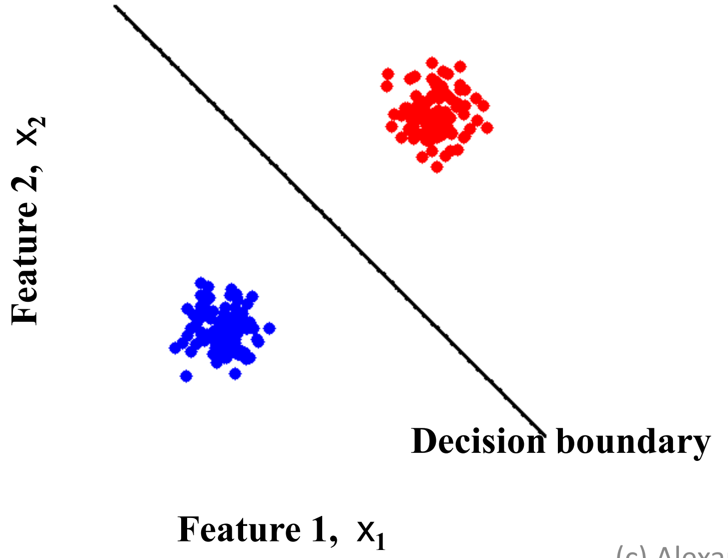
Neural Networks

Adopted from slides by Alexander Ihler and Andrew Ng

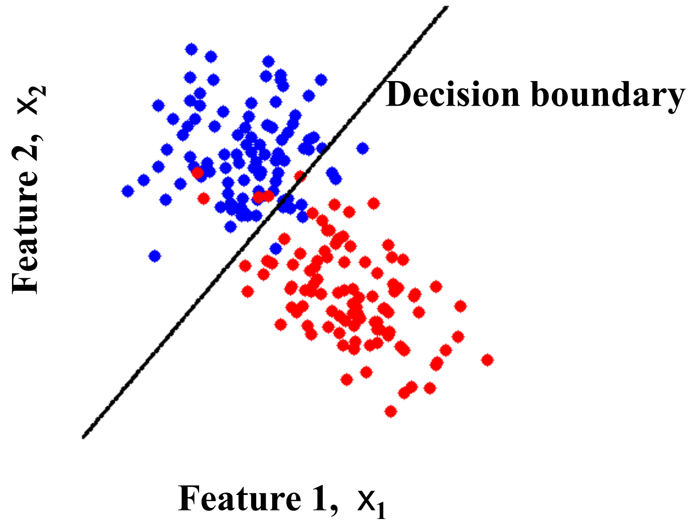
Linear classifiers (perceptrons)

- Linear Classifiers
  - a linear classifier is a mapping which partitions feature space using a linear function (a straight line, or a hyperplane)
  - separates the two classes using a straight line in feature space
  - in 2 dimensions the decision boundary is a straight line

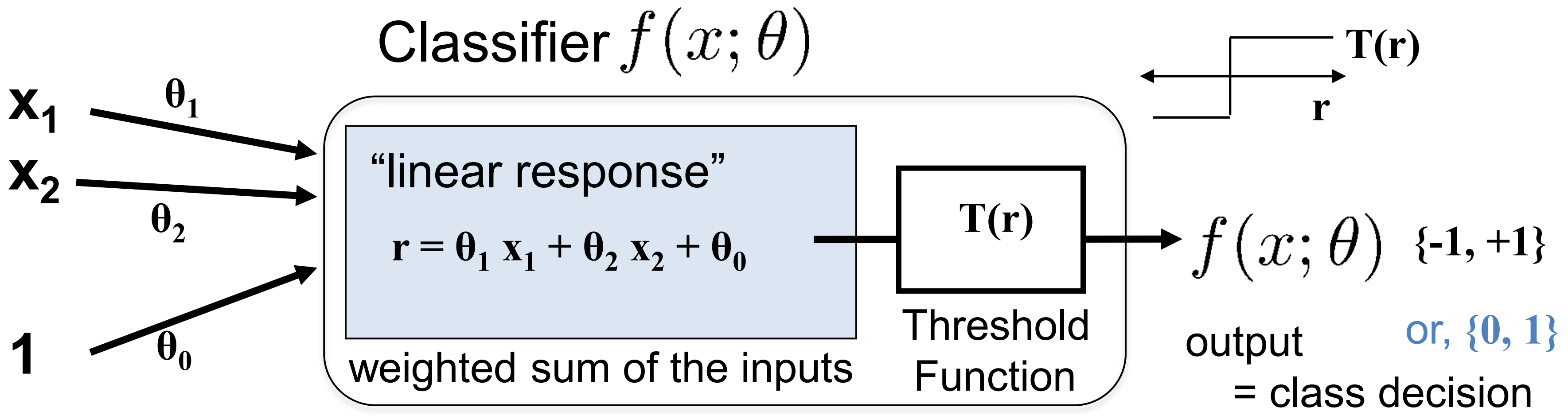
Linearly separable data



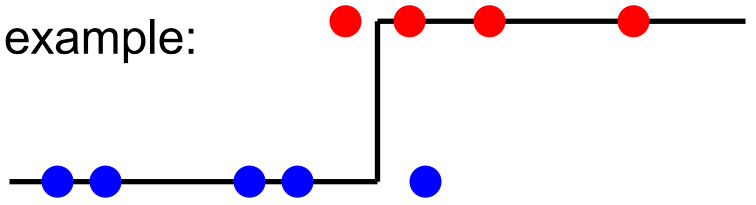
Linearly non-separable data



Perceptron Classifier (2 features)



1D example:

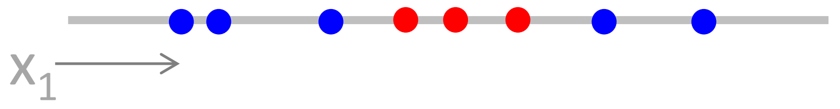


$T(r) = -1$ if $r < 0$

$T(r) = +1$ if $r > 0$

Decision boundary = "x such that $T(w_1 x + w_0)$ transitions"

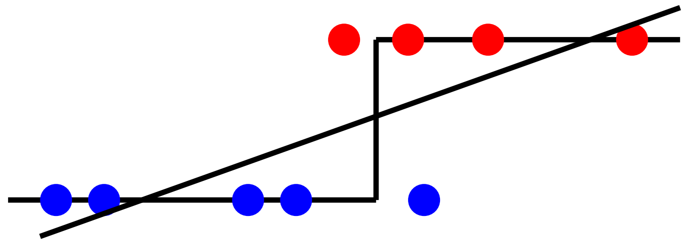
Non-linear classifiers



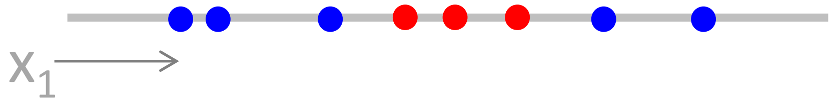
Not linearly separable

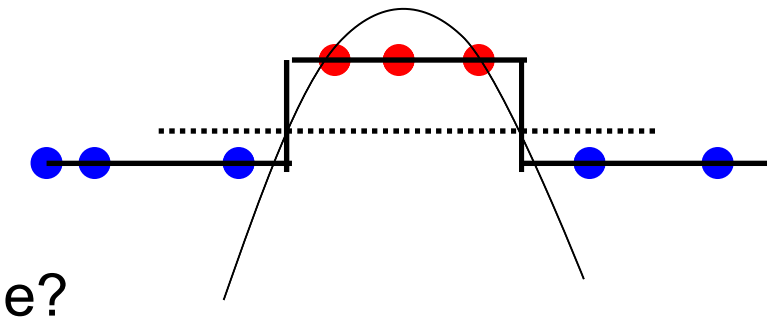
- How to obtain non-linear classifiers?
- Kernel SVM: use feature function $\Phi(x)$ to transform original features to new features (with higher dimensions)

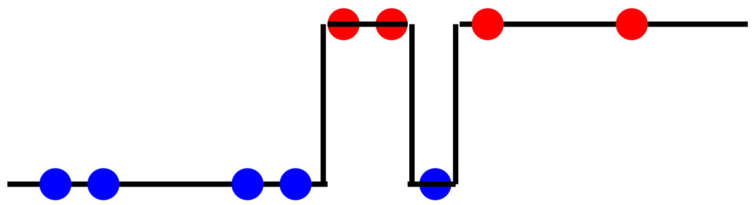
Features and perceptrons



- Recall the role of features
  - We can create extra features that allow more complex decision boundaries
  - Linear classifiers
  - Features [1,x]
    - Decision rule: T(ax+b) = ax + b >/< 0
    - Boundary ax+b =0 => point
  - Features [1,x,x^2]
    - Decision rule T(ax^2+bx+c)
    - Boundary ax^2+bx+c = 0 = ?
  - What features can produce this decision rule?



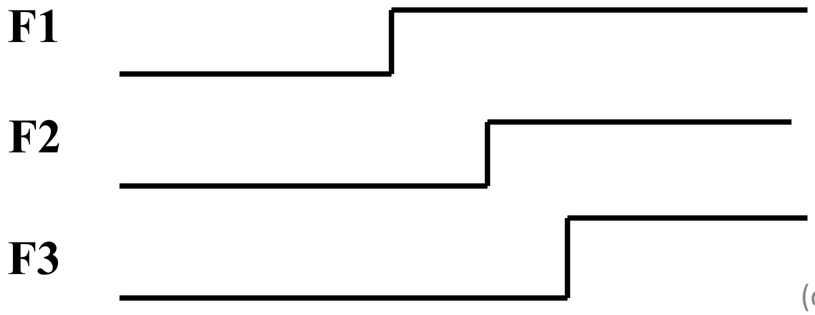




Features and perceptrons

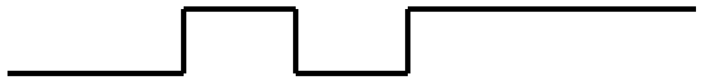
- Recall the role of features
  - We can create extra features that allow more complex decision boundaries
  - For example, polynomial features
    $\Phi(x) = [1 \;\; x \;\; x^2 \;\; x^3 \; \dots]$

- What other kinds of features could we choose?
  - Step functions?



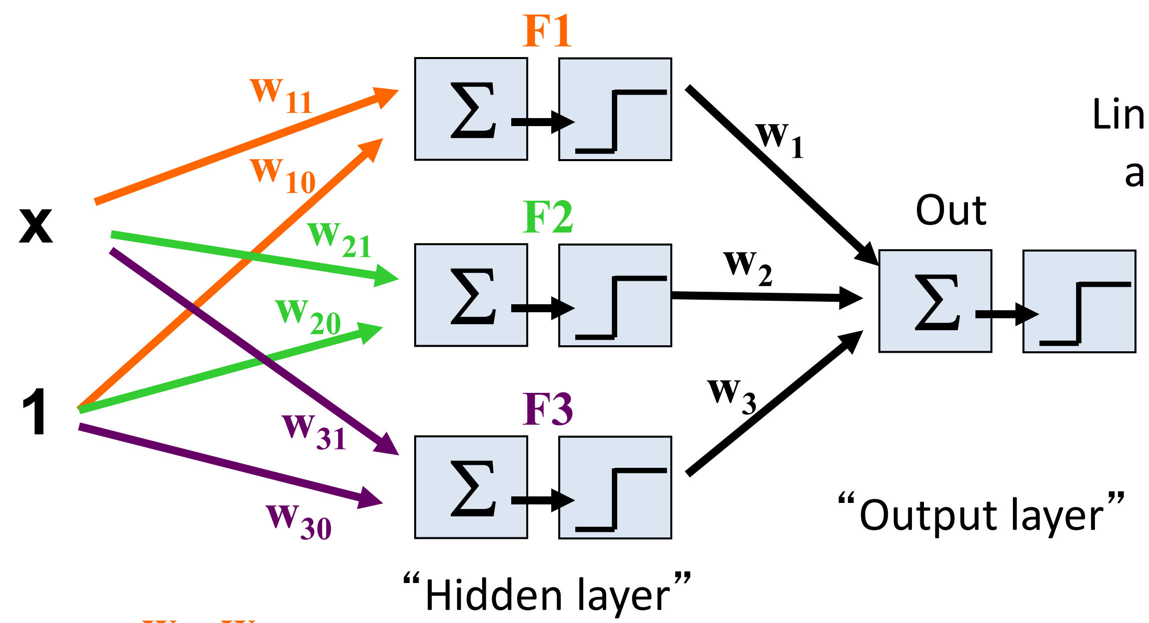
Linear function of features
a F1 + b F2 + c F3 + d

Ex: F1 – F2 + F3



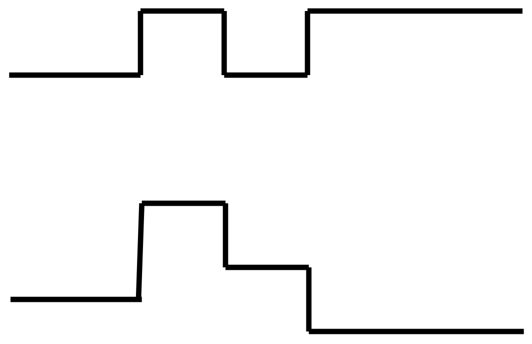
Multi-layer perceptron model

- Step functions are just perceptrons!
  - "Features" are outputs of a perceptron
  - Combination of features output of another



Linear function of features:
a F1 + b F2 + c F3 + d

Ex: F1 – F2 + F3



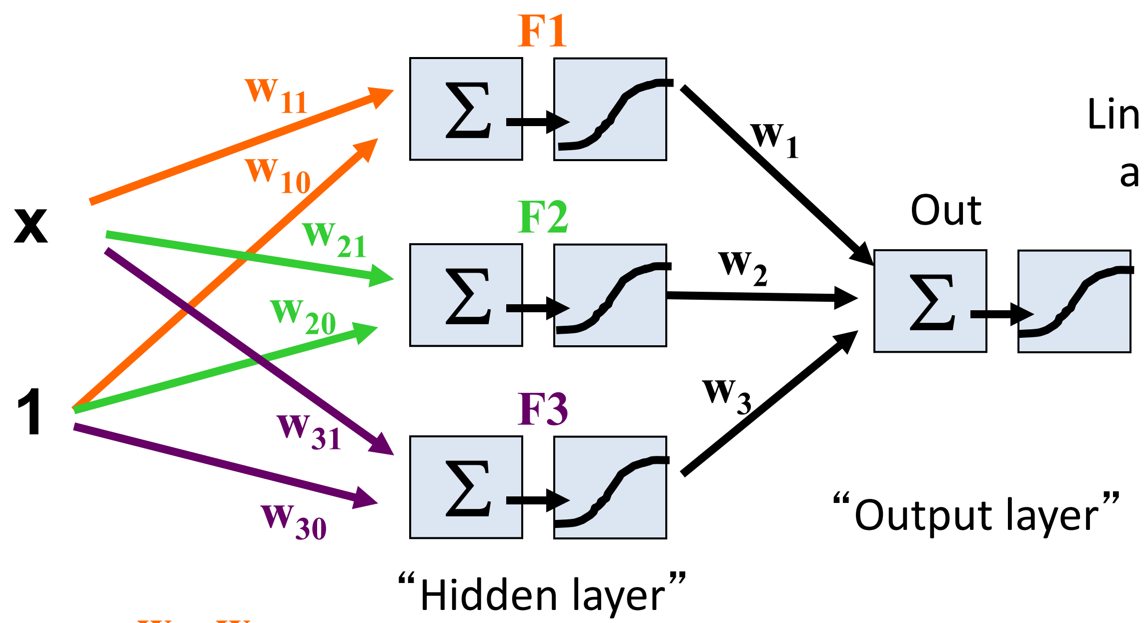
$$W^1 = \begin{matrix} w_{10} & w_{11} \\ w_{20} & w_{21} \\ w_{30} & w_{31} \end{matrix}$$

$$W^2 = w_1 \;\; w_2 \;\; w_3$$

Multi-layer perceptron model

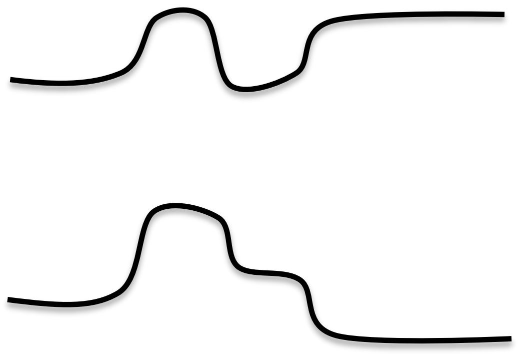
- Step functions are just perceptrons!
  - "Features" are outputs of a perceptron
  - Combination of features output of another

Regression version:
Remove activation
function from output



Linear function of features:
a F1 + b F2 + c F3 + d

Ex: F1 – F2 + F3

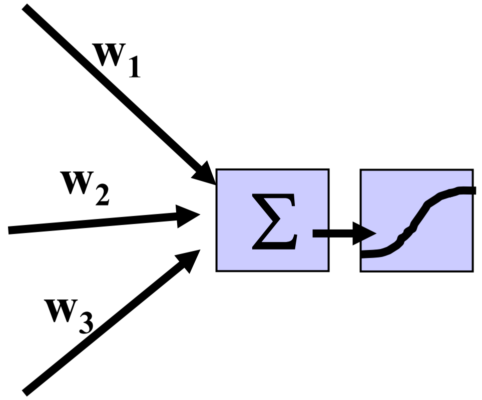


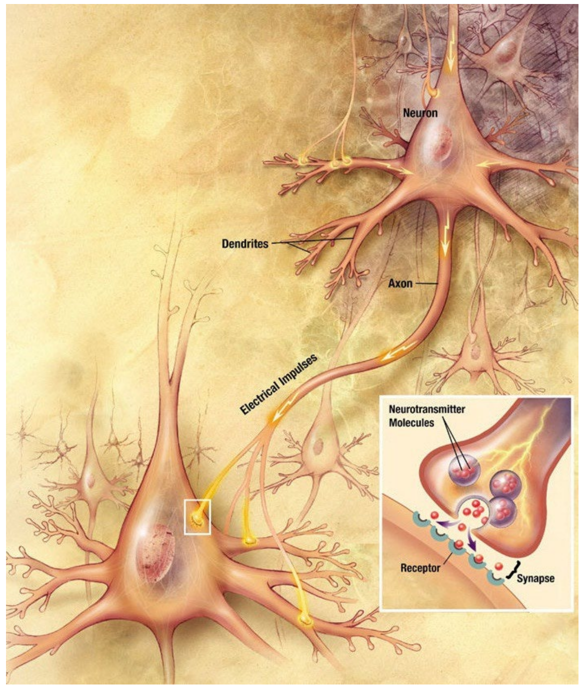
$$W^1 = \begin{matrix} w_{10} & w_{11} \\ w_{20} & w_{21} \\ w_{30} & w_{31} \end{matrix} \qquad W^2 = \begin{matrix} w_1 & w_2 & w_3 \end{matrix}$$

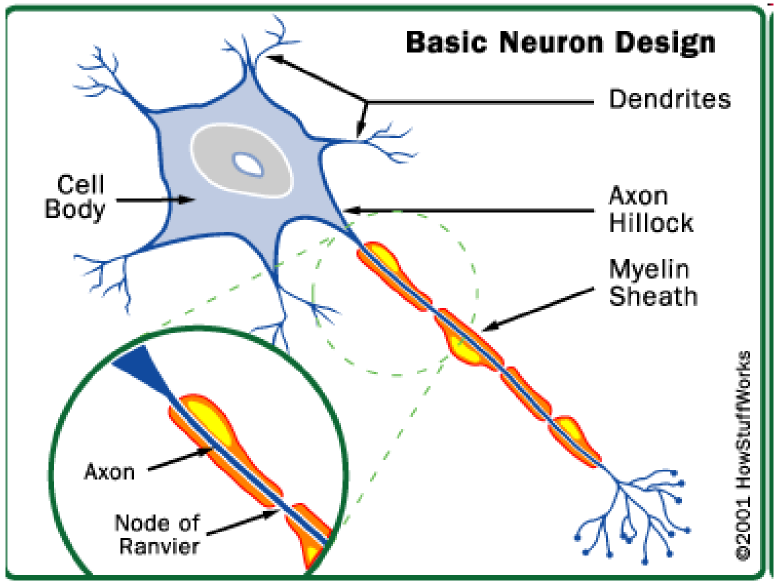
Neural networks

- Another term for MLPs
- Biological motivation

- Neurons
  - "Simple" cells
  - Dendrites sense charge
  - Cell weighs inputs
  - "Fires" axon







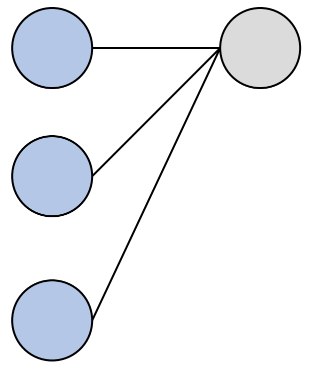
"How stuff works: the brain"

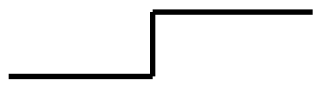
Representation

Features of MLPs

- Simple building blocks
  - Each element is just a perceptron f'n

- Can build upwards



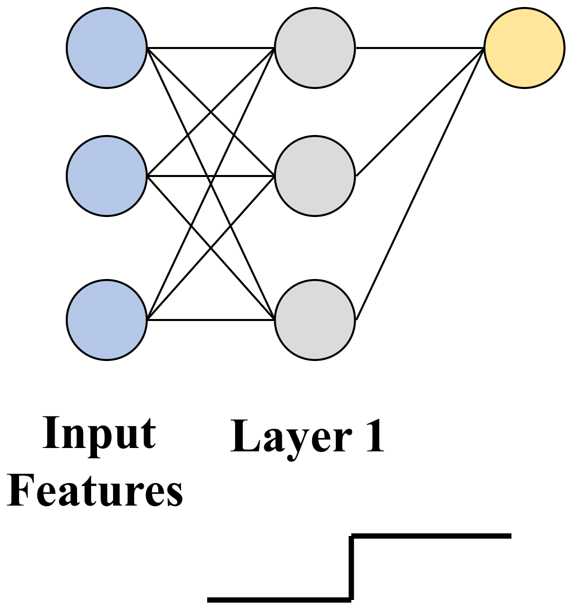


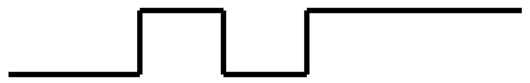
Perceptron:
Step function /
Linear partition

Features of MLPs

- Simple building blocks
  - Each element is just a perceptron f'n

- Can build upwards



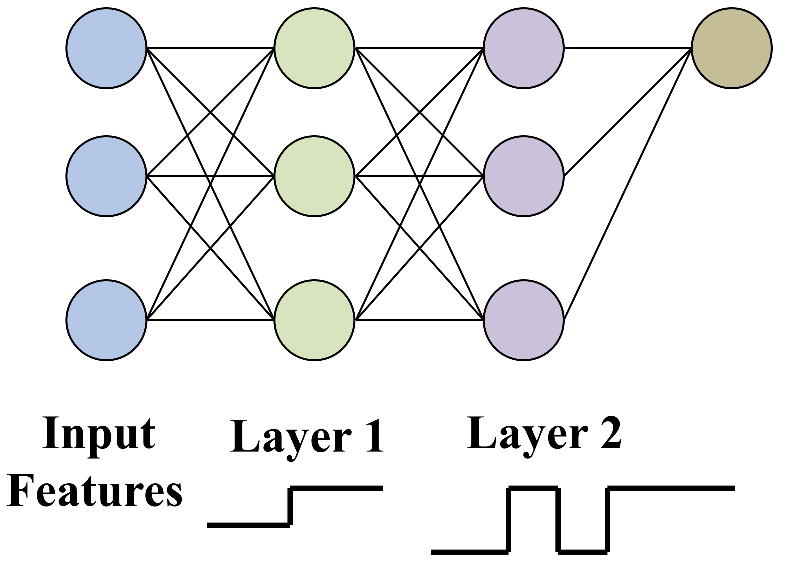


2-layer:
"Features" are now partitions
All linear combinations of those partitions

Features of MLPs

- Simple building blocks
  - Each element is just a perceptron f'n

- Can build upwards

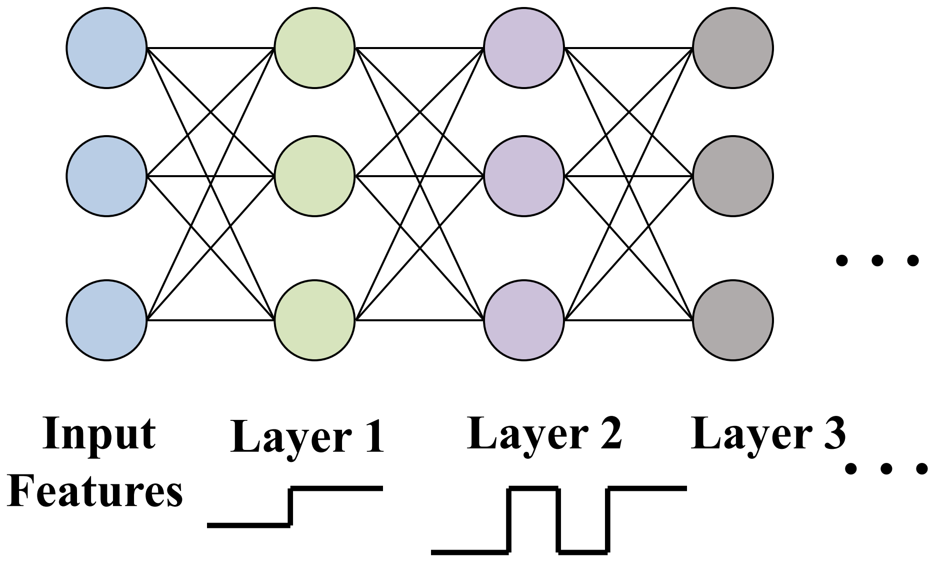


3-layer:
"Features" are now complex functions
Output any linear combination of those

Features of MLPs

- Simple building blocks
  - Each element is just a perceptron f'n

- Can build upwards

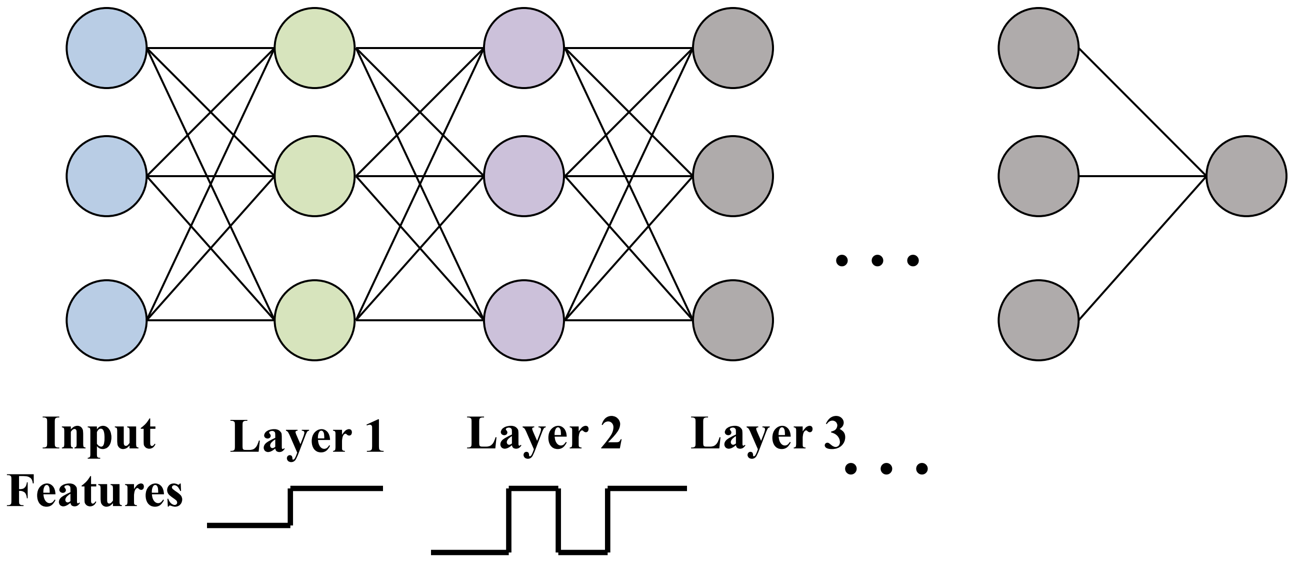


Current research:
"Deep" architectures
(many layers)

Features of MLPs

- Simple building blocks
  - Each element is just a perceptron f'n

- Can build upwards

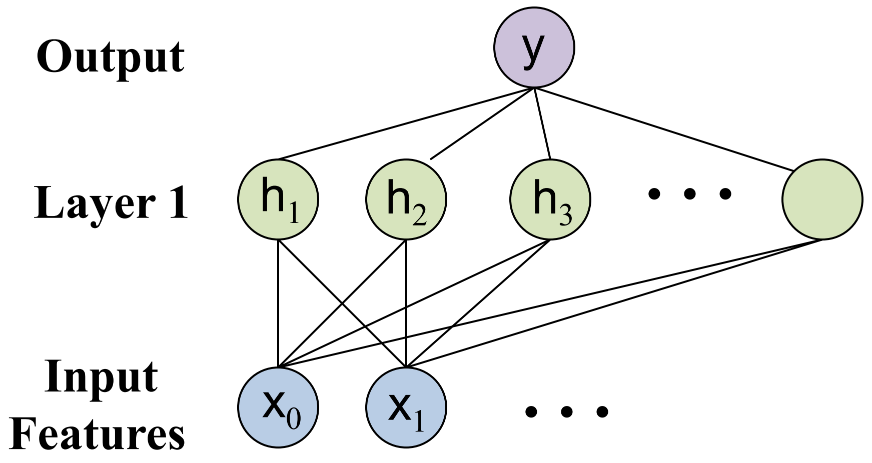


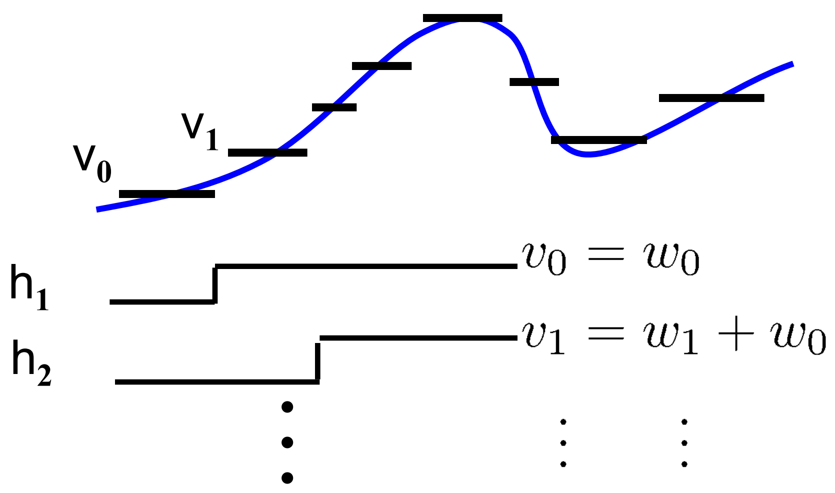
Features of MLPs

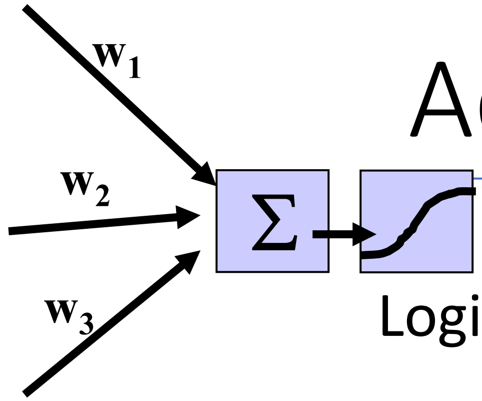
- Simple building blocks
  - Each element is just a perceptron function

- Can build upwards

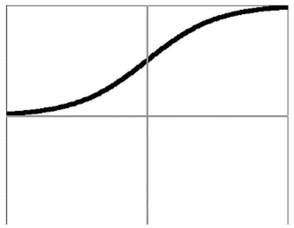
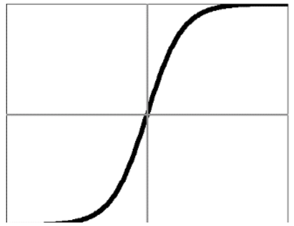
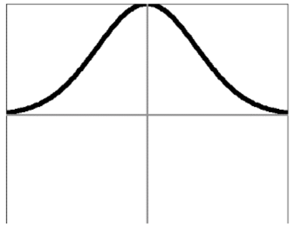
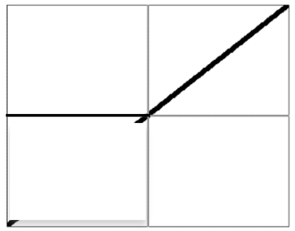
- Flexible function approximation
  - Approximate arbitrary functions with enough hidden nodes







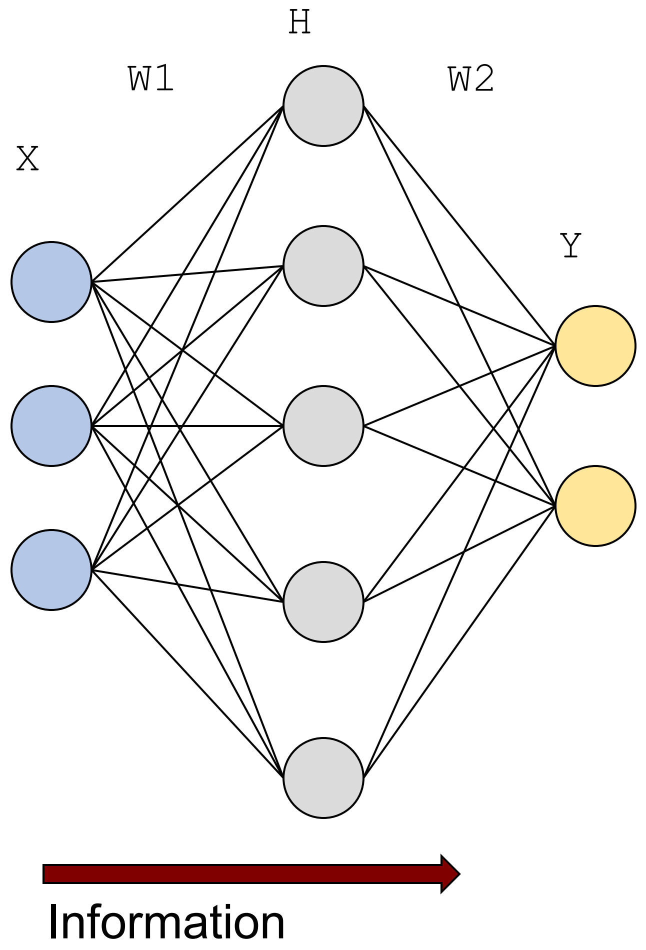
Activation functions

Logistic	$\sigma(z) = \frac{1}{1+\exp(-z)}$		$\frac{\partial\sigma}{\partial z}(z) = \sigma(z)(1-\sigma(z))$
Hyperbolic Tangent	$\sigma(z) = \frac{1-\exp(-2z)}{1+\exp(-2z)}$		$\frac{\partial\sigma}{\partial z}(z) = 1-(\sigma(z))^2$
Gaussian	$\sigma(z) = \exp(-z^2/2)$		$\frac{\partial\sigma}{\partial z}(z) = -z\sigma(z)$
ReLU (rectified linear)	$\sigma(z) = \max(0,z)$		$\frac{\partial\sigma}{\partial z}(z) = \mathbb{1}[z>0]$
Linear	$\sigma(z) = z$		and many others…

Feed-forward networks

- Information flows left-to-right
  - Input observed features
  - Compute hidden nodes (parallel)
  - Compute next layer…

- Alternative: recurrent NNs…



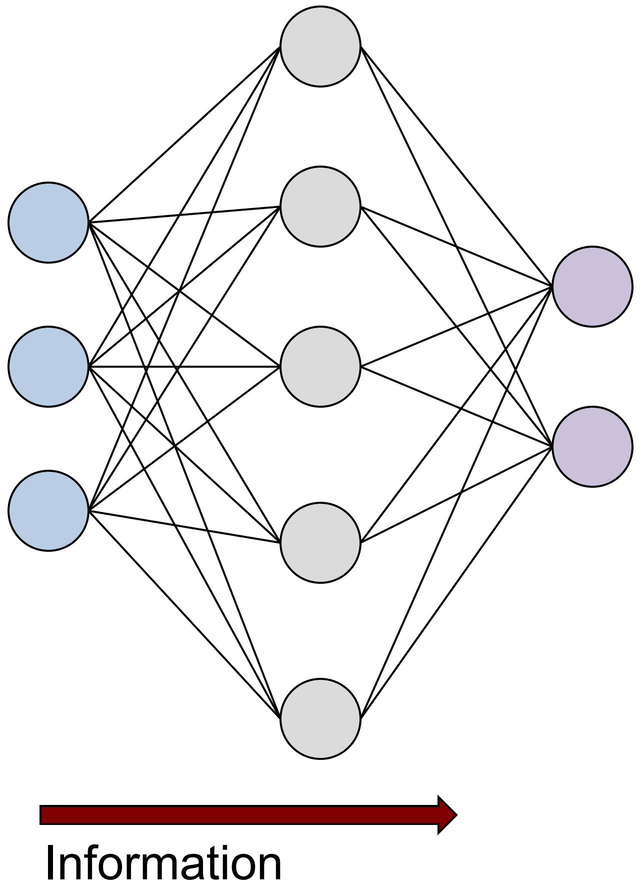
Feed-forward networks

A note on multiple outputs:

- Regression:
  - Predict multi-dimensional y
  - "Shared" representation = fewer parameters

- Classification
  - Predict binary vector
  - Multi-class classification y = 2 = [0 0 1 0 …]
  - Multiple, joint binary predictions (image tagging, etc.)

  - Often trained as regression (MSE), with saturating activation



Multiclass classification

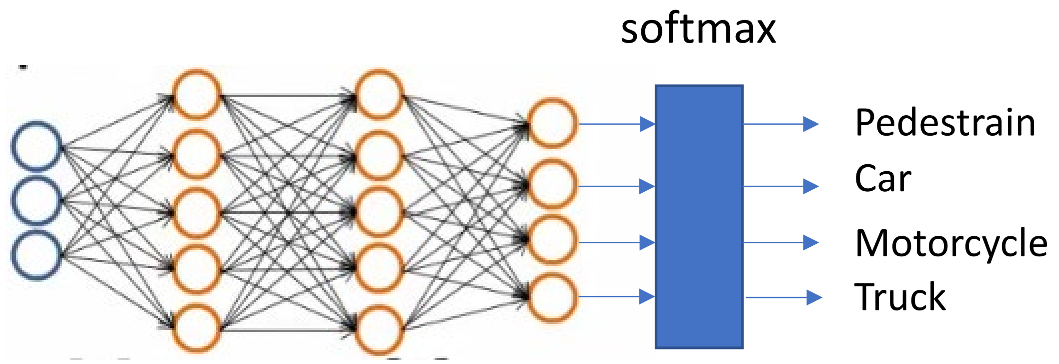
Pedestrian

Car

Motorcycle

Truck

- E.g., Recognizing pedestrian, car, motorbike or truck
- Output a vector of four variables
  - 1 is 0/1 pedestrian
  - 2 is 0/1 car
  - 3 is 0/1 motorcycle
  - 4 is 0/1 truck
- Training set here is images of our four classes



$y^{(i)}$ one of $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

Softmax and Cross Entropy Loss

- Cross Entropy Loss

$$\mathcal{L} = -\sum_i y^{(i)} \cdot \log \hat{y}^{(i)}$$

where

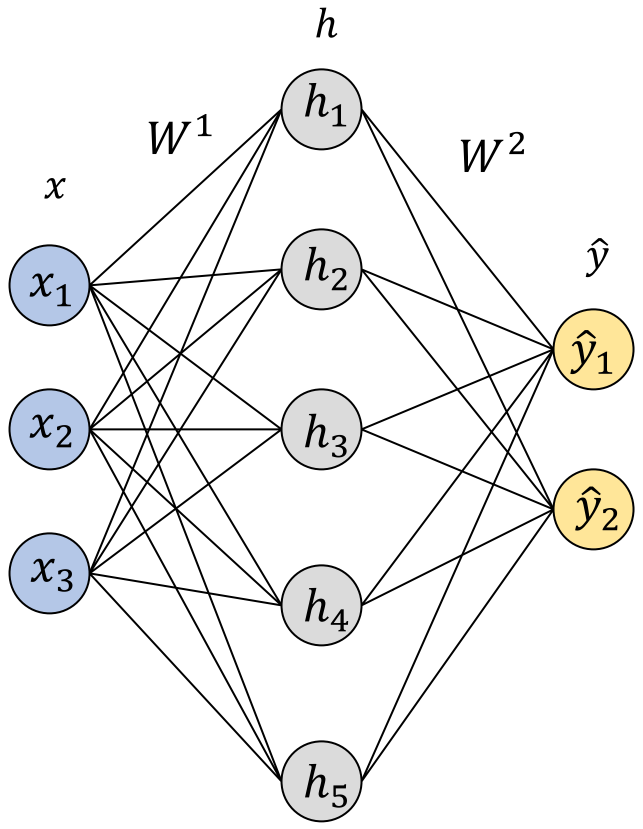
$$\hat{y}^{(i)} = \mathrm{softmax}(z^{(i)})$$

or $\hat{y}_c^{(i)} = \dfrac{e^{z_c^{(i)}}}{\sum_{c'} e^{z_{c'}^{(i)}}}$

Notations

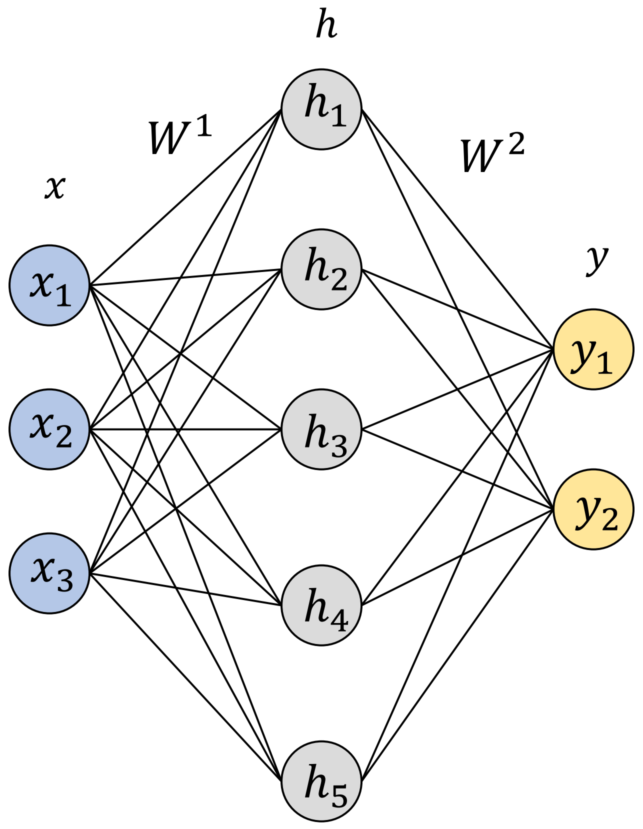
- $h_1 = \sigma\left(w_{1,1}^1 x_1 + w_{1,2}^1 x_2 + w_{1,3}^1 x_3\right)$
- $h_2 = \sigma\left(w_{2,1}^1 x_1 + w_{2,2}^1 x_2 + w_{2,3}^1 x_3\right)$
- …
- $\hat{y}_1 = \text{softmax}(w_{1,1}^2 h_1 + w_{1,2}^2 h_2 + w_{1,3}^2 h_3 + w_{1,4}^2 h_4 + w_{1,5}^2 h_5)$
- …

- $t = W^1 x, \quad h = \sigma(t)$
- $s = W^2 h, \quad \hat{y} = \text{softmax}(s)$
- $W^1 = \begin{pmatrix} w_{1,1}^1 & \cdots & w_{1,3}^1 \\ \vdots & \ddots & \vdots \\ w_{5,1}^1 & \cdots & w_{5,3}^1 \end{pmatrix}$
- $W^2 = \begin{pmatrix} w_{1,1}^2 & \cdots & w_{1,5}^2 \\ \vdots & \ddots & \vdots \\ w_{2,1}^2 & \cdots & w_{2,5}^2 \end{pmatrix}$

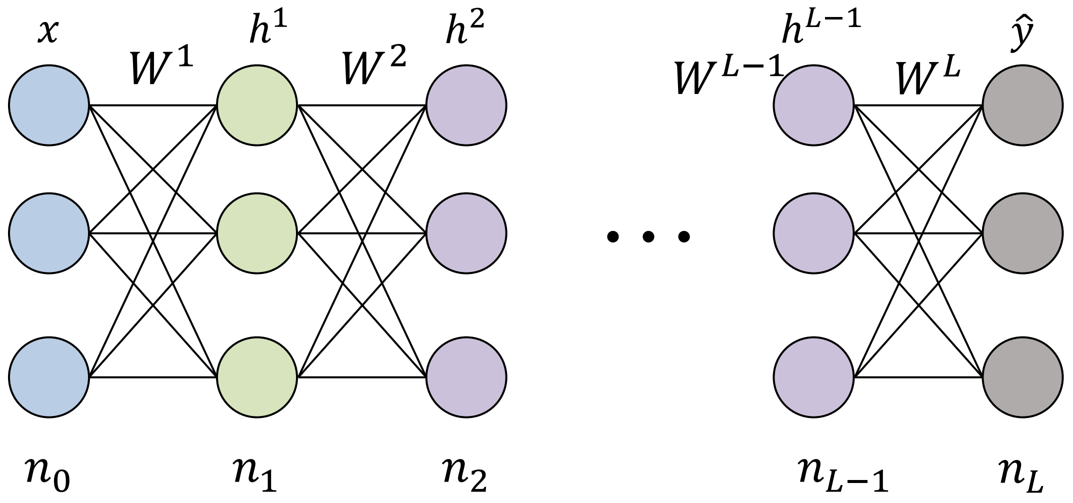


Forward Propagation

- $W^1 = \begin{pmatrix} w^1_{1,1} & \cdots & w^1_{1,3} \\ \vdots & \ddots & \vdots \\ w^1_{5,1} & \cdots & w^1_{5,3} \end{pmatrix}$
- $W^2 = \begin{pmatrix} w^2_{1,1} & \cdots & w^2_{1,5} \\ \vdots & \ddots & \vdots \\ w^2_{2,1} & \cdots & w^2_{2,5} \end{pmatrix}$
- $X = \begin{pmatrix} x^{(1)} \\ \cdots \\ x^{(m)} \end{pmatrix}$
- $T = X \cdot (W^1)^T, \qquad H = \sigma(T)$
- $S = H \cdot (W^2)^T, \qquad \hat{Y} = \text{softmax}(S)$



Neural networks



$$W^l = \begin{pmatrix} w^l_{1,1} & \cdots & w^l_{1,n_{l-1}} \\ \vdots & \ddots & \vdots \\ w^l_{n_l,1} & \cdots & w^l_{n_l,n_{l-1}} \end{pmatrix}$$

Forward propagation

$$z^1 = W^1 x \qquad h^1 = \sigma(z^1)$$

$$z^2 = W^2 h^1 \qquad h^2 = \sigma(z^2)$$

…

$$z^{L-1} = W^{L-1} h^{L-2} \qquad h^{L-1} = \sigma(z^{L-1})$$

$$z^L = W^L h^{L-1} \qquad \hat{y} = \sigma(z^L)$$

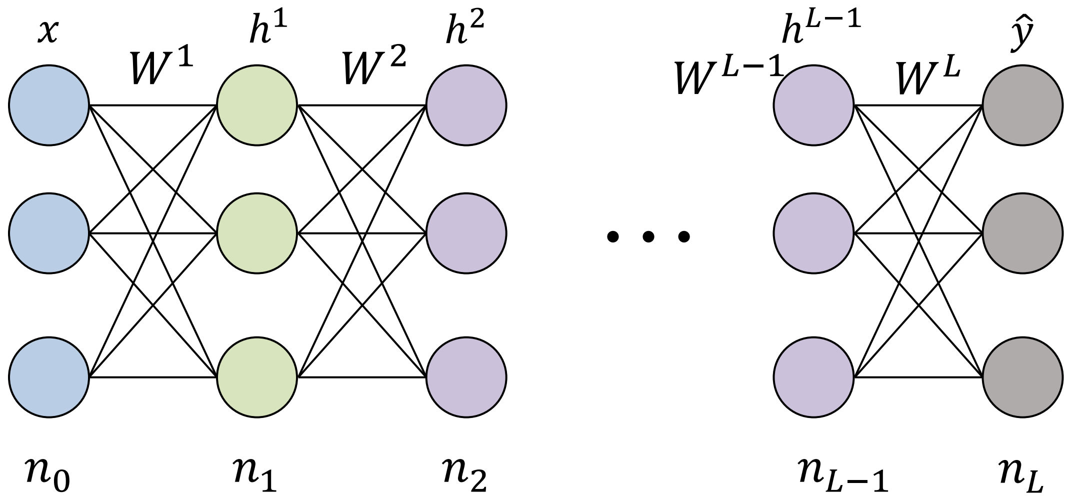
Overall, we can denote $\hat{y} = h_W(x)$

Neural Networks: Backpropagation

Adopted from slides by Alexander Ihler and Andrew Ng

Evaluation – Loss Functions

Neural networks



$$W^l = \begin{pmatrix} w^l_{1,1} & \cdots & w^l_{1,n_{l-1}} \\ \vdots & \ddots & \vdots \\ w^l_{n_l,1} & \cdots & w^l_{n_l,n_{l-1}} \end{pmatrix}$$

Forward propagation

$$z^1 = W^1 x \qquad h^1 = \sigma(z^1)$$

$$z^2 = W^2 h^1 \qquad h^2 = \sigma(z^2)$$

…

$$z^{L-1} = W^{L-1} h^{L-2} \qquad h^{L-1} = \sigma(z^{L-1})$$

$$z^L = W^L h^{L-1} \qquad \hat{y} = \sigma(z^L)$$

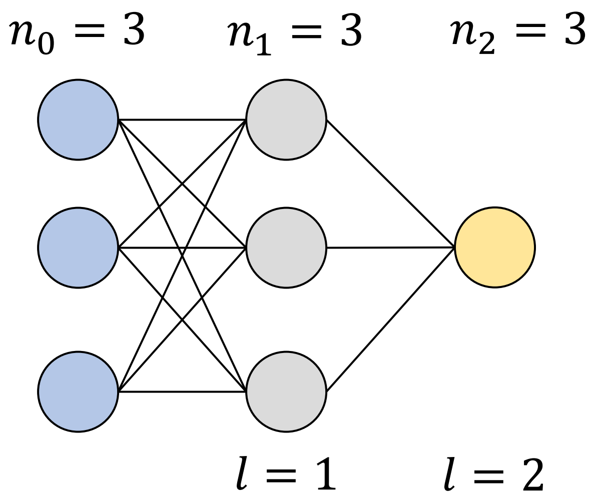
Overall, we can denote $\hat{y} = h_W(x)$

Loss Functions

- Similar to other machine learning models
- For continuous target, use MSE

$$\begin{aligned} J(W) &= \frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)} - \hat{y}^{(i)}\right)^2 \\ &= \frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)} - h_W(x^{(i)})\right)^2 \end{aligned}$$

- For discrete target, use cross entropy loss



Loss function (single output node)

- Regularized logistic regression:

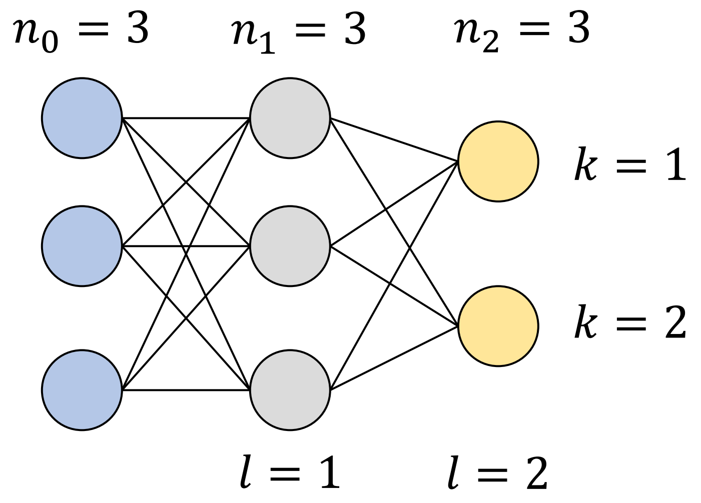
$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log \underset{\sigma(\theta \cdot x^{(i)})}{\boxed{h_\theta\left(x^{(i)}\right)}} + \left(1-y^{(i)}\right)\log\left(1-h_\theta\left(x^{(i)}\right)\right)\right) + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

- Neural network with cross entropy loss:

$$J(W) = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log \boxed{h_W\left(x^{(i)}\right)} + \left(1-y^{(i)}\right)\log\left(1-h_W\left(x^{(i)}\right)\right)\right) + \frac{\lambda}{2m}\sum_{l=1}^{L}\sum_{j=1}^{n_l}\sum_{i=1}^{n_{l-1}}\left(w_{j,i}^{l}\right)^2$$

Cross Entropy Loss (multiple output nodes)

$$J(W) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \cdot \log \mathrm{softmax}\left(h_w\left(x^{(i)}\right)\right) + \frac{\lambda}{2m}\sum_{l=1}^{L}\sum_{j=1}^{n_l}\sum_{i=1}^{n_{l-1}}\left(w_{j,i}^{l}\right)^2$$

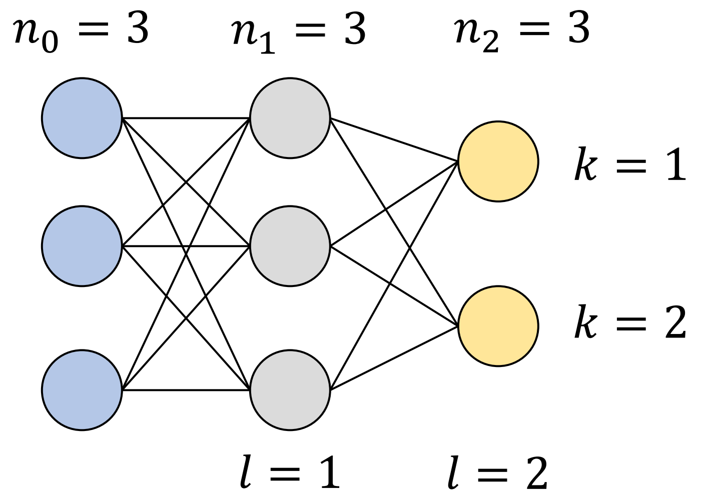


Loss function (multiple output nodes)

$$J(W) =$$

$$-\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left(y_k^{(i)}\log h_w\left(x^{(i)}\right)_k + \left(1 - y_k^{(i)}\right)\log\left(1 - h_w\left(x^{(i)}\right)_k\right)\right)$$

$$+\frac{\lambda}{2m}\sum_{l=1}^{L}\sum_{j=1}^{n_l}\sum_{i=1}^{n_{l-1}}\left(w_{j,i}^{l}\right)^2$$



Loss function

$$J(W) = \underbrace{\frac{1}{m}\sum_{i=1}^{m}\mathcal{L}\Big(y^{(i)}, \overset{\hat{y}^{(i)}}{\boxed{h_W(x^{(i)})}}\Big)}_{\textbf{empirical loss}} + \underbrace{\frac{\lambda}{2m}\mathcal{R}(W)}_{\textbf{regularizer}}$$

Optimization – Backpropagation

Optimization - gradient descent

```
Initialize $W$
Do {
   $W \leftarrow W - \alpha \nabla J(W)$
} while (stop condition)
```

- Compute $\frac{\partial J(W)}{\partial w_{j,i}^{l}}$
- Different from logistic regression (or SVM), where all parameters are equivalent in terms of positions in the loss function, parameters in neural network are different due to their locations in the network structure.

Gradient calculations

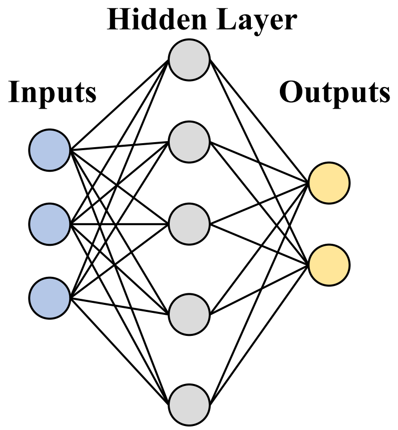
$$J(W) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}\left(y^{(i)}, h_W(x^{(i)})\right) + \frac{\lambda}{2m}\mathcal{R}(W)$$

- Compute $\frac{\partial\mathcal{L}(y,\hat{y})}{\partial w_{j,i}^{l}}$ where $\hat{y} = h_W(x)$

Gradient calculations

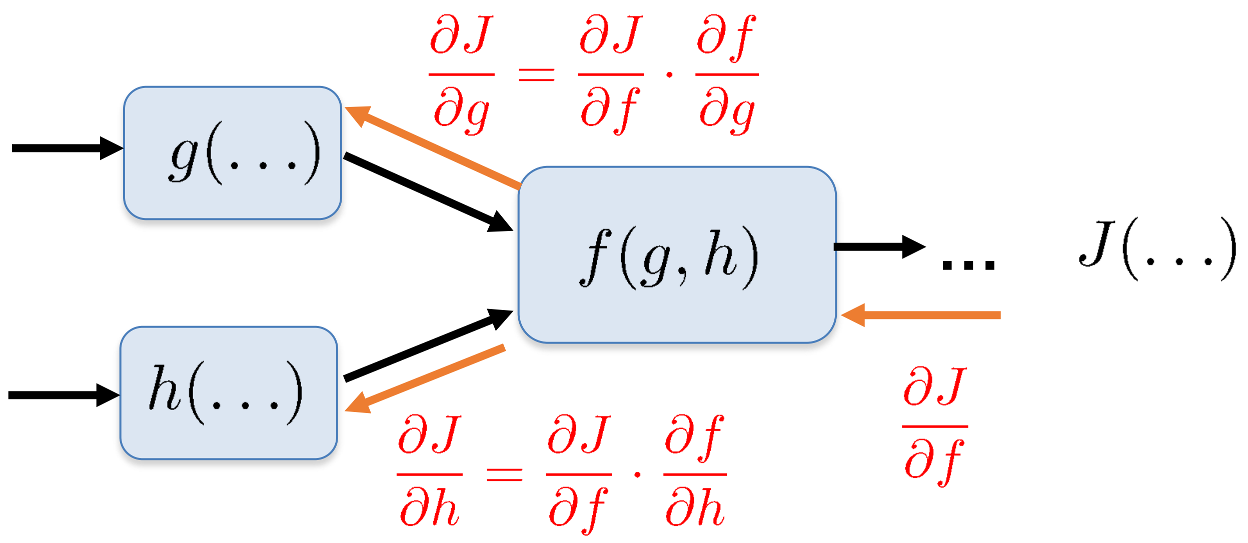
- Think of NNs as “schematics” made of smaller functions
  - Building blocks: summations & nonlinearities
  - For derivatives, just apply the chain rule, etc!

$$\begin{tikzcd}
{} \arrow[r] & g(\ldots) \arrow[dr] & & & \\
{} \arrow[ur] & & f(g,h) \arrow[r] & \ldots & J(\ldots) \\
{} \arrow[r] & h(\ldots) \arrow[ur] & & &
\end{tikzcd}$$



Gradient calculations

- Chain rule:
  - If $F(x) = f\big(g(x)\big)$, then $F'(x) = f'\big(g(x)\big)g'(x)$
  - Alternative form, if z depends on y, and y depends on x, then $\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$



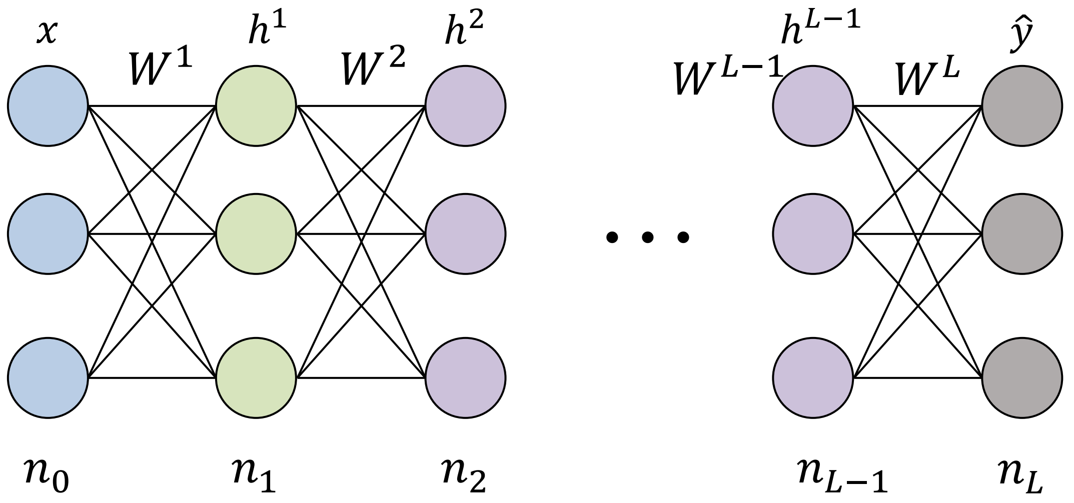
$\frac{\partial J}{\partial g} = \frac{\partial J}{\partial f} \cdot \frac{\partial f}{\partial g}$
$g(\ldots)$
$f(g,h)$
...
$J(\ldots)$
$h(\ldots)$
$\frac{\partial J}{\partial h} = \frac{\partial J}{\partial f} \cdot \frac{\partial f}{\partial h}$
$\frac{\partial J}{\partial f}$

Ex: $f(g,h) = g^2 h$

$$\frac{\partial J}{\partial g} = \frac{\partial J}{\partial f} \cdot 2\, g(\cdot)\, h(\cdot) \qquad \frac{\partial J}{\partial h} = \frac{\partial J}{\partial f} \cdot g^2(\cdot)$$

save & reuse info (g,h) from forward computation!

Backpropagation



Forward propagation

$$z^1 = W^1 x \qquad h^1 = \sigma(z^1)$$

$$z^2 = W^2 h^1 \qquad h^2 = \sigma(z^2)$$

…

$$z^{L-1} = W^{L-1} h^{L-2} \qquad h^{L-1} = \sigma(z^{L-1})$$

$$z^L = W^L h^{L-1} \qquad \hat{y} = \sigma(z^L)$$

$$W^l = \begin{pmatrix} w^l_{1,1} & \cdots & w^l_{1,n_{l-1}} \\ \vdots & \ddots & \vdots \\ w^l_{n_l,1} & \cdots & w^l_{n_l,n_{l-1}} \end{pmatrix}$$

$$J = \mathcal{L}(y, \hat{y}) \text{ or } \sum_k \mathcal{L}(y_k, \hat{y}_k)$$

Backpropagation

$$z^1 = W^1 x \qquad h^1 = \sigma(z^1)$$

$$z^2 = W^2 h^1 \qquad h^2 = \sigma(z^2)$$

$$\ldots$$

$$z^{L-1} = W^{L-1} h^{L-2} \quad h^{L-1} = \sigma(z^{L-1})$$

$$z^L = W^L h^{L-1} \qquad \hat{y} = \sigma(z^L)$$

$$J = \sum_k \mathcal{L}(y_k, \hat{y}_k)$$

$$\frac{\partial J}{\partial w_{k,j}^L} =$$

$$\frac{\partial J}{\partial w_{j,i}^{L-1}} =$$

Backpropagation

$$\frac{\partial J}{\partial w_{k,j}^{L}} = \frac{\partial \mathcal{L}}{\partial \hat{y}_k} \cdot \frac{\partial \hat{y}_k}{\partial z_k^L} \cdot \frac{\partial z_k^L}{\partial w_{k,j}^L} = \overbrace{\mathcal{L}'(y_k, \hat{y}_k) \cdot \sigma'\left(z_k^L\right)}^{\beta_k^L} \cdot h_j^{L-1}$$

$$\frac{\partial J}{\partial w_{j,i}^{L-1}} = \sum_k \frac{\partial \mathcal{L}}{\partial \hat{y}_k} \cdot \frac{\partial \hat{y}_k}{\partial z_k^L} \cdot \frac{\partial z_k^L}{\partial h_j^{L-1}} \cdot \frac{\partial h_j^{L-1}}{\partial z_j^{L-1}} \cdot \frac{\partial z_j^{L-1}}{\partial w_{j,i}^{L-1}}$$

$$= \underbrace{\sum_k \underbrace{\mathcal{L}'(y_k, \hat{y}_k) \cdot \sigma'\left(z_k^L\right)}_{\beta_k^L} \cdot w_{k,j}^L \cdot \sigma'\left(z_j^{L-1}\right)}_{\beta_j^{L-1}} \cdot h_i^{L-2}$$

$$z^1 = W^1 x \qquad h^1 = \sigma(z^1)$$

$$z^2 = W^2 h^1 \qquad h^2 = \sigma(z^2)$$

$$\dots$$

$$z^{L-1} = W^{L-1} h^{L-2} \qquad h^{L-1} = \sigma(z^{L-1})$$

$$z^L = W^L h^{L-1} \qquad \hat{y} = \sigma(z^L)$$

$$J = \mathcal{L}(y, \hat{y}) \text{ or } \sum_k \mathcal{L}(y_k, \hat{y}_k)$$

Backpropagation

$$\frac{\partial J}{\partial w_{j,i}^{l}} = \beta_j^l \cdot \underbrace{h_i^{l-1}}_{x_i \text{ if } l = 1}$$

$$\beta_j^l = \begin{cases} \mathcal{L}_j'\left(y_j, \hat{y}_j\right) \cdot \sigma'\left(z_j^l\right) & \text{if } l = L \\ \left(\sum_{k=1}^{n_{l+1}} \beta_k^{l+1} \cdot w_{k,j}^{l+1}\right) \cdot \sigma'\left(z_j^l\right) & \text{otherwise} \end{cases}$$

Backpropagation

X if $l = 1$

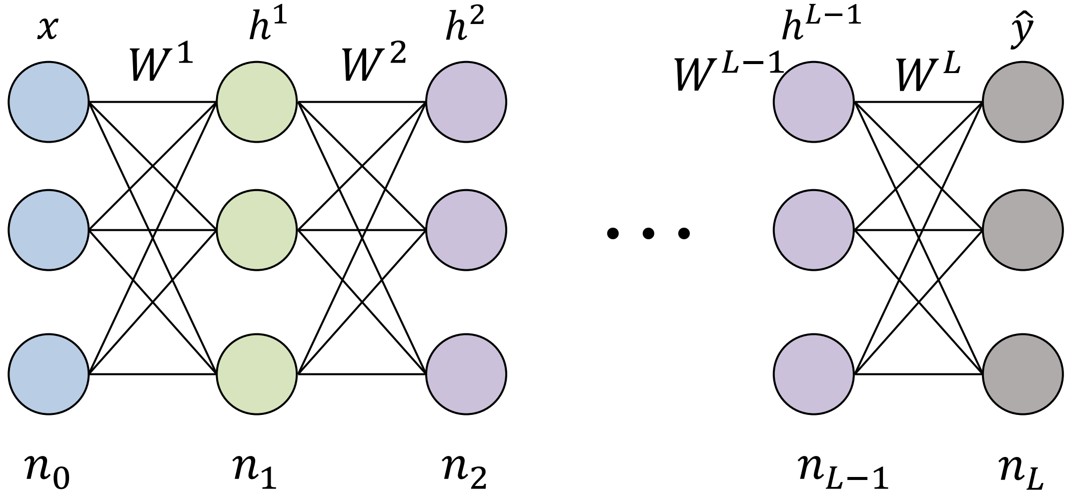
$$\nabla_{W^l} J = \left(\beta^l\right)^T \cdot h^{l-1}$$

$$\beta^l = \begin{cases} \mathcal{L}'(y, \hat{y}) * \sigma'(z^L) & \text{if } l = L \\ \left(\beta^{l+1} \cdot W^{l+1}\right) * \sigma'\left(z^l\right) & \text{otherwise} \end{cases}$$

Forward and back propagation

- $x = h^0$
- $z^l = W^l \cdot h^{l-1}$
- $h^l = \sigma\left(z^l\right)$
- $\hat{y} = \sigma(z^L), \ \ \mathcal{L}(y, \hat{y})$

- $\beta^L = \mathcal{L}'(y, \hat{y}) * \sigma'(z^L)$
- $\beta^l = \left(\beta^{l+1} \cdot W^{l+1}\right) * \sigma'\left(z^l\right)$
- $\nabla_{W^l} J = \left(\beta^l\right)^T \cdot h^{l-1}$



Training a neural network

1. Randomly initialize weights to small values
2. Implement forward propagation to get $h_w(x^{(i)})$ for any $x^{(i)}$
3. Implement code to compute loss function $J(w)$
4. Implement backprop to compute partial derivatives $\frac{\partial J}{\partial w_{j,i}^{l}}$

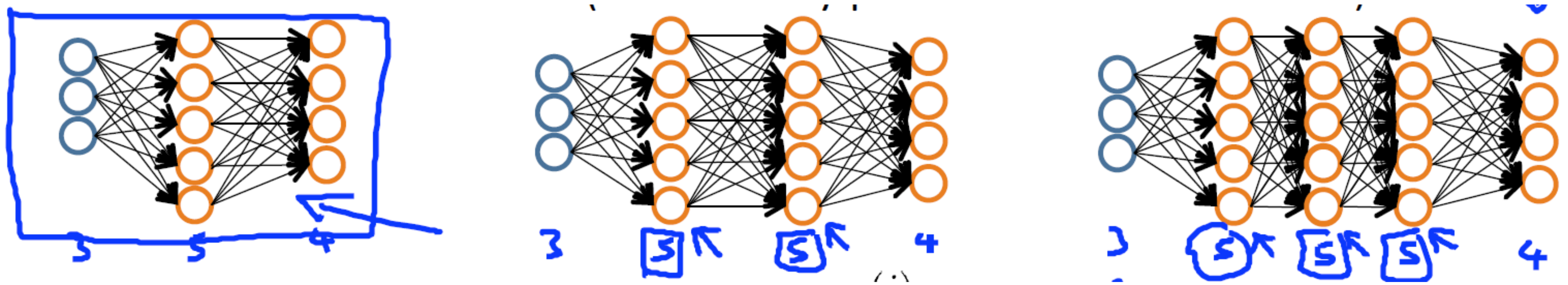
```
for i=1:m
    Perform forward propagation and backpropagation
    using example $(x^{(i)}, y^{(i)})$
```

Training a neural network

5. Use gradient checking to compare $\frac{\partial J}{\partial w_{j,i}^{l}}$ computed using backpropagation vs. using numerical estimate of gradient of $J(w)$
   - Then disable gradient checking code
6. Use gradient descent or advanced optimization method with backpropagation to try to minimize $J(w)$

Training a neural network

- Pick a network architecture



- No. of input nodes: dimension of features x
- No. of output nodes: number of classes
- Reasonable default: 1 hidden layer, or if >1 hidden layer, have same no. of hidden nodes in every layer (usually the more the better)

Automatic Differentiation

- In deep learning packages like TensorFlow and PyTorch, backpropagation is not implemented by hand.
  - Users only need to define the layers and loss function and backpropagation will be implemented automatically.
  - Autograd is the name of the PyTorch autodiff package

- Autograd is not gradApprox.
- Autograd is not symbolic differentiation either.
- Autograd is a procedure for computing derivatives.

Autograd

- Consider a neural network with 0 hidden layer and sigmoid function

Forward propagation

$$z = wx + b$$
$$y = \sigma(z)$$
$$L = \frac{1}{2}(y - t)^2$$

Backpropagation

$$\frac{\partial L}{\partial y} = y - t$$
$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial z} = \frac{\partial L}{\partial y} \cdot \sigma'(z)$$
$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial z} \cdot \frac{\partial z}{\partial w} = \frac{\partial L}{\partial z} \cdot x$$
$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial z} \cdot \frac{\partial z}{\partial b} = \frac{\partial L}{\partial z} \cdot 1$$

Backpropagation

$$\bar{L} = 1$$
$$\bar{y} = \bar{L} \cdot (y - t)$$
$$\bar{z} = \bar{y} \cdot \sigma'(z)$$
$$\bar{w} = \bar{z} \cdot x$$
$$\bar{b} = \bar{z} \cdot 1$$

Primitive Operations

Sequence of primitive operations

Original program

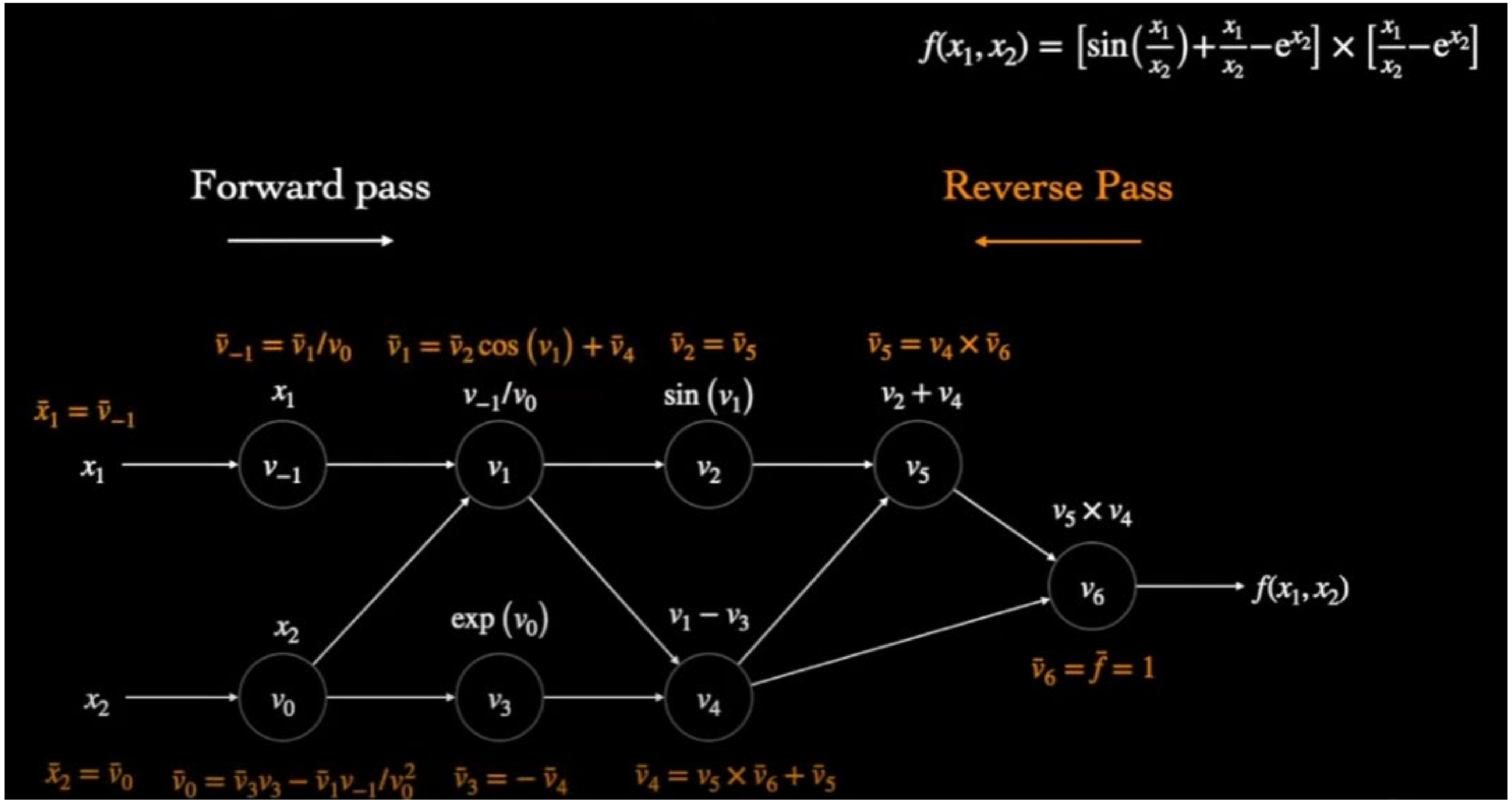
$$z = wx + b$$
$$y = \frac{1}{1 + e^{-z}}$$
$$L = \frac{1}{2}(y - t)^2$$

$$t_1 = wx$$
$$z = t_1 + b$$
$$t_3 = -z$$
$$t_4 = e^{t_3}$$
$$t_5 = 1 + t_4$$
$$y = \frac{1}{t_5}$$
$$t_6 = y - t$$
$$t_7 = t_6^2$$
$$L = \frac{t_7}{2}$$

Backpropagation

…

Computation Graph



$f(x_1, x_2) = [\sin(\frac{x_1}{x_2}) + \frac{x_1}{x_2} - e^{x_2}] \times [\frac{x_1}{x_2} - e^{x_2}]$
Forward pass
Reverse Pass
$\bar{v}_{-1} = \bar{v}_1 / v_0$
$\bar{v}_1 = \bar{v}_2 \cos(v_1) + \bar{v}_4$
$\bar{v}_2 = \bar{v}_5$
$\bar{v}_5 = v_4 \times \bar{v}_6$
$\bar{x}_1 = \bar{v}_{-1}$
x_1
v_{-1}/v_0
$\sin(v_1)$
$v_2 + v_4$
x_1
v_{-1}
v_1
v_2
v_5
$v_5 \times v_4$
v_6
$f(x_1, x_2)$
x_2
$\exp(v_0)$
$v_1 - v_3$
x_2
v_0
v_3
v_4
$\bar{v}_6 = \bar{f} = 1$
$\bar{x}_2 = \bar{v}_0$
$\bar{v}_0 = \bar{v}_3 v_3 - \bar{v}_1 v_{-1} / v_0^2$
$\bar{v}_3 = -\bar{v}_4$
$\bar{v}_4 = v_5 \times \bar{v}_6 + \bar{v}_5$

https://www.youtube.com/watch?v=wG_nF1awSSY

PyTorch Example Code

- Define a neural network

```
import torch.nn as nn
import torch.nn.functional as F

class TwoLayerNet(nn.Module):
        def __init__(self, D_in, H, D_out):
                Super(TwoLayerNet, self).__init__()
                self.linear1 = nn.Linear(D_in, H)
                self.linear2 = nn.Linear(H, D_out)
        def forward(self, x):
                h_relu = F.relu(self.linear1(x))
                y_pred = self.linear2(h_relu)
                return y_pred

D_in, H, D_out = 100, 50, 10
model = TwoLayerNet(D_in, H, D_out)
```

PyTorch Example Code

- Define the loss function and optimizer

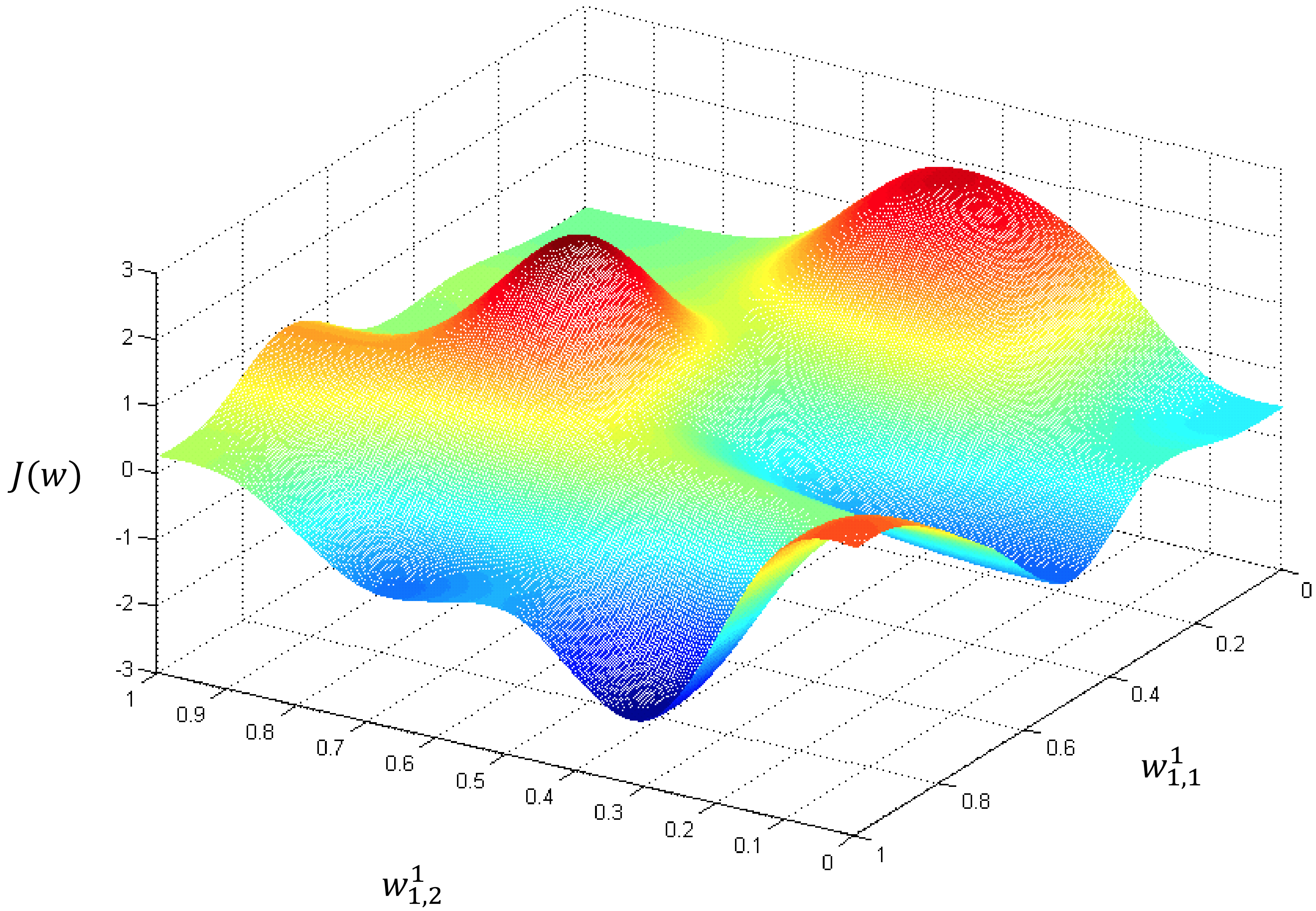
```
import torch.optim as optim


loss_fn = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr = 0.01)
```

PyTorch Example Code

- Train the neural network

```
output_batch = model(train_batch)           # compute model output
loss = loss_fn(output_batch, labels_batch)  # calculate loss
optimizer.zero_grad()                       # clear previous gradients
loss.backward()                             # compute gradients of all variables wrt loss
optimizer.step()                            # perform updates using calculated gradients
```

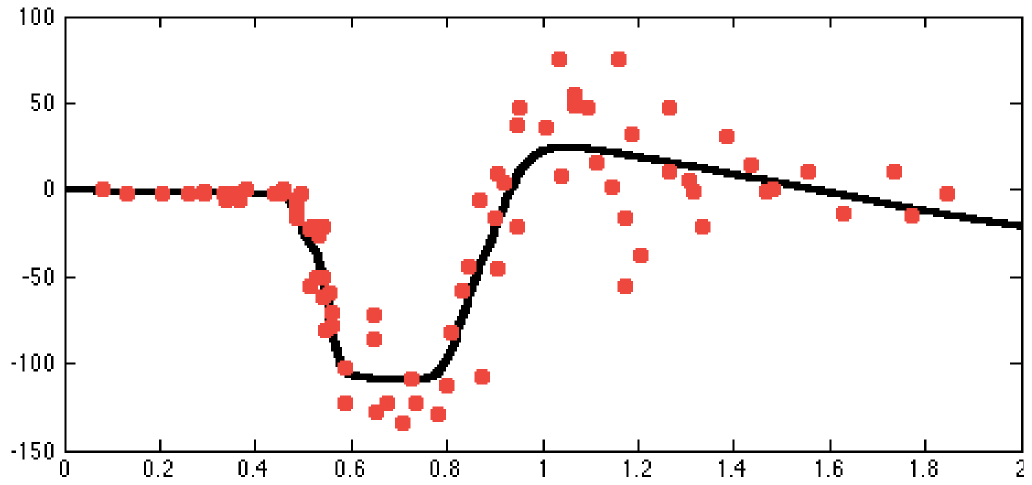


$J(w)$
3
2
1
0
-1
-2
-3
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
$w^1_{1,2}$
0
0.2
0.4
0.6
0.8
1
$w^1_{1,1}$

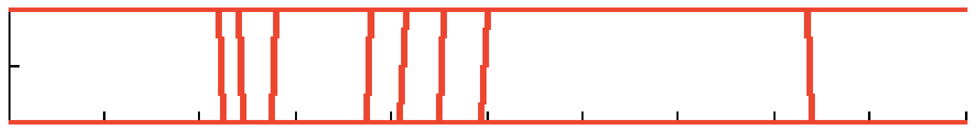
Example: Regression, MCycle data

- Train NN model, 2 layer
  - 1 input feature => 1 input node
  - 10 hidden nodes
  - 1 target => 1 output node
  - Logistic sigmoid activation for hidden layer, linear for output layer

Data:
+
learned prediction f’n:

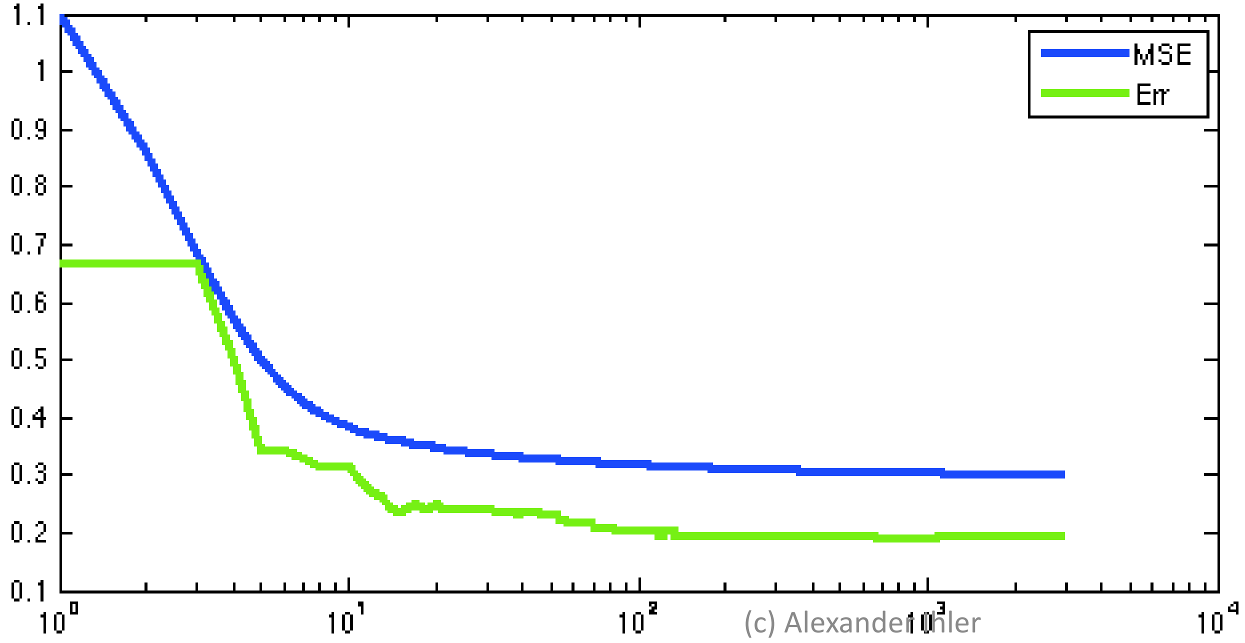


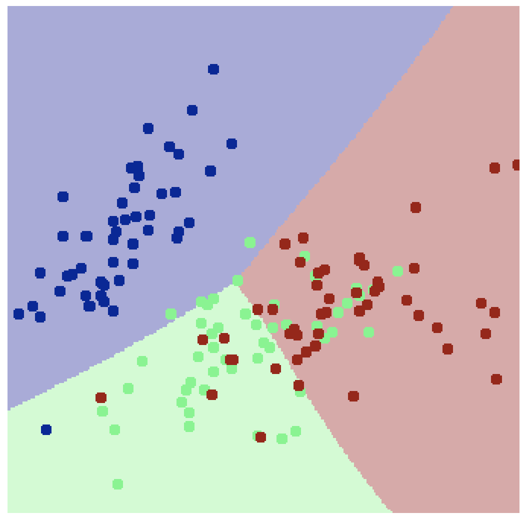
Responses of hidden nodes
(= features of linear regression):
select out useful regions of “x”



Example: Classification, Iris data

- Train NN model, 2 layer
  - 2 input features => 2 input nodes
  - 10 hidden nodes
  - 3 classes => 3 output nodes (y = [0 0 1], etc.)
  - Logistic sigmoid activation functions





Summary

- Neural networks, multi-layer perceptrons

- Cascade of simple perceptrons
  - Each just a linear classifier
  - Hidden units used to create new features

- Together, general function approximators
  - Enough hidden units (features) = any function
  - Can create nonlinear classifiers
  - Also used for function approximation, regression, …

- Training via backprop
  - Gradient descent; logistic; apply chain rule. Building block view.

- Advanced: deep nets, conv nets, …