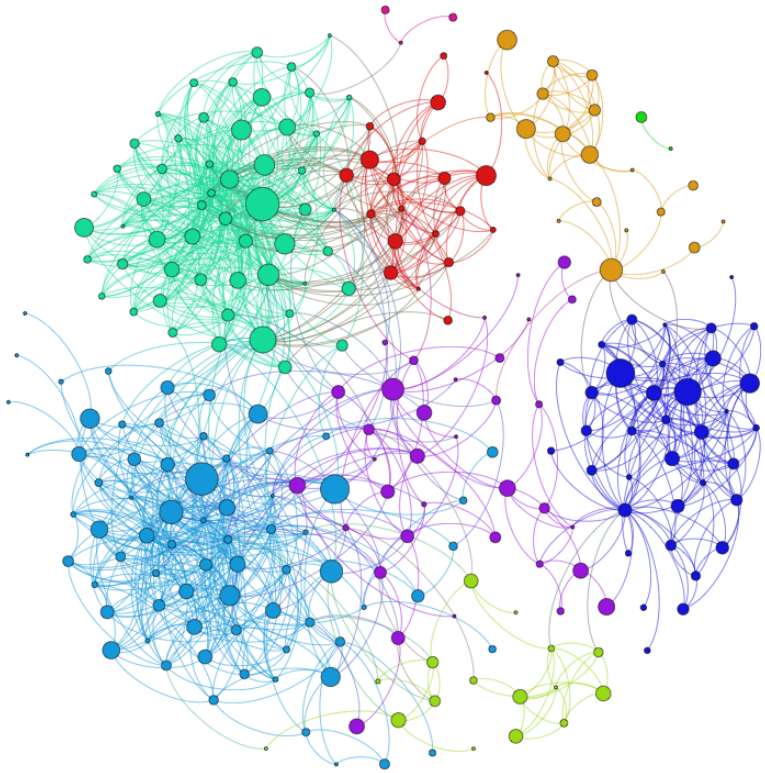
Neural Networks

Jiaming Mao

Xiamen University

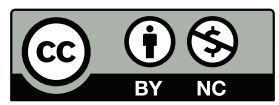


Copyright © 2017–2019, by Jiaming Mao

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

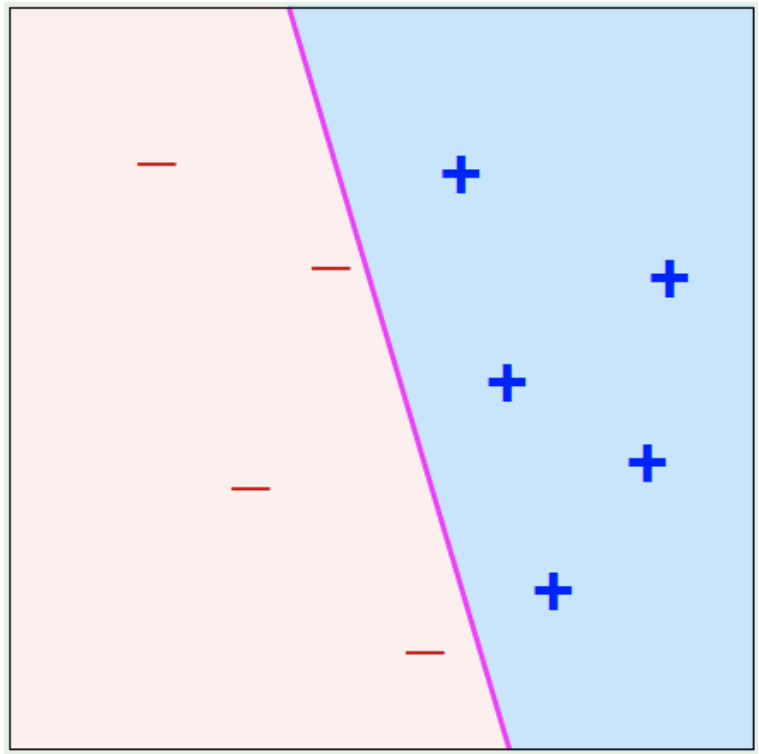
Consider a binary classification problem. Let $y \in \{-1, 1\}$ and $x = (1, x_1, \ldots, x_p)$. The **perceptron** is a linear classifier

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

that minimizes the misclassification loss

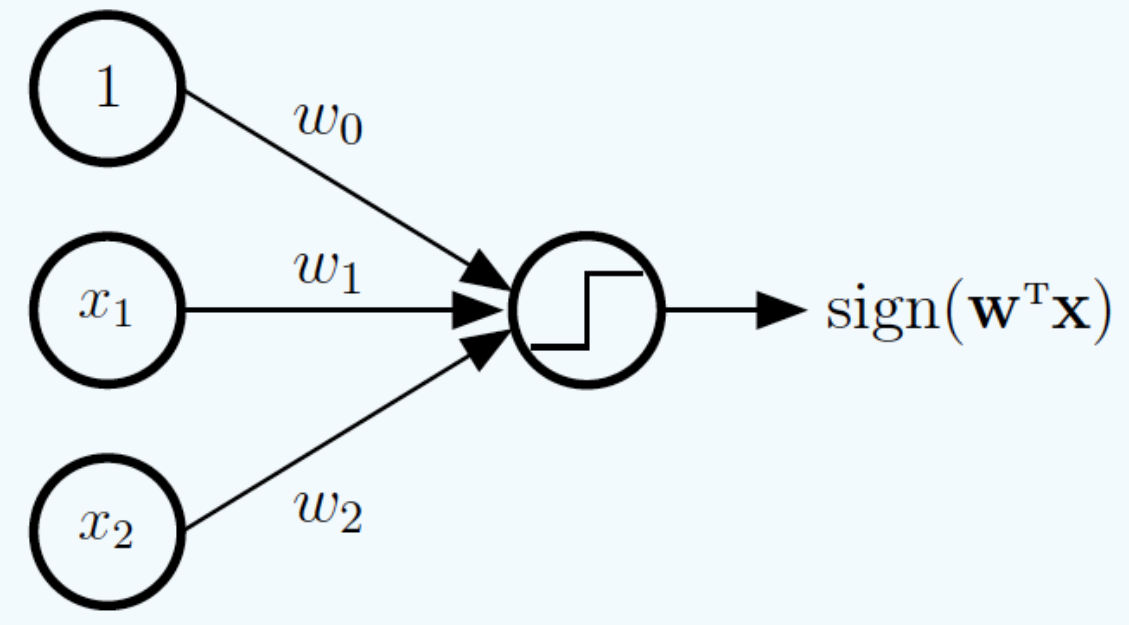
$$\ell\left(y, h(x)\right) = \mathcal{I}\left(y \neq h(x)\right)$$

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



© Jiaming Mao

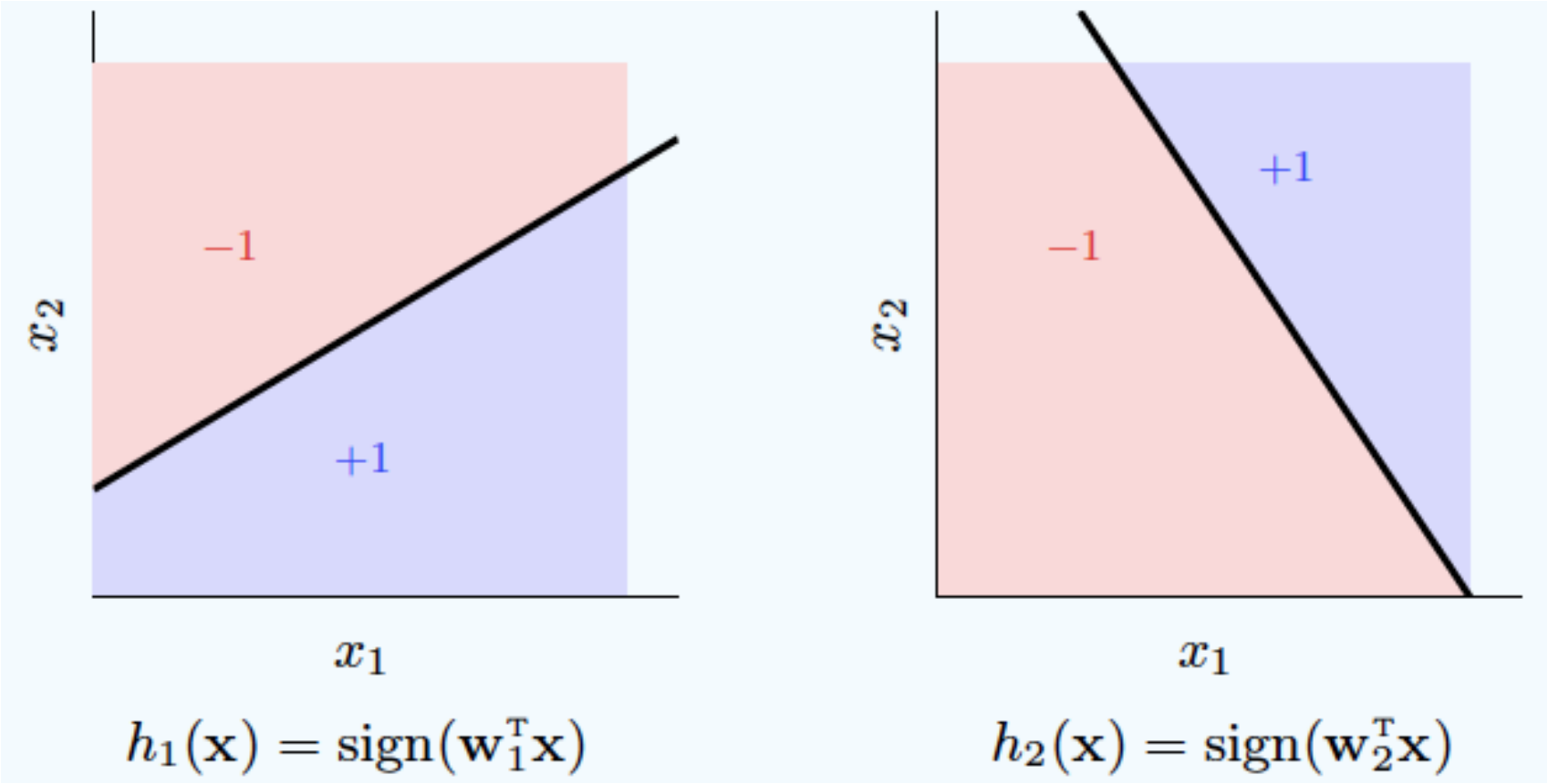
The Perceptron



Graph representation of the perceptron

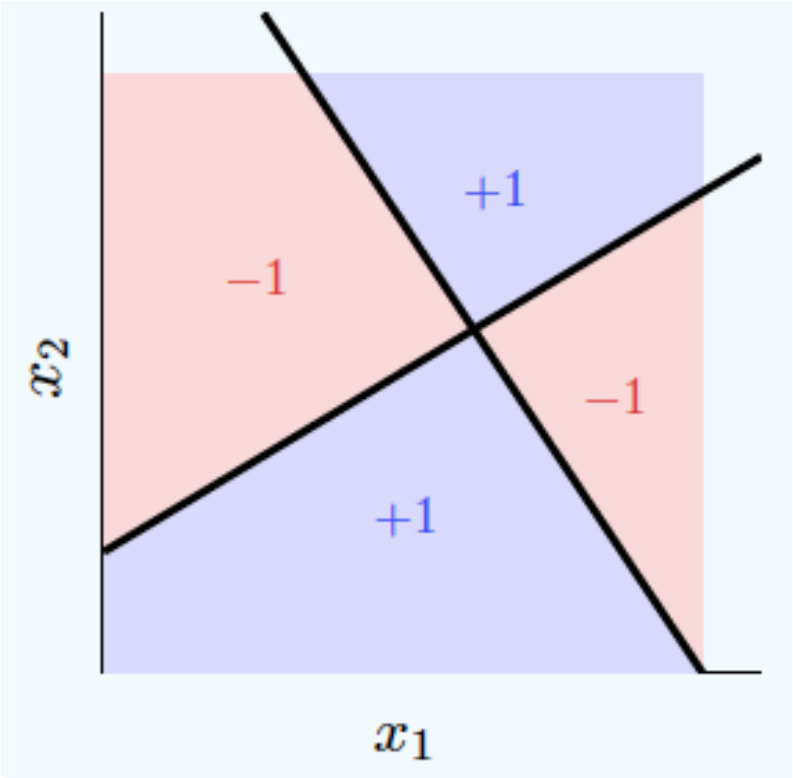
© Jiaming Mao

Combining Perceptrons



© Jiaming Mao

Combining Perceptrons



$$f = \text{XOR}(h_1, h_2)$$
$$= h_1\overline{h_2} + \overline{h_1}h_2$$

[1]

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

© Jiaming Mao

Perceptrons for Boolean Functions

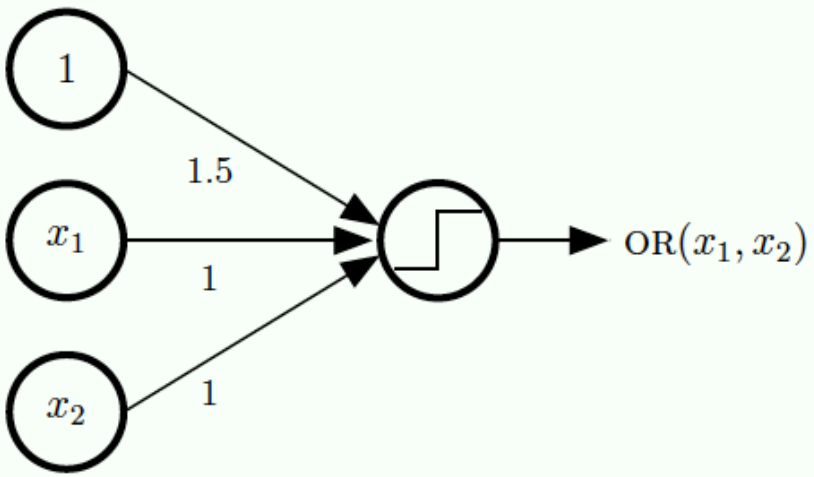
The Boolean functions AND and OR themselves can be implemented by the perceptron: for $x_1, x_2 \in \{-1, 1\}$,

$$\begin{aligned} \mathrm{OR}\,(x_1, x_2) &= \mathrm{sign}\,(x_1 + x_2 + 1.5) \\ \mathrm{AND}\,(x_1, x_2) &= \mathrm{sign}\,(x_1 + x_2 - 1.5) \end{aligned}$$

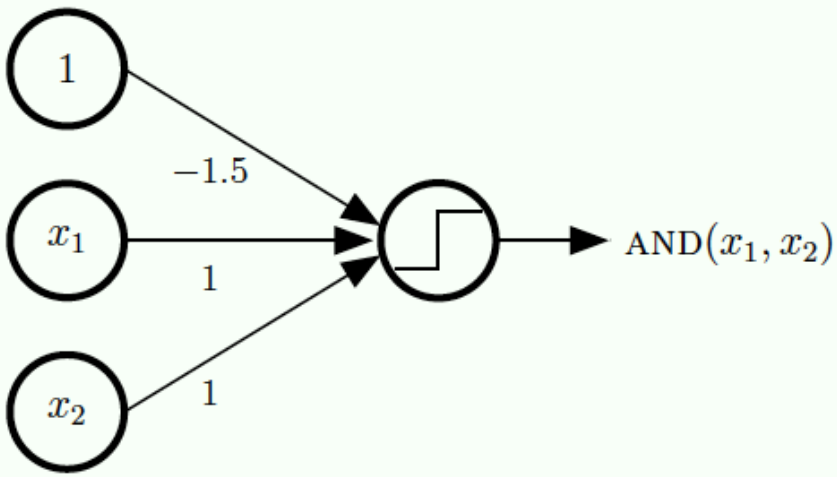
© Jiaming Mao

Perceptrons for Boolean Functions

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



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

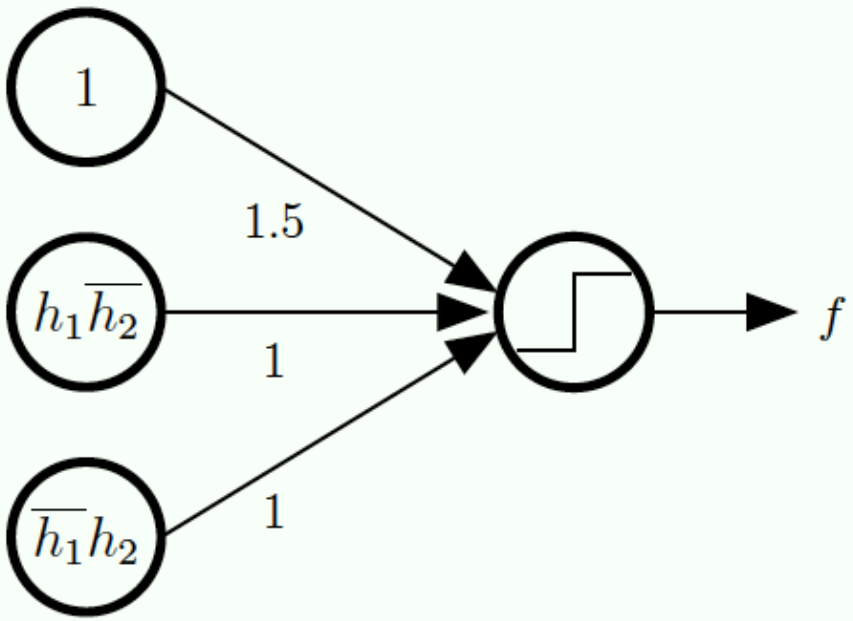


Graph representation of perceptrons for Boolean functions

© Jiaming Mao

Combining Perceptrons

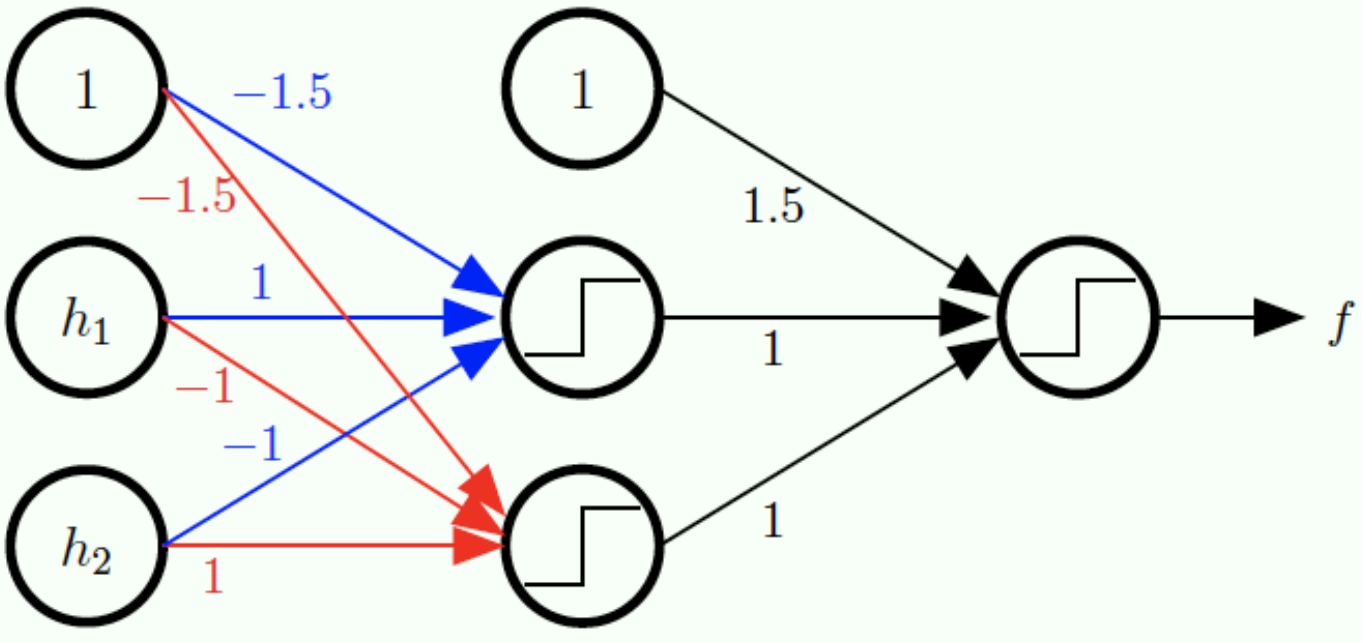
$$f = h_1\overline{h_2} + \overline{h_1}h_2$$



© Jiaming Mao

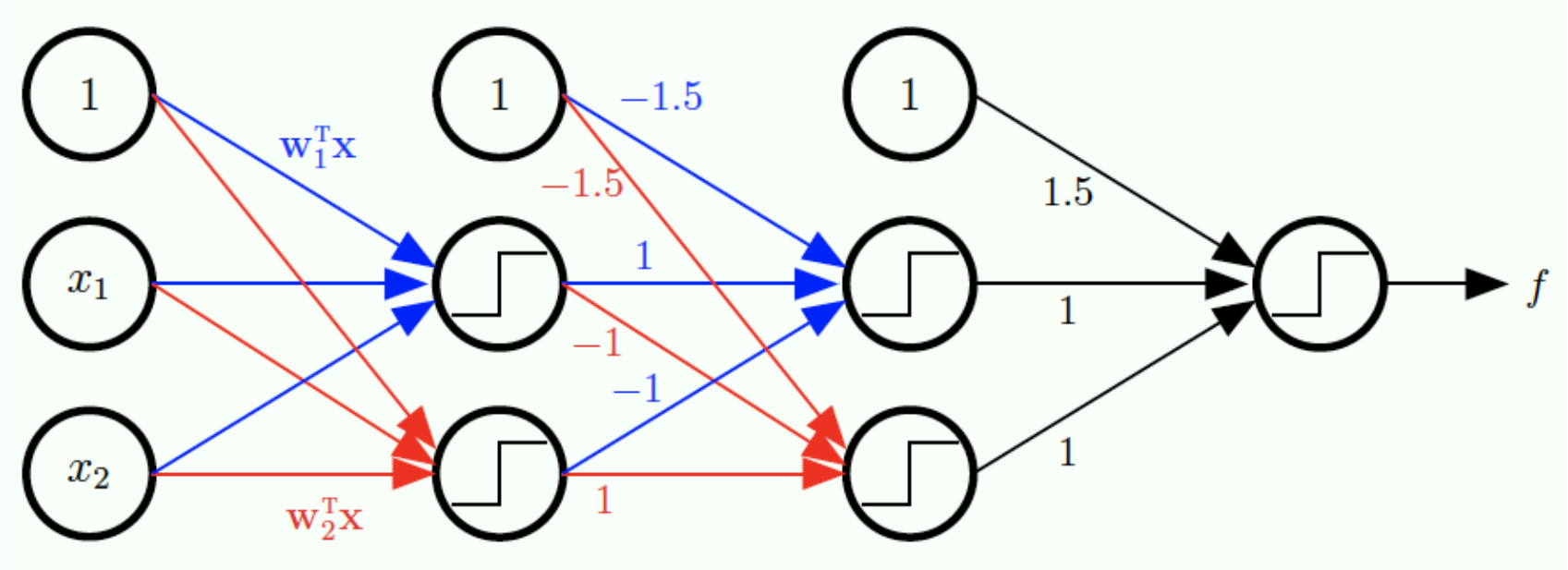
Combining Perceptrons

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



© Jiaming Mao

Combining Perceptrons



$$f(x) = \text{sign}\left\{\text{sign}\left[\text{sign}\left(w_1'x\right) - \text{sign}\left(w_2'x\right) - 1.5\right] + \text{sign}\left[-\text{sign}\left(w_1'x\right) + \text{sign}\left(w_2'x\right) - 1.5\right] + 1.5\right\}$$

© Jiaming Mao

The Multilayer Perceptron

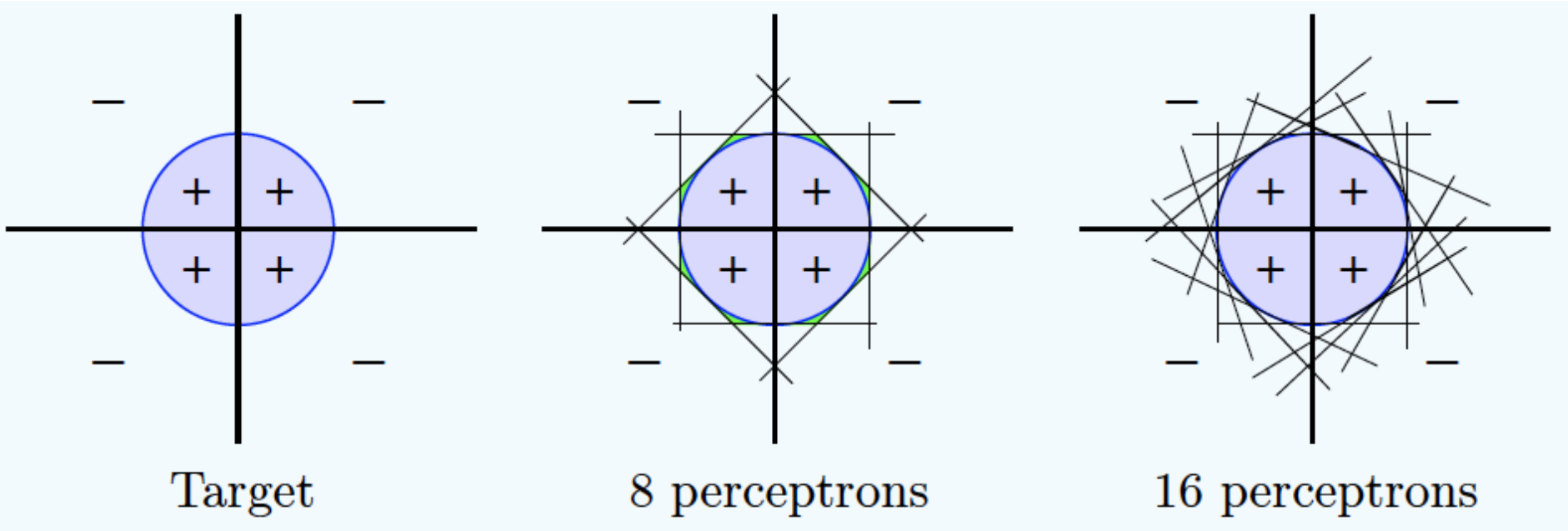
f is called a **multilayer perceptron** (**MLP**)

- 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.

© Jiaming Mao

Universal Approximation

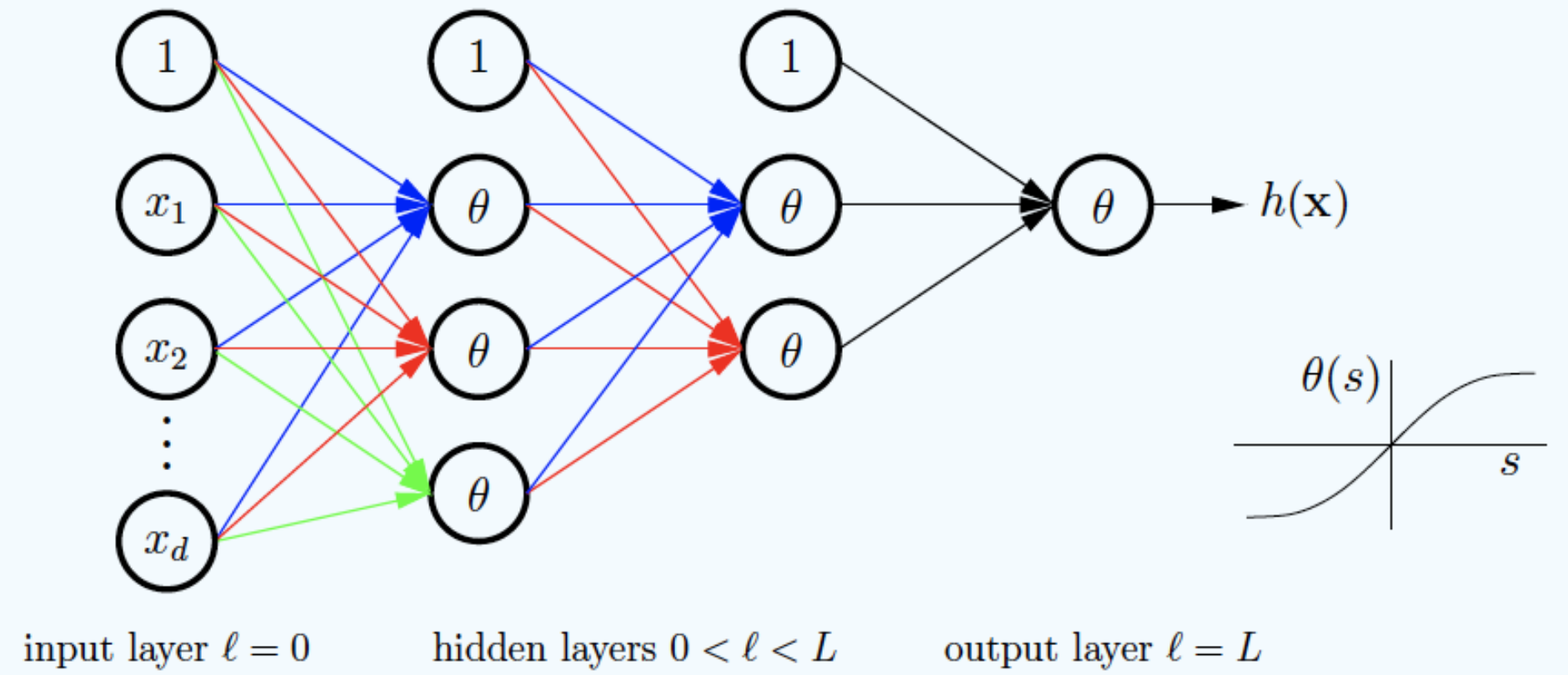


A sufficiently smooth decision boundary can "essentially" be decomposed into linear separators and hence approximated arbitrarily closely by a 3-layer MLP[2].

[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.

© Jiaming Mao

The Neural Network



© Jiaming Mao

The Neural Network

- A neural network of L layers contains $L-1$ hidden layers and one output layer. The layers are numbered $\ell = 0, 1, \ldots, L$, with $\ell = 0$ denoting the input layer[3].
- Each hidden layer ℓ contains $1 + d^{(\ell)}$ **units** (**neurons**), indexed $0, 1, \ldots, d^{(\ell)}$, 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]:

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

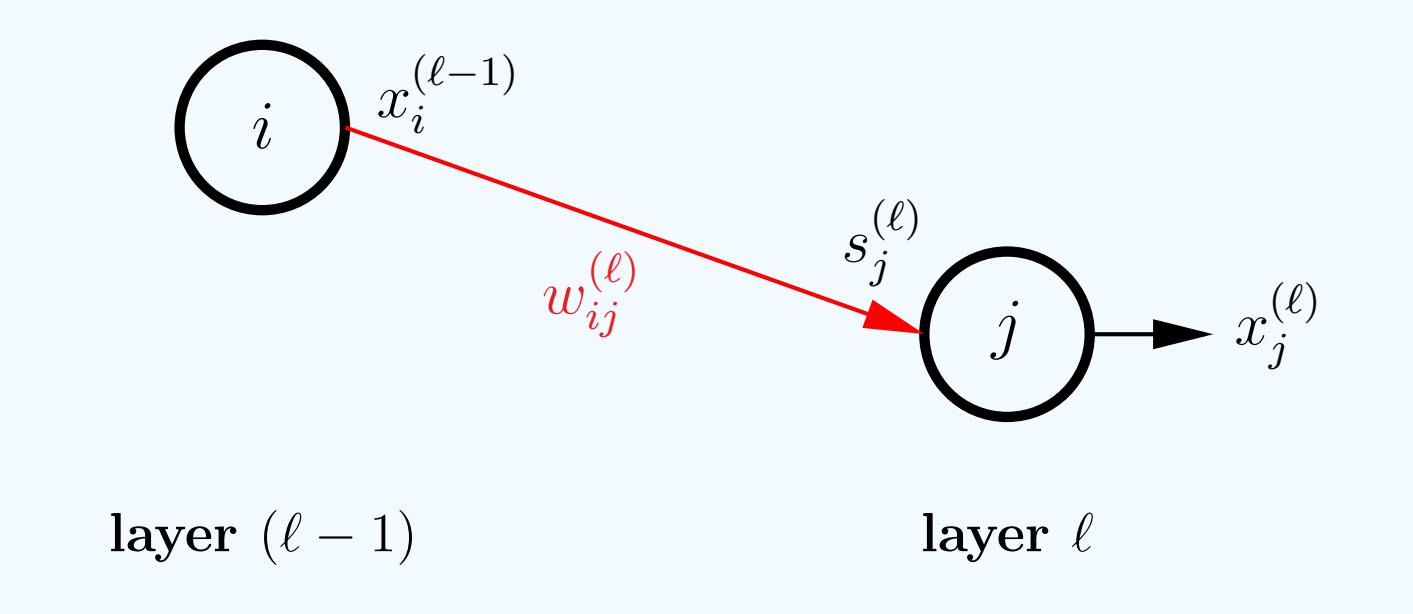
, where $s_i^{(\ell)}$ is the weighted sum of the outputs of the previous layer: $s_i = \sum_{j=0}^{d^{(\ell-1)}} x_j^{(\ell-1)} w_{ji}^{(\ell)}$. The bias unit is set to have an output 1.

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

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

© Jiaming Mao

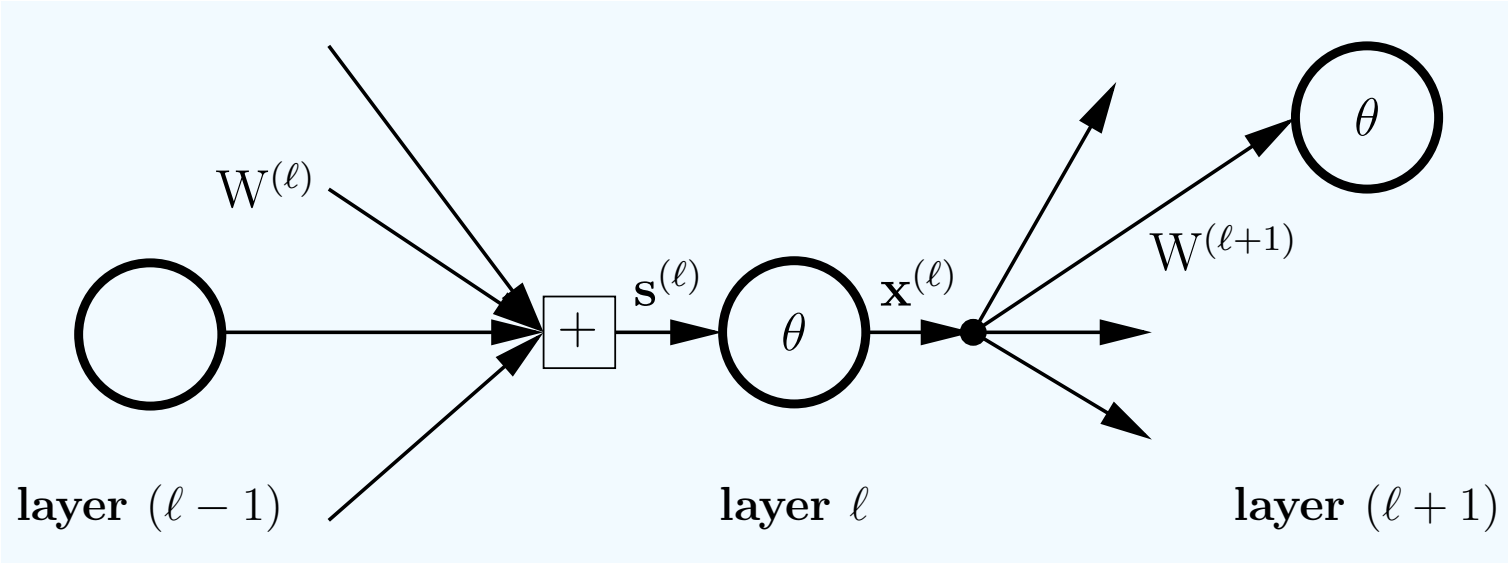
The Neural Network



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

© Jiaming Mao

The Neural Network



layer ℓ parameters

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

© Jiaming Mao

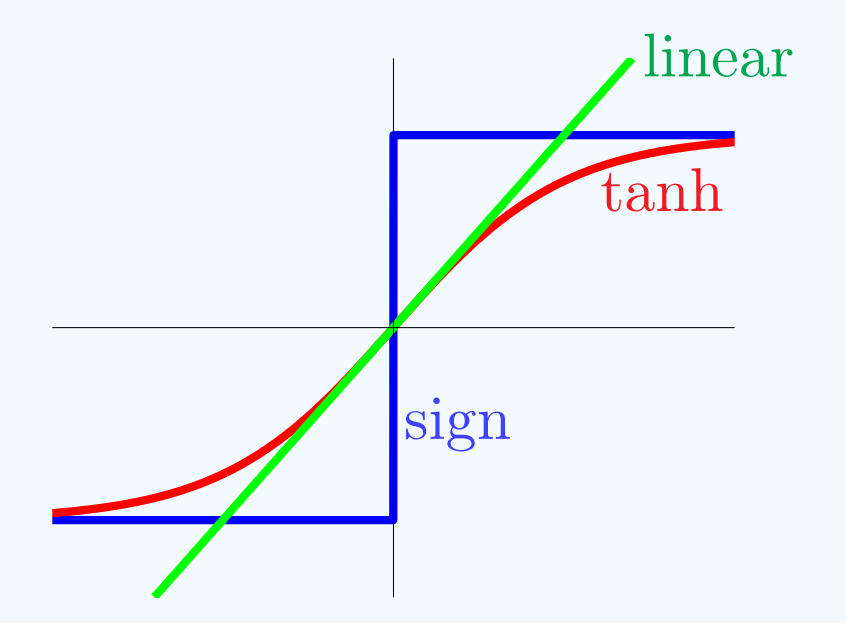
The Neural Network

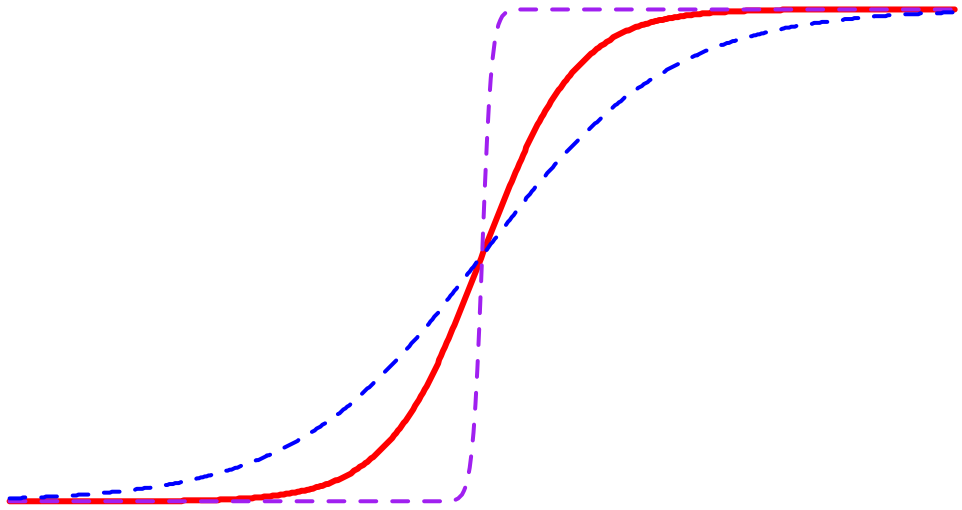
- The MLP uses the step function $\text{sign}(.)$ as its activation functions, which makes it hard to optimize.
- In neural networks, for the hidden layers, we can replace the hard-threshold $\text{sign}(.)$ function with the soft-threshold $\tanh(.)$ function.
- Other popular activation functions for hidden layers:
  - Logistic sigmoid[5]: $\sigma(x) = \frac{1}{1+e^{-x}}$
  - The Rectified Linear Unit (ReLU): $f(x) = \max(0, x)$

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

© Jiaming Mao

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)

© Jiaming Mao

The Neural Network

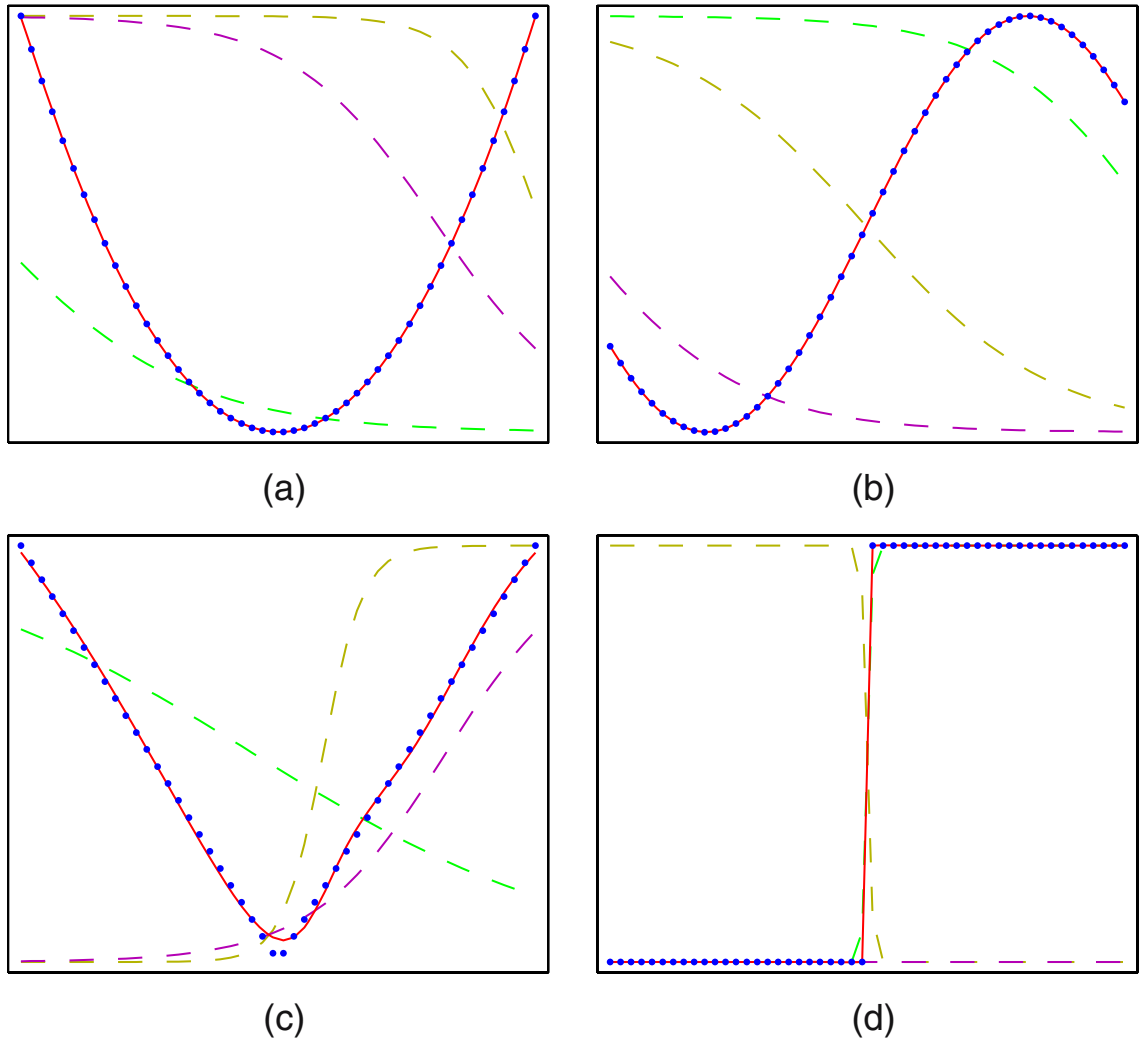
- For regression, the output layer typically contains a single unit[6], with the activation function being the identity function $f(x) = x$ and the error measure being the sum of squares error.
- For binary classification, the output layer contains a single unit. We can choose the activation function 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 error[7].

---

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

[7] Alternatively, 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: $y = [y_1, \ldots, y_K]'$, where $y_j \in \{0, 1\}, j \in \{1, \ldots, K\}$.

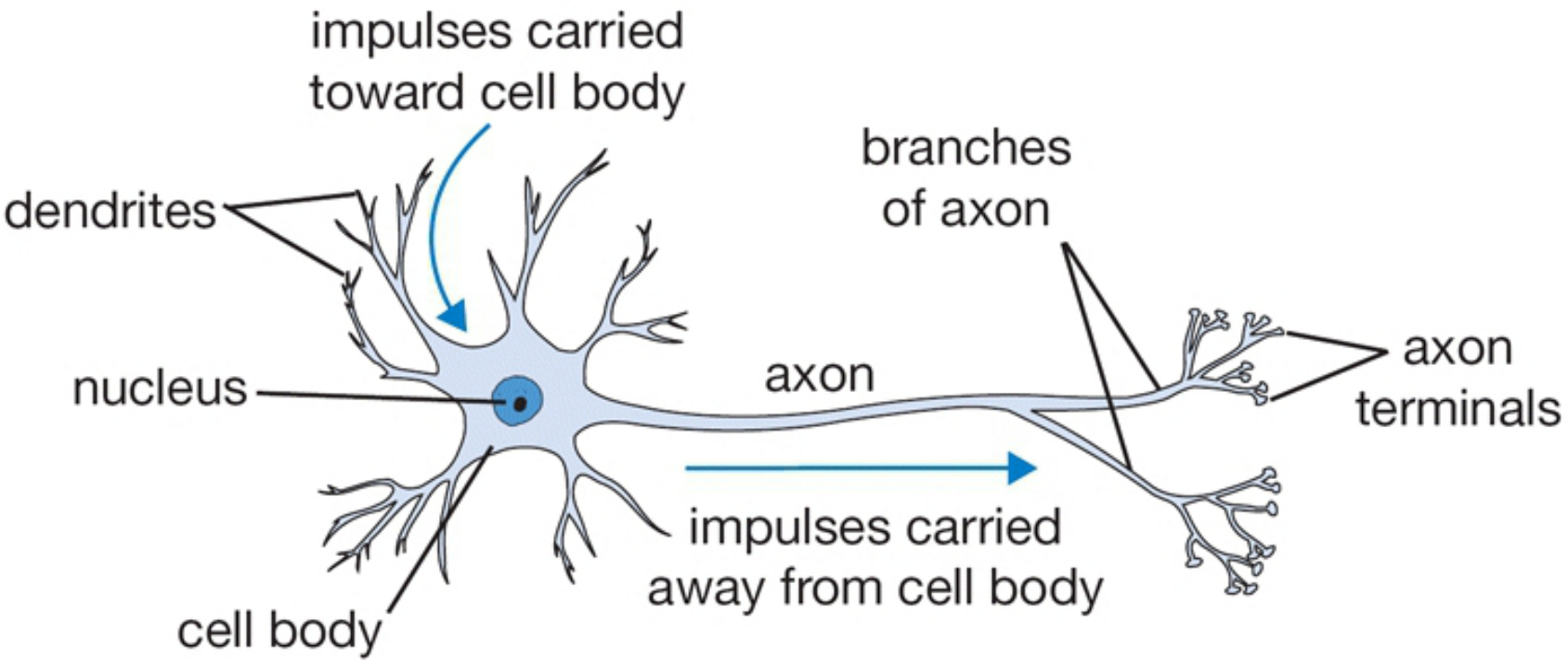
© Jiaming Mao



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.

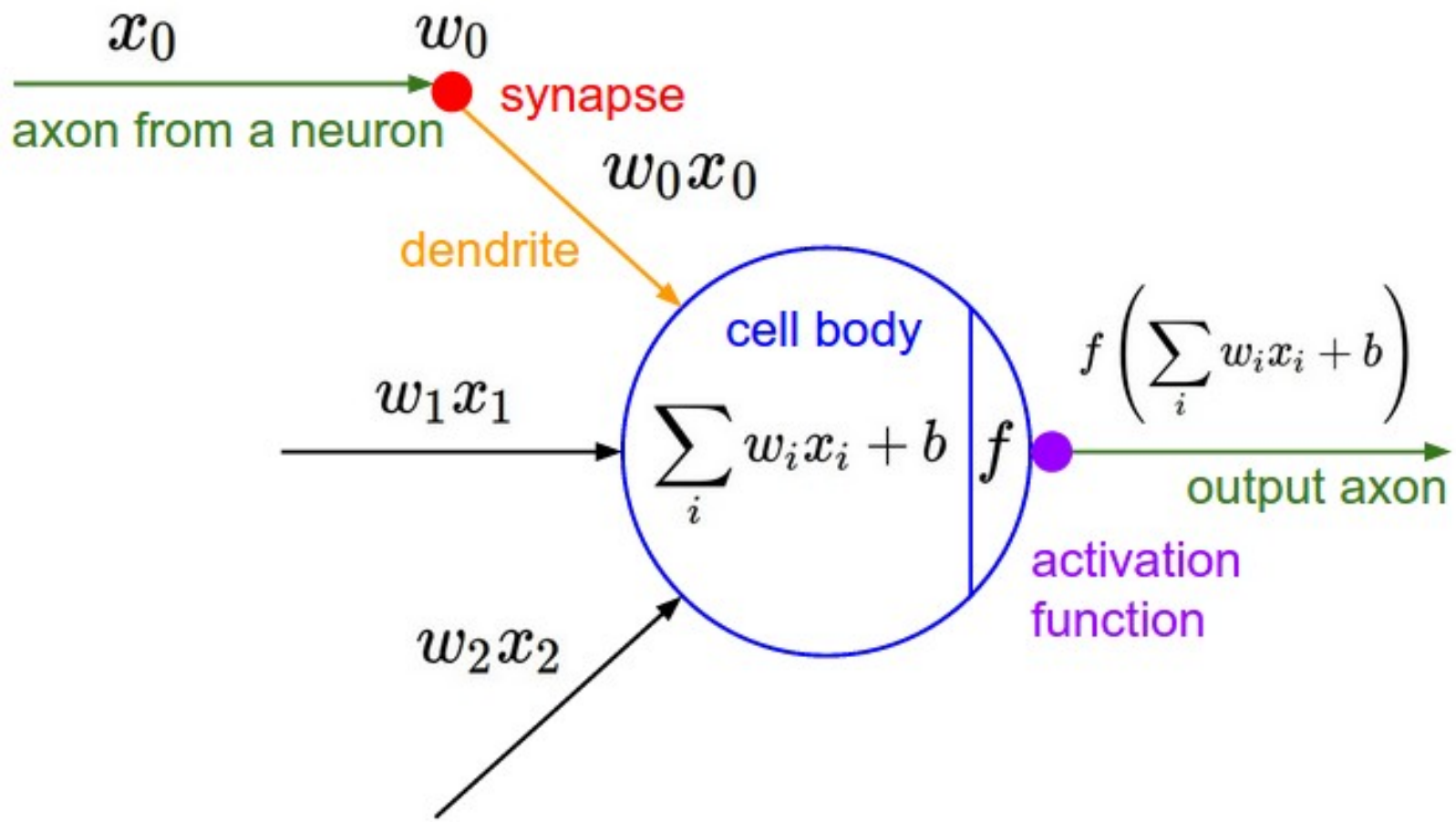
© Jiaming Mao

Biological Inspiration



© Jiaming Mao

Biological Inspiration



© Jiaming Mao

Estimation

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

The training error for a neural network is

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

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

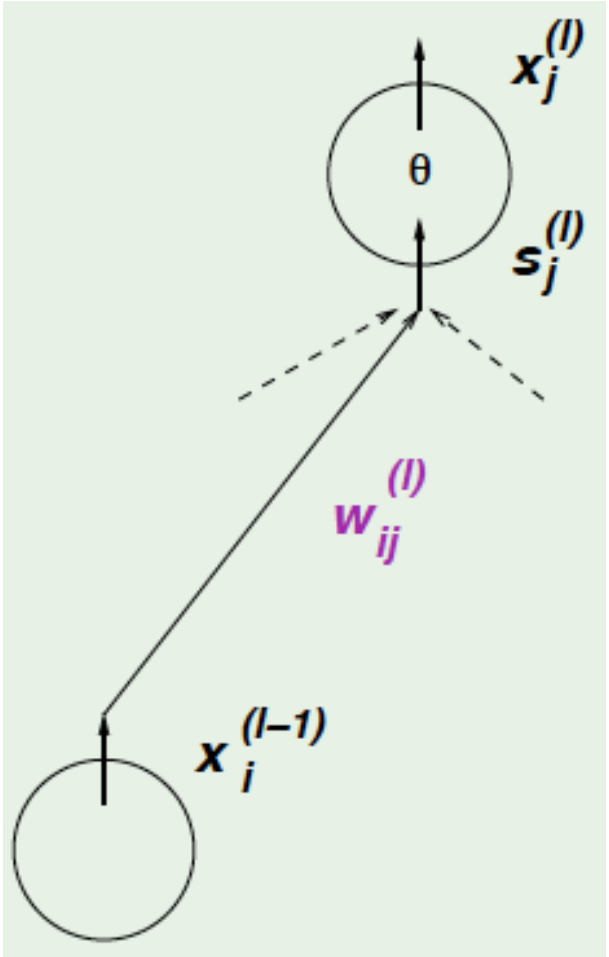
© Jiaming Mao

Estimation

An efficient way to compute (1):

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

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



© Jiaming Mao

Forward Propagation

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

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

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

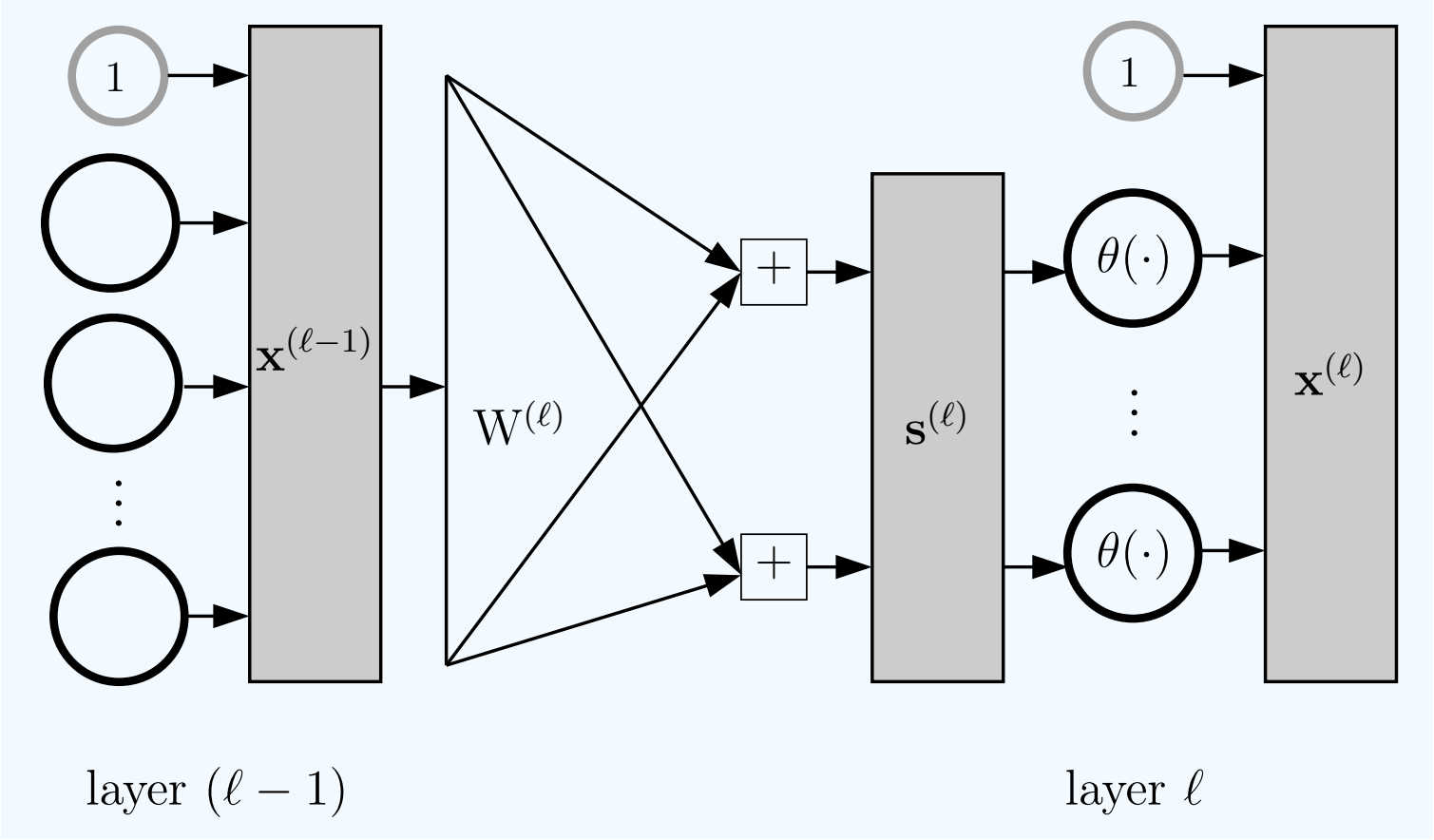
Therefore, starting with $x^{(0)} = x_n$, we can calculate $x^{(\ell)}$ for all ℓ as follows:

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

This is called **forward propagation**.

© Jiaming Mao

Forward Propagation



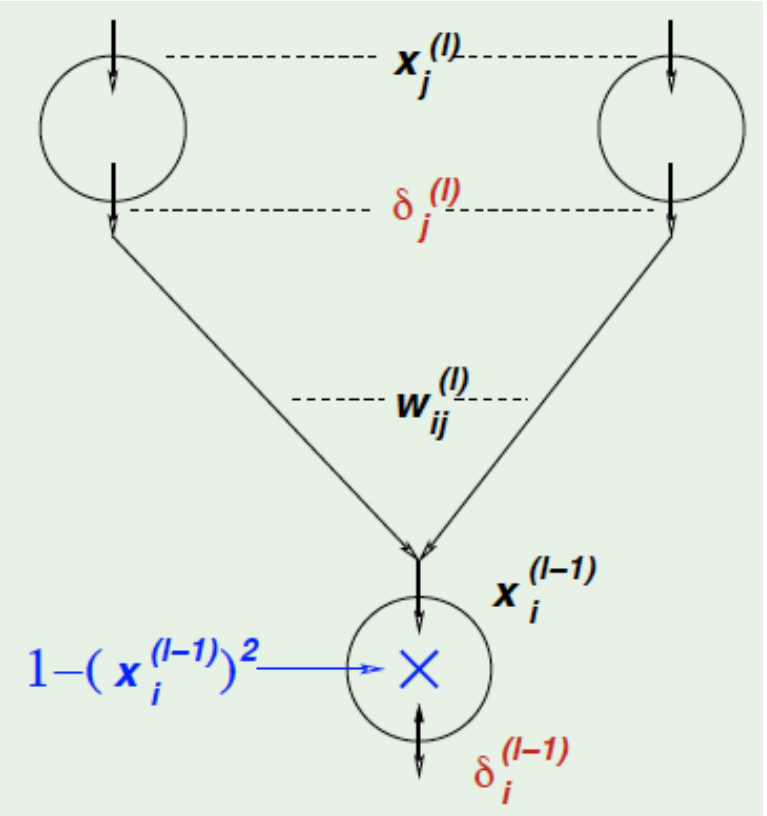
© Jiaming Mao

Backward Propagation

$\delta^{(l)} = \left[\delta_1^{(\ell)}, \ldots, \delta_{d^{(\ell)}}^{(\ell)}\right]'$ 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:

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



© Jiaming Mao

Backward Propagation

For $\ell < L$,

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

, where, for $\theta(.) = \tanh(.)$, we have
$\tanh'\left(s_i^{(\ell-1)}\right) = 1 - \left[\tanh\left(s_i^{(\ell-1)}\right)\right]^2 = 1 - \left(x_i^{(\ell-1)}\right)^2$.

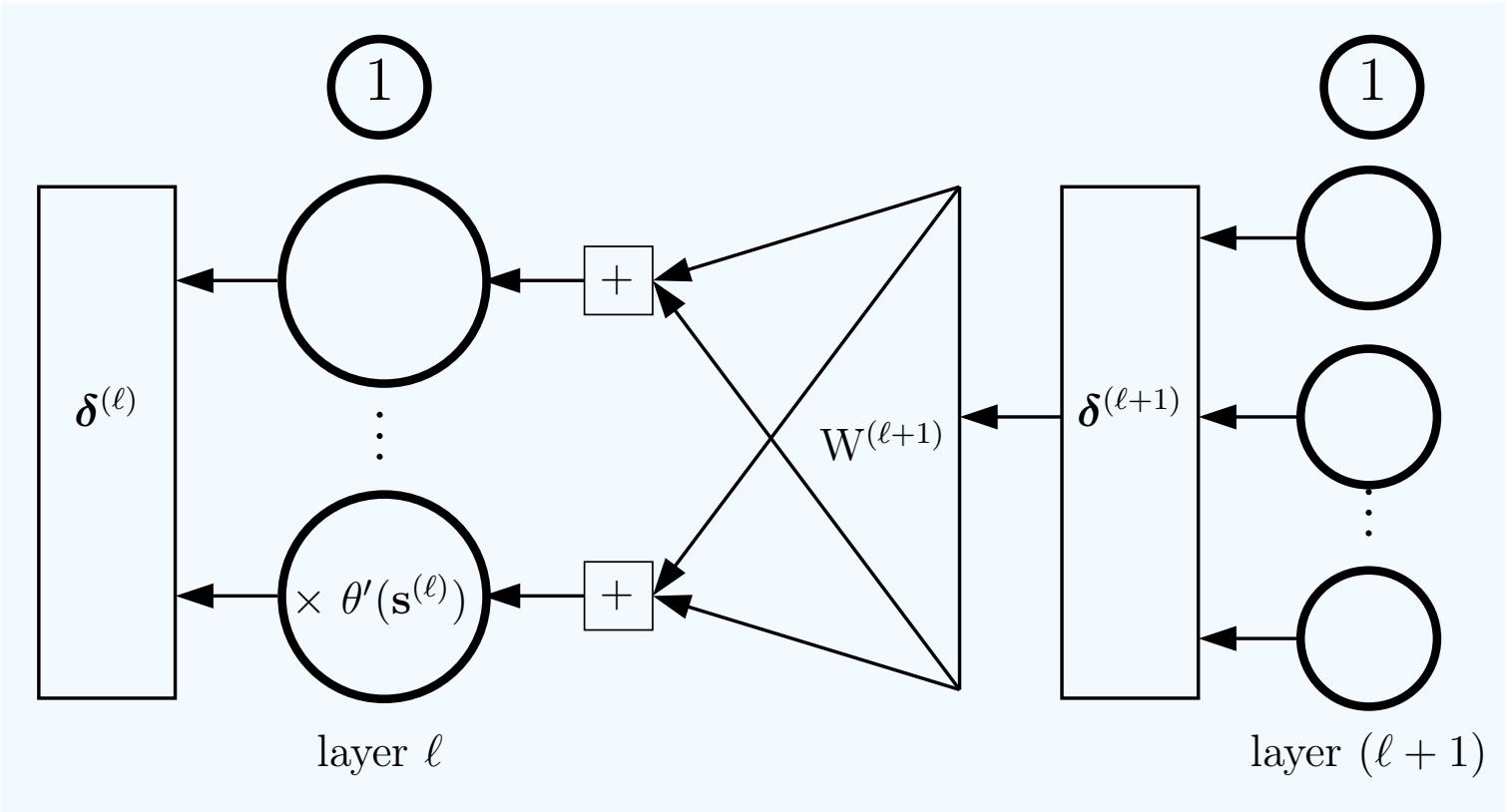
© Jiaming Mao

Backward Propagation

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

$$\delta^{(1)} \longleftarrow \delta^{(2)} \ldots \longleftarrow \delta^{(L-1)} \longleftarrow \delta^{(L)}$$

This is called **backward propagation**.



© Jiaming Mao

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.

© Jiaming Mao

Gradient Descent

Gradient Descent (Batch)

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

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

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

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

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

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

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

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

© Jiaming Mao

Gradient Descent

Stochastic Gradient Descent (SGD)

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

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

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

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

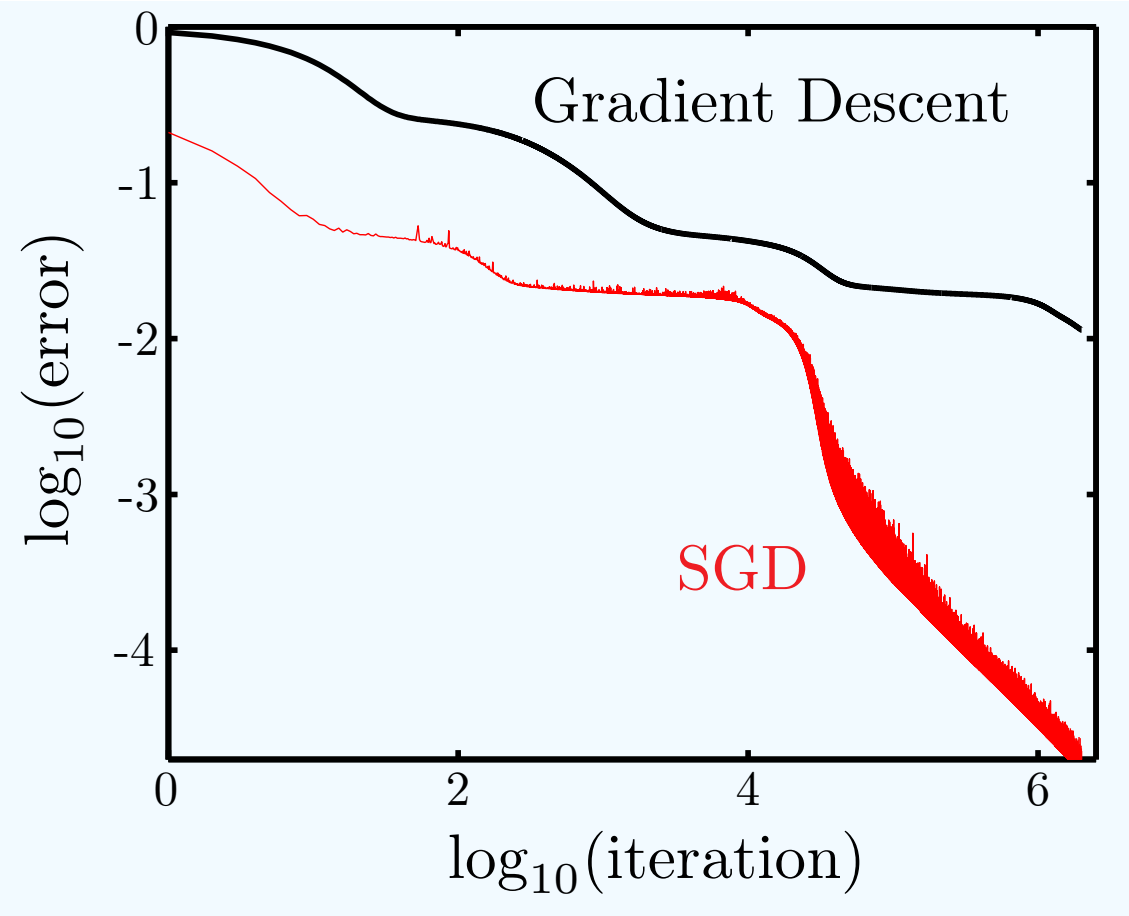
    Compute $\delta^{(\ell)}$ for $\ell = L, \ldots, 1$ (backward propagation)

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

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

© Jiaming Mao

Gradient Descent



© Jiaming Mao

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. $w_i \sim \mathcal{N}\left(0, \sigma_w^2\right)$, 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 $\tanh\left(w'x\right) \approx \pm 1$ 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.

© Jiaming Mao

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:

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

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.

© Jiaming Mao

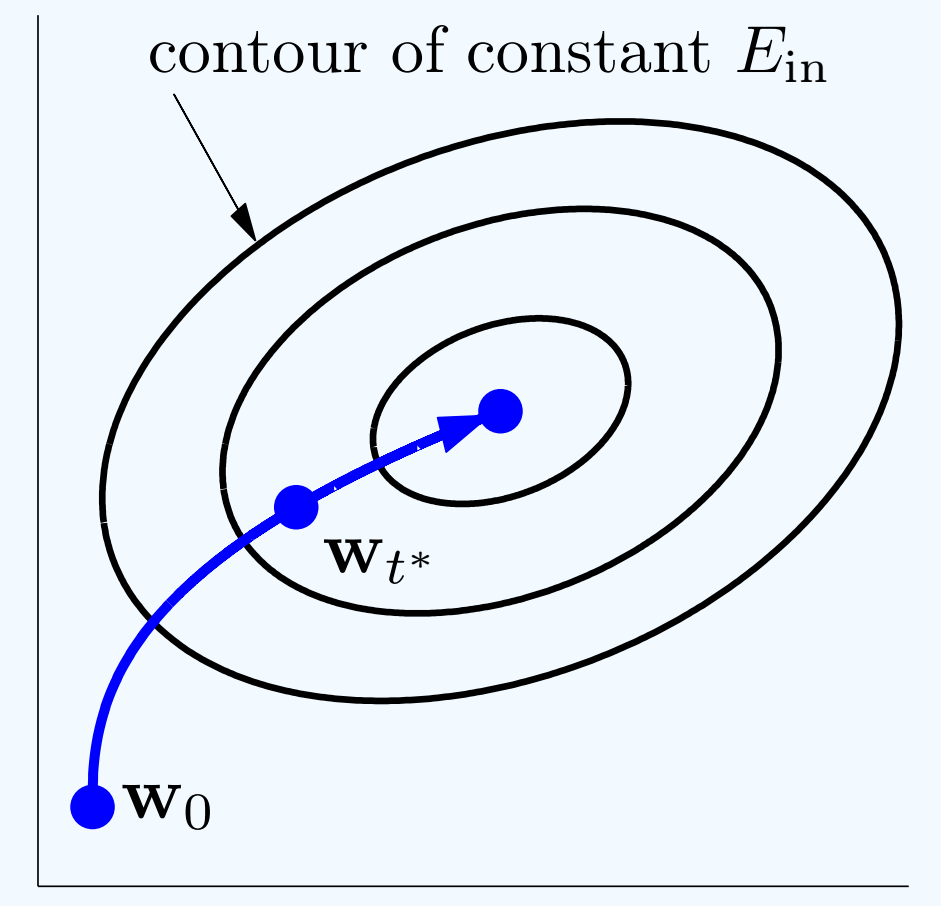
Training Your Neural Network: Regularization

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.

© Jiaming Mao

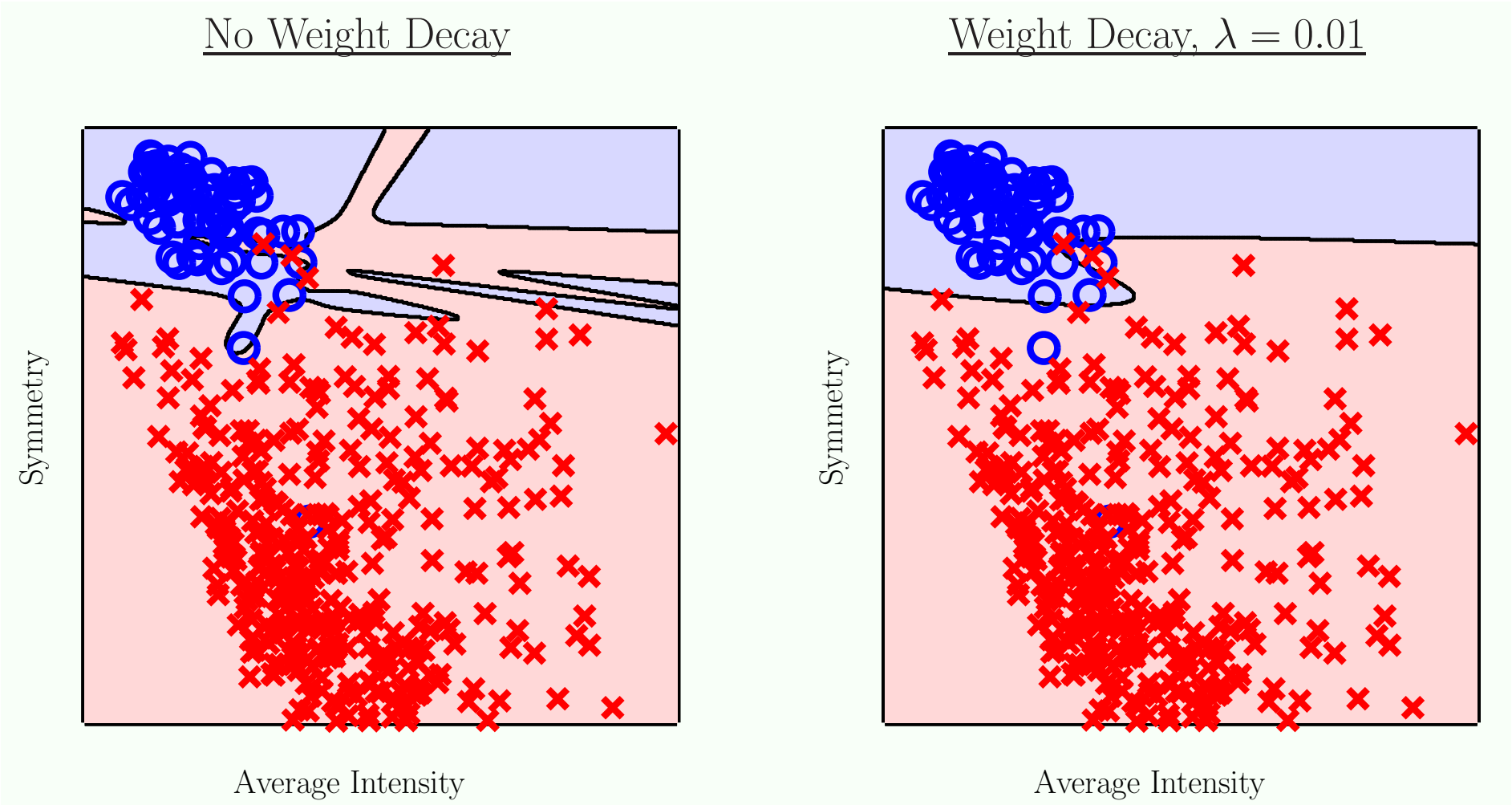
Training Your Neural Network: Regularization



Early Stopping

© Jiaming Mao

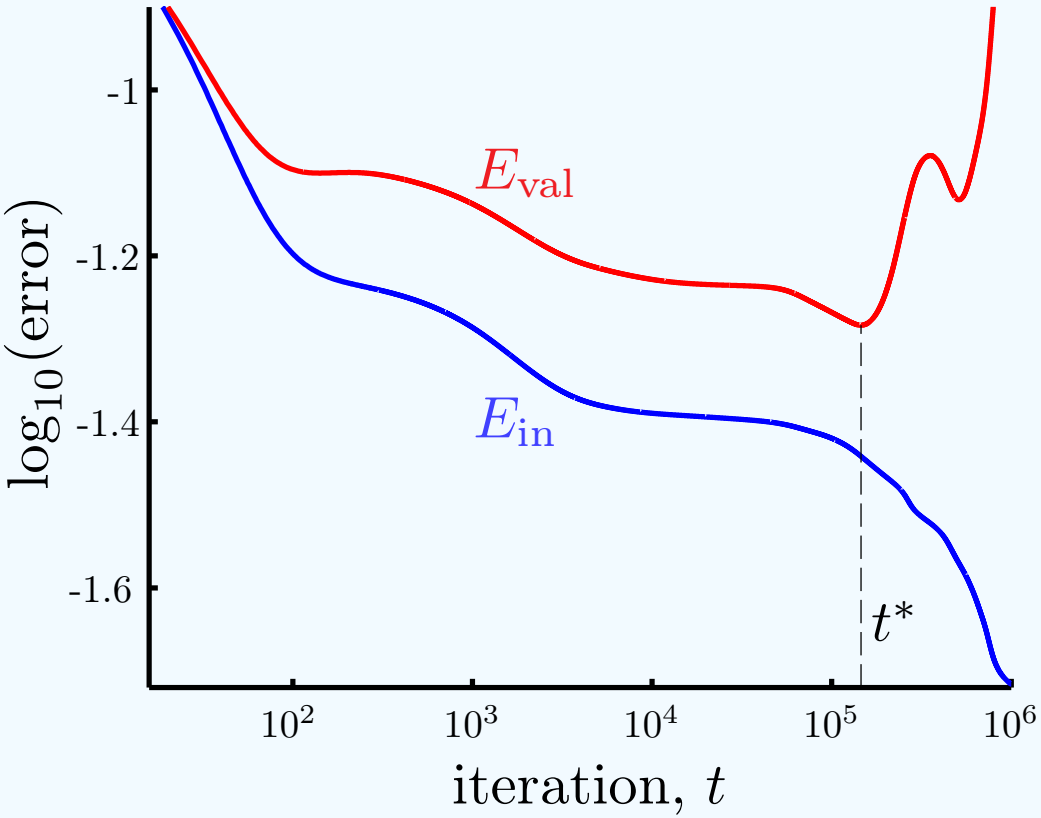
Training Your Neural Network: Regularization



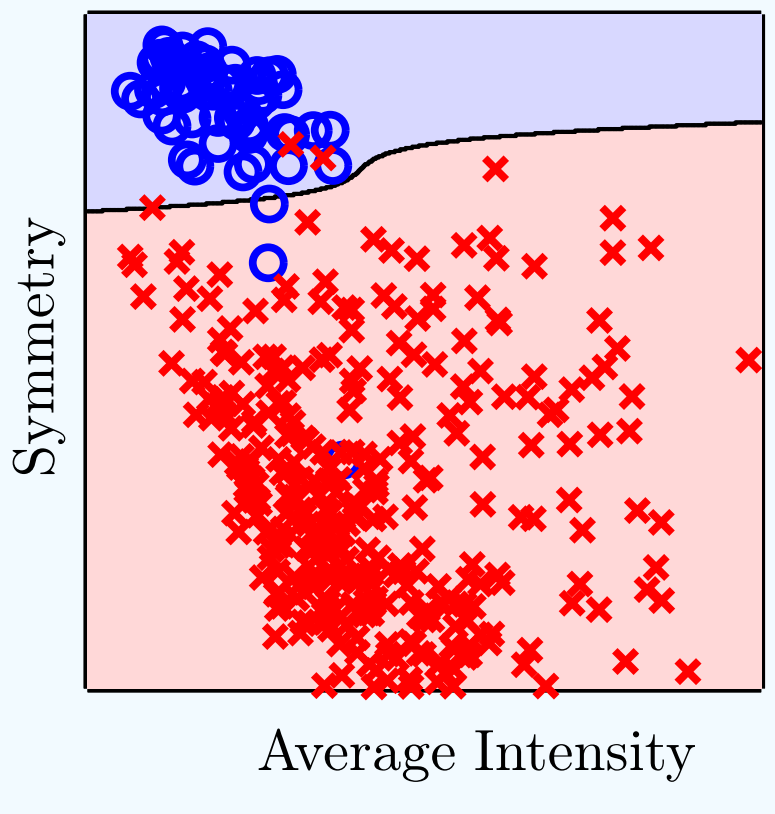
Weight Decay

© Jiaming Mao

Training Your Neural Network: Regularization



(a) Training dynamics



(b) Final hypothesis

Early Stopping

© Jiaming Mao

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 it better to have a reasonably large number of units and train them with regularization.

© Jiaming Mao

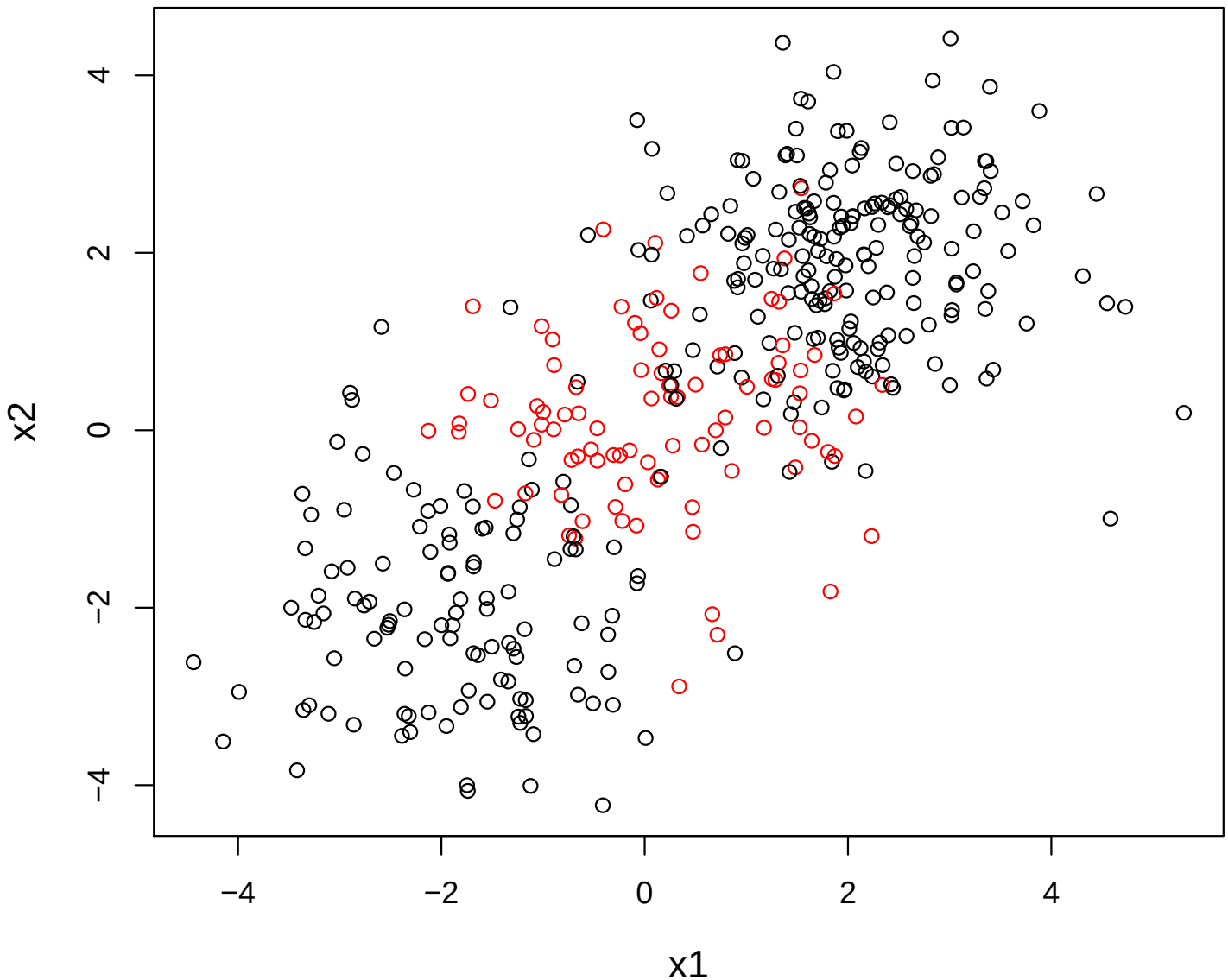
Simulation

```r
# 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,]
```

© Jiaming Mao

Simulation



© Jiaming Mao

Simulation

```r
# 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))
```

```r
fit$bestTune
```

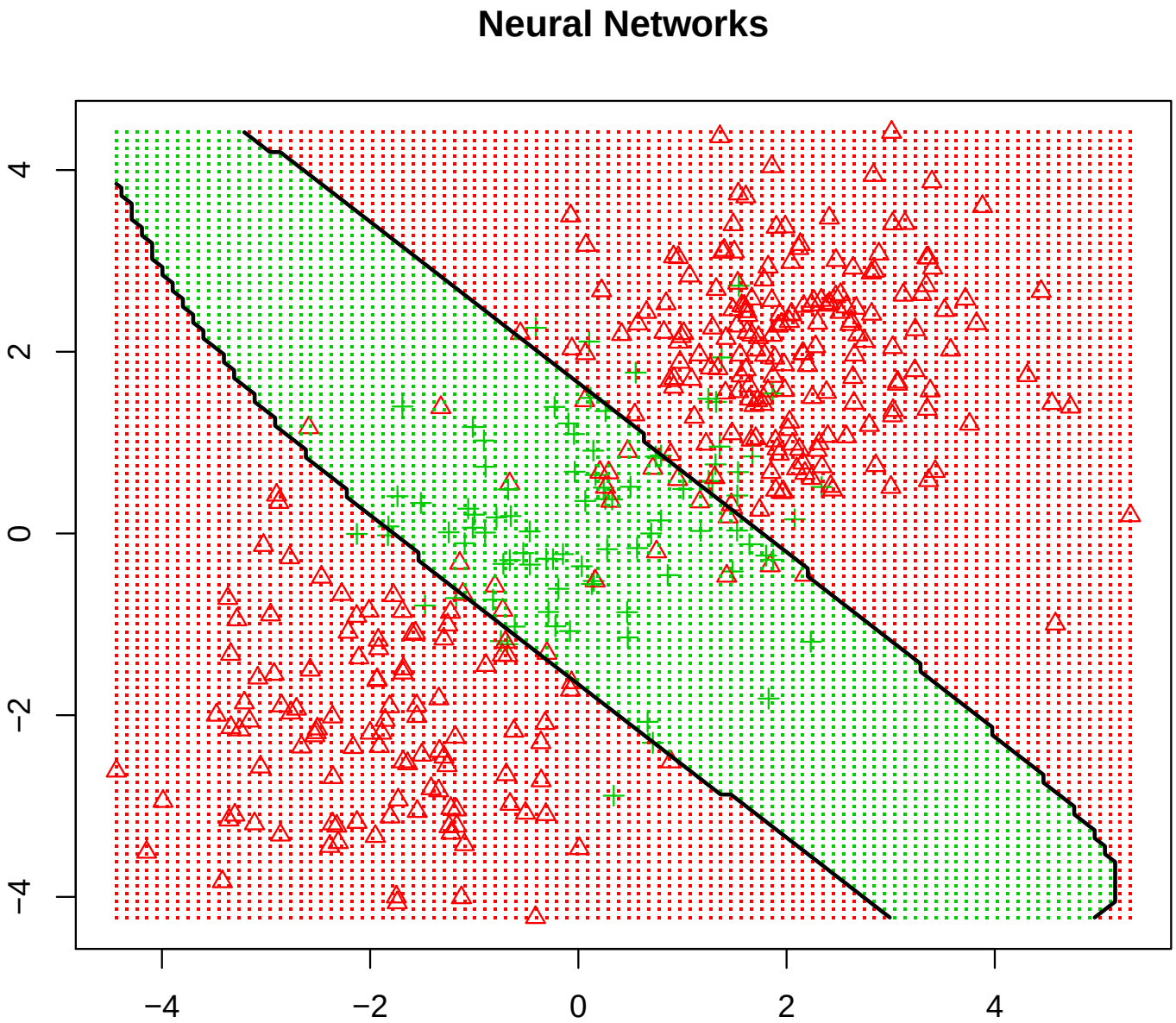
```
##   size decay
## 3   10   0.1
```

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

```
## [1] 0.1033333
```

© Jiaming Mao

Simulation



© Jiaming Mao

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.

© Jiaming Mao

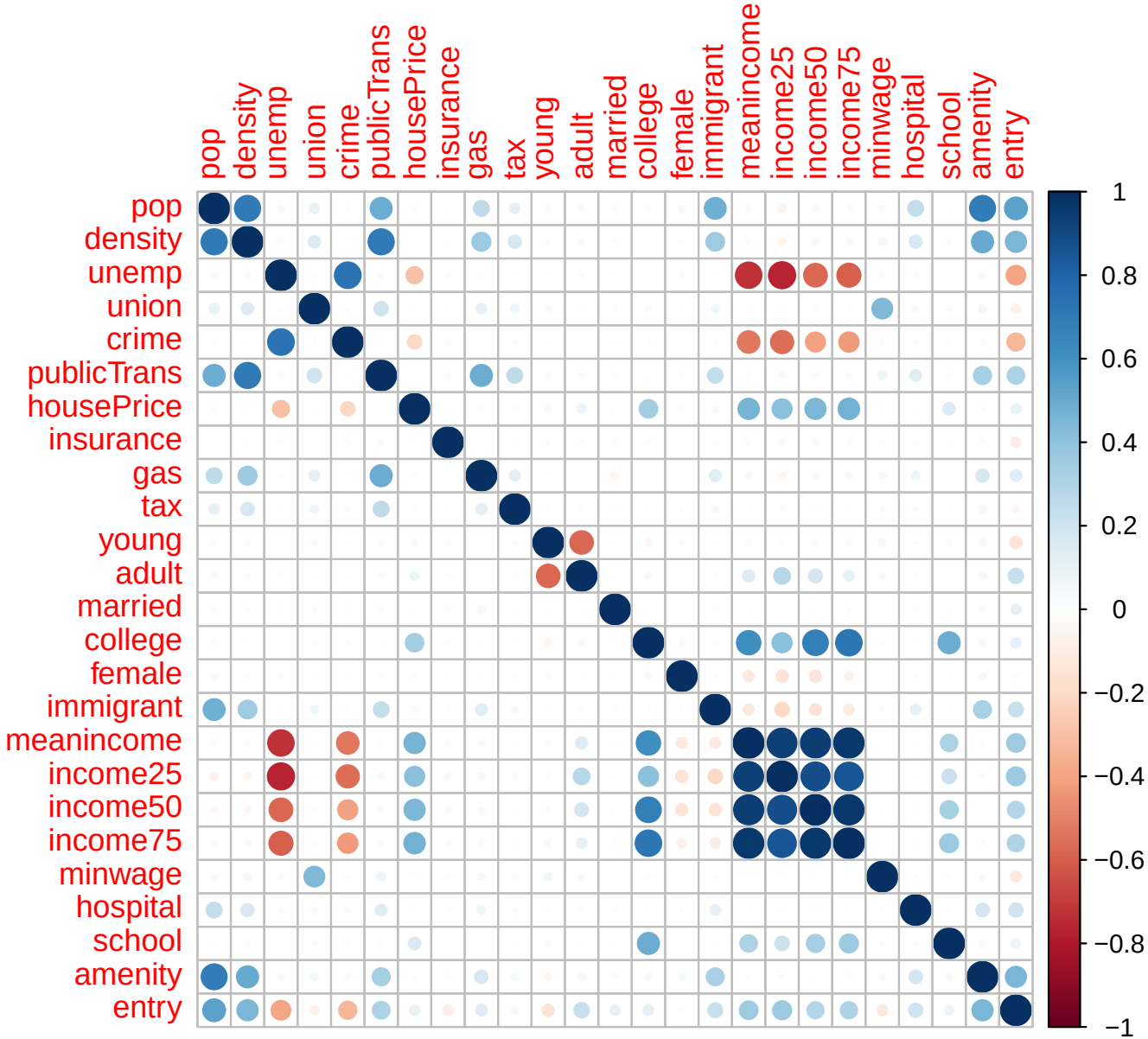
Supermarket Entry

```
# 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,]
```

© Jiaming Mao

Supermarket Entry



© Jiaming Mao

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
```

© Jiaming Mao

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
```

© Jiaming Mao

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
```

© Jiaming Mao

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
```

© Jiaming Mao

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
```

© Jiaming Mao

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
```

© Jiaming Mao

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
```

© Jiaming Mao

Supermarket Entry

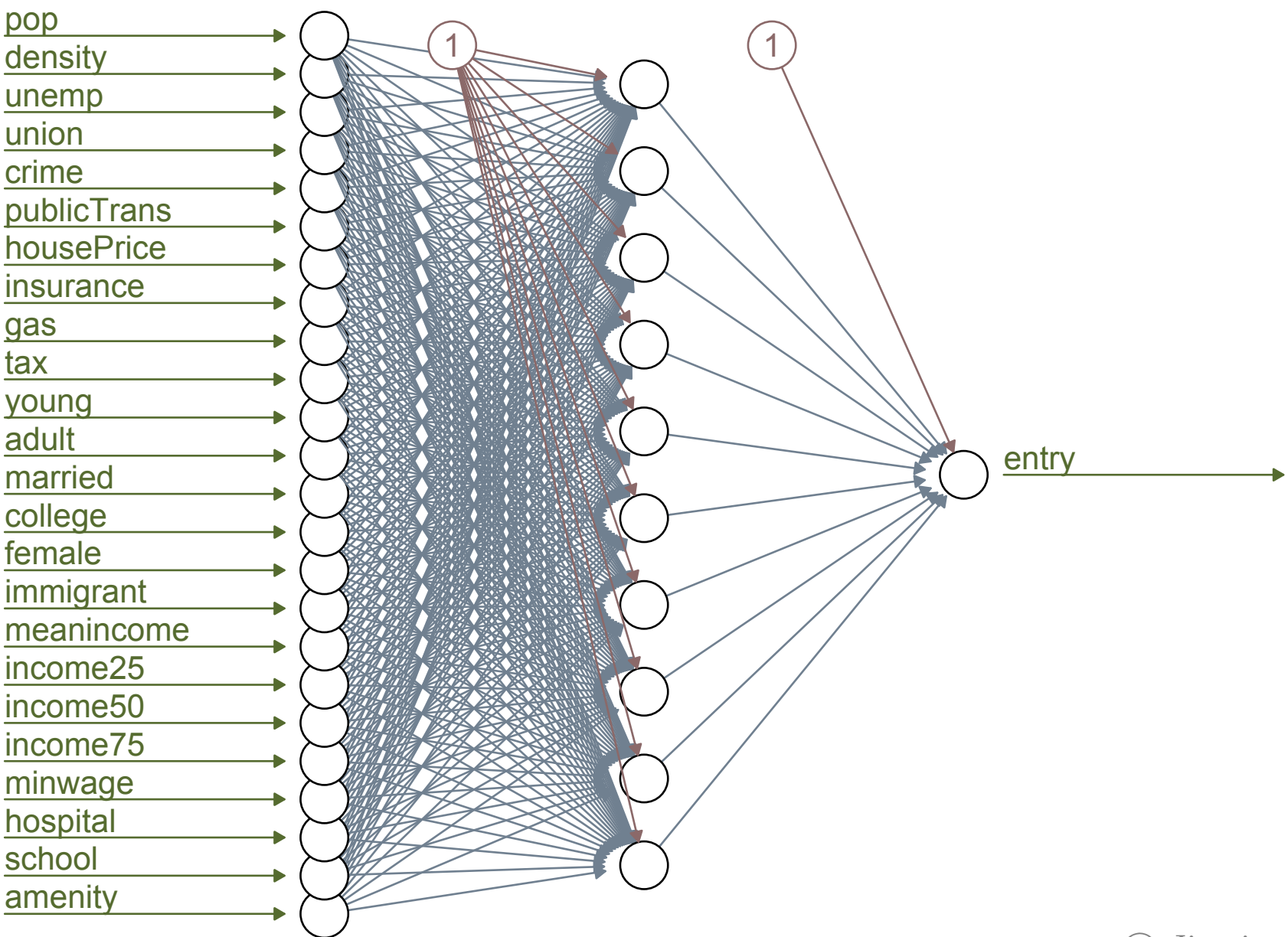
```
###################
# Neural Network #
###################
# Fit a single-hidden-layer network
require(nnet)
fit = nnet(entry ~.,data_train,maxit=10000,
           size=10,
           decay=0.1) # parameters selected by cv
```

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

## [1] 0.04394904
```

© Jiaming Mao

Supermarket Entry



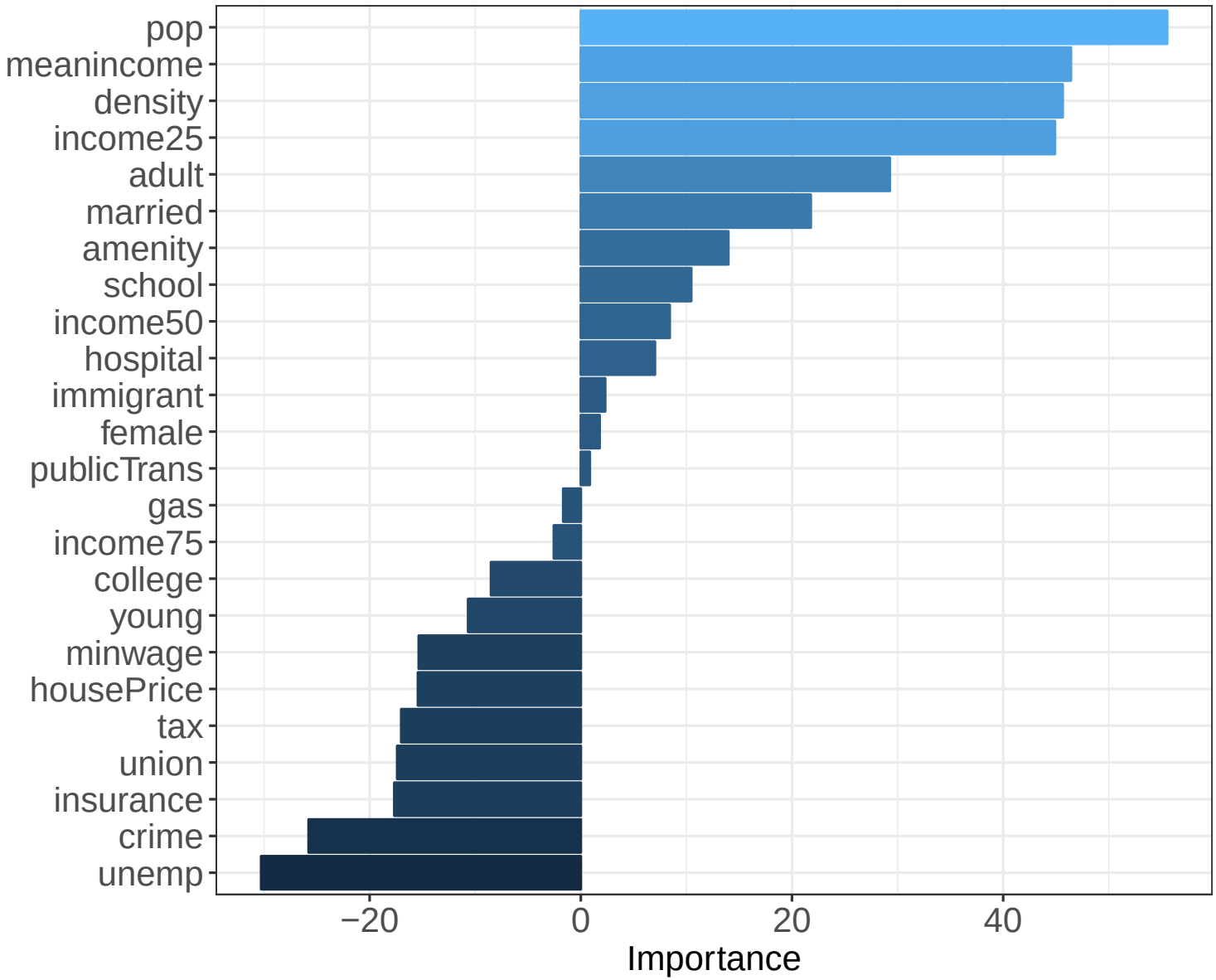
© Jiaming Mao

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.

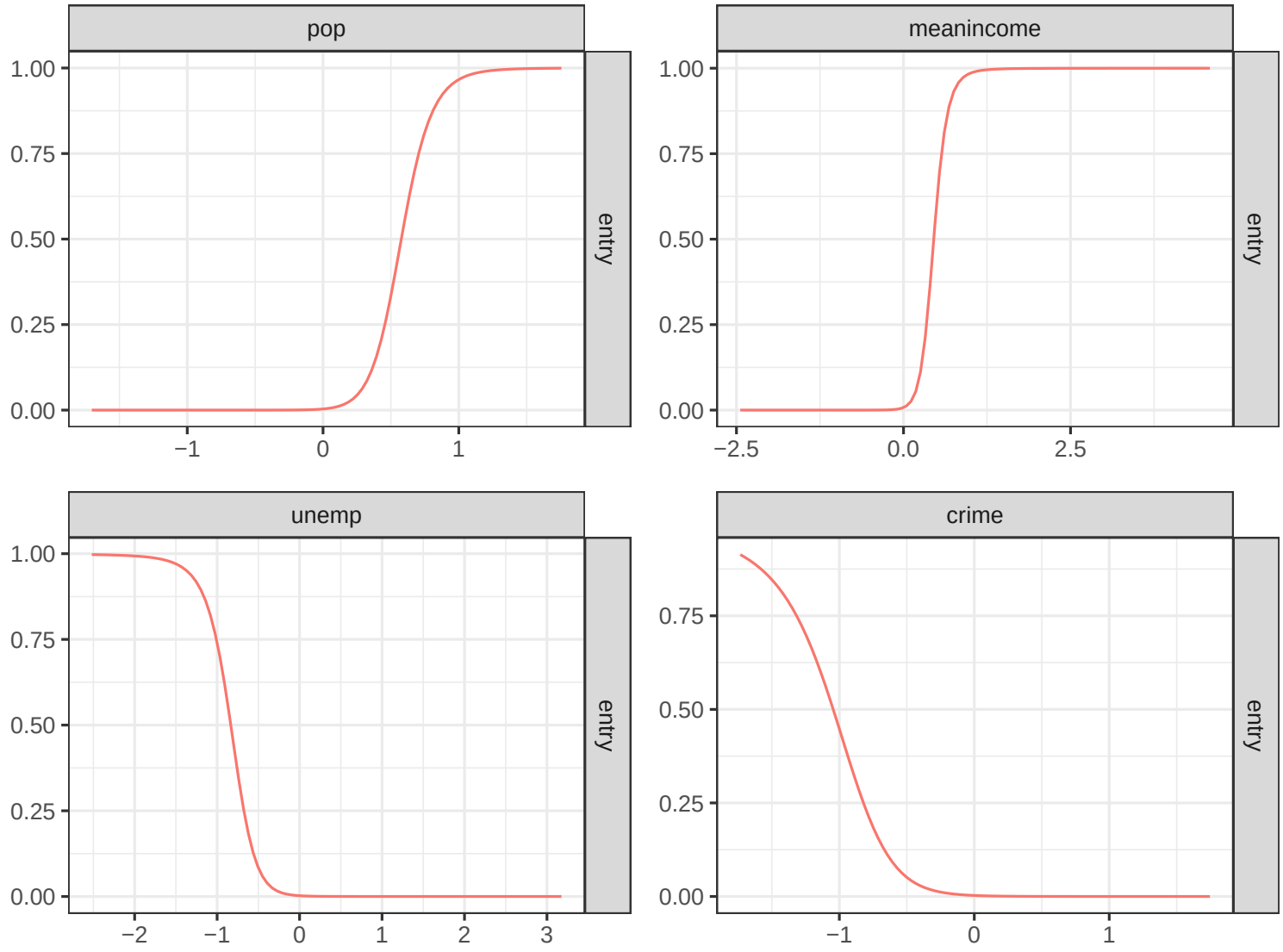
© Jiaming Mao

Supermarket Entry



© Jiaming Mao

Supermarket Entry



Partial dependence (other variables held at median)

© Jiaming Mao

The Neural Network: Discussion

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

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

, 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.

© Jiaming Mao

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]:

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

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

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

© Jiaming Mao

The Neural Network: Discussion

Difference between (3) and (4):

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

The $\theta\left(\gamma_m' x\right)$'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.

[9] As in boosting

© Jiaming Mao

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.

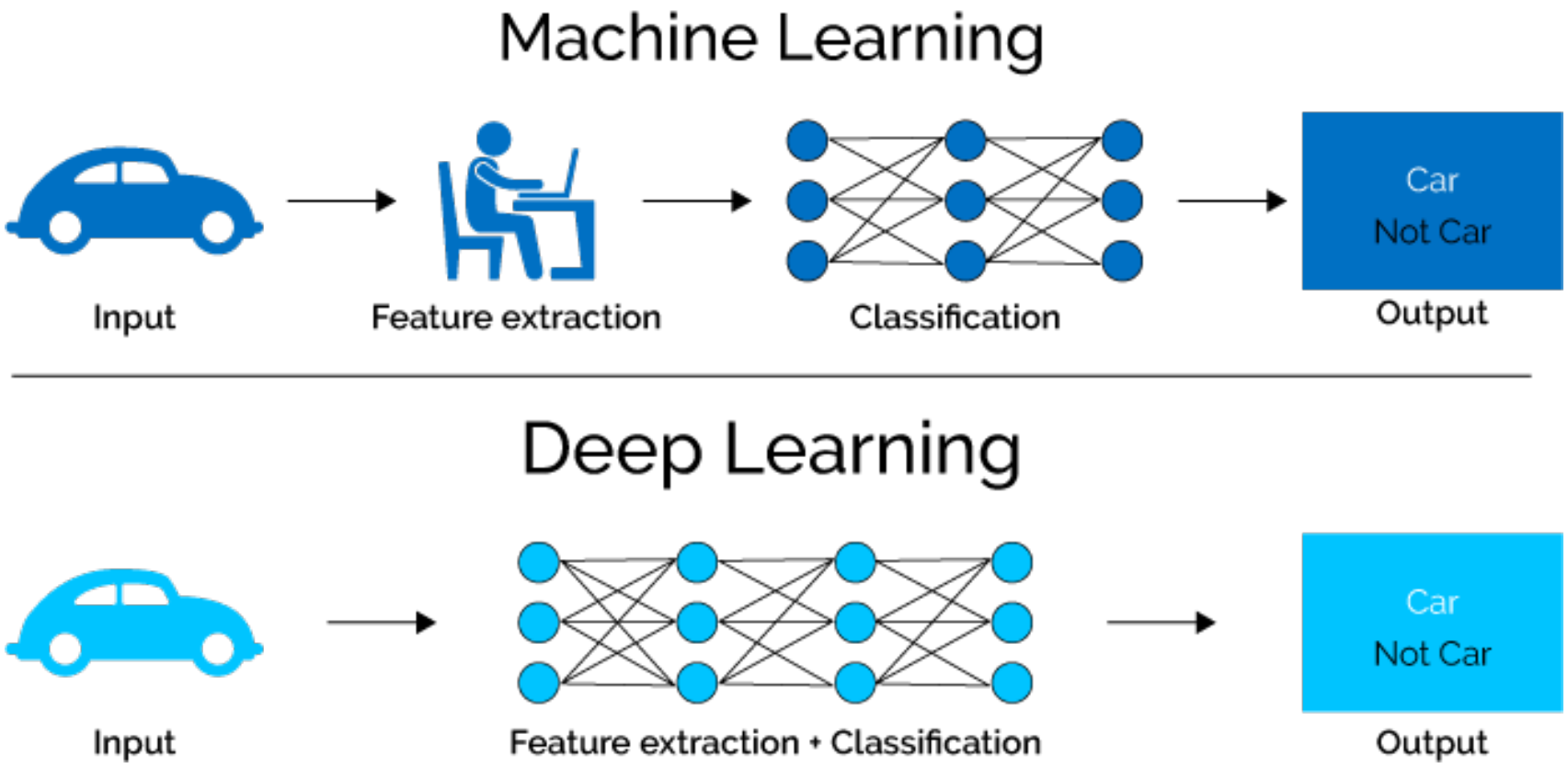
© Jiaming Mao

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.

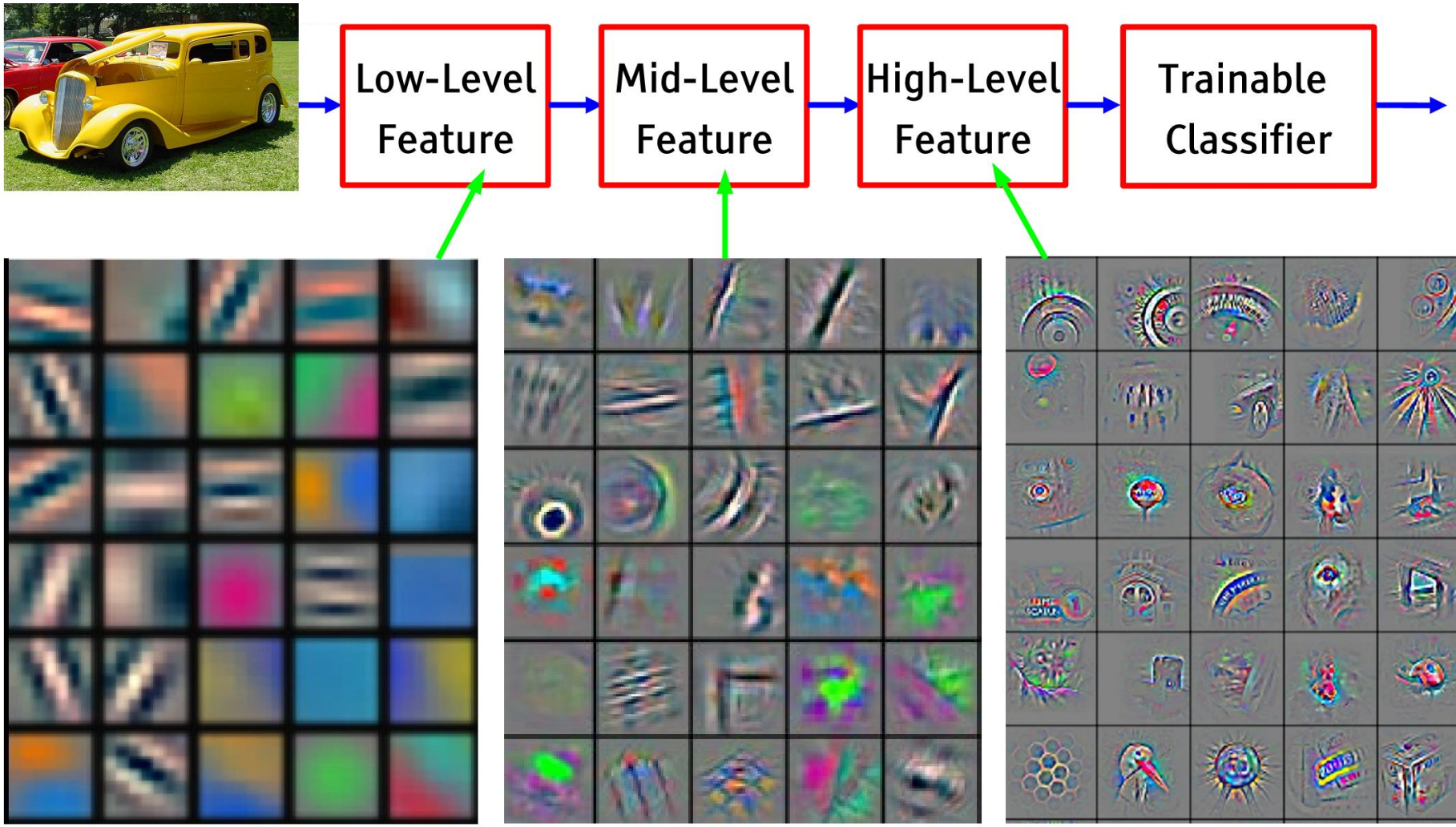
© Jiaming Mao

Deep Learning



© Jiaming Mao

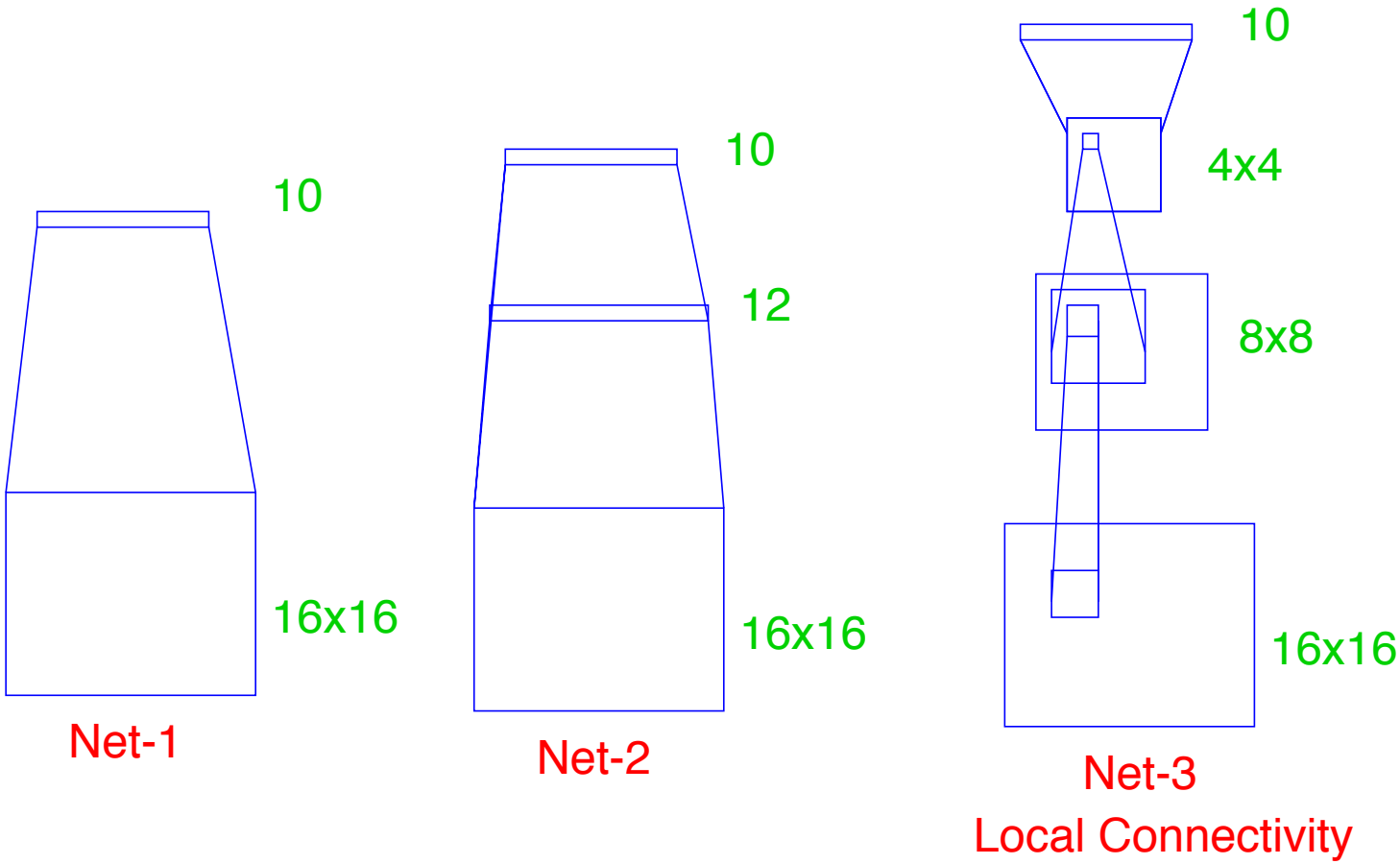
Deep Learning



© Jiaming Mao

Deep Learning

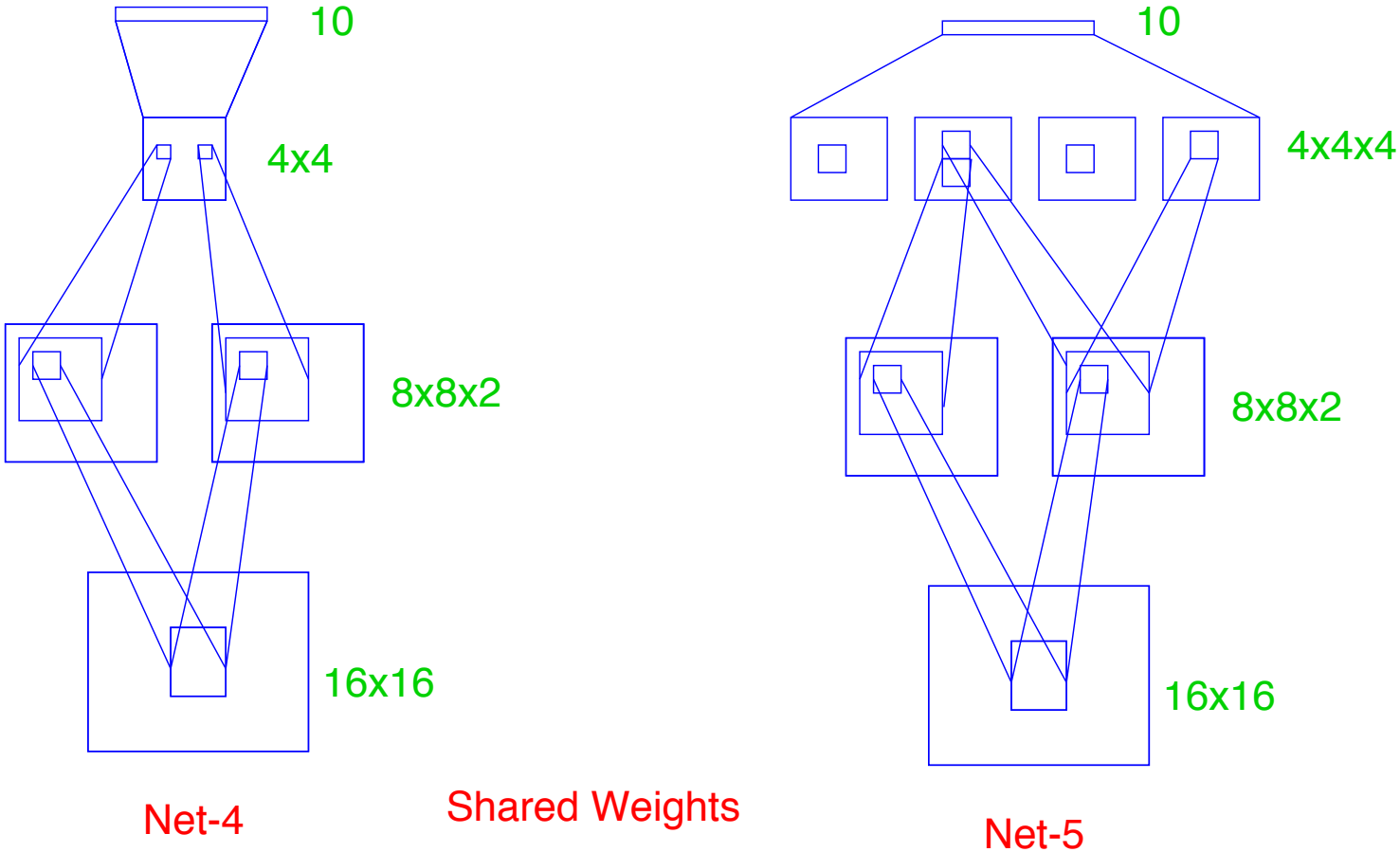
Deep networks are highly flexible and allow a wide variety of architectures:



Net-1: no hidden layer; Net-2: one hidden layer
Net-3: two hidden layers locally connected

© Jiaming Mao

Deep Learning

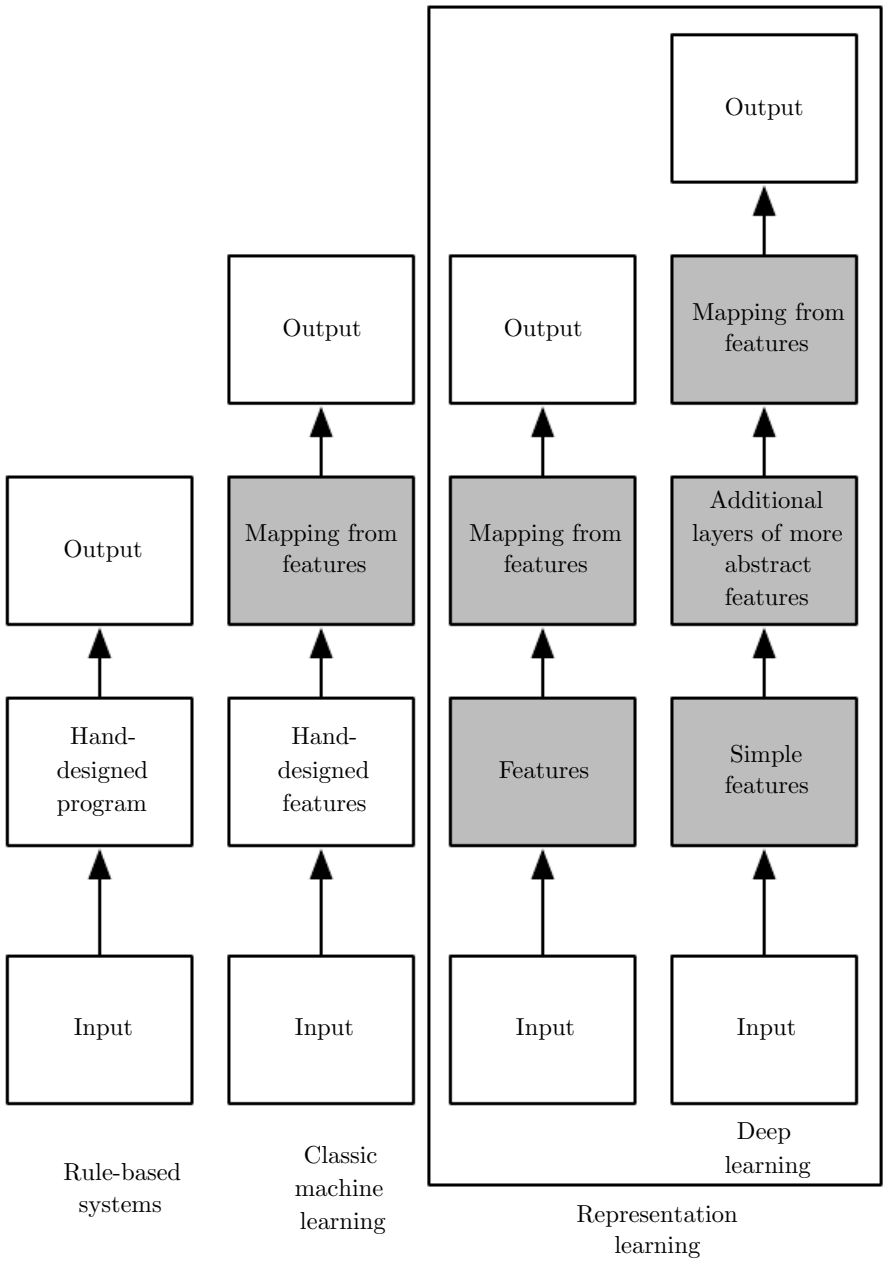


Net-4: two hidden layers, locally connected with weight sharing

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

© Jiaming Mao

Deep Learning



© Jiaming Mao

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. Friedmand. 2008. *The Elements of Statistical Learning* (2^{nd} ed.). Springer.

© Jiaming Mao