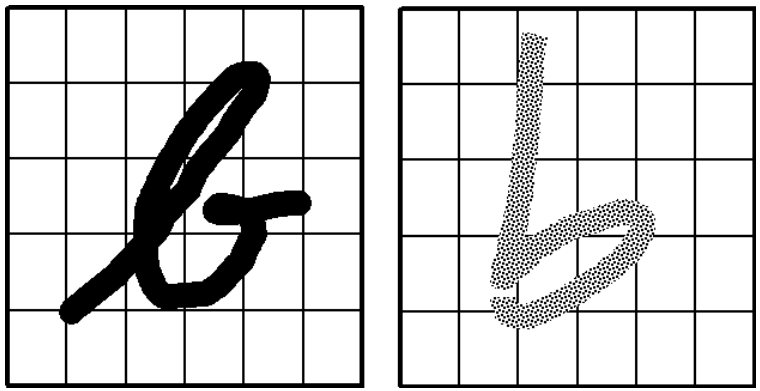
Adaptation and Empirical Models

Ravi Kothari, Ph.D.

"Let's see what is out there..." – Star Trek

Learning from Examples

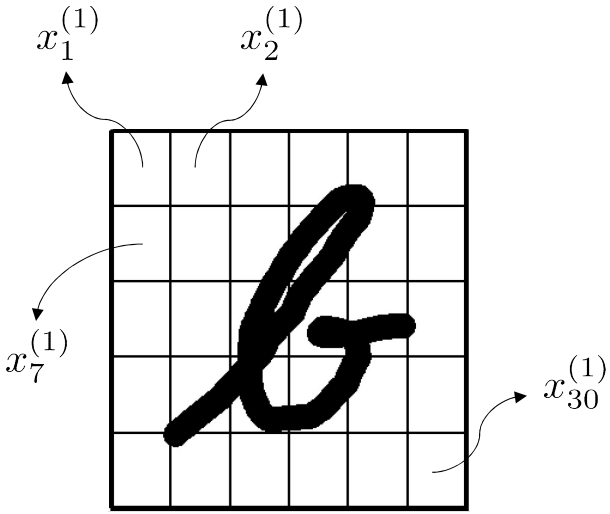
- Often, enough is not known (or it is too cumbersome) to construct first-principles based models
- Try designing an optical character recognition system to recognize characters written by different people, with different instruments, on different types of paper, with a variable amount of soiling/smudging etc.



- How about a system for recognizing faces or for a driverless car?
- Empirical model construction, in which we construct models based on training examples, become very attractive

Training Examples

- Training examples are (input, desired output) i.e. $\{(x^{(l)}, y^{(l)})\}$ pairs
  - Components of the inputs, $x_i^{(l)}$, are called features
  - A feature can be binary, numerical, categorical, ordered (and noisy!)
  - The desired output can be a real number or belong to one of a few possibilities. In the latter case, it is also called the class label. For example, if we were only dealing with lower case alphabets in the OCR case, the desired output will be 1 of the 26 possibilities (classes)
- There will typically be many many thousands of training examples – in the OCR case, representing different characters written by different people



A character from John's writing

features

$$x^{(l)} = [x_1^{(l)}\ x_2^{(l)}\ \dots\ x_n^{(l)}]$$

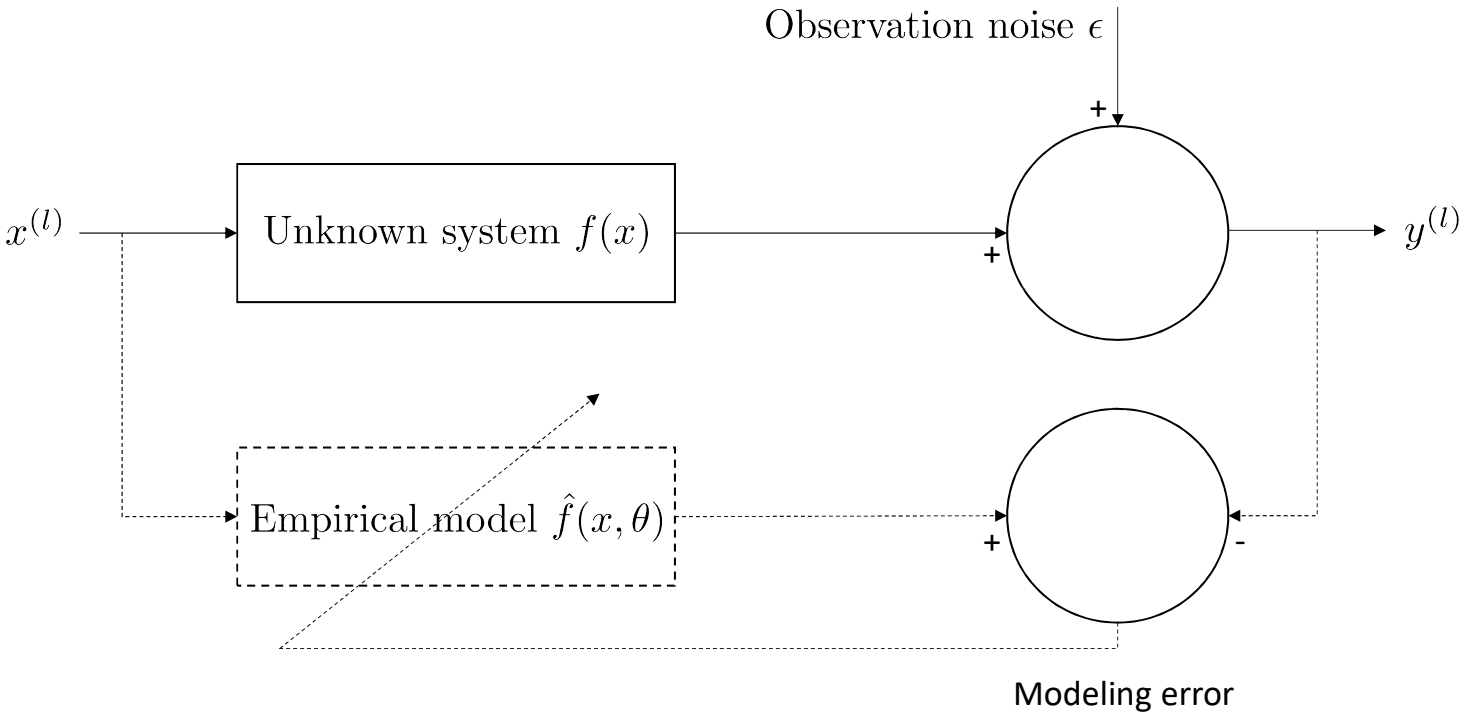
$$x_i^{(l)} \in \{0, 1\}$$

$$y^{(l)} \in \{1, 2, \dots, 26\}$$

A character from John's writing	$x^{(1)}$	1	0	...	0	1	1 $y^{(1)}$
A character from Rita's writing	$x^{(2)}$	0	0	...	1	0	2 $y^{(2)}$
				...			
Another character from Rita's writing	$x^{(N)}$	0	1	...	0	1	1 $y^{(N)}$
				Input			Desired output

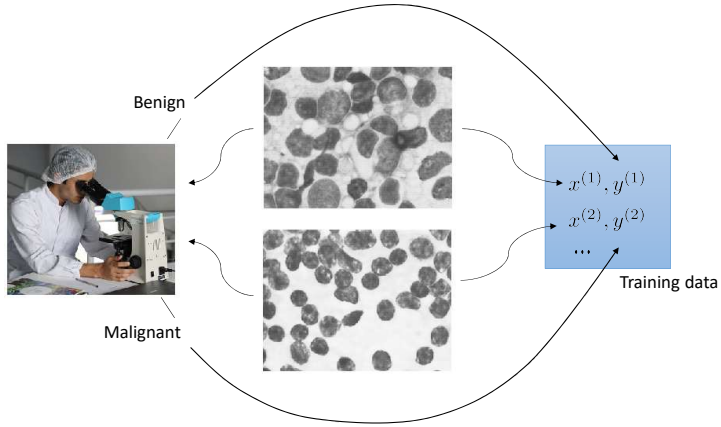
Training data $\rightarrow \{(x^{(l)}, y^{(l)})\}$

Empirical Learning – A Typical Setting

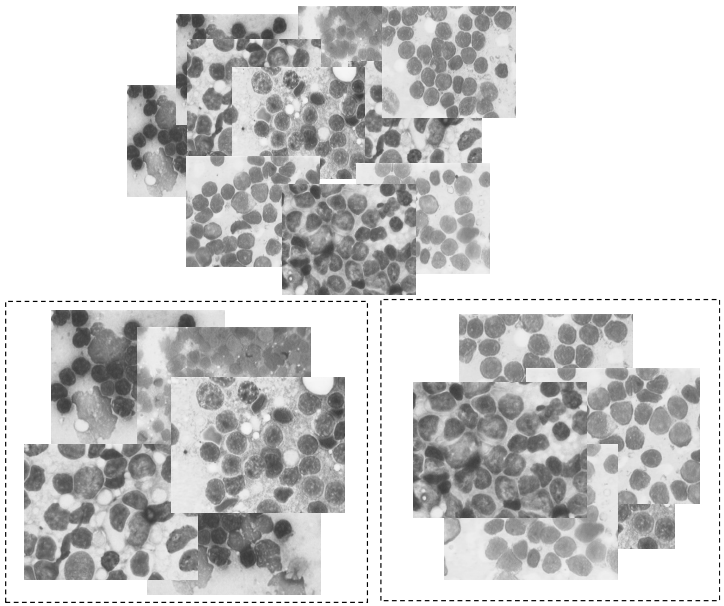


(Un)Supervised Learning

- Supervised learning: A class label is available



- Unsupervised learning: A class label is not available



Supervised Learning – A Formal Definition

- Based on N (possibly noisy) observations $\mathcal{X} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$ of the input and output of a fixed though unknown system $f(x)$, construct an estimator $\hat{f}(x;\theta)$ so as to minimize,

$$E\left[\left(L(f(x), \hat{f}(x;\theta))\right)\right] \tag{1}$$

- Note that $f(x)$ is not known. What we know is $y(x) = f(x) + \epsilon$ though our intent is to estimate $f(x)$

Why $E[\cdot]$

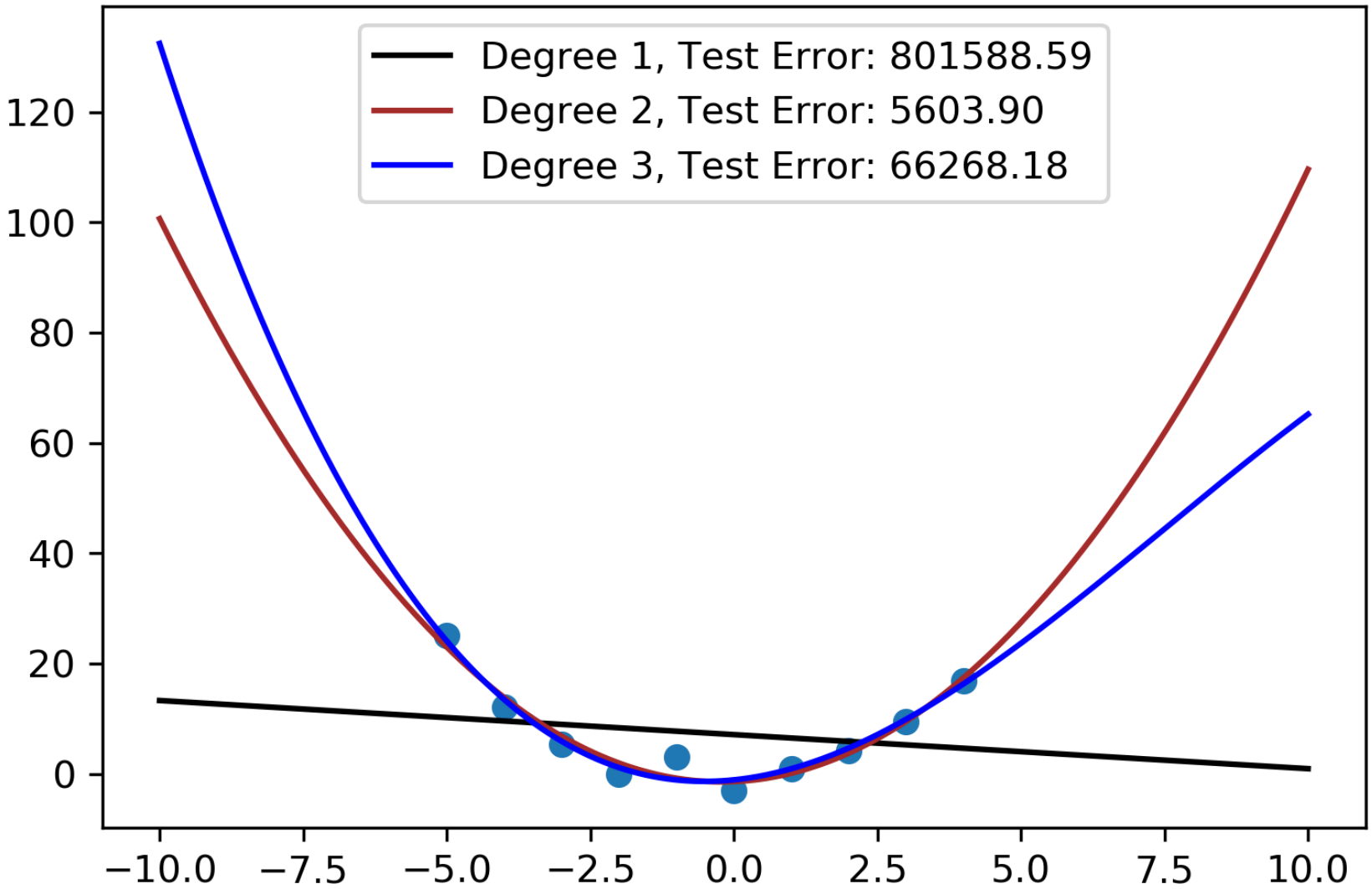
- Our goal is not simply to learn the given data. We would like our model to generalize to unseen data once we have trained it
  - If I sit alongside you and show you how to drive on 1000 trips, you should be able to drive by yourself thereafter i.e. you can generalize. If I show you 1000 handwritten `a`'s, you should be able to recognize a thereafter
- e.g. Given 10 (possibly noisy) points (which come from an unknown system), I can always fit a polynomial of degree 9 so that it has 0 error on the training data (the given 10 points). But such a model will also fit the noise in the given data. Recall we want to estimate $f(x)$

Let's Implement This and See What Happens

```
1  import random
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5  # Generate training data
6  np.random.seed(12345678)
7  dom = 5
8
9  X_train = np.array(range(-dom,+dom))
10 Y_train = X_train**2 + 2 * (np.random.randn(*X_train.shape) - 0.5)
11
12 X_test = np.linspace(-2*dom, +2*dom, 500).reshape(-1,1)
```

```python
# Fit simple polynomials of different degrees. Plot fit and compute test error
plt.figure()
plt.scatter(X_train,Y_train)

theta = np.polyfit(X_train.flatten(), Y_train.flatten(), 1)
x = sum((np.polyval(theta, X_test) - X_test ** 2) ** 2)
a = "Degree {}, Test Error: {:.2f}".format(1, x[0])
plt.plot(X_test, np.polyval(theta, X_test), color="black", label=a)

theta = np.polyfit(X_train.flatten(), Y_train.flatten(), 2)
x = sum((np.polyval(theta, X_test) - X_test ** 2) ** 2)
a = "Degree {}, Test Error: {:.2f}".format(2, x[0])
plt.plot(X_test, np.polyval(theta, X_test), color="brown", label=a)

theta = np.polyfit(X_train.flatten(), Y_train.flatten(), 3)
x = sum((np.polyval(theta, X_test) - X_test ** 2) ** 2)
a = "Degree {}, Test Error: {:.2f}".format(3, x[0])
plt.plot(X_test, np.polyval(theta, X_test), color="blue", label=a)

plt.legend()

plt.savefig("polyfit.png", dpi=300, bbox_inches='tight')
plt.show()
```

The Generalization Error



The Generalization Error

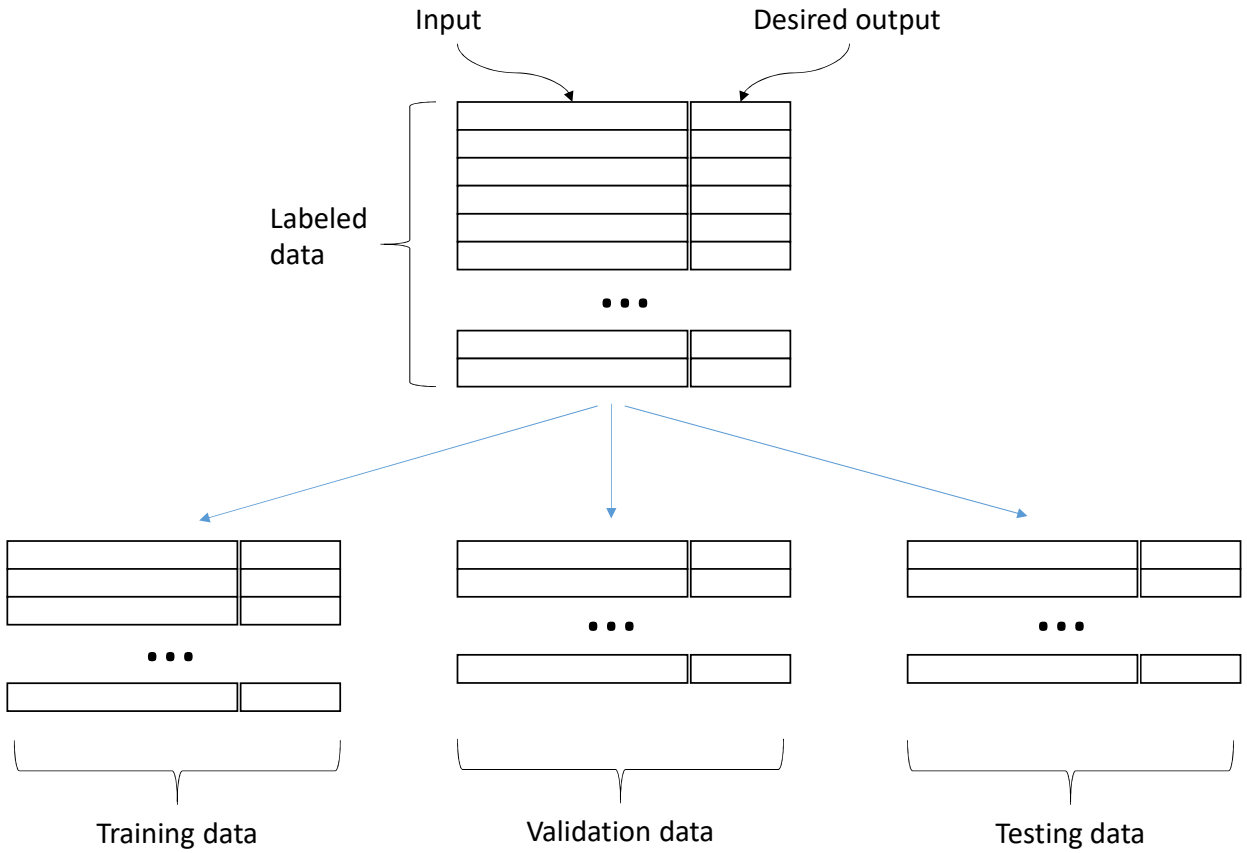
- The training error,

$$J = \frac{1}{N}\sum_{l=1}^{N} J^{(l)} = \frac{1}{N}\sum_{l=1}^{N} L\left(y^{(l)}, \hat{f}(x^{(l)}; \theta)\right) \tag{2}$$

  where, $L(\cdot)$ is a loss function. J is an optimistically biased estimate of the error we expect when using this trained network on unseen data
- We need a better estimate of the generalization performance
- There are theoretical characterizations of the prediction error - in the average case (using the Bias-Variance Decomposition or the Minimum Description Length) or the worst case (using the Vapnik-Chevronenkis (VC) Dimension)

Empirically Estimating the Generalization Performance

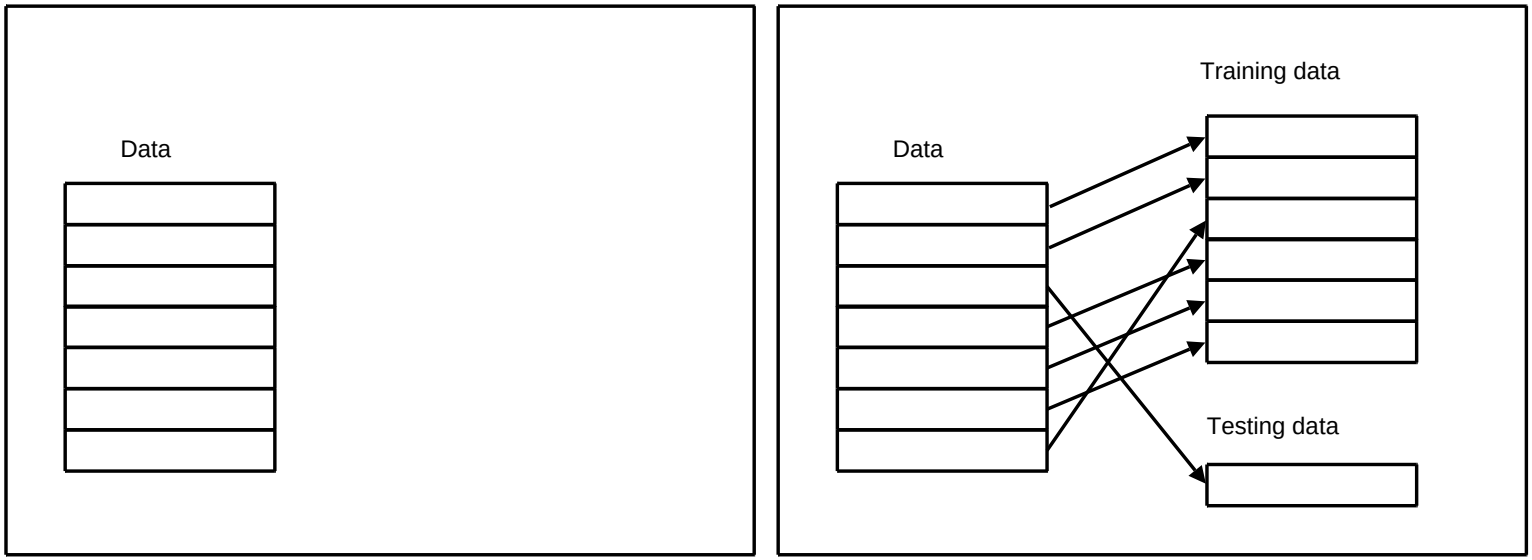
- One possibility – divide the available data into,
  - Training data – a large fraction used to construct the model
  - Validation data – a smaller fraction used for model validation
  - Test data – used to assess the performance of the model on unseen data *in lieu* of the loss function above



More Accurately Estimating the Prediction Error

- A drawback of the above approach is its dependence on the specific partitioning
- There are many methods of estimating the prediction error which do not have this drawback. A popular sampling without replacement method is the k-fold cross validation estimate. A popular *sampling with replacement* method is the bootstrap estimate

k-fold Cross Validation Estimate

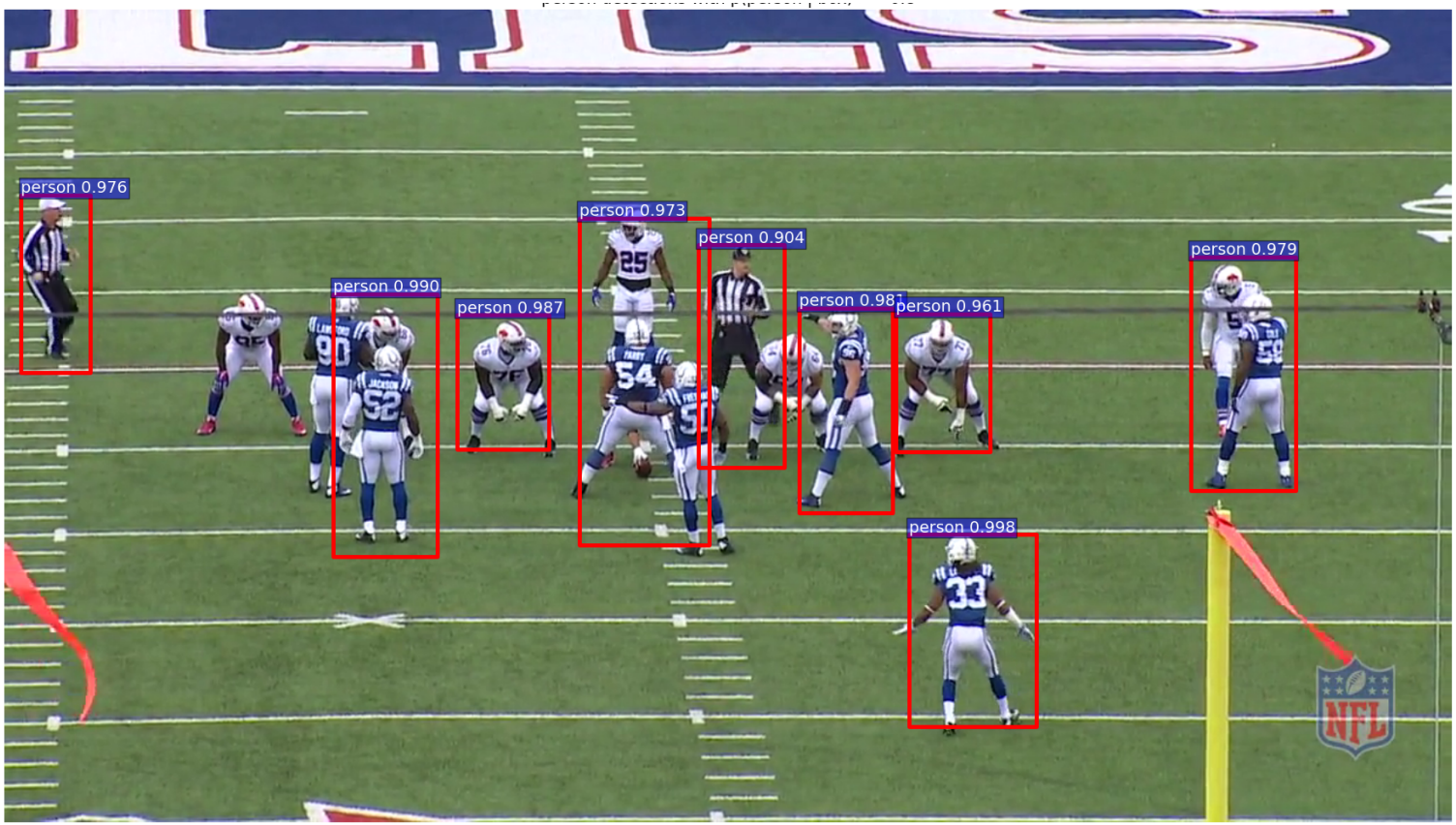


- Partition the data in to k disjoint equal sized blocks. Use $(k-1)$ partitions for training and the remaining partition for testing. Repeat k times ensuring that each of the k partition is used for testing once
- Prediction error estimate is the average of the k-error estimates
- When $k = N$, we get the leave-one-out estimate
- Time consuming. Likely to overestimate the prediction error (why?)

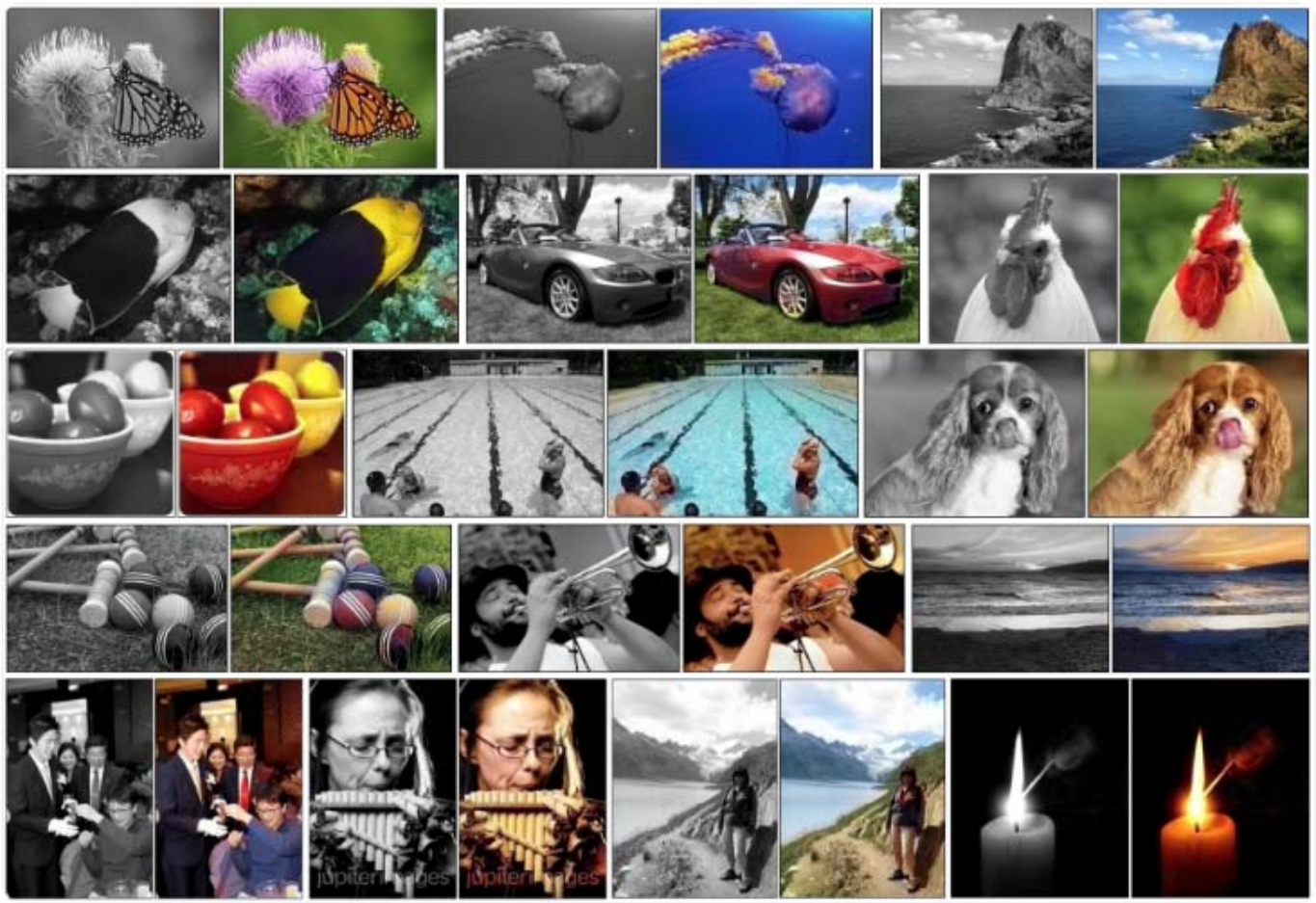
Some Learning Scenarios

- Determining (credit-card or other types of) fraud
- Speaker identification, speech recognition
- Drug design and bio-informatics
- Driverless cars
- Problem determination
- Intrusion detection, surveillance

Object Detection



Automatic Colorization



From https://arxiv.org/pdf/1603.08511.pdf

Automatic Caption Generation

"man in black shirt is playing guitar."

"construction worker in orange safety vest is working on road."

"two young girls are playing with lego toy."

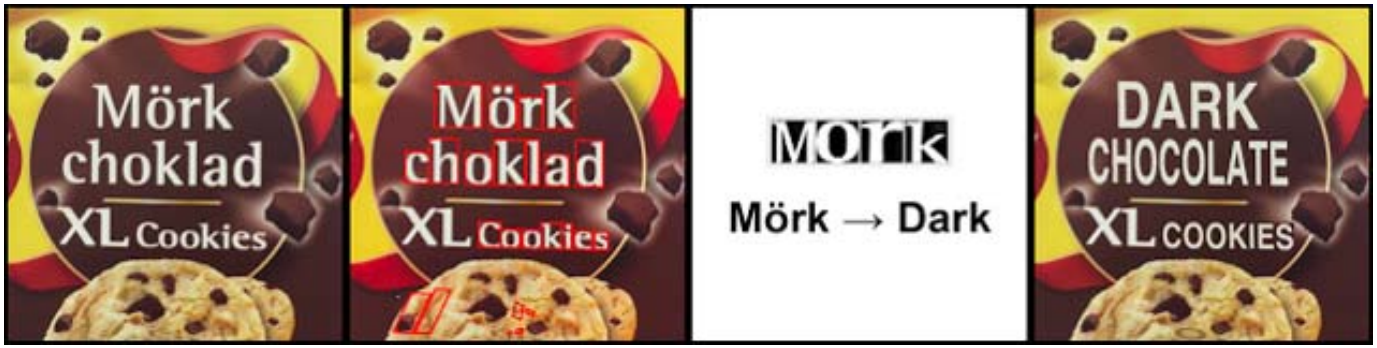
"girl in pink dress is jumping in air."

"black and white dog jumps over bar."

"young girl in pink shirt is swinging on swing."

From `http://cs.stanford.edu/people/karpathy/cvpr2015.pdf`

Language Translation



From https://research.googleblog.com/2015/07/how-google-translate-squeezes-deep.html

Additional Reading

- R. O. Duda, and P. E. Hart, *Pattern Classification and Scene Analysis*, John Wiley, 1973.
- C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2010.
- I. A. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, MIT Press, 2016.

Loss Functions

Ravi Kothari, Ph.D.

"If you cannot measure it, you cannot improve it" – Lord Kelvin

Loss Functions for Classification

Indicator Loss

- $y(x) \in \{\omega_1, \omega_2, \ldots, \omega_c\}$,

$$L(y(x), \hat{f}(x;\theta)) = \frac{1}{N} \sum_{l=1}^{N} \begin{cases} 0, \text{ if } & f(x^{(l)}) = \hat{f}(x^{(l)};\theta) \\ 1, \text{ if } & f(x^{(l)}) \neq \hat{f}(x^{(l)};\theta) \end{cases} \tag{1}$$

- Can be asymmetric
- Not differentiable

Hinge Loss

- For two classes, i.e., $y(x) \in \{\omega_1, \omega_2\}$ with classes coded as ± 1. $\hat{f}(x^{(l)}; \theta)$ is the raw output

$$L(y(x), \hat{f}(x; \theta)) = \frac{1}{N} \sum_{l=1}^{N} \max \left[0, 1 - y(x^{(l)}) \hat{f}(x^{(l)}; \theta)\right] \tag{2}$$

- Often used for maximum margin classification
  - When $y(x^{(l)})$ and $\hat{f}(x^{(l)}; \theta)$ have the same sign and $|\hat{f}(x^{(l)}; \theta)| \geq 1$, the loss is 0
  - When they have opposite signs the loss increases linearly with $\hat{f}(x^{(l)}; \theta)$
  - Finally, even if they have the same sign but $|\hat{f}(x^{(l)}; \theta)| < 1$ (i.e. not enough margin), the loss increases linearly

Cross-Entropy Loss

- Cross-Entropy Loss: For two classes, i.e., $y(x) \in \{\omega_1, \omega_2\}$ with classes coded as $0, 1$. $\hat{f}(x^{(l)}; \theta)$ is the output of the classifier

$$L(y(x), \hat{f}(x;\theta)) = -\frac{1}{N}\sum_{l=1}^{N}\Big[y(x^{(l)})\log \hat{f}(x^{(l)};\theta) + (1 - y(x^{(l)}))\log(1 - \hat{f}(x^{(l)};\theta))\Big] \quad (3)$$

- Note that the cross entropy between two distributions p and q can be computed as,

$$H(p, q) = -\sum_{x \in \mathcal{X}} p(x)\log q(x) \quad (4)$$

i.e. the message-length when a wrong distribution q is assumed while the data is actually from some other distribution p. The above equation thus follows if we also use that $P(\omega_1) = 1 - P(\omega_2)$

Let $p = [0, 1, 0, 0, 1, 0, 0, 1, 1, 1]$ and $q = [0.2, 0.8, 0.4, 0.6, 0.4, 0.3, 0.7, 0.7, 0.1, 0.8]$. Let us use a small proram to calculate this

```
A,B = [],[]
for i in range(len(p)):
        a = -(p[i] * log(q[i]) + (1 - p[i]) * log(1-q[i]))
        A.append(a)
        B.append((p[i]-q[i])**2)

# Mean Cross Entropy and Mean Squared Error
aa = ["%.2f"%item for item in A]
print(aa)
print("Mean Cross Entropy is {:.2f}".format(mean(A)))
aa = ["%.2f"%item for item in B]
print(aa)
print("Mean error is {:.2f}".format(mean(B)))
```

[0.22, 0.22, 0.51, 0.92, 0.92, 0.36, 1.20, 0.36, 2.30, 0.22]

Mean Cross Entropy is 0.72

[0.04, 0.04, 0.16, 0.36, 0.36, 0.09, 0.49, 0.09, 0.81, 0.04]

Mean error is 0.25

Mean Squared (MSE)

- MSE is defined as,

$$L(y(x), \hat{f}(x;\theta)) = \frac{1}{N}\sum_{l=1}^{N}\left[y(x^{(l)}) - \hat{f}(x^{(l)};\theta)\right]^2 \tag{5}$$

- Magnifies larger error; attenuates smaller errors
- Differentiable
- Positive errors and negative errors are equivalent

Loss Functions for Regression

Mean Squared Error (MSE)

- $y(x) \in \mathcal{R}^m$, $\hat{f}(x^{(l)}; \theta) \in \mathcal{R}^m$,

$$L(y(x), \hat{f}(x; \theta)) = \frac{1}{N} \sum_{l=1}^{N} \left[y(x^{(l)}) - \hat{f}(x^{(l)}; \theta) \right]^2 \tag{6}$$

- Magnifies larger error; attenuates smaller errors
- Differentiable
- Positive errors and negative errors are equivalent

Mean Absolute Error or $L1$ Loss

- $y(x) \in \mathcal{R}^m$, $\hat{f}(x^{(l)}; \theta) \in \mathcal{R}^m$,

$$L(y(x), \hat{f}(x; \theta)) = \frac{1}{N} \sum_{l=1}^{N} \left| y(x^{(l)}) - \hat{f}(x^{(l)}; \theta)) \right| \tag{7}$$

Huber Loss – Smooth Mean Absolute Error

- $y(x) \in \mathcal{R}^m$,

$$L(y(x), \hat{f}(x;\theta)) =$$
$$\frac{1}{N}\sum_{l=1}^{N}\begin{cases} \frac{1}{2}\left[y(x^{(l)}) - \hat{f}(x^{(l)};\theta)\right]^2 & \text{, if } \left|y(x^{(l)}) - \hat{f}(x^{(l)};\theta)\right| \leq \\ \delta\left|y(x^{(l)}) - \hat{f}(x^{(l)};\theta)\right| - \frac{1}{2}\delta^2 & \text{, otherwise} \end{cases}$$

where, δ is a hyper-parameter. This loss function is less sensitive to outliers

- Note that the loss is quadratic for small values of δ and linear for large values δ

Additional Reading

- R. O. Duda, and P. E. Hart, *Pattern Classification and Scene Analysis*, John Wiley, 1973.
- C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2010.

Numerical Optimization

Ravi Kothari, Ph.D.

"Don't worry about getting perfect; just keep getting better" – Frank Peretti

Learning

- Once a loss function is defined, learning becomes an optimization process that minimizes the loss function by adjusting the parameters (θ)
- This is the prevalent and popular definition of learning (though completely inappropriate in my view – more on it later)

Numerical Optimization

- One starts with some values of the parameters, present a training pattern, adjust the parameters to reduce the loss function, present the next training pattern and repeating until the loss function is below an acceptable threshold (or stops decreasing)
- i.e.

$$\theta(t+1) = \theta(t) + \Delta\theta(t) \tag{1}$$

such that,

$$L(t+1) < L(t) \tag{2}$$

- A popular approach to finding θ is based on gradient descent

Taylor Series

- Given $L(\theta)$, we want to expand this function about a chosen point,

$$L(\theta) = a_0 + a_1(\theta - \theta_0) + a_2(\theta - \theta_0)^2 + a_3(\theta - \theta_0)^3 + \ldots$$

- To obtain θ_0, let $\theta = \theta_0$ in the above. We get $a_0 = L(\theta_0)$
- For a_1, first take the derivative, i.e.,

$$\frac{d}{d\theta}L(\theta) = a_1 + 2a_2(\theta - \theta_0) + 3a_3(\theta - \theta_0)^2 + \ldots$$

- Now choose $\theta = \theta_0$, We get $a_1 = \left.\frac{dL(\theta)}{d\theta}\right|_{\theta=\theta_0}$
- For a_2, we take the derivative of the equation above and choose $\theta = \theta_0$

Gradient Descent

- Using Taylor Series ($\theta(t+1) = \theta(t) + \Delta\theta(t)$),

$$L(\theta(t+1)) = L(\theta(t)) + \nabla L(\theta)^T_{\theta(t)} \Delta\theta(t) + \dots \tag{3}$$

- To lower the LHS,

$$\nabla L(\theta)^T_{\theta(t)} \Delta\theta(t) < 0 \tag{4}$$

- This happens maximally when,

$$\Delta\theta(t) \propto -\nabla L(\theta)_{\theta(t)} \tag{5}$$

- i.e.,

$$\Delta\theta(t) = -\eta \nabla L(\theta)_{\theta(t)} \tag{6}$$

where, η is the step size. So,

$$\theta(t+1) = \theta(t) - \eta \nabla L(\theta)_{\theta(t)} \tag{7}$$

Example

- Say we want to find the minimum of,

$$L(\theta) = \theta_1^2 + (\theta_2 - 3)^2 \tag{8}$$

- Let the starting guess be,

$$\theta(0) = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 0.7 \end{bmatrix} \tag{9}$$

- So,

$$\nabla L(\theta) = \begin{bmatrix} \frac{\partial L(\theta)}{\partial(\theta_1)} \\ \frac{\partial L(\theta)}{\partial(\theta_2)} \end{bmatrix} = \begin{bmatrix} 2\theta_1 \\ 2\theta_2 - 6 \end{bmatrix} \tag{10}$$

- Using $\eta = 0.1$ in $\theta(t+1) = \theta(t) - \eta \nabla_{\theta(t)}$ we get,

$$\theta(1) = \begin{bmatrix} 1 \\ 0.7 \end{bmatrix} - 0.1 \begin{bmatrix} 2\theta_1 \\ 2\theta_2 - 6 \end{bmatrix} = \begin{bmatrix} 1 \\ 0.7 \end{bmatrix} - 0.1 \begin{bmatrix} 2 \\ -4.4 \end{bmatrix} = \begin{bmatrix} 0.8 \\ 1.14 \end{bmatrix} \tag{11}$$

- We see $L(\theta(1)) < L(\theta(0))$. Repeat until a stopping criteria is met

Other Optimization Methods

- Gradient descent can get trapped in a local minima (like all local search methods)
- It can be slow to converge. One can use higher order derivatives. They are expensive to compute though provide a better approximation of the error surface and can converge faster
- Quasi-Newton's method (such as Broyden–Fletcher–Goldfarb–Shanno (BFGS)) do not explicitly compute the Hessian but rather approximate it. They usually converge much faster than first-order gradient descent. We will look at some if time permits

Additional Reading

- C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2010.
- J. Nocedal, and S. J. Wright, *Numerical Optimization*, Springer, NY, 2006.

Linear Regression

... and Ridge, Lasso, and ElasticNet

Ravi Kothari, Ph.D.

"Begin at the beginning and go on till you come to the end: then stop" – Lewis Caroll, Alice in Wonderland

Linear Models

- We start with the simplest of models – those consisting of linear functions, i.e. of the form,

$$\hat{f}(x; w) = w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n = \sum_{k=0}^{n} w_k x_k$$

- Assumptions,
  - The effect of x's are linear and additive
  - x's are non-interacting
  - Recall that, $y(x) = f(x) + \epsilon$. The noise is independent and identically normally distributed

Formulation

- What is desired is,

$$\begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \dots & x_n^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \dots & x_n^{(2)} \\ \dots & & & & \\ 1 & x_1^{(N)} & x_2^{(N)} & \dots & x_n^{(N)} \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \dots \\ w_n \end{bmatrix} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \dots \\ y^{(N)} \end{bmatrix}$$

- More succinctly,

$$X_{N\times(n+1)}\, W_{(n+1)\times 1} = Y_{N\times 1}$$

- Note that the model output is linear in the parameters (w's). It also happens to be linear in x's

Pseudo Inverse

- The least squares solution to the above system of linear equations is given by the pseudo inverse or the Moore-Penrose inverse and is given by,

$$W = X^* Y = (X^T X)^{-1} X^T Y$$

- In the event that an inverse exists, then the pseudo inverse is equal to the inverse
- Not practical for ML where N and n are both large. In general, computing the pseudo inverse requires a QR decomposition or a SVD decomposition which is $O(Nn^2)$ if $n > N$ and $O(nN^2)$ is $N > n$

An Iterative Approach

- Recall that we have training data, $\mathcal{X} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$
- The performance of our model on $\mathcal{X}$ can be captured using a loss function, L,

$$J(w) = \sum_{l=1}^{N} L\left(y^{(l)}, \hat{f}(x^{(l)}; w)\right)$$

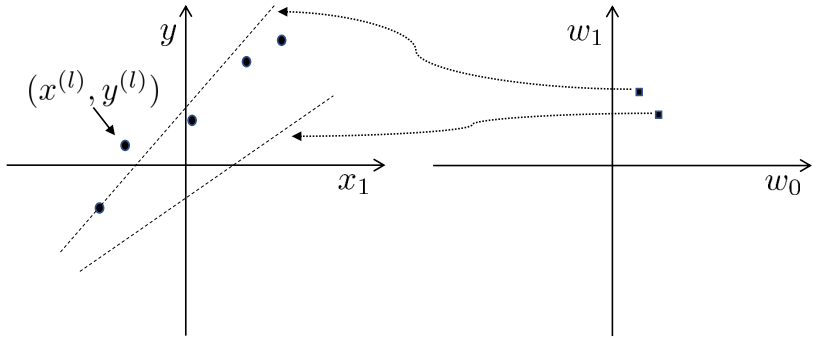
- In regression, we often use a sum-of-squared error loss function,

$$\begin{aligned} J(w) &= \sum_{l=1}^{N} \left(y^{(l)} - \hat{f}(x^{(l)}; w)\right)^2 \\ &= \sum_{l=1}^{N} \left(y^{(l)} - \sum_{k=0}^{n} w_k x_k\right)^2 \end{aligned}$$

- The cost function is convex

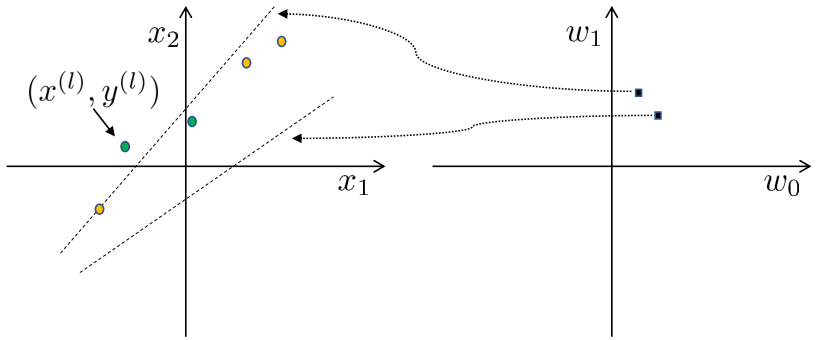
Updating the Parameter Vector

- Note that as the parameter vector w changes, $J(w)$ will change. We would like to find the parameter vector w that minimizes $J(w)$



Input space
Regression

Parameter space



Input space
Classification

Parameter space

- One way to adjust w is using gradient descent, i.e.,

$$w(\text{new}) = w(\text{old}) - \eta \nabla J_w(w(\text{old}))$$

where, $\nabla J_w(w(\text{old}))$ is the gradient evaluated at the old values of the weights, η is the step-size or the learning rate

- i.e.,

$$w_k(\text{new}) = w_k(\text{old}) - \eta \frac{\partial J(w)}{\partial w_k}$$

- Now,

$$\frac{\partial J(w)}{\partial w_k} = -\frac{2}{N} \sum_{l=1}^{N} \left(y^{(l)} - \hat{f}(x^{(l)}; w) \right) x_k^{(l)}$$

- So, the overall learning rule is,

$$\boxed{w_k(\text{new}) = w_k(\text{old}) + \frac{2\eta}{N} \sum_{l=1}^{N} \left(y^{(l)} - \hat{f}(x^{(l)}; w) \right) x_k^{(l)}}$$

The Overall Algorithm (Batch Learning)

1: Initialize weights to small random values (e.g $\in [-0.25, 0.25]$)
2: **while** J is large **do**
3: $\quad \Delta(w_k) = 0, \;\; k = 0, 1, 2, \ldots, n$
4: $\quad$ **for** $l = 1, 2, \ldots, N$ **do**
5: $\quad\quad$ Do a forward pass i.e. compute $\hat{f}^{(l)}$
6: $\quad\quad$ **for** $k = 0, 1, 2, \ldots, n$ **do**
$\quad\quad\quad \Delta(w_k) += (y^{(l)} - \hat{f}^{(l)})\, x_k^{(l)}$
7: $\quad\quad$ **end for**
8: $\quad$ **end for**
9: $\quad$ **for** $k = 0, 1, 2, \ldots, n$ **do**
$\quad\quad w_k(\text{new}) = w_k(\text{old}) + \eta\, \Delta(w_k)$
10: $\quad$ **end for**
11: **end while**

Regularization

- Often, we want to specify additional constraints in order to solve an ill-posed problem or to prevent overfitting
- Typically, the modified loss function takes the form,

$$J(w) = \sum_{l=1}^{N} L\left(y^{(l)}, \hat{f}(x^{(l)}; w)\right) + \lambda \left(\boxed{xx}\right)$$

- λ is the regularization parameter that controls the relative influence of the fidelity term and the regularization term. $\boxed{xx}$ is the regularization term and specifies the additional constraints

Ridge Regression

- A regularization term is added to restrict the magnitude of the parameter vector, i.e.,

$$J(w) = \sum_{l=1}^{N} L\left(y^{(l)}, \hat{f}(x^{(l)}; w)\right) + \lambda \sum_{k=0}^{n} w_k^2$$

- Encourages smoother solutions. Why?

Lasso Regression

- Least Absolute Shrinkage Selection Operator (Lasso) is similar to Ridge regression but uses the $L1$ norm instead of the $L2$ norm used in Ridge Regression, i.e.,

$$J(w) = \sum_{l=1}^{N} L\left(y^{(l)}, \hat{f}(x^{(l)}; w)\right) + \lambda \sum_{k=0}^{n} |w_k|$$

- In contrast to Ridge Regression, Lasso tends to result in more components of w tending towards 0

Elastic Net Regression

- Elastic Net Regression combines Ridge and Lasso Regression and applies regularization of the following form,

$$J(w) = \sum_{l=1}^{N} L\left(y^{(l)}, \hat{f}(x^{(l)}; w)\right) + \lambda_1 \sum_{k=0}^{n} w_k^2 + \lambda_2 \sum_{k=0}^{n} |w_k|$$

Additional Reading

- R. Tibshirani, "Regression Shrinkage and Selection via the Lasso," *Journal of the Royal Statistical Society, Series B*, Vo. 58, pp. 267–288, 1996.
- G. A. F. Seber, and A. J. Lee, *Linear Regression Analysis*, Wiley, 2014.

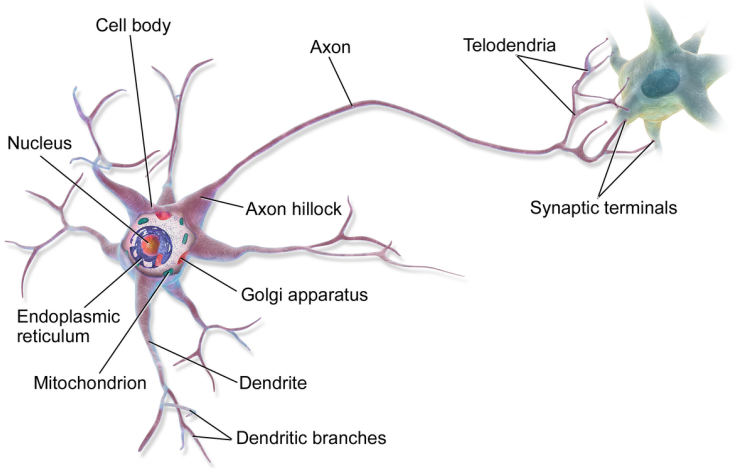
Artificial Neural Networks

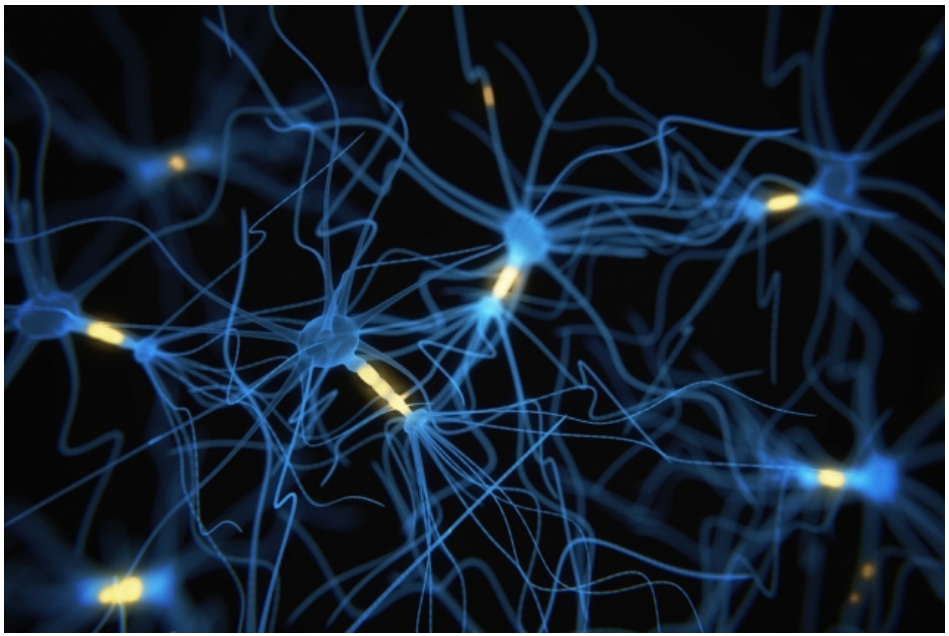
Single Layered Perceptrons and the Delta Rule

Ravi Kothari, Ph.D.

"Failure is simply the opportunity to begin again; this time more intelligently" – Henry Ford

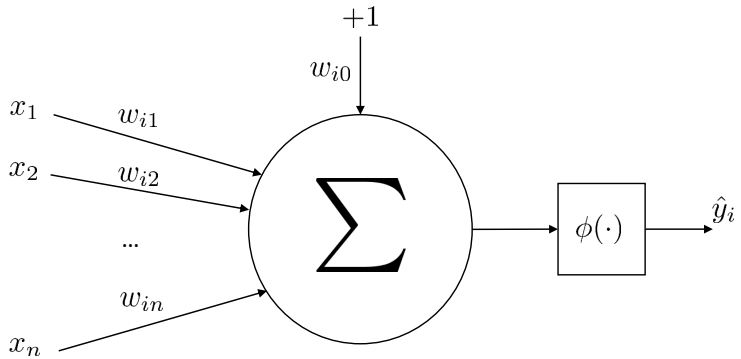
Biological Neuron and Neural Network





From http://en.wikipedia.com and http://news.mit.edu/2017/new-tool-offers-snapshots-neuron-activity-0626

Basic Model of an Artificial Neuron

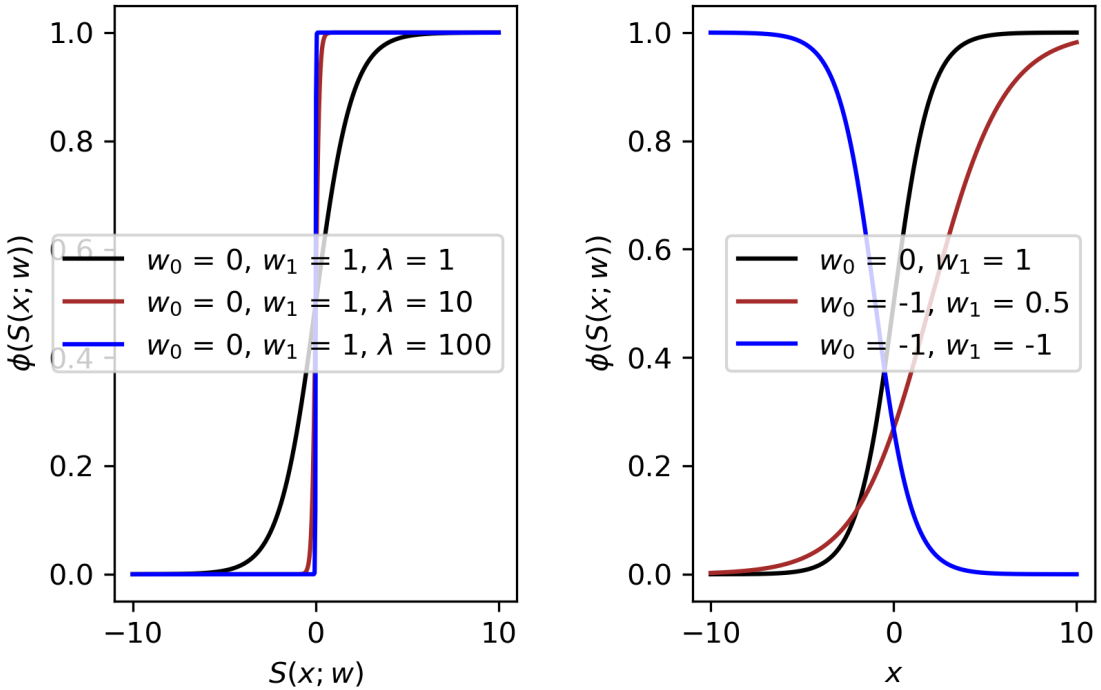


Neuron i

$$\hat{y}_i(x; w) = \phi\left(S_i(x; w)\right) = \phi\left(\sum_{j=1}^{n} w_{ij}x_j + w_{i0}\right) = f\left(\sum_{j=0}^{n} w_{ij}x_j\right)$$

- An artificial neuron. Other models such as spiking neurons have also been proposed
- The output of a neuron can be the input to another neuron
- $\phi(\cdot)$ is the activation function, $S_i(x, w)$ is the weighted sum, w_{ij} is the weight from the j^{th} input and w_{i0} is the "bias". $x_0 = +1$

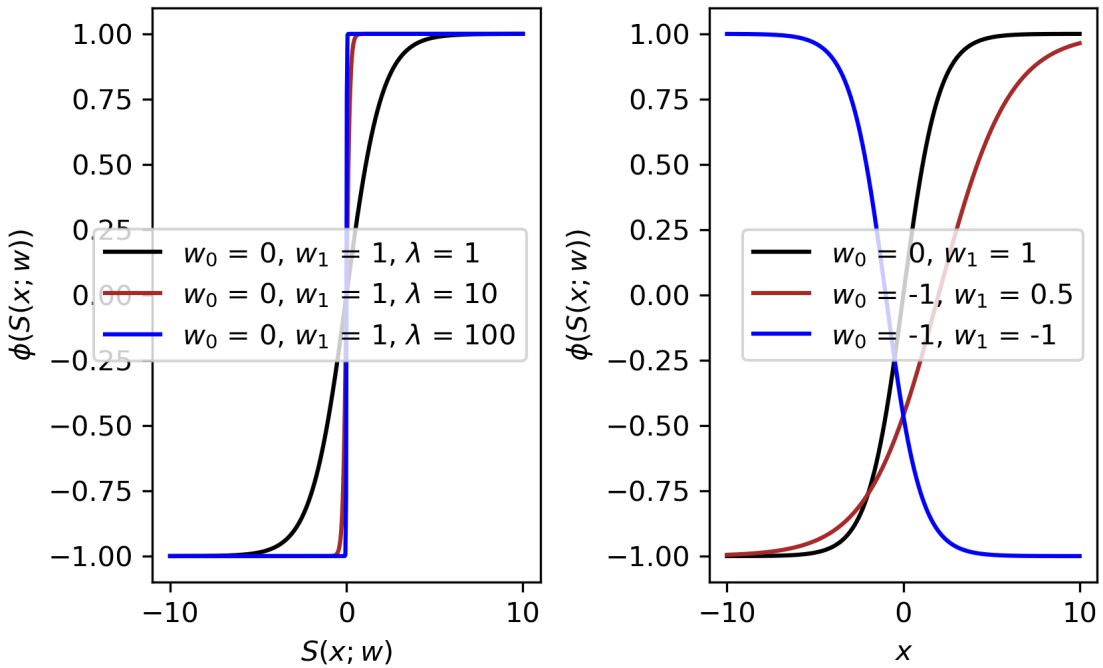
Binary Sigmoid Activation Function



$$\hat{y}_i(x; w) = \phi\left(S_i(x; w)\right) = \frac{1}{1 + e^{-\lambda S_i(x;w)}}$$

- Differentiable
- With increasing λ, becomes more and more like the step function

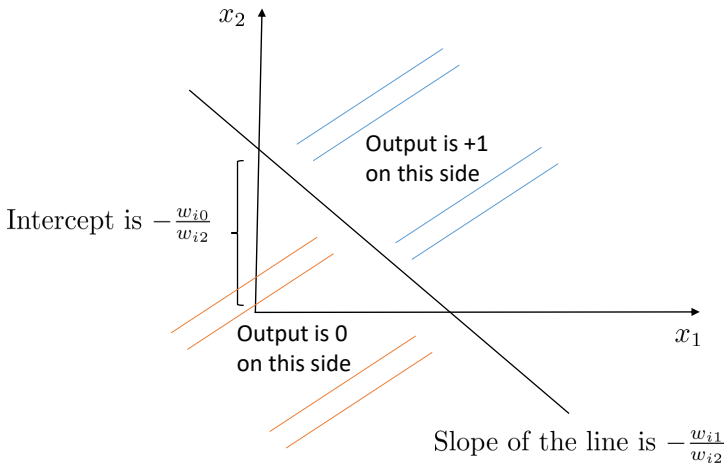
Bipolar Sigmoid Activation Function



$$\hat{y}_i(x; w) = \phi\left(S_i(x; w)\right) = \frac{2}{1 + e^{-\lambda S_i(x;w)}} - 1$$

- Differentiable
- With increasing λ, becomes more and more steeper

What does a Simple Neuron With 2 Inputs Do?



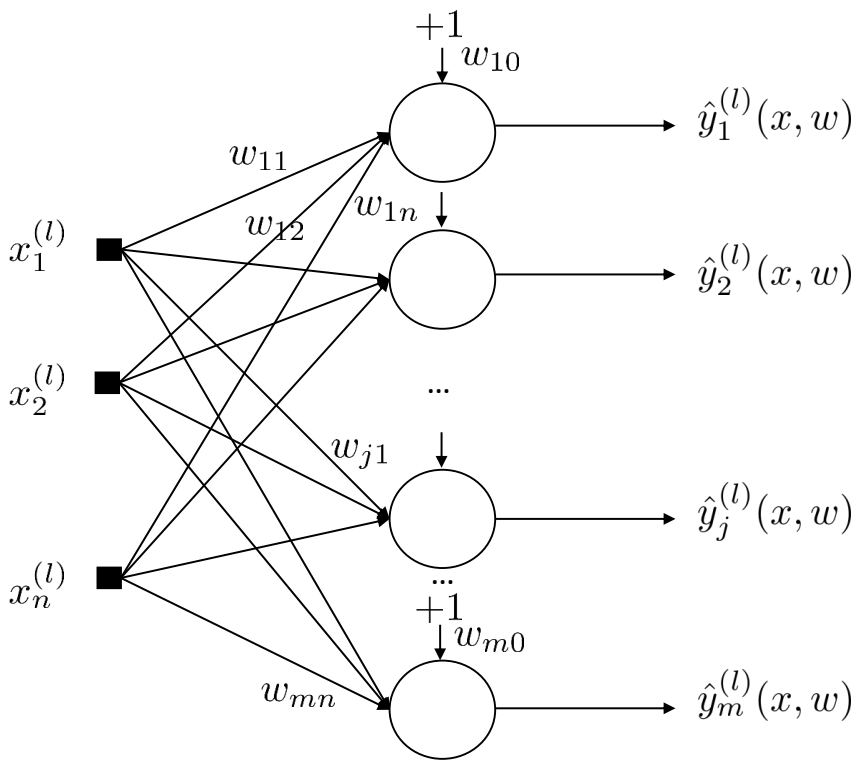
$$\hat{y}_i(x; w) = \phi\left(S_i(x; w)\right) = \phi\left(\sum_{j=1}^{2} w_{ij}x_j + w_{i0}\right) = \phi\left(w_{i1}x_1 + w_{i2}x_2 + w_{i0}\right)$$

- $x_2 = -\frac{w_{i1}}{w_{i2}}x_1 - \frac{w_{i0}}{w_{i2}}$; at $S_i(x; w) = 0$
- $S_i(x; w) > 0$ on one side of the line and $S_i(x; w) < 0$ on the other side. A neuron establishes a linear decision boundary in the input space
- What if the bias (w_{i0}) is not there?

Learning (Adjusting the Weights)

- Based on the training data, we would like to come up with an algorithm that would "evolve" the weights such that the network starts producing the desired output i.e. the decision boundary is adapted such that Class 1 patterns are on one side and Class 2 patterns are on the other side
- To reduce clutter, we will not show some of the dependencies explicitly. So, we will simplify $\hat{y}_i(x; w)$ to $\hat{y}_i$ and $S_i^(x; w)$ to S_i and so on

Single Layered Perceptron



Given: $\left\{\left(x^{(l)}, y^{(l)}\right)\right\}_{l=1}^{N}$; $x^{(l)} \in \mathcal{R}^n$, $y^{(l)} \in \mathcal{R}^m$

The Cost Function

$$J^{(l)} = \sum_{i=1}^{m} \left(y_i^{(l)} - \hat{y}_i^{(l)} \right)^2$$

$$\begin{aligned} J &= \sum_{l=1}^{N} J^{(l)} \\ &= \sum_{l=1}^{N} \sum_{i=1}^{m} \left(y_i^{(l)} - \hat{y}_i^{(l)} \right)^2 \end{aligned}$$

A Simplification

From now on, we will include the bias term implicitly, i.e.,

$$x^{(l)} = \begin{bmatrix} 1 \\ x_1^{(l)} \\ x_2^{(l)} \\ \vdots \\ x_n^{(l)} \end{bmatrix} \qquad w_i = \begin{bmatrix} w_{i0} \\ w_{i1} \\ w_{i2} \\ \vdots \\ w_{in} \end{bmatrix}$$

$$\hat{y}_i^{(l)} = \phi\left(\sum_{j=0}^{n} w_{ij} x_j^{(l)}\right)$$

The Delta Rule

$$
\begin{aligned}
J &= \sum_{l=1}^{N}\sum_{i=1}^{m}\left(y_i^{(l)} - \hat{y}_i^{(l)}\right)^2 \\
&= \sum_{l=1}^{N}\sum_{i=1}^{m}\left(y_i^{(l)} - \phi\left(\sum_{j=0}^{n} w_{ij}x_j\right)\right)^2
\end{aligned}
$$

- So, we can start with random values of w's and adopt the weights by moving opposite to the gradient computed from the equation above, i.e.

$$\Delta w = -\eta \nabla J$$

- η is the learning rate and is typically a small value e.g. 0.1

Batch and Online Weight Updates

In batch learning, we aggregate the gradients computed for each pattern, i.e.,

$$\nabla J = \nabla J^{(1)} + \nabla J^{(2)} + \ldots + \nabla J^{(N)}$$

In online learning, we make the approximation that as long as η is small, we can,

- Feed the next pattern l
- Compute $\nabla J^{(l)}$
- Update w's and repeat

The Delta Rule Derived

$$
\begin{aligned}
J &= \sum_{l=1}^{N}\sum_{i=1}^{m}\left(y_i^{(l)} - \phi\left(\sum_{j=0}^{n} w_{ij}x_j\right)\right)^2 \\
&= \sum_{l=1}^{N}\sum_{i=1}^{m}\left(y_i^{(l)} - \hat{y}_i^{(l)}\right)^2 = \sum_{l=1}^{N}\sum_{i=1}^{m} {e_i^{(l)}}^2
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial J^{(l)}}{\partial w_{ij}} &= \frac{\partial J^{(l)}}{\partial e_i^{(l)}} \quad \frac{\partial e_i^{(l)}}{\partial \hat{y}_i^{(l)}} \quad \frac{\partial \hat{y}_i^{(l)}}{\partial S_i^{(l)}} \quad \frac{\partial S_i^{(l)}}{\partial w_{ij}} \\
&= 2e_i^{(l)} \cdot (-1) \cdot \phi'(S_i^{(l)}) \cdot x_j^{(l)} \\
&= -2\left(y_i^{(l)} - \hat{y}_i^{(l)}\right) \cdot \phi'(S_i^{(l)}) \cdot x_j^{(l)}
\end{aligned}
$$

What is $\phi'(S_i)$

Recall that,

$$\hat{y}_i^{(l)} = \phi\left(S_i^{(l)}\right) \tag{1}$$

If the activation function is linear, $\hat{y}_i^{(l)} = S_i^{(l)}$

$$\phi'\left((S_i^{(l)}\right) = \frac{\hat{y}_i^{(l)}}{S_i^{(l)}} = 1 \tag{2}$$

If the activation function is a binary sigmoid, $\hat{y}_i^{(l)} = 1/(1 + e^{-S_i^{(l)}})$

$$\phi'\left(S_i^{(l)}\right) = \hat{y}_i^{(l)}\left(1 - \hat{y}_i^{(l)}\right) \tag{3}$$

If the activation function is a bipolar sigmoid, $\hat{y}_i^{(l)} = \left(2/(1 + e^{-S_i^{(l)}})\right) - 1$

$$\phi'\left(S_i^{(l)}\right) = \frac{1}{2}\left(1 - \hat{y}^{(l)_i{}^2}\right) \tag{4}$$

The Overall Algorithm (Online Learning)

1: Initialize weights to small random values (e.g $\in [-0.25, 0.25]$)
2: **while** J is large **do**
3: **for** $l = 1, 2, \ldots, N$ **do**
4: Do a forward pass i.e. compute $\hat{y}_1^{(l)}, \hat{y}_2^{(l)}, \ldots, \hat{y}_m^{(l)}$
5: **for** $i = 1, 2, \ldots, n$ **do**
6: **for** $j = 1, 2, \ldots, m$ **do**
7: $\frac{\partial J^{(l)}}{\partial w_{ij}} = -\left(y_i^{(l)} - \hat{y}_i^{(l)}\right) \cdot \phi'\left(S_i^{(l)}\right) \cdot x_j^{(l)}$
 $w_{ij}(\text{new}) = w_{ij}(\text{old}) - \eta \frac{\partial J^{(l)}}{\partial w_{ij}}$
8: **end for**
9: **end for**
10: **end for**
11: **end while**

The Overall Algorithm (Batch Learning)

Initialize weights to small random values (e.g $\in [-0.25, 0.25]$)
while J is large **do**
  $\Delta w_{ij} = 0; \quad i = 1, 2, \ldots n; \; j = 1, 2, \ldots, m$
  for $l = 1, 2, \ldots, N$ **do**
    Do a forward pass i.e. compute $\hat{y}_1^{(l)}, \hat{y}_2^{(l)}, \ldots, \hat{y}_m^{(l)}$
    for $i = 1, 2, \ldots, n$ **do**
      for $j = 1, 2, \ldots, m$ **do**
        $\frac{\partial J^{(l)}}{\partial w_{ij}} = -\left(y_i^{(l)} - \hat{y}_i^{(l)}\right) \cdot \phi'\left(S_i^{(l)}\right) \cdot x_j^{(l)}$
        $\Delta w_{ij}(\text{new}) = \Delta w_{ij}(\text{old}) - \eta \frac{\partial J^{(l)}}{\partial w_{ij}}$
      end for
    end for
  end for
  $w_{ij}(\text{new}) = w_{ij}(\text{old}) + \Delta w_{ij}$
end while

Additional Comments

- The Delta Rule is also called the LMS Rule or the Perceptron Learning Rule
- Obviously, the activation function needs to be differentiable for gradient descent based optimizations. The sigmoid is a nice choice for this reason (more reasons later)
- The loss function is quadratic
- For classification, we can interpret an output of greater than 0 as Class 1 and an output of less than 0 as Class 2 if we are using a bipolar sigmoid
- Note that a neuron establishes a linear decision boundary. Thus, such networks can only perform linearly separable pattern classification

An Illustrative Example – Online Learning

Assume we are given the following training data,

	Input	Desired Output	
$x^{(1)} \rightarrow$	1	1	$\leftarrow y^{(1)}$
$x^{(2)} \rightarrow$	-0.5	-1	$\leftarrow y^{(2)}$
$x^{(3)} \rightarrow$	3	1	$\leftarrow y^{(3)}$
$x^{(4)} \rightarrow$	-1	-1	$\leftarrow y^{(4)}$

So, we have $n = 1$ input and $m = 1$ output. Let us initialize the parameters as: $w_{10} = -0.5$, $w_{11} = 0.5$, take bipolar activations (outputs are +1 and -1), and $\eta = 0.1$

An Illustrative Example – Online Learning (contd.)

- Present $x^{(1)}$
  - $S_1 = w_{10} \times 1 + w_{11} \times 1 = 0$. $\hat{y}^{(1)} = (2/(1 + e^{-0})) - 1 = 0$
  - Adjust the weights
    - ★ $w_{ij}(new) = w_{ij}(old) + \eta \left(y_i^{(l)} - \hat{y}_i^{(l)} \right) \cdot \phi' \left(S_i^{(l)} \right) \cdot x_j^{(l)}$
    - ★ $w_{10}(new) = -0.5 + 0.1\,(1 - 0)) \cdot \frac{1}{2}\,(1 - 0) \cdot 1 = -0.45$
    - ★ $w_{11}(new) = 0.5 + 0.1\,(1 - 0)) \cdot \frac{1}{2}\,(1 - 0) \cdot 1 = 0.55$
- Present $x^{(2)}$
  - $S_1 = w_{10} \times 1 + w_{11} \times (-0.5) =$ -3.2. $\hat{y}^{(1)} = (2/(1 + e^{-3.2})) - 1 = ...$
  - ...

Additional Reading

- C. M. Bishop, *Pattern Recognition and Machine Learning*, Springer, 2010.
- R. A. Rescorla, and A. R. Wagner, "A theory of Pavlovian conditioning: Variations in the effectiveness of reinforcement and nonreinforcement," *Classical Conditioning II*, A. H. Black & W. F. Prokasy, Eds., pp. 64–99. Appleton-Century-Crofts, 1972.