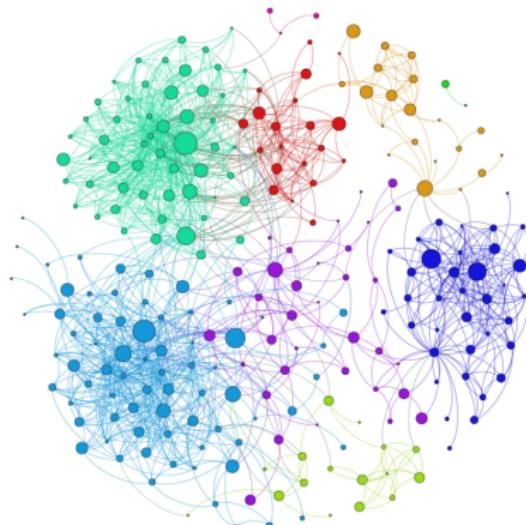


Neural Networks

Jiaming Mao

Xiamen University



Copyright © 2017–2019, by Jiaming Mao

This version: Spring 2019

Contact: jmao@xmu.edu.cn

Course homepage: jiamingmao.github.io/data-analysis



All materials are licensed under the [Creative Commons Attribution-NonCommercial 4.0 International License](#).

The Perceptron

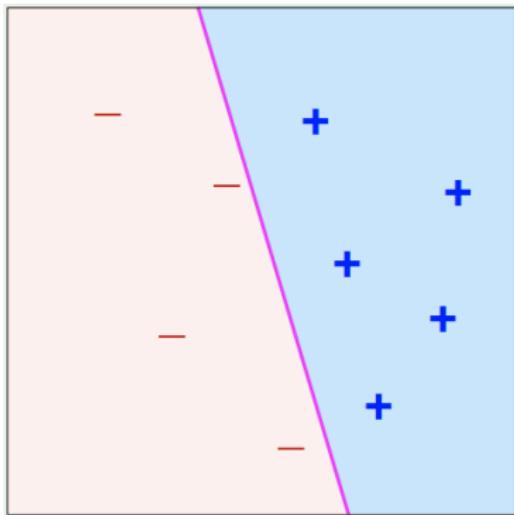
Consider a binary classification problem.
Let $y \in \{-1, 1\}$ and $x = (1, x_1, \dots, x_p)$.
The **perceptron** is a linear classifier

$$h(x) = \text{sign}(w'x)$$

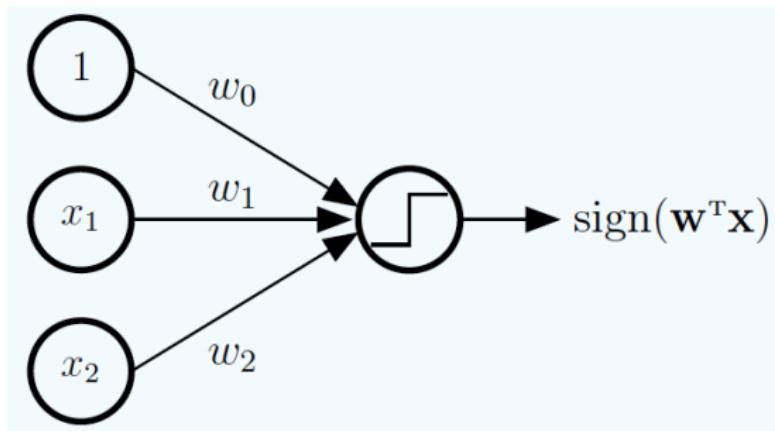
that minimizes the misclassification loss

$$\ell(y, h(x)) = \mathcal{I}(y \neq h(x))$$

- $w = (w_0, w_1, \dots, w_p)$ are **weights**.

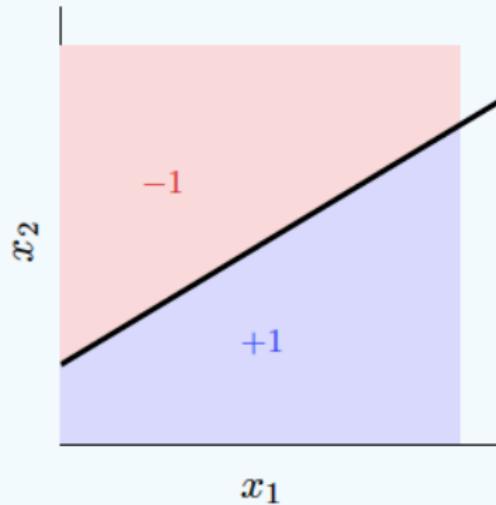


The Perceptron

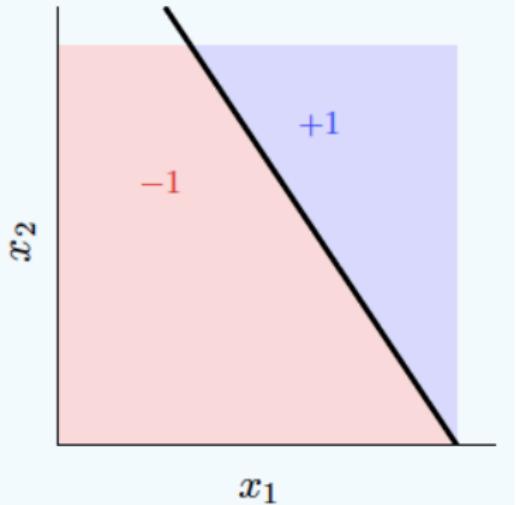


Graph representation of the perceptron

Combining Perceptrons

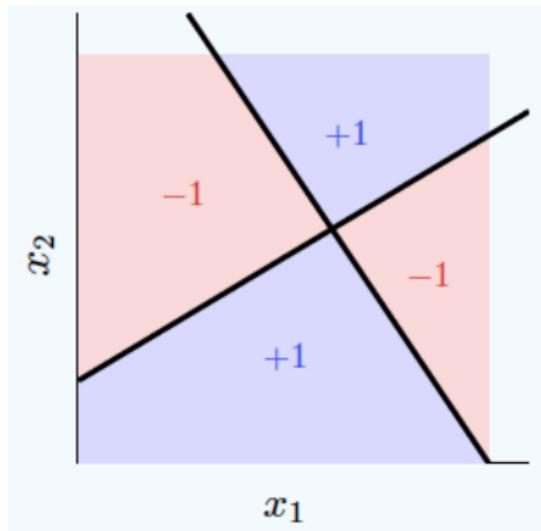


$$h_1(\mathbf{x}) = \text{sign}(\mathbf{w}_1^T \mathbf{x})$$



$$h_2(\mathbf{x}) = \text{sign}(\mathbf{w}_2^T \mathbf{x})$$

Combining Perceptrons



$$\begin{aligned}f &= \text{XOR}(h_1, h_2) \\&= h_1 \overline{h_2} + \overline{h_1} h_2\end{aligned}$$

1

¹Using standard Boolean notation: multiplication for AND, addition for OR, and overbar for negation.

Perceptrons for Boolean Functions

The Boolean functions AND and OR themselves can be implemented by the perceptron: for $x_1, x_2 \in \{-1, 1\}$,

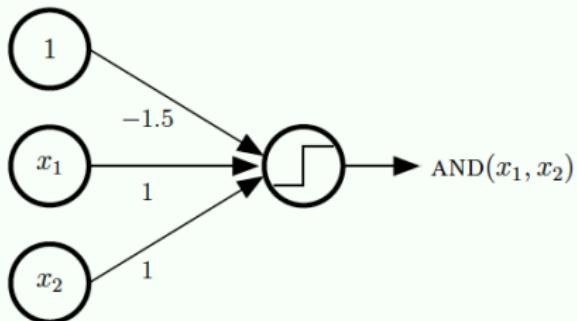
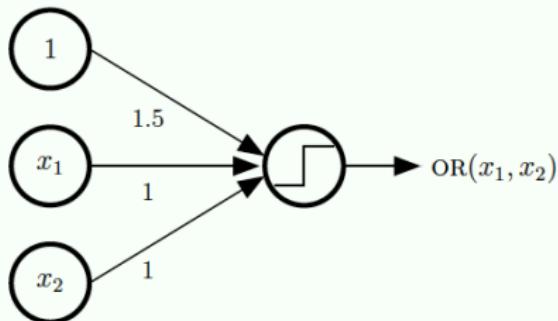
$$\text{OR}(x_1, x_2) = \text{sign}(x_1 + x_2 + 1.5)$$

$$\text{AND}(x_1, x_2) = \text{sign}(x_1 + x_2 - 1.5)$$

Perceptrons for Boolean Functions

$$\text{OR}(x_1, x_2) = \text{sign}(x_1 + x_2 + 1.5)$$

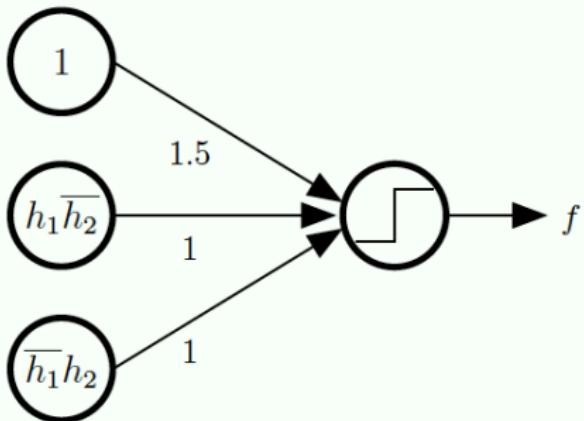
$$\text{AND}(x_1, x_2) = \text{sign}(x_1 + x_2 - 1.5)$$



Graph representation of perceptrons for Boolean functions

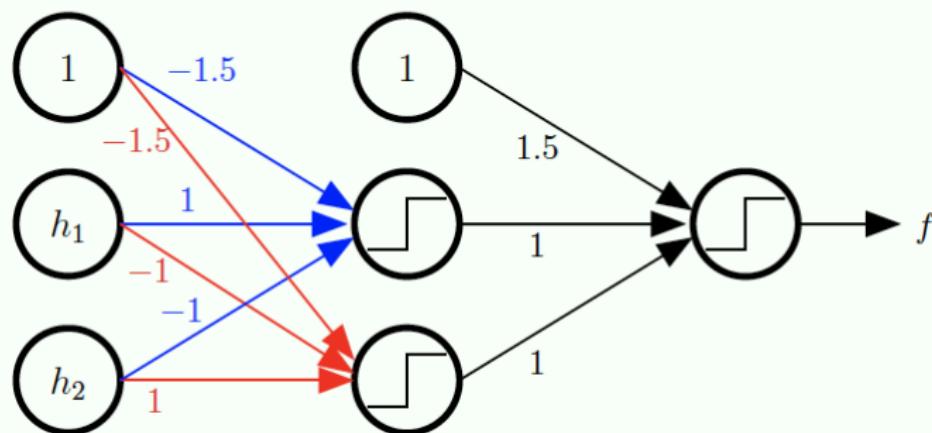
Combining Perceptrons

$$f = h_1 \overline{h_2} + \overline{h_1} h_2$$

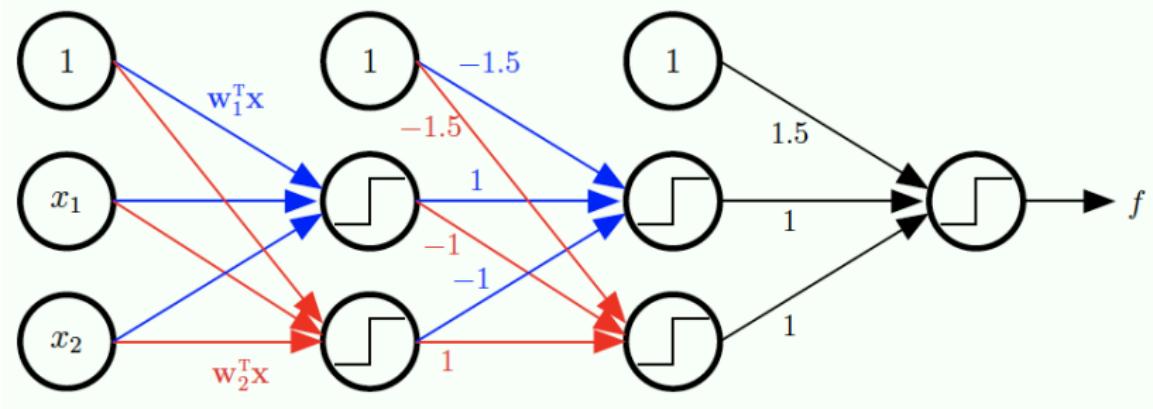


Combining Perceptrons

$$f = h_1 \overline{h_2} + \overline{h_1} h_2$$



Combining Perceptrons



$$f(x) = \text{sign} \left\{ \text{sign} [\text{sign}(w'_1 x) - \text{sign}(w'_2 x) - 1.5] + \text{sign} [-\text{sign}(w'_1 x) + \text{sign}(w'_2 x) - 1.5] + 1.5 \right\}$$

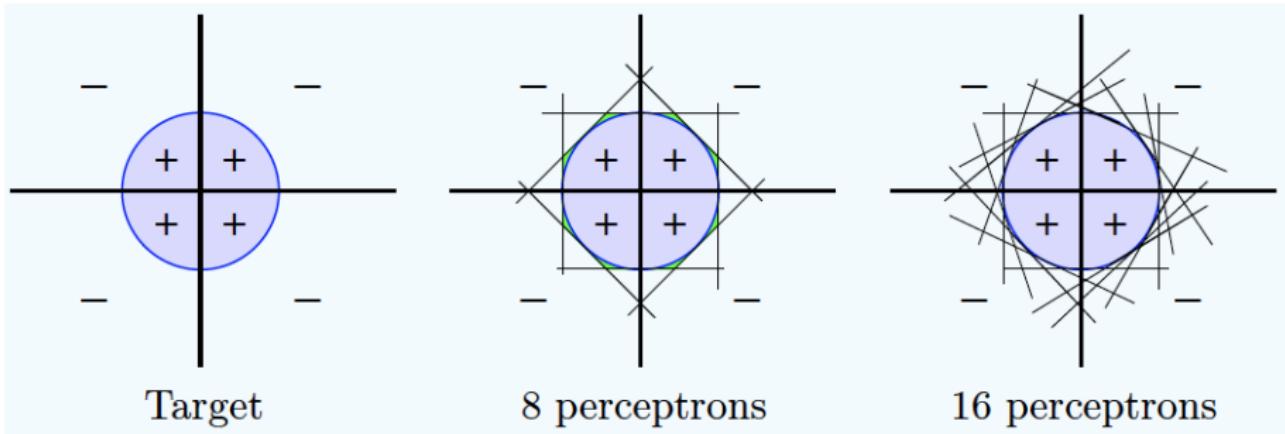
The Multilayer Perceptron

f is called a **multilayer perceptron (MLP)**

- Compared with the simple perceptron, the MLP has more layers. The additional layers are called **hidden layers**.
 - ▶ f on the preceding page has 3 layers: 2 hidden layers with 3 nodes each, and an output layer with 1 node.

Any target function f that can be decomposed into linear separators can be implemented by a 3-layer MLP.

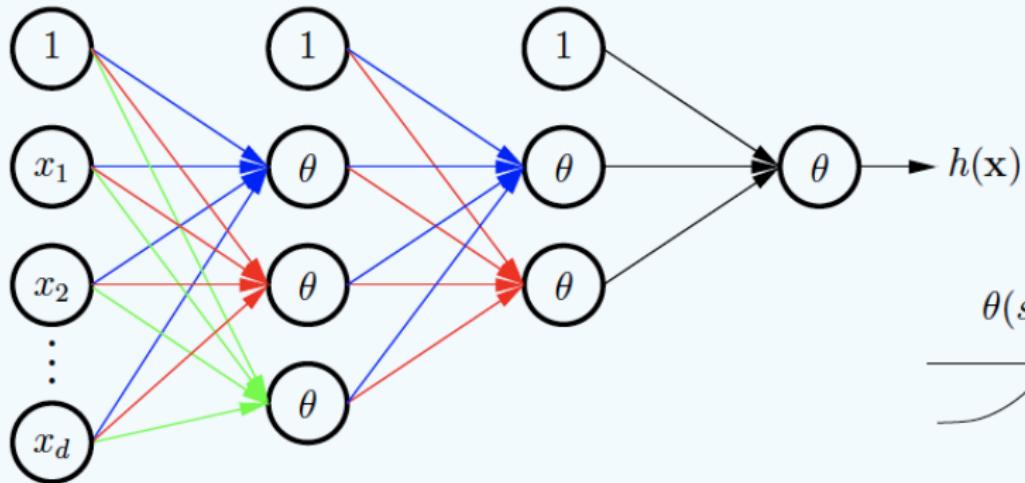
Universal Approximation



A sufficiently smooth decision boundary can “essentially” be decomposed into linear separators and hence approximated arbitrarily closely by a 3-layer MLP².

² Just like any continuous function on a compact set can be approximated arbitrarily closely using step functions. The perceptron is the analog of the step function.

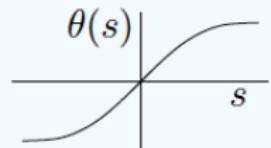
The Neural Network



input layer $\ell = 0$

hidden layers $0 < \ell < L$

output layer $\ell = L$



The Neural Network

- A neural network of L layers contains $L - 1$ hidden layers and one output layer. The layers are numbered $\ell = 0, 1, \dots, L$, with $\ell = 0$ denoting the input layer³.
- Each hidden layer ℓ contains $1 + d^{(\ell)}$ **units (neurons)**, indexed $0, 1, \dots, d^{(\ell)}$, with unit 0 being the **bias unit**. Each unit i in layer ℓ other than the bias unit takes a **signal** $s_i^{(\ell)}$ as **input** and transforms it into an **output** $x_i^{(\ell)}$ using the **activation function** θ^4 :

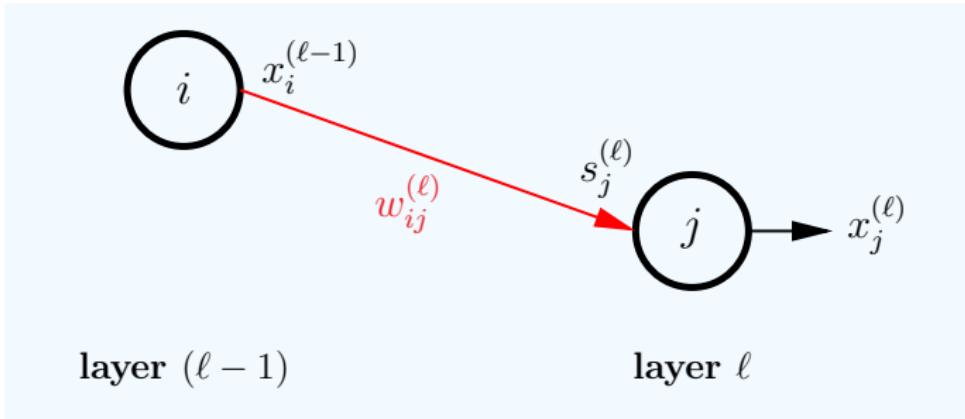
$$x_i^{(\ell)} = \theta(s_i^{(\ell)})$$

, where $s_i^{(\ell)}$ is the weighted sum of the outputs of the previous layer:
 $s_i = \sum_{j=0}^{d^{(\ell-1)}} x_j^{(\ell-1)} w_{ji}^{(\ell)}$. The bias unit is set to have an output 1.

³which, despite the name, is not considered a layer of the neural network.

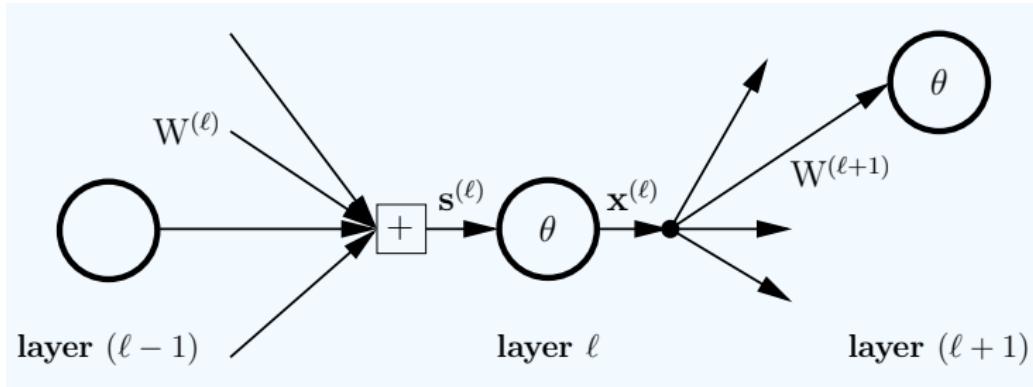
⁴When all $\theta(\cdot)$'s are identity functions, the neural network collapses to a linear model.

The Neural Network



$$x_j^{(\ell)} = \theta(s_j^{(\ell)}) = \theta \left(\sum_{i=0}^{d^{(\ell-1)}} w_{ij}^{(\ell)} x_i^{(\ell-1)} \right)$$

The Neural Network



layer ℓ parameters

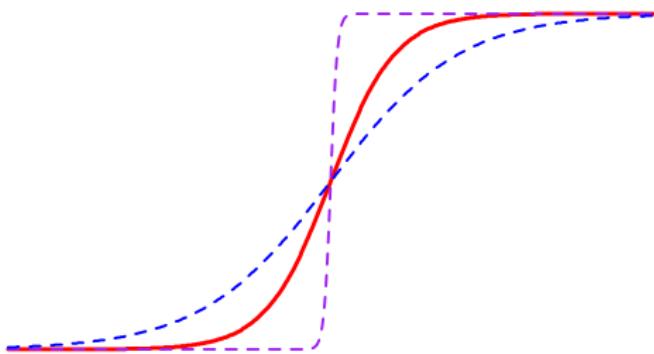
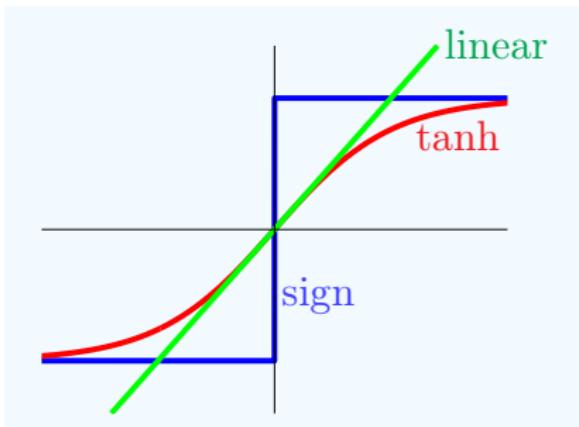
signals in	$s^{(\ell)}$	$d^{(\ell)} \times 1$
outputs	$x^{(\ell)}$	$(1 + d^{(\ell)}) \times 1$
weights in	$w^{(\ell)}$	$(1 + d^{(\ell-1)}) \times d^{(\ell)}$
weights out	$w^{(\ell+1)}$	$(1 + d^{(\ell)}) \times d^{(\ell+1)}$

The Neural Network

- The MLP uses the step function $\text{sign}(\cdot)$ as its activation functions, which makes it hard to optimize.
- In neural networks, for the hidden layers, we can replace the **hard-threshold** $\text{sign}(\cdot)$ function with the **soft-threshold** $\tanh(\cdot)$ function.
- Other popular activation functions for hidden layers:
 - ▶ Logistic sigmoid⁵: $\sigma(x) = \frac{1}{1+e^{-x}}$
 - ▶ The Rectified Linear Unit (ReLU): $f(x) = \max(0, x)$

⁵The tanh is a scaled logistic sigmoid (σ): $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1$

The Neural Network



Left: the \tanh activation function. Note that for small x , $\tanh(x) \approx x$.

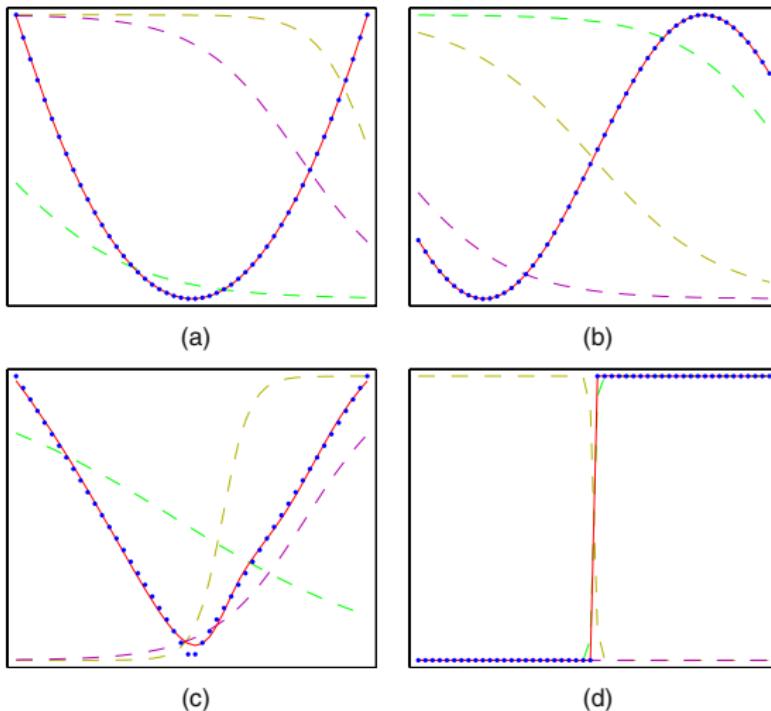
Right: $\tanh(wx)$ for $w = 0.5$ (blue), $w = 1$ (red), and $w = 10$ (purple)

The Neural Network

- For regression, the output layer typically contains a single unit⁶, with the activation function being the identity function $f(x) = x$ and the error measure being the sum of squares error.
- For binary classification, the output layer contains a single unit. We can choose the activation function to be the logistic sigmoid, with the error measure being the cross-entropy error.
- For K -class multiclass classification, the output layer would contain K units. We can choose the activation function to be the softmax function, with the corresponding multiclass cross-entropy error⁷.

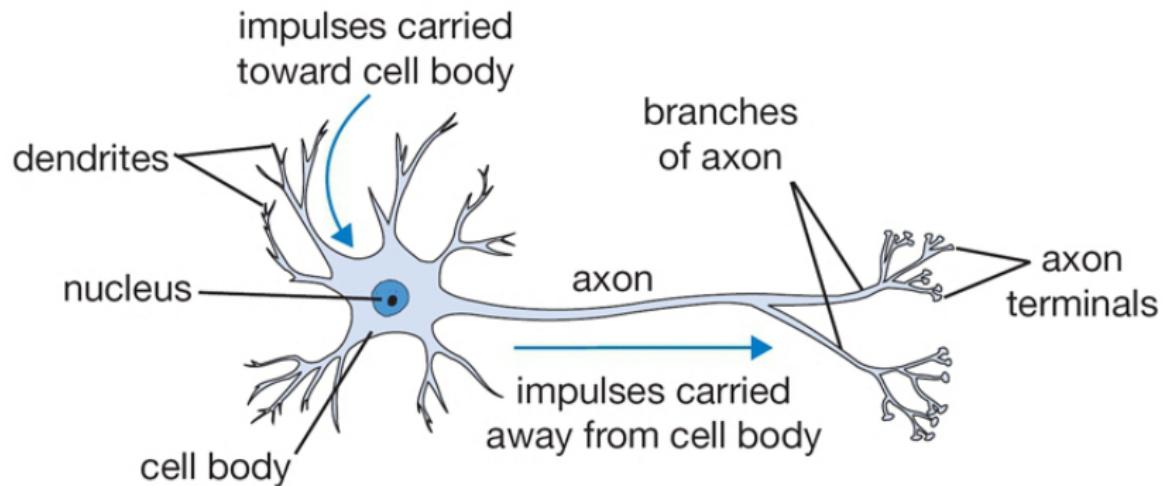
⁶It is also possible for have multiple output units for multiple quantitative responses as in multivariate regressions.

⁷Alternatively, using a *one-versus-all* approach, we can choose the activation function to be the logistic sigmoid and let the y be coded as a K -dimensional vector:
 $y = [y_1, \dots, y_K]',$ where $y_j \in \{0, 1\}, j \in \{1, \dots, K\}.$

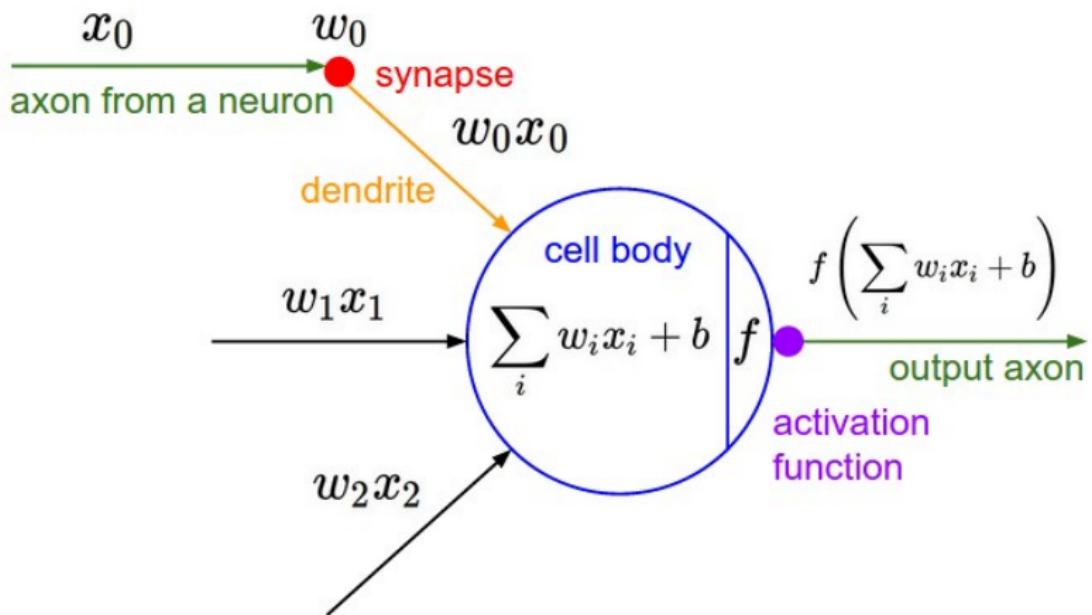


Approximation of (a) $f(x) = x^2$; (b) $f(x) = \sin(x)$; (c) $f(x) = |x|$; (d) $f(x) = \text{sign}(x)$ using a two-layer neural network with three hidden units and a linear output. Blue dots: data from the underlying function; Red line: neural network output; Dashed lines: outputs of the three hidden units.

Biological Inspiration



Biological Inspiration



Estimation

Let $w = \{w^{(1)}, \dots, w^{(L)}\}$. A neural network model is $\mathcal{H} = \{h(x; w)\}$.

The training error for a neural network is

$$E_{in}(w) = \frac{1}{N} \sum_{n=1}^N \ell(h(x_n; w), y_n)$$

, where ℓ is the appropriate loss function.

To estimate w , we need to compute $\nabla E_{in}(w)$, i.e., for each data point (x_n, y_n) , we need to compute

$$\frac{\partial e_n(w)}{\partial w_{ij}^{(\ell)}} \quad \forall i, j, \ell \tag{1}$$

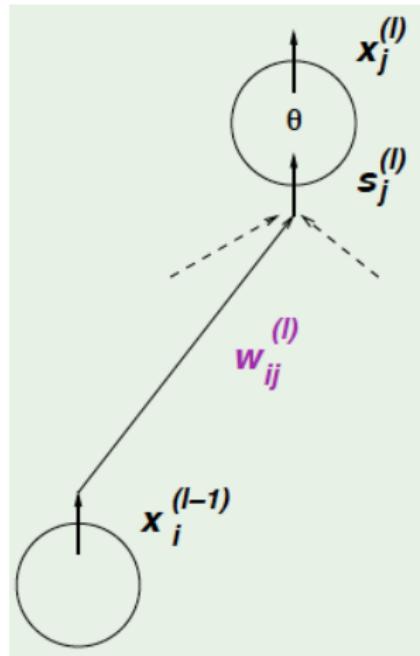
, where $e_n(w) = \ell(h(x_n; w), y_n)$.

Estimation

An efficient way to compute (1):

$$\begin{aligned}\frac{\partial e_n(w)}{\partial w_{ij}^{(\ell)}} &= \frac{\partial e_n(w)}{\partial s_j^{(\ell)}} \times \frac{\partial s_j^{(\ell)}}{\partial w_{ij}^{(\ell)}} \\ &= x_i^{(\ell-1)} \delta_j^{(\ell)}\end{aligned}$$

, where $\delta_j^{(\ell)} \equiv \frac{\partial e_n(w)}{\partial s_j^{(\ell)}}$.



Forward Propagation

We can calculate $x^{(\ell)} = [1, x_1^{(\ell)}, \dots, x_{d^{(\ell)}}^{(\ell)}]'$ based on $x^{(\ell-1)}$:

$$s^{(\ell)} = (W^{(\ell)})' x^{(\ell-1)}$$

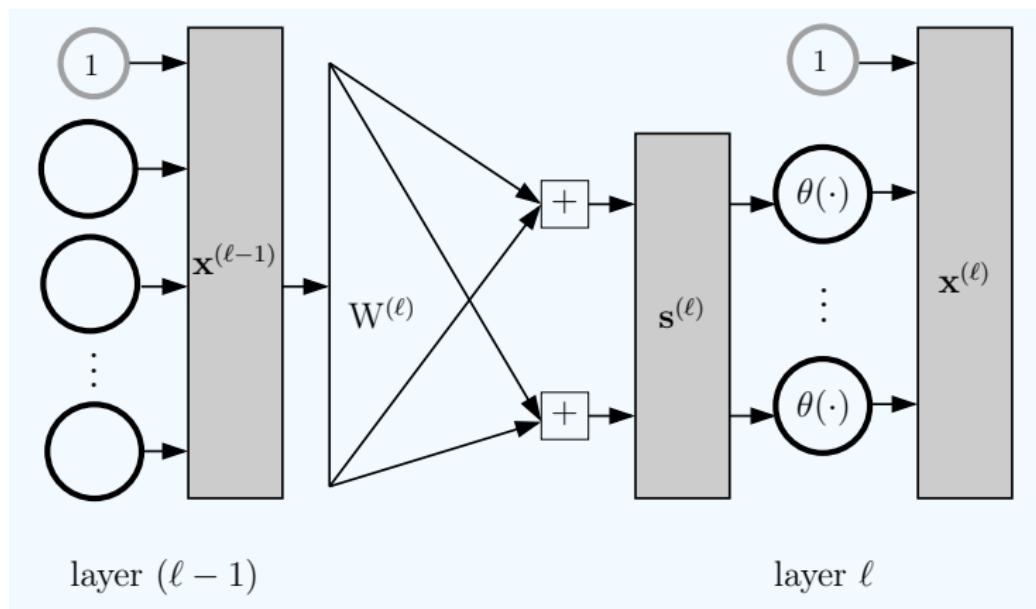
$$x^{(\ell)} = \begin{bmatrix} 1 \\ \theta(s^{(\ell)}) \end{bmatrix}$$

Therefore, starting with $x^{(0)} = x_n$, we can calculate $x^{(\ell)}$ for all ℓ as follows:

$$x_n = x^{(0)} \xrightarrow{W^{(1)}} s^{(1)} \xrightarrow{\theta} x^{(1)} \longrightarrow \dots s^{(L)} \xrightarrow{\theta} x^{(L)} = h(x_n)$$

This is called **forward propagation**.

Forward Propagation

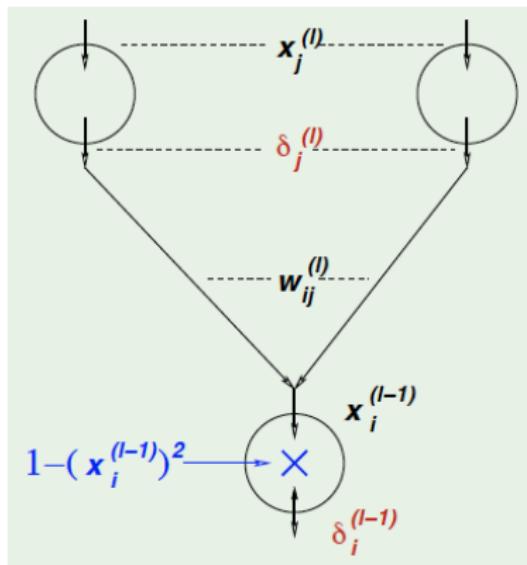


Backward Propagation

$\delta^{(l)} = [\delta_1^{(l)}, \dots, \delta_{d^{(l)}}^{(l)}]'$ can be calculated from $\delta^{(l+1)}$.

Starting from the output layer, assuming single unit output, the identity activation function, and sum-of-squares error, we have:

$$\begin{aligned}\delta^{(L)} &= \frac{\partial e_n(w)}{\partial s^{(L)}} \\ &= \frac{\partial (\theta(s^{(L)}) - y_n)^2}{\partial s^{(L)}} \\ &= 2(x^{(L)} - y_n)\end{aligned}$$



Backward Propagation

For $\ell < L$,

$$\begin{aligned}\delta_i^{(\ell-1)} &= \frac{\partial e_n(w)}{\partial s_j^{(\ell-1)}} \\ &= \sum_{j=1}^{d^{(\ell)}} \frac{\partial e_n(w)}{\partial s_j^{(\ell)}} \times \frac{\partial s_j^{(\ell)}}{\partial x_i^{(\ell-1)}} \times \frac{\partial x_i^{(\ell-1)}}{\partial s_i^{(\ell-1)}} \\ &= \sum_{j=1}^{d^{(\ell)}} \delta_j^{(\ell)} \times w_{ij}^{(\ell)} \times \theta'(s_i^{(\ell-1)})\end{aligned}$$

, where, for $\theta(.) = \tanh(.)$, we have

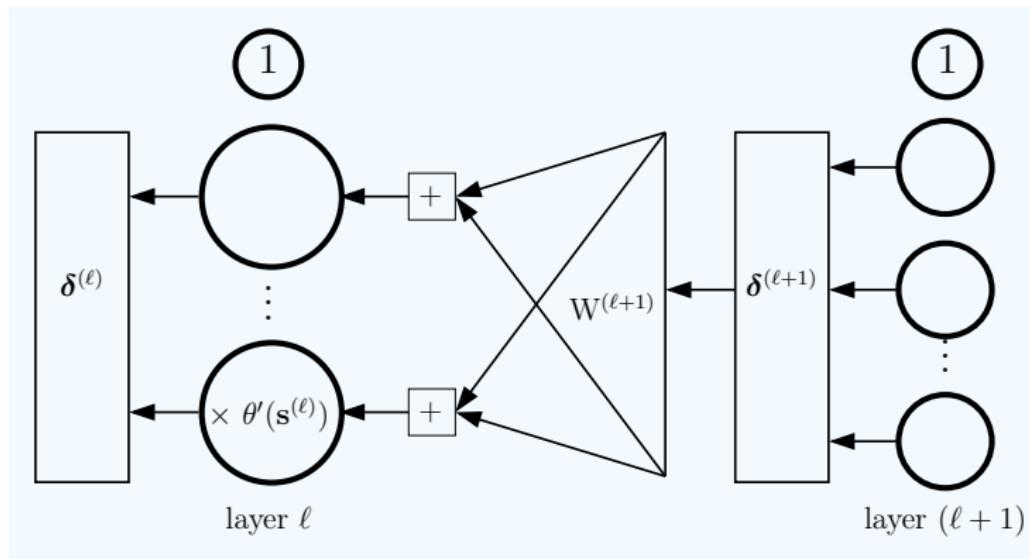
$$\tanh'(s_i^{(\ell-1)}) = 1 - [\tanh(s_i^{(\ell-1)})]^2 = 1 - (x_i^{(\ell-1)})^2.$$

Backward Propagation

Therefore, starting with $\delta^{(L)}$, we can calculate $\delta^{(\ell)}$ for all ℓ :

$$\delta^{(1)} \leftarrow \delta^{(2)} \dots \leftarrow \delta^{(L-1)} \leftarrow \delta^{(L)}$$

This is called **backward propagation**.



Gradient Descent

- Forward and backward propagation provide an efficient method to compute $\nabla E_{in}(w)$. We can then search for the optimal w^* that minimizes $E_{in}(w)$ using optimization techniques such as batch and stochastic gradient descent.
- In practice, there is quite an art in training neural networks, as the optimization problem is nonconvex with many local minima.

Gradient Descent

Gradient Descent (Batch)

Initialize all weights $w = \{w_{ij}^{(\ell)}\}$ at random.

For $t = 0, 1, 2, \dots$, do until the convergence of w

 For each data point (x_n, y_n) , $n = 1, \dots, N$, do

 Compute $x^{(\ell)}$ for $\ell = 1, \dots, L$ (forward propagation)

 Compute $\delta^{(\ell)}$ for $\ell = L, \dots, 1$ (backward propagation)

 Compute $\nabla e_n(w) : \partial e_n(w) / \partial w_{ij}^{(\ell)} = x_i^{(\ell-1)} \delta_j^{(\ell)}$

 Compute $\nabla E_{in}(w) = \frac{1}{N} \sum_{n=1}^N \nabla e_n(w)$

 Update $w : w \leftarrow w - \eta \nabla E_{in}(w)$

Gradient Descent

Stochastic Gradient Descent (SGD)

Initialize all weights $w = \{w_{ij}^{(\ell)}\}$ at random.

For $t = 0, 1, 2, \dots$, do until the convergence of w

 For each data point (x_n, y_n) , $n = 1, \dots, N$, do

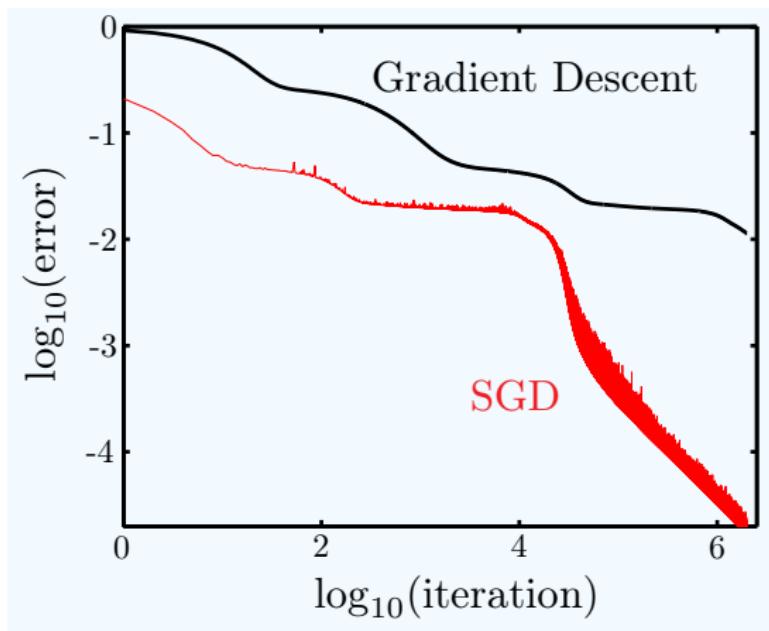
 Compute $x^{(\ell)}$ for $\ell = 1, \dots, L$ (forward propagation)

 Compute $\delta^{(\ell)}$ for $\ell = L, \dots, 1$ (backward propagation)

 Compute $\nabla e_n(w) : \partial e_n(w) / \partial w_{ij}^{(\ell)} = x_i^{(\ell-1)} \delta_j^{(\ell)}$

 Update $w : w \leftarrow w - \eta \nabla e_n(w)$

Gradient Descent



Training Your Neural Network: Initialization

To train a neural network, it is best to standardize all inputs to have mean zero and standard deviation one. The starting values for w are usually chosen to be random values near zero, e.g. $w_i \sim \mathcal{N}(0, \sigma_w^2)$, where σ_w is small.

- Use of exact zero weights leads to zero derivatives and perfect symmetry, and the algorithm never moves.
- Starting with large weights so that $\tanh(w'x) \approx \pm 1$ also leads to close to zero derivatives.
- Hence the initial values for w should be random values near zero so that the model starts out *nearly linear*, and becomes *nonlinear* as the weights increase.

Training Your Neural Network: Regularization

The neural network model is in essence a hyper-parametrized nonlinear model that can contain an arbitrary number of parameters to approximate any smooth functions.

To avoid overfitting, we typically do not want the global minimizer of $E_{in}(w)$. Instead, some regularization is needed, which can be achieved by adding a penalty term. For example, we can minimize the following augmented error:

$$E_{\text{aug}}(w) = E_{in}(w) + \lambda \|w\|^2 \quad (2)$$

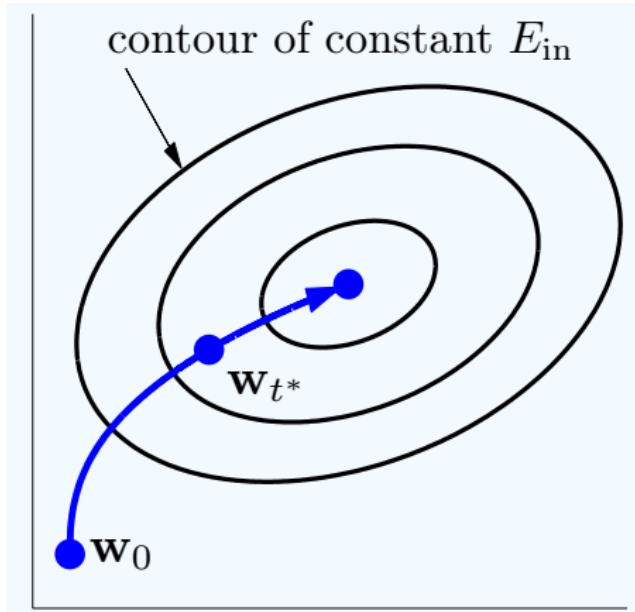
The L2 regularizer (2) is also called **weight decay** in the context of neural networks. The tuning parameter λ is typically selected by cross validation.

Training Your Neural Network: Regularization

Alternatively, regularization can be achieved by **early stopping**: train the model only for a while, and stop well before we approach the global minimum.

- Since we initialize weights near zero, early stopping has the effect of achieving smaller weights, similar to weight decay.
 - ▶ Equivalently, it has the effect of shrinking the final model toward a linear model.
- A validation set is used for determining when to stop.

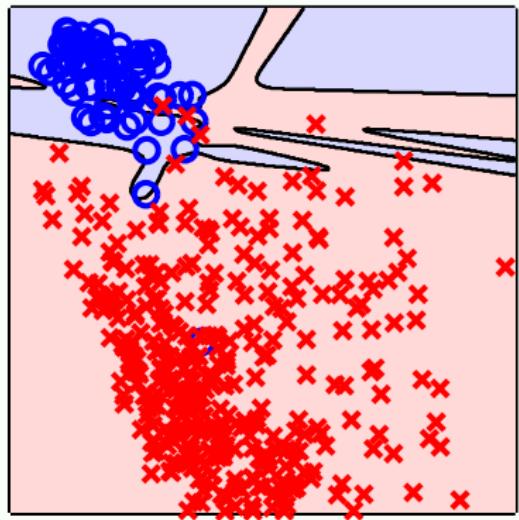
Training Your Neural Network: Regularization



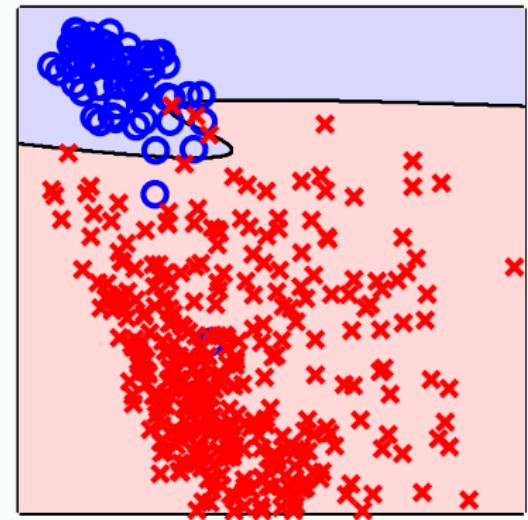
Early Stopping

Training Your Neural Network: Regularization

No Weight Decay



Weight Decay, $\lambda = 0.01$



Symmetry

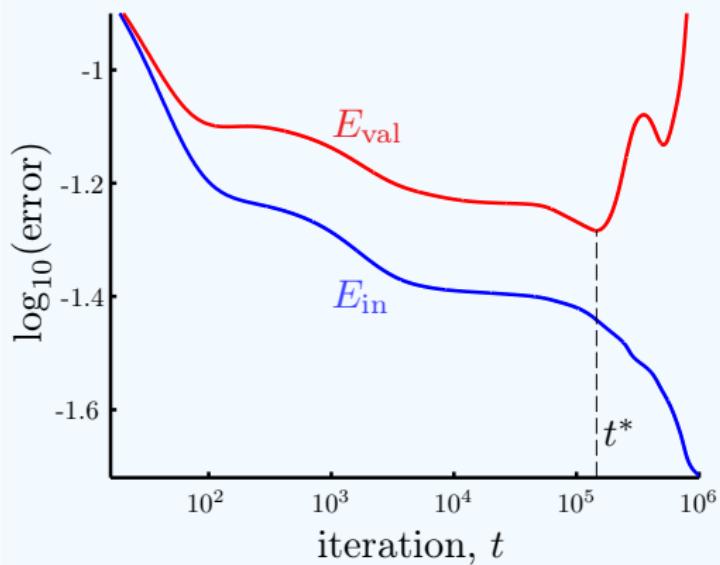
Average Intensity

Symmetry

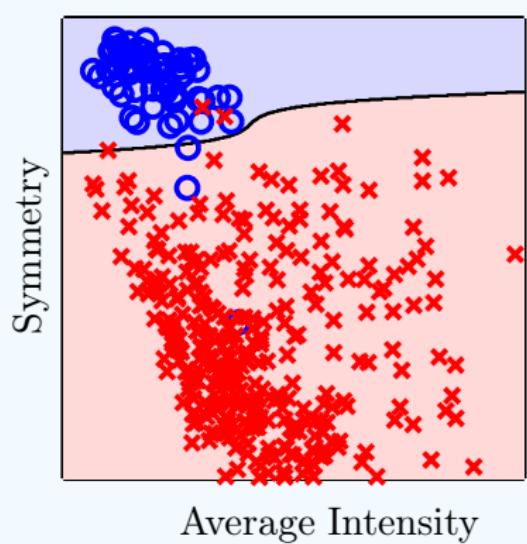
Average Intensity

Weight Decay

Training Your Neural Network: Regularization



(a) Training dynamics



(b) Final hypothesis

Early Stopping

Training Your Neural Network: Hidden Units

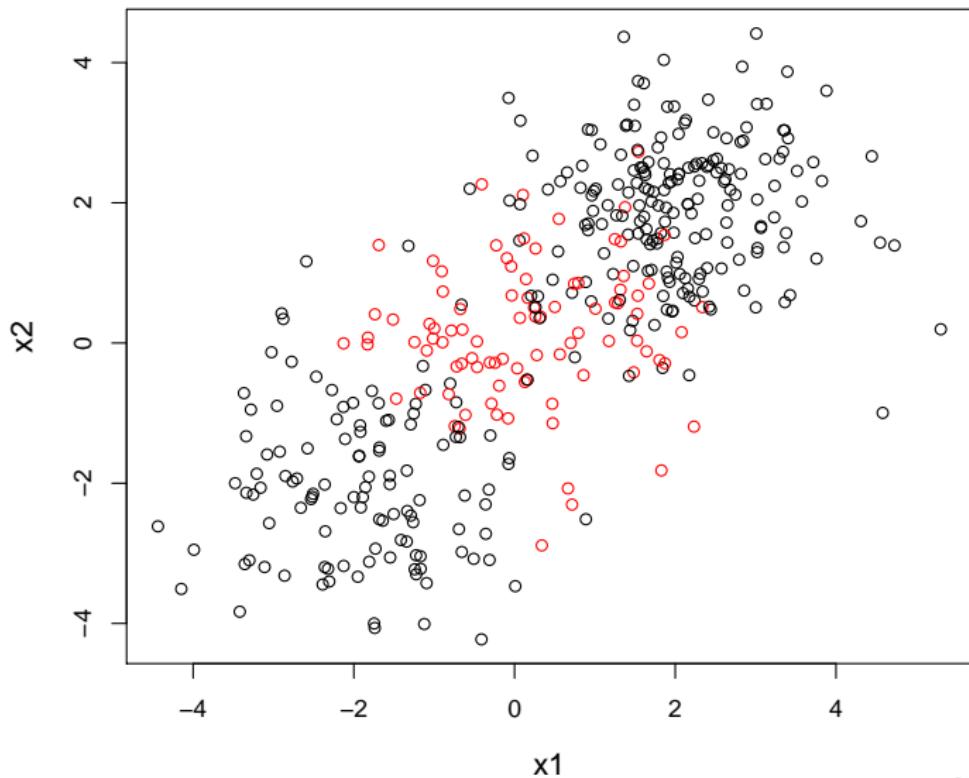
- One can control the complexity of a model by regularization or by limiting the number of hidden units.
- In practice, regularization strength – rather than the number of neurons – is the preferred way to control overfitting. It is better to have a reasonably large number of units and train them with regularization.

Simulation

```
# Simulation
n = 1000
x = matrix(rnorm(n*2), ncol=2)
x[1:n/2,] = x[1:n/2,] + 2
x[(n/2+1):(n/4*3),] = x[(n/2+1):(n/4*3),] - 2
y = c(rep(1,(n/4*3)),rep(2,(n/4)))

# Create training and test sets
data = data.frame(x1=x[,1],x2=x[,2],y=as.factor(y))
train = sample(n,n*0.4)
data_train = data[train,]
data_test = data[-train,]
```

Simulation



Simulation

```
# Here we fit a single-hidden-layer network with 10 hidden units
# and choose the weight decay parameter by cross-validation
require(caret)
fit = train(y ~ ., data=data_train, method="nnet",
            preprocess=c("center", "scale"),
            tuneGrid=expand.grid(size=1:20,
                                  decay=c(0.001, 0.01, 0.1, 1)),
            trControl=trainControl(method="repeatedcv", repeats=3))

fit$bestTune

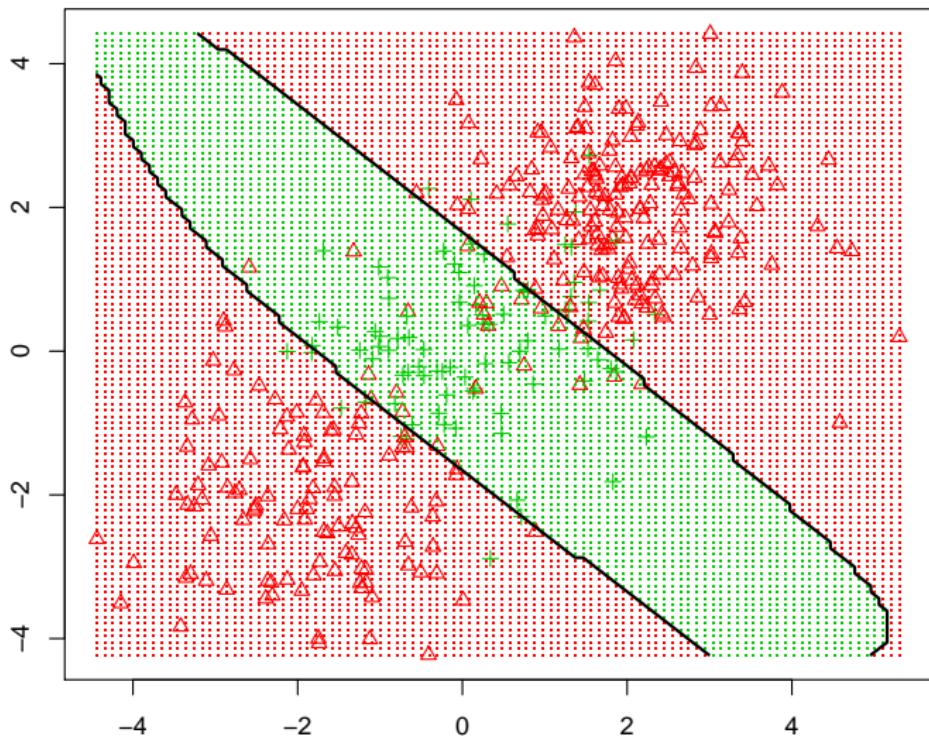
##    size decay
## 3    10    0.1

# test err
ytrue = data_test[, "y"]
yhat = predict(fit, data_test) # predict using fit$bestTune
1-mean(yhat==ytrue)

## [1] 0.1033333
```

Simulation

Neural Networks



Supermarket Entry

Supermarket entry in geographical markets. For each market, data include:

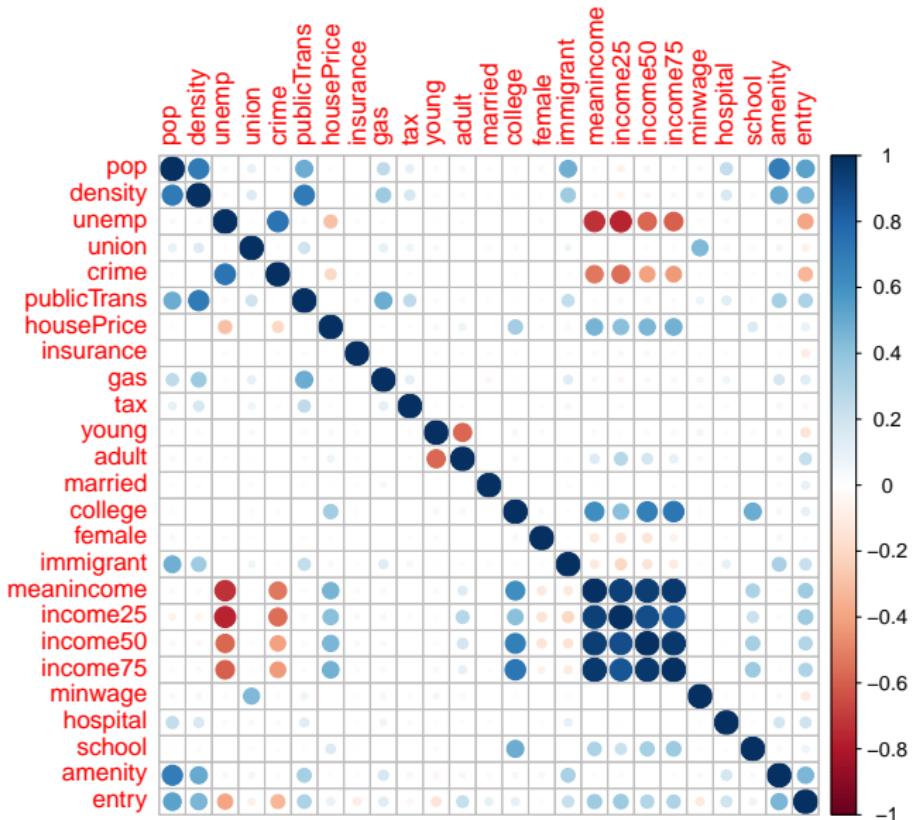
- total population
- population density
- average income
- percentage population with college degree
- minimum wage
- unemployment rate
- crime rate
- etc.

Supermarket Entry

```
# read data
data = read.csv("supermarket_entry.csv")
data[,1:24] = scale(data[,1:24]) # center & scale data
data$entry = as.factor(data$entry)

# create training and test sets
require(caret)
train = createDataPartition(data$entry,p=0.5,list=F)
data_train = data[train,]
data_test = data[-train,]
```

Supermarket Entry



Supermarket Entry

```
#####
# Logistic Regression #
#####

require(AER)
fit = glm(entry~.,data_train,family="binomial")
results = coeftest(fit)
results[1:10,]

##           Estimate Std. Error     z value   Pr(>|z|)
## (Intercept) -2.453152189  0.1967120 -12.47077924 1.077670e-35
## pop          2.926116034  0.2739735  10.68028881 1.258794e-26
## density      1.772984978  0.2275741   7.79080418 6.658404e-15
## unemp         -2.765055539  0.5330852  -5.18689228 2.138323e-07
## union         -1.101124570  0.1531317  -7.19070145 6.445929e-13
## crime         -0.976543245  0.1792566  -5.44773919 5.101409e-08
## publicTrans   0.259851907  0.1745623   1.48859149 1.365950e-01
## housePrice   -0.737637260  0.1415481  -5.21121131 1.876116e-07
## insurance    -1.047932648  0.1389732  -7.54053789 4.680369e-14
## gas           0.002968719  0.1322816   0.02244242 9.820950e-01
```

Supermarket Entry

```
results[11:25,]
```

	Estimate	Std. Error	z value	Pr(> z)
## tax	-0.9355652	0.1379065	-6.7840523	1.168510e-11
## young	-0.3732518	0.1916235	-1.9478395	5.143417e-02
## adult	1.6714084	0.2659985	6.2835250	3.309811e-10
## married	1.2394382	0.1467281	8.4471788	2.984265e-17
## college	1.4075509	0.4098684	3.4341534	5.944078e-04
## female	-0.2281599	0.1520545	-1.5005139	1.334813e-01
## immigrant	-0.1331635	0.1748719	-0.7614918	4.463634e-01
## meanincome	1.0099997	0.9532922	1.0594860	2.893785e-01
## income25	1.6291830	0.5389211	3.0230454	2.502447e-03
## income50	-0.7924468	0.5308270	-1.4928531	1.354756e-01
## income75	-1.7747426	0.7851714	-2.2603251	2.380108e-02
## minwage	-0.9216894	0.1454276	-6.3377867	2.330892e-10
## hospital	0.6793887	0.1350251	5.0315743	4.864686e-07
## school	0.6074254	0.1415092	4.2924806	1.766880e-05
## amenity	0.8117051	0.1674314	4.8479857	1.247214e-06

Supermarket Entry

```
# test err
ytrue = data_test$entry
phat = predict(fit,data_test,type="response")
yhat = as.numeric(phat > 0.5)
table(ytrue,yhat)

##          yhat
## ytrue   0    1
##       0 907  64
##       1  79 520

1-mean(yhat==ytrue)

## [1] 0.0910828
```

Supermarket Entry

```
#####
# Classification Tree #
#####

require(rpart)
fit = rpart(entry~.,data_train)

# test err
yhat = predict(fit,data_test,type="class")
1-mean(yhat==ytrue)

## [1] 0.1656051
```

Supermarket Entry

```
#####
# Random Forest #
#####
require(randomForest)
fit = randomForest(entry~.,data=data_train,mtry=6) # mtry selected by cv

# test err
yhat = predict(fit,data_test)
1-mean(yhat==ytrue)

## [1] 0.1031847
```

Supermarket Entry

```
#####
# Boosting #
#####
require(gbm)
data_boost = transform(data_train,entry=as.numeric(entry)-1)
fit = gbm(entry~.,data=data_boost,distribution="adaboost",
           n.trees=2000,
           interaction.depth=10,
           shrinkage = 0.01) # parameters selected by cv
```

```
# test err
phat = predict(fit,data_test,n.trees=2000,type="response")
yhat = as.numeric(phat>0.5)
1-mean(yhat==ytrue)

## [1] 0.06305732
```

Supermarket Entry

```
#####
# SVM #
#####

require(e1071)
fit= svm(entry~,data_train,kernel="radial", # RBF kernel
         scale=F,
         gamma=0.01,
         cost=30) # parameters selected by cv

# test err
yhat = predict(fit,data_test,type="class")
1-mean(yhat==ytrue)

## [1] 0.07197452
```

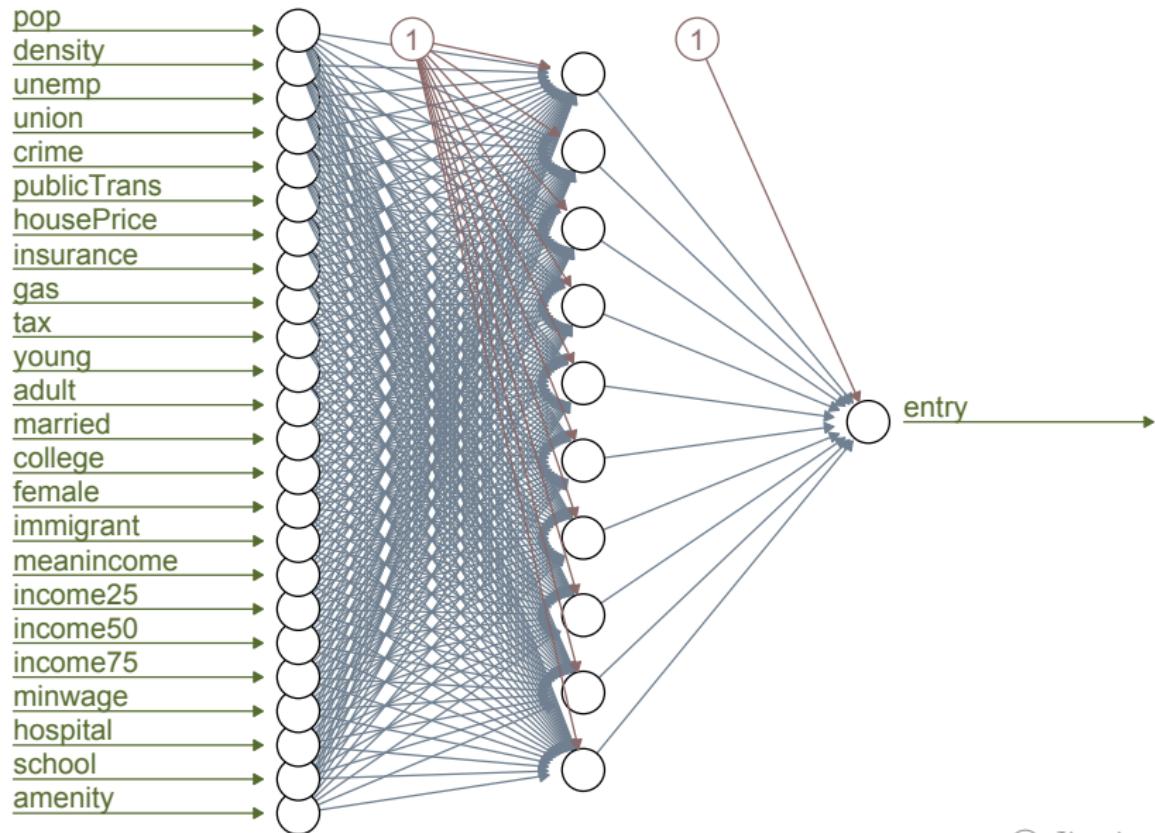
Supermarket Entry

```
#####
# Neural Network #
#####
# Fit a single-hidden-layer network
require(nnet)
fit = nnet(entry ~.,data_train,maxit=10000,
            size=10,
            decay=0.1) # parameters selected by cv
```

```
# test err
yhat = predict(fit,data_test,type="class")
1-mean(yhat==ytrue)

## [1] 0.04394904
```

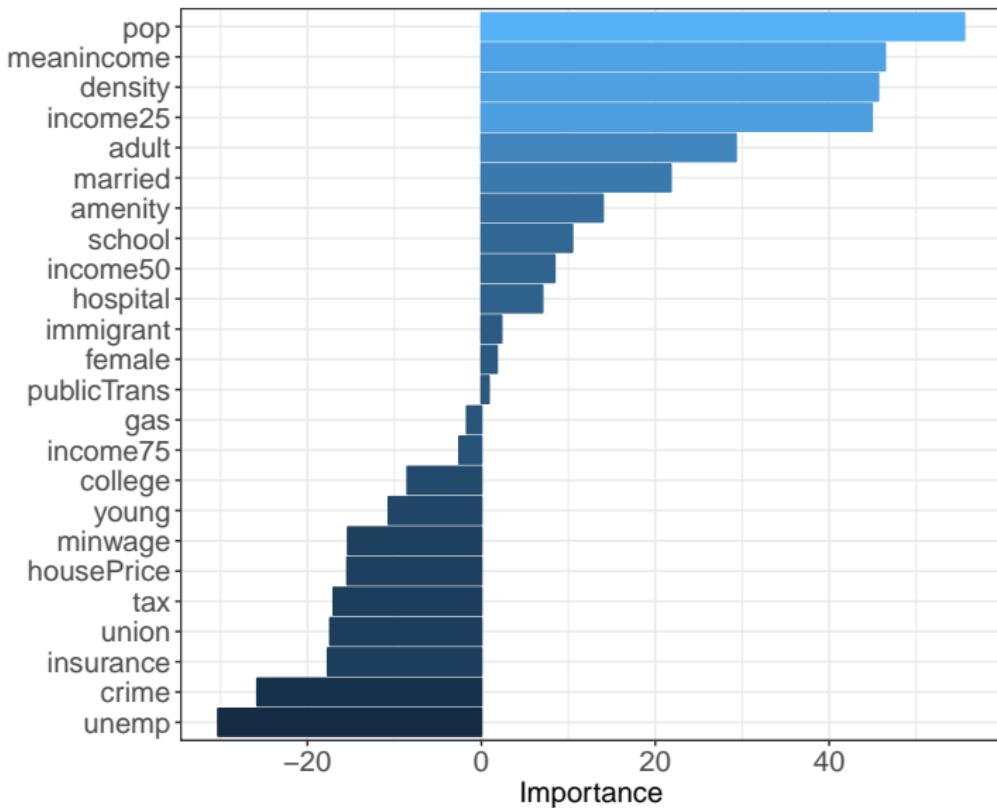
Supermarket Entry



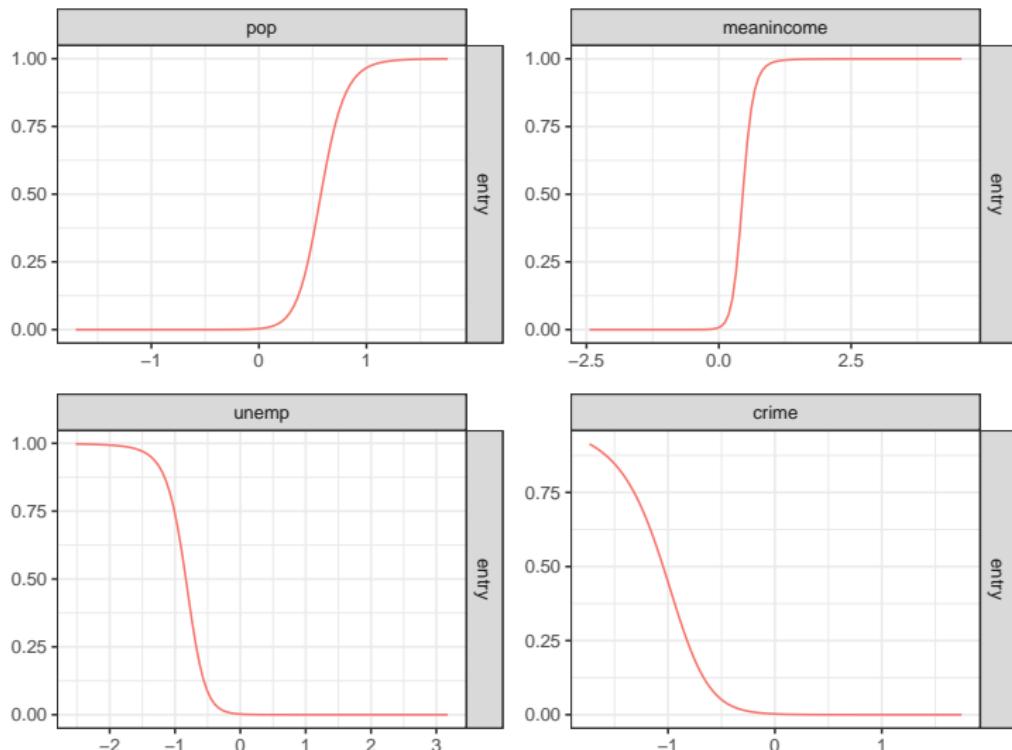
Supermarket Entry

- Like ensemble methods, neural networks are “**black boxes**” that lack interpretability and offer little insight into the relationships among variables.
- To better interpret a neural network model, we can deconstruct the model weights and calculate each variable’s **variable importance** based on its contribution to the output.
- Alternatively, we can calculate the **partial dependence** of the output on each input variable by either integrating out all the other input variables, or holding them constant.

Supermarket Entry



Supermarket Entry



Partial dependence (other variables held at median)

The Neural Network: Discussion

Consider a neural network with a single hidden layer, which implements a function of the form

$$h(x) = f \left(\beta_0 + \sum_{m=1}^M \beta_m \theta(\gamma'_m x) \right) \quad (3)$$

, where $\theta(.)$ is the hidden layer activation function and $f(.)$ is the output layer activation function. There are $M + 1$ number of units in the hidden layer. γ_m are the weights going into the m^{th} unit of the hidden layer, and β are the weights going into the output layer.

The Neural Network: Discussion

(3) is equivalent in functional form to a linear model with feature transform Φ and a nonlinear output transform f ⁸:

$$h(x) = f \left(\beta_0 + \sum_{m=1}^M \beta_m \phi_m(x) \right) \quad (4)$$

, where $\Phi = (\phi_1, \dots, \phi_M)$ are the basis functions.

⁸If $f(\cdot)$ is the identity function, then (4) is a linear basis function model.

The Neural Network: Discussion

Difference between (3) and (4):

- In (4), the basis functions ϕ_m are **fixed** before seeing the data.
- In (3), $\theta(\gamma'_m x)$ contains the parameters γ_m that are to be **learned** from the data.

The $\theta(\gamma'_m x)$'s are therefore called **tunable** or **adaptive basis functions**⁹ that give the neural network much more flexibility to fit the data than fixed basis functions.

⁹As in boosting

Feature Engineering vs. Feature Learning

- Because in neural networks, neurons in hidden layers can be thought of as features that are *learned* from the data, neural networks can be thought of as enabling **feature learning** or **representation learning**.
- In contrast, in “classical” machine learning, features are manually constructed by human experts – a practice called **feature engineering**. Algorithms are then applied to find the mapping from features to output.

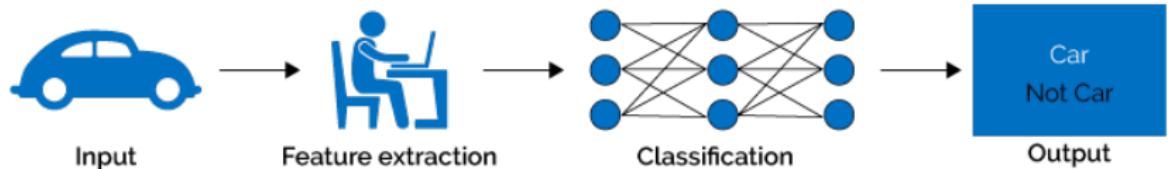
Deep Learning

Deep learning models are neural networks with many hidden layers. This allows the construction of **hierarchical** features and enables representation learning with multiple levels of representation that correspond to multiple levels of abstraction.

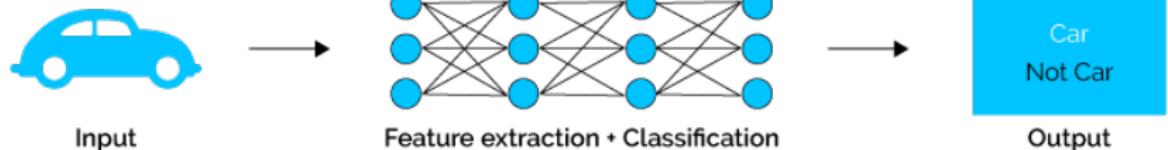
- Shallow neurons represent low level features. Deep neurons represent high level features.
- More complex representations are expressed in terms of other, simpler representations.

Deep Learning

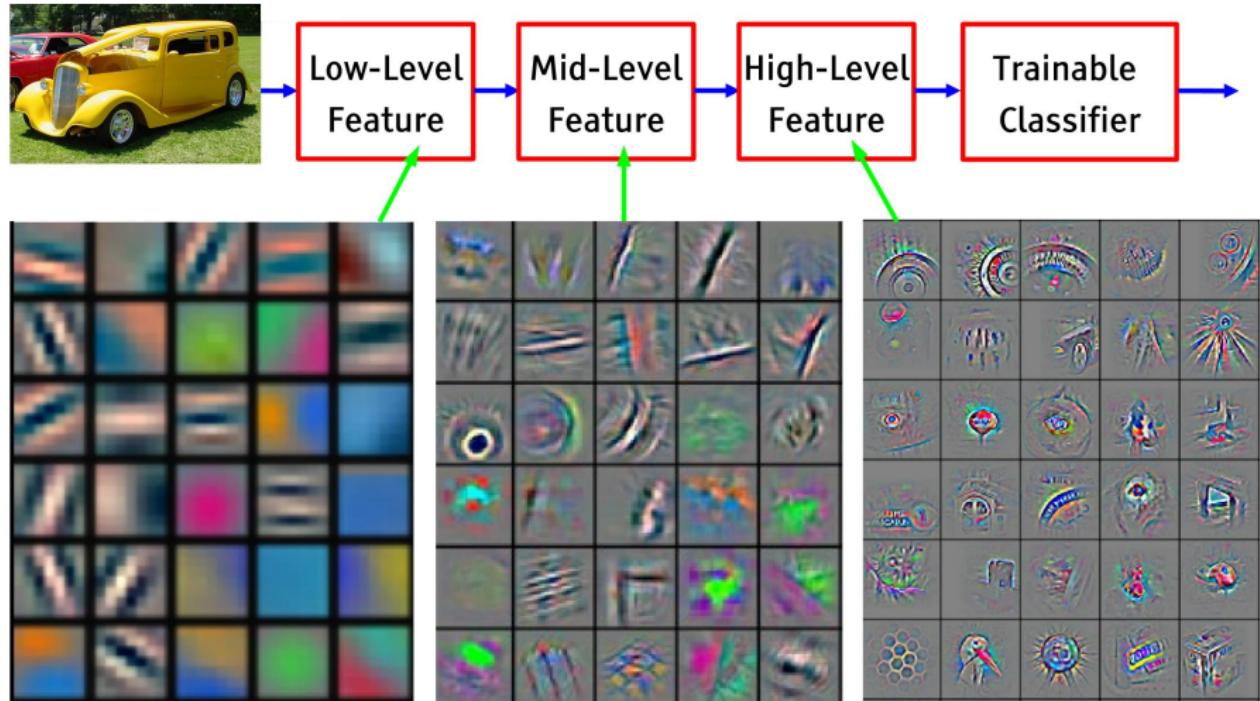
Machine Learning



Deep Learning



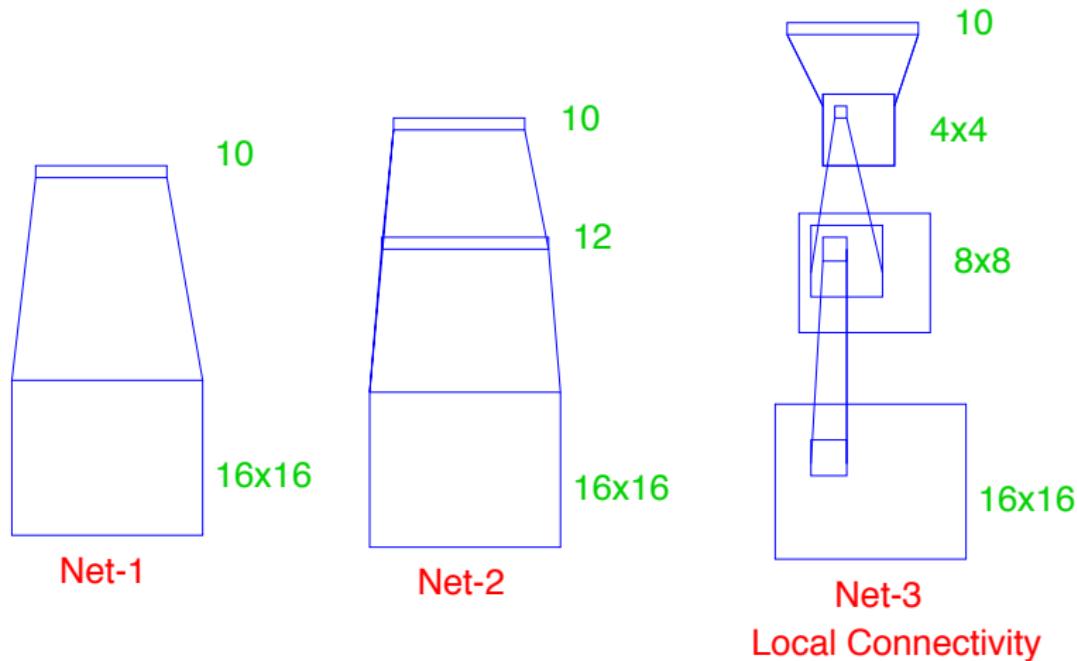
Deep Learning



Deep Learning

- Deep networks are highly flexible and allow a wide variety of **architectures**, which allow the incorporation of *domain-specific knowledge* (e.g., translational invariance in image recognition) into the model.
- Like parametric models, neural networks have specific parameters to learn, but like nonparametric models, their complexity can grow with data.
- While “classical” nonparametric methods rely mainly on the smoothness prior, deep neural nets impose a structure of hierarchical representation while incorporating domain-specific knowledge in order to defeat the curse of dimensionality.

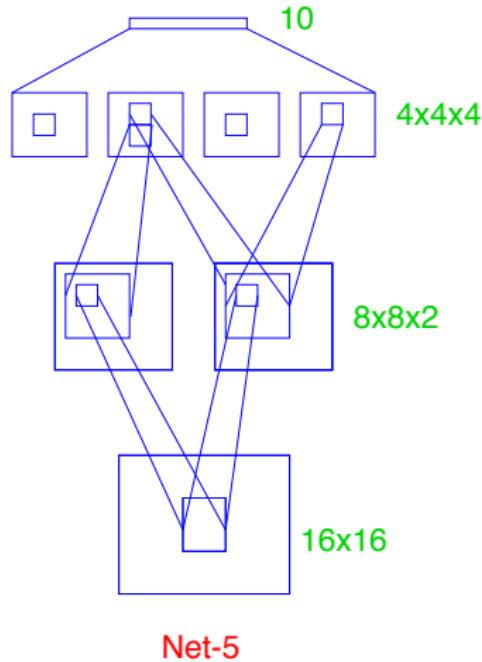
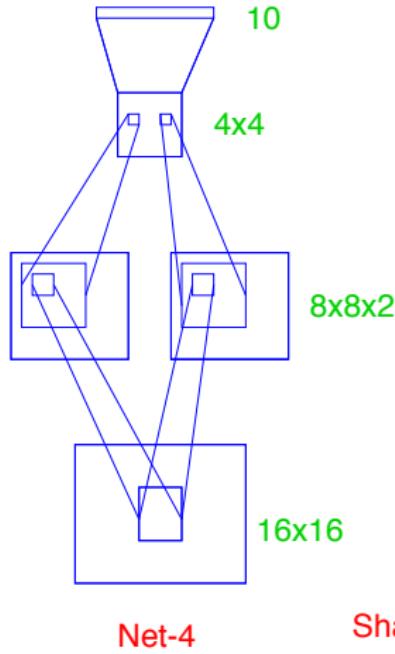
Deep Learning



Net-1: no hidden layer; Net-2: one hidden layer

Net-3: two hidden layers **locally connected**

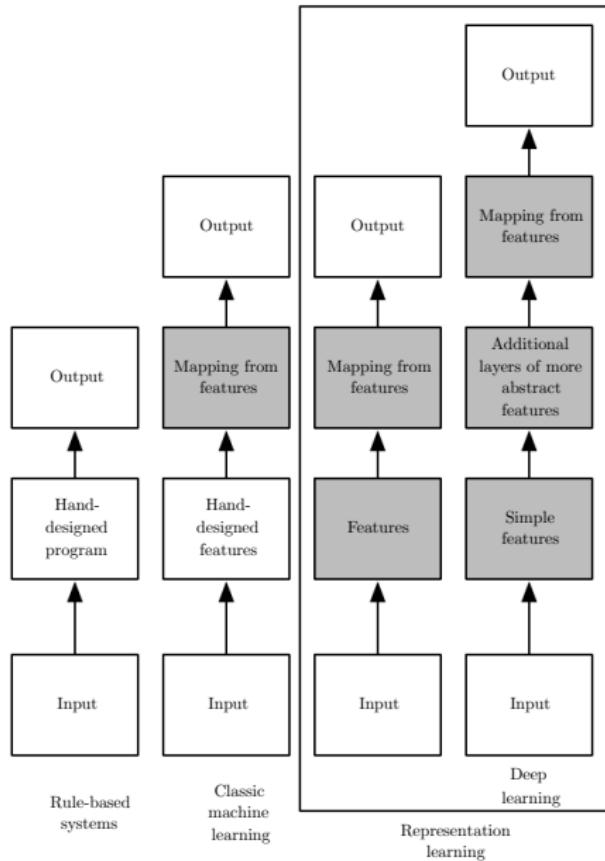
Deep Learning



Net-4: two hidden layers, **locally connected** with **weight sharing**

Net-5: two hidden layers, **locally connected**, **two levels of weight sharing**

Deep Learning



Acknowledgement

Part of this lecture is adapted from the following sources:

- Abu-Mostafa, Y. S., M. Magdon-Ismail, and H. Lin. 2012. *Learning from Data*. AMLBook.
- Bishop, C. M. 2011. *Pattern Recognition and Machine Learning*. Springer.
- Goodfellow, I., Y. Bengio, and A. Courville. 2016. *Deep Learning*. The MIT Press.
- Hastie, T., R. Tibshirani, and J. Friedman. 2008. *The Elements of Statistical Learning* (2nd ed.). Springer.