boldsymbol{x} = [x_1, x_2, \dots, x_N]^\top$
  - An encoder function f parameterized by $\boldsymbol{\theta}$
  - A coding representation $\boldsymbol{z} = [z_1, z_2, \dots, z_K]^\top = f_{\boldsymbol{\theta}}(\boldsymbol{x})$
  - A decoder function g parameterized by $\boldsymbol{\varphi}$
  - An output, also called reconstruction
    $$\boldsymbol{r} = [r_1, r_2, \dots, r_N]^\top = g_{\boldsymbol{\varphi}}(\boldsymbol{z}) = g_{\boldsymbol{\varphi}}\big(f_{\boldsymbol{\theta}}(\boldsymbol{x})\big)$$
  - A loss function $\mathcal{J}$ that computes a scalar $\mathcal{J}(\boldsymbol{x}, \boldsymbol{r})$ to measure how good of a reconstruction $\boldsymbol{r}$ of the given input $\boldsymbol{x}$, e.g., mean square error loss:
    $$\mathcal{J}(\boldsymbol{x}, \boldsymbol{r}) = \sum_{n=1}^{N} (x_n - r_n)^2$$
  - Learning objective: $\underset{\boldsymbol{\theta}, \boldsymbol{\varphi}}{\operatorname{argmin}}\, \mathcal{J}(\boldsymbol{x}, \boldsymbol{r})$



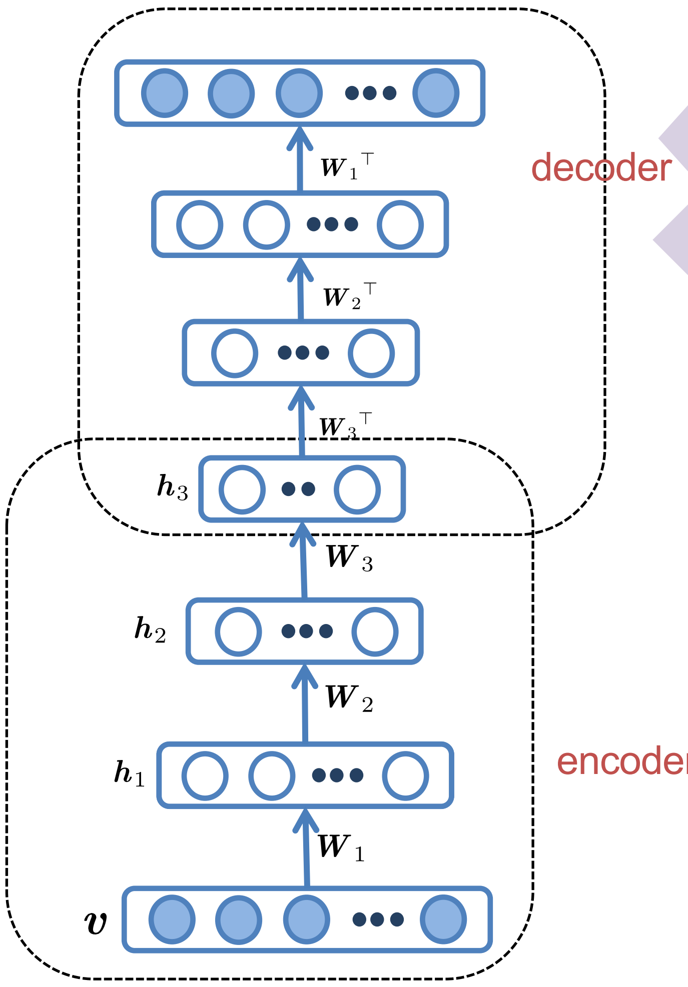
Deep Autoencoder

- Autoencoders are another way of feature learning.

- The solution is trivial unless there are constraints (such as sparsity) on the number of nodes in the hidden layers.

- Linear autoencoder with $K \leq N$ acts as PCA.

- However, it is nonlinearity that makes it powerful.



Deep Autoencoder

Resources

- Courses/Tutorials
  - Hinton Coursera: https://www.coursera.org/course/neuralnets
  - Hugo Youtube channel: https://www.youtube.com/user/hugolarochelle
  - Colah’s blog (http://colah.github.io/): very nice visualizations for easier and better understanding.
  - Karpathy’s blog (http://karpathy.github.io/): deep learning in web browsers.

Resources

- Tools
  - Tensorflow (Google)
    - Keras
  - Theano (Montreal – Bengio et. al.)
    - Lasange, Keras
  - Caffe (Berkeley)
  - Torch (Facebook)
  - Deep Scalable Sparse Tensor Network Engine (DSSTNE) (Amazon)
  - MatConvNet (Oxford, MATLAB)

Thank You.