Machine Learning Notes

Haotian Chen

Contents

1 Machine Learning Introduction

A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E.

1.1 Supervised Learning

Supervised learning algorithms build a mathematical model of a set of data that contains both the inputs and the desired outputs.

In supervised learning, each example is a pair consisting of an input object (typically a vector) and a desired output value (also called the supervisory signal).

1.1.1 Regression

Regression analysis is a set of statistical processes for estimating the relationships between a dependent variable (often called the 'outcome variable') and one or more independent variables (often called 'predictors', 'covariates', or 'features').

Training(Learning) Process:
observed data(training set) $\rightarrow$ *learning algorithm* $\rightarrow$ *h(hypothesis)*

Predicting Process:
independent variable $\rightarrow$ *h(hypothesis)* $\rightarrow$ *dependent variable*

1.1.2 Classification

Classification is the problem of identifying to which of a set of categories (subpopulations) a new observation belongs, on the basis of a training set of data containing observations (or instances) whose category membership is known.

1.2 Unsupervised Learning

Unsupervised learning algorithms take a set of data that contains only inputs, and find structure in the data, like grouping or clustering of data points.

Draw inferences from data sets consisting of input data without labeled responses.

1.3 Reinforcement learning

Reinforcement learning is an area of machine learning concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward.

2 Regression

2.1 Linear Regression(from Wikipedia)

Given a data set $\{y_i, x_{i1}, ..., x_{ip}\}_{i=1}^{n}$ of n statistical units, a linear regression model assumes that the relationship between the dependent variable y and the p-vector of regressors x is linear.

$$y_i = \beta_0 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip} + \epsilon_i,\ i = 1, \ldots, n$$
$$y = X\beta + \epsilon$$

where

$$y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, X = \begin{bmatrix} x_1^T \\ x_2^T \\ \vdots \\ x_n^T \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & \ldots & x_{1p} \\ 1 & x_{21} & \ldots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \ldots & x_{np} \end{bmatrix}, \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{bmatrix}, \epsilon = \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{bmatrix}$$

y is a vector of observed values y_i $(i = 1, \ldots, n)$ of the variable called the regressand, endogenous variable, response variable, measured variable, criterion variable, or dependent variable.

X may be seen as a matrix of row-vectors x_i or of n-dimensional column-vectors X_j, which are known as regressors, exogenous variables, explanatory variables, covariates, input variables, predictor variables, or independent variables. Usually a constant is included as one of the regressors. In particular, $x_{i0} = 1$ for $i = 1, \ldots, n$. The corresponding element of β is called the intercept.

β is a $(p + 1)$-dimensional parameter vector, where β_0 is the intercept term (if one is included in the model—otherwise β is p-dimensional). Its elements are known as effects or regression coefficients (although the latter term is sometimes reserved for the estimated effects).

ϵ is a vector of values ϵ_i. This part of the model is called the error term, disturbance term, or sometimes noise (in contrast with the "signal" provided by the rest of the model).

Matrix Concepts

identity matrix I(or $I_{n\times n}$):

$$A \cdot I = I \cdot A = A$$

$$I = \begin{bmatrix} 1 & 0 & \ldots & 0 & 0 \\ 0 & 1 & \ldots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & 1 & 0 \\ 0 & 0 & \ldots & 0 & 1 \end{bmatrix}$$

inverse matrix A^{-1}: If A is an $m \times m$ matrix, and if it has an inverse.

$$A \cdot A^{-1} = A^{-1} \cdot A = I$$

transpose matrix A^T: Let A be an $m \times n$ matrix. Then A^T is an $n \times m$ matrix.

$$A^T_{ij} = A_{ji}$$

2.1.1 Linear Regression Learning (from Machine Learning course)

For convenience reasons, define $x_0 = 1$

$$x = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix}, \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix}$$

hypothesis:

$$y = h_\theta(x) = \theta^T x = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

training data:

$$(x^{(i)}, y^{(i)})\ for\ i = 1, \ldots, m$$

cost function:
convex: second derivative (hessian matrix) is non-negative definite.
goal: $\underset{\theta}{\text{minimize}}\, J(\theta)$

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

gradient descent:
repeat until convergence {

$$\theta_j =: \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) = \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

simultaneously update θ_j for $j = 0, \ldots, n$

}

vectorized implementation:
repeat until convergence {

$$\theta =: \theta - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x^{(i)}$$

}

batch size:
The number of training samples to use in a training action.

epoch:
The number of times of training which uses all the training samples.

Feature scaling:
The range of all features should be normalized so that each feature contributes approximately proportionately to the result. Also it helps gradient descent converge much faster.

mean normalization: (don't apply to x_0)

$$x_i = \frac{x_i - mean(x_i)}{max(x_i) - min(x_i)}$$

standardization: (don't apply to x_0) * math stuff needed for this part
Feature standardization makes the values of each feature in the data have zero-mean (when subtracting the mean in the numerator) and unit-variance.

$$\mu : \text{mean}, \sigma^2 : \text{variance}, \sigma : \text{standard deviation}$$

$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_i^{(j)}$$

$$\sigma_i^2 = \frac{1}{m}\sum_{j=1}^{m} (x_i^{(j)} - \mu_i)^2$$

$$x_i = \frac{x_i - \mu_i}{\sigma_i}$$

learning rate α:
If α is too small: slow convergence.
If α is too large: may not decrease on every iteration and thus may not converge.
try a range of learning rate to find a good one:

$$\alpha = \dots, 0.001, \dots, 0.01, \dots, 0.1, \dots$$

Feature choosing:
Choose the right features to fit the data set.
housing price example, choose from:

$$h_\theta(x) = \theta_0 + \theta_1 \times frontage + \theta_2 \times depth$$

$$h_\theta(x) = \theta_0 + \theta_1 \times area$$

polynomial regression:

Use the polynomial model with the machinery of multivariant linear regression. housing price example, choose from:

$$h_\theta(x) = \theta_0 + \theta_1 \times area + \theta_2 \times area^2$$
$$h_\theta(x) = \theta_0 + \theta_1 \times area + \theta_2 \times \sqrt{area}$$

2.1.2 Normal Equation

Let:

$$X = \begin{bmatrix} (x^{(1)})^T \\ (x^{(2)})^T \\ \vdots \\ (x^{(m)})^T \end{bmatrix}, y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

Calculate best θ: * math stuff needed for this part

$$\theta = (X^T X)^{-1} X^T y$$

If $X^T X$ is noninvertible, the common causes might be having:
1. Redundant features, where two features are very closely related (i.e. they are linearly dependent)
2. Too many features (e.g. $m < n$). In this case, delete some features or use "regularization".

Gradient Descent vs **Normal Equation**:

Gradient Descent	**Normal Equation**
Need to choose α	No need to choose α
Needs many iterations	No need to iterate
$O(kn^2)$	$O(n^3)$. Need to calculate $(X^T X)^{-1}$
Works well when n is large	Slow if n is very large

Vectorized Cost Function:

$$J(\theta) = \frac{1}{2m}(X\theta - y)^T (X\theta - y)$$

2.2 Logistic Regression

sigmoid function: (Logistic Function)
Maps any real number from $(-\infty, +\infty)$ to the $(0, 1)$ interval.

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

hypothesis:

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}}$$

$h_\theta(x)$ will give us the probability that our output is 1. Our probability that our prediction is 0 is just the complement of our probability that it is 1.

$$h_\theta(x) = P(y = 1|x;\theta) = 1 - P(y = 0|x;\theta)$$
$$P(y = 1|x;\theta) + P(y = 0|x;\theta) = 1$$

decision boundary:
In order to get our discrete 0 or 1 classification, we can translate the output of the hypothesis function as follows:

$$h_\theta(x) \geq 0.5 \rightarrow y = 1$$
$$h_\theta(x) < 0.5 \rightarrow y = 0$$

Decision is made when:

$$g(\theta^T x) \geq 0.5, \text{ when } \theta^T x \geq 0$$
$$g(\theta^T x) < 0.5, \text{ when } \theta^T x < 0$$

Now we get:

$$\theta^T x \geq 0 \rightarrow y = 1$$
$$\theta^T x < 0 \rightarrow y = 0$$

The decision boundary is the line that separates the area where $y = 0$ and where $y = 1$:
$$\theta^T x = 0$$

cost function:
If we use the the same cost function that we use for linear regression for logistic regression, the cost function will have many local optima. It will not be a convex function. Instead, we construct logistic regression cost function as follows:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} Cost(h_\theta(x^{(i)}), y^{(i)})$$
$$Cost(h_\theta(x), y) = -log(h_\theta(x)), \text{ if } y = 1$$
$$Cost(h_\theta(x), y) = -log(1 - h_\theta(x)), \text{ if } y = 0$$

We can find it guarantees that $J(\theta)$ is convex for logistic regression: * math stuff needed for this part

$$Cost(h_\theta(x), y) = 0 \text{ if } h_\theta(x) = y, \text{ at } h_\theta(x) = y = 0 \text{ and } h_\theta(x) = y = 1$$

$$Cost(h_\theta(x), y) \rightarrow +\infty \text{ if } y = 0 \text{ and } h_\theta(x) \rightarrow 1$$
$$Cost(h_\theta(x), y) \rightarrow +\infty \text{ if } y = 1 \text{ and } h_\theta(x) \rightarrow 0$$

We can compress our cost function's two conditional cases into one case:

$$Cost(h_\theta(x), y) = -ylog(h_\theta(x)) - (1-y)log(1-h_\theta(x))$$

simplified cost function:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}log(h_\theta(x^{(i)})) + (1-y^{(i)})log(1-h_\theta(x^{(i)}))]$$

vectorized cost function:

$$h = g(X\theta)$$

$$J(\theta) = \frac{1}{m}(-y^T log(h) - (1-y)^T log(1-h))$$

gradient descent:
repeat until convergence {

$$\theta_j =: \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta) = \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

simultaneously update θ_j for $j = 0, \dots, n$

}

vectorized implementation:
repeat until convergence {

$$\theta =: \theta - \alpha\frac{1}{m}X^T(g(X\theta) - y)$$

}

advanced optimization methods:
With the same cost function, we can choose different optimization method to optimize θ: * math stuff needed for this part

- gradient descent
- conjugate gradient
- BFGS
- L-BFGS
- ...

multiclass classification: (one vs all)
Instead of $y = \{0, 1\}$, we will expand our definition so that $y = \{0, 1, ..., n\}$. Then we divide our problem into n+1 binary classification problems, by constructing n + 1 hypothesis functions; in each one, we choose one class and then

lump all the others into a single second class, then predict the probability that y is a member of the chosen class.

To summarize:
Train a logistic regression classifier $h_\theta^{(i)}(x)$ for each class to predict the probability that $y = i$. On a new input x, to make a prediction, pick the class i that maximizes $h_\theta^{(i)}(x)$.

$$
\begin{gathered}
y \in \{0, 1 ... n\} \\
h_\theta^{(0)}(x) = P(y = 0|x; \theta) \\
h_\theta^{(1)}(x) = P(y = 1|x; \theta) \\
\cdots \\
h_\theta^{(n)}(x) = P(y = n|x; \theta) \\
\text{prediction} = \max_i(h_\theta^{(i)}(x))
\end{gathered}
$$

2.3 Overfitting Problem

underfitting:
High bias, which means hypothesis fits training data poorly, is usually caused by a function that is too simple or using too few features.

overfitting:
High variance, which means hypothesis fits training data well, but does not generalize well to predict new data. It is usually caused by a complicated function with too many features.

address overfitting:

1. Reduce the number of features:
    - Manually select which features to keep.
    - Use a model selection algorithm (studied later in the course).
2. Regularization:
    - Keep all the features, but reduce the magnitude of parameters θ_j.
    - Regularization works well when we have a lot of slightly useful features.

2.3.1 Linear Regression Regularization

regularized cost function:
We can reduce(penalize) the weight of the features in our function carry by increasing their cost. The λ is the regularization parameter. It determines how much the costs of our theta parameters are inflated.

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

Using the above cost function with the extra summation, we can smooth the output of our hypothesis function to reduce overfitting. If λ is too small, it would be hard to see a difference. If λ is too large, it may smooth out the function too much and cause underfitting.

regularized gradient descent:
repeat until convergence {

$$\theta_0 =: \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)}$$

$$\begin{aligned}\theta_j =: \theta_j - \alpha((\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}) + \frac{\lambda}{m}\theta_j) \\ = \theta_j(1 - \alpha\frac{\lambda}{m}) - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}\end{aligned}$$

simultaneously update θ_0 and θ_j for $j = 1, \ldots, n$

}

The first term in the above equation, $1 - \alpha\frac{\lambda}{m}$ will always be less than 1. Intuitively you can see it as reducing the value of θ_j by some amount on every update.

regularized normal equation: * math stuff needed for this part

$$\theta = (X^TX + \lambda L)^{-1}X^Ty$$

$$L = \begin{bmatrix} 0 & 0 & \ldots & 0 & 0 \\ 0 & 1 & \ldots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & 1 & 0 \\ 0 & 0 & \ldots & 0 & 1 \end{bmatrix}$$

L has a dimension of $(n+1)\times(n+1)$. Intuitively, this is the identity matrix (though we are not including x_0), multiplied with a single real number λ.

Recall that if $m < n$, then X^TX is non-invertible. However, when we add the term λL ($\lambda > 0$), the term $X^TX + \lambda L$ becomes invertible.

2.3.2 Logistic Regression Regularization

regularized cost function:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}log(h_\theta(x^{(i)})) + (1-y^{(i)})log(1-h_\theta(x^{(i)}))] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

regularized gradient descent:
repeat until convergence {

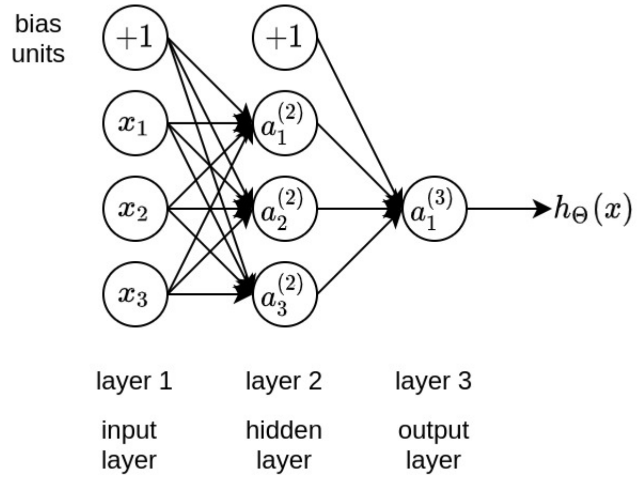
$$\theta_0 =: \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)}$$

$$\begin{aligned}\theta_j =: \theta_j - \alpha((\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}) + \frac{\lambda}{m}\theta_j) \\ = \theta_j(1-\alpha\frac{\lambda}{m}) - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}\end{aligned}$$

simultaneously update θ_0 and θ_j for $j = 1, \ldots, n$
}

3 Neural Network

3.1 Basic Structure



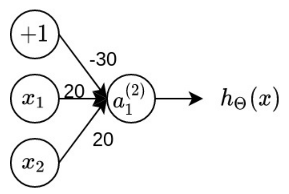
$$a_i^{(j)} = \text{"activation" of unit i in layer j}$$

$$\Theta^{(j)} = \text{matrix of weights (edges) from layer j to j + 1}$$

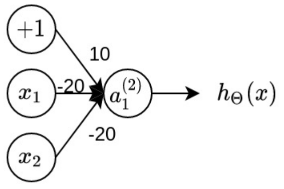
$$a_1^{(2)} = g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3)$$
$$a_2^{(2)} = g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3)$$
$$a_3^{(2)} = g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3)$$
$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)})$$

If network has s_j units in layer j, s_{j+1} units in layer $j+1$, then $\Theta^{(j)}$ will be of dimension $s_{j+1} \times (s_j + 1)$.

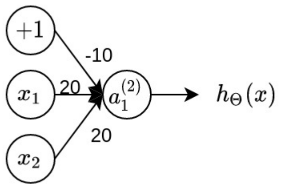
3.2 Simple Applications



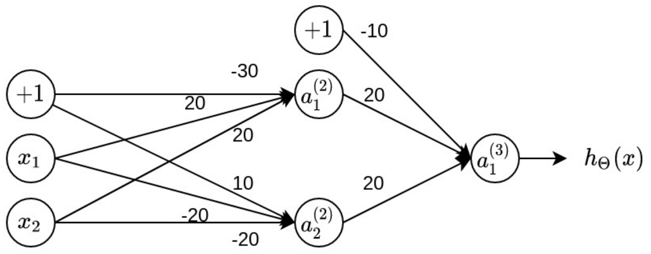
x_1 AND x_2



(NOT x_1) AND (NOT x_2)

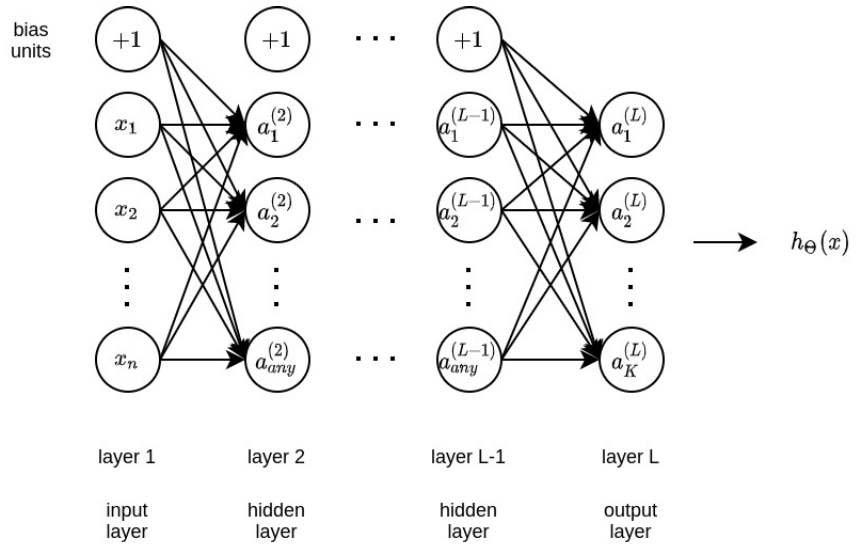


x_1 OR x_2



x_1 XNOR x_2

3.3 Generalized Model (one vs all)



For a neural network that has:

$$L = \text{total number of layers in the network}$$
$$s_l = \text{number of units (not counting bias unit) in layer } l$$
$$K = \text{number of output units/classes}$$

assume $a^{(1)} = x, a^{(L)} = h_\Theta(x)$, let:

$$z^{(l)} = \Theta^{(l-1)} a^{(l-1)}$$
$$a^{(l)} = g(z^{(l)})$$
$$h_\Theta(x) = a^{(L)} = g(z^{(L)})$$

Notice that bias units ($a_0^{(l)} = 1$) are considered as input only when calculating the next layer. They are not included in the output generated by previous layer.

regularized cost function:

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}[y_k^{(i)} log(h_\Theta(x^{(i)})_k) + (1-y_k^{(i)}) log(1-h_\Theta(x^{(i)})_k)] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{j,i}^{(l)})^2$$

3.4 Backpropagation Preliminary

matrix calculus: see Wikipedia
chaine rule:
Suppose the variable J depends on the variables $\theta_1, \ldots, \theta_p$ via the intermediate variable $z_1, \ldots, z_k$.

$$z_j = z_j(\theta_1, \ldots, \theta_p), \forall j \in \{1, \ldots, k\}$$
$$J = J(z_1, \ldots, z_k)$$

Expand J, we can find:

$$\frac{\partial J}{\partial \theta_i} = \sum_{j=1}^{k} \frac{\partial J}{\partial z_j} \frac{\partial z_j}{\partial \theta_i}, \forall i \in \{1, \ldots, p\}$$

chain rule derivation for matrix:
Suppose J is a real-valued output variable, $z \in \mathbb{R}^m$ is the intermediate variable and $\Theta \in \mathbb{R}^{m \times d}$, $a \in \mathbb{R}^d$ are the input variables. Suppose they satisfy:

$$z = \Theta a$$
$$J = J(z)$$

Then we can get:

$$\frac{\partial J}{\partial a} = \begin{bmatrix} \frac{\partial J}{\partial a_1} \\ \vdots \\ \frac{\partial J}{\partial a_d} \end{bmatrix} = \begin{bmatrix} \sum_{j=1}^{m} \frac{\partial J}{\partial z_j} \frac{z_j}{a_1} \\ \vdots \\ \sum_{j=1}^{m} \frac{\partial J}{\partial z_j} \frac{z_j}{a_d} \end{bmatrix} = \begin{bmatrix} \frac{\partial z_1}{\partial a_1} & \cdots & \frac{\partial z_m}{\partial a_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_1}{\partial a_d} & \cdots & \frac{\partial z_m}{\partial a_d} \end{bmatrix} \begin{bmatrix} \frac{\partial J}{\partial z_1} \\ \vdots \\ \frac{\partial J}{\partial z_m} \end{bmatrix} = \frac{\partial z}{\partial a} \frac{\partial J}{\partial z} = \Theta^T \frac{\partial J}{\partial z}$$

$$\frac{\partial J}{\partial \Theta_{ij}} = \sum_{k=1}^{m} \frac{\partial J}{\partial z_k} \frac{\partial z_k}{\partial \Theta_{ij}} = \frac{\partial J}{\partial z_i} \frac{\partial z_i}{\partial \Theta_{ij}} = \frac{\partial J}{\partial z_i} a_j$$

$$\frac{\partial J}{\partial \Theta} = \begin{bmatrix} \frac{\partial J}{\partial z_1} \\ \vdots \\ \frac{\partial J}{\partial z_m} \end{bmatrix} \begin{bmatrix} a_1 & \ldots & a_d \end{bmatrix} = \frac{\partial J}{\partial z} a^T$$

element-wise chain rule:
Assume $z, a \in \mathbb{R}^d$:

$$a = \sigma(z), \text{ where } \sigma \text{ is an element-wise activation}$$
$$J = J(a)$$

Then we have:

$$\frac{\partial J}{\partial z} = \frac{\partial J}{\partial a} \odot \sigma'(z)$$

Where σ' is the element-wise derivative of the activation function σ.

3.5 Backpropagation

To train the model, we need to update Θ for each epoch: (gradient decent)

$$\Theta := \Theta - \alpha \frac{\partial}{\partial \Theta} J(\Theta)$$

We can see $\frac{\partial}{\partial \Theta} J(\Theta)$ is hard to get directly. There is an easier way to calculate it. For each training example $(x^{(q)}, y^{(q)}), q \in \{1, \ldots, m\}$, define **loss function**:

$$J = -\sum_{k=1}^{K} [y_k^{(q)} log(h_\Theta(x^{(q)})_k) + (1 - y_k^{(q)}) log(1 - h_\Theta(x^{(q)})_k)]$$

Apply chain rule we have:

$$\begin{aligned}
\frac{\partial J}{\partial \Theta^{(l)}} &= \frac{\partial J}{\partial z^{(l+1)}} (a^{(l)})^T \\
\frac{\partial J}{\partial a^{(l)}} &= (\Theta^{(l)})^T \frac{\partial J}{\partial z^{(l+1)}} \\
\frac{\partial J}{\partial z^{(l)}} &= \frac{\partial J}{\partial a^{(l)}} \odot g'(z^{(l)}) \\
&= (\Theta^{(l)})^T \frac{\partial J}{\partial z^{(l+1)}} \odot g'(z^{(l)}) \\
&= (\Theta^{(l)})^T \frac{\partial J}{\partial z^{(l+1)}} \odot (a^{(l)} \odot (1 - a^{(l)}))
\end{aligned}$$

For $p \in \{1, ..., K\}$:

$$\begin{aligned}
\frac{\partial J}{\partial z_p^{(L)}} &= \frac{\partial}{\partial z_p^{(L)}} \sum_{k=1}^{K} -[y_k^{(q)} log(h_\Theta(x^{(q)})_k) + (1 - y_k^{(q)}) log(1 - h_\Theta(x^{(q)})_k)] \\
&= \frac{\partial}{\partial z_p^{(L)}} \{-[y_p^{(q)} log(\frac{1}{1 + e^{-z_p^{(L)}}}) + (1 - y_p^{(q)}) log(1 - \frac{1}{1 + e^{-z_p^{(L)}}})]\} \\
&= -[y_p^{(q)} (1 + e^{-z_p^{(L)}}) \frac{0 - (-e^{-z_p^{(L)}})}{(1 + e^{-z_p^{(L)}})^2} + (1 - y_p^{(q)}) \frac{1 + e^{-z_p^{(L)}}}{e^{-z_p^{(L)}}} \frac{(-e^{-z_p^{(L)}})(1 + e^{-z_p^{(L)}}) - e^{-z_p^{(L)}}(-e^{-z_p^{(L)}})}{(1 + e^{-z_p^{(L)}})^2}] \\
&= -[y_p^{(q)} \frac{e^{-z_p^{(L)}}}{1 + e^{-z_p^{(L)}}} + (1 - y_p^{(q)}) \frac{-1}{1 + e^{-z_p^{(L)}}}] \\
&= -\frac{y_p^{(q)} e^{-z_p^{(L)}} + y_p^{(q)} - 1}{1 + e^{-z_p^{(L)}}} \\
&= \frac{1}{1 + e^{-z_p^{(L)}}} - y_p^{(q)} \\
&= a_p^{(L)} - y_p^{(q)}
\end{aligned}$$

Then we get:

$$\frac{\partial J}{\partial z^{(L)}} = \begin{bmatrix} \frac{\partial J}{\partial z_1^{(L)}} \\ \vdots \\ \frac{\partial J}{\partial z_K^{(L)}} \end{bmatrix} = a^{(L)} - y^{(q)}$$

For convenience, define **error term**:

$$\delta^{(l)} = \frac{\partial J}{\partial z^{(l)}}$$

Then we get:

$$\frac{\partial J}{\partial \Theta^{(l)}} = \delta^{(l+1)}(a^{(l)})^T$$

$$\delta^{(l)} = (\Theta^{(l)})^T \delta^{(l+1)} \odot (a^{(l)} \odot (1 - a^{(l)}))$$

$$\delta^{(L)} = a^{(L)} - y^{(q)}$$

backpropagation algorithm: (compute $\frac{\partial}{\partial \Theta} J(\Theta)$)

```
training set: $(x^{(q)}, y^{(q)}), q \in \{1, \ldots, m\}$
set $\Delta_{ij}^{(l)} = 0, l \in \{1, \ldots, L-1\}$
for $q \in \{1, \ldots, m\}$:
    forward propagation: compute $a^{(l)}$ for $l \in \{2, \ldots, L\}$
    compute $\delta^{(L)} = a^{(L)} - y^{(q)}$
    for $l \in \{L-1, \ldots, 2\}$:
        compute $\delta^{(l)} = (\Theta^{(l)})^T \delta^{(l+1)} \odot (a^{(l)} \odot (1 - a^{(l)}))$
    for $l \in \{1, \ldots, L-1\}$:
        compute $\Delta^{(l)} := \Delta^{(l)} + \frac{\partial J}{\partial \Theta^{(l)}} = \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T$
compute $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \begin{cases} \frac{1}{m}\Delta_{ij}^{(l)} + \frac{\lambda}{m}\Theta_{ij}^{(l)}, & \text{if } j \neq 0 \\ \frac{1}{m}\Delta_{ij}^{(l)}, & \text{if } j = 0 \end{cases}$
```

3.6 Backpropagation in Practice

unrolling parameters:
We can concat flattened matrices into a single vector for the convenience of some calculations. Also after slice and reshape on the vector we can get matrices back.

$$thetaVector = concat(\Theta^{(1)}.flatten(), \ldots, \Theta^{(L-1)}.flatten())$$

$$deltaVector = concat(D^{(1)}.flatten(), \ldots, D^{(L-1)}.flatten())$$

gradient checking:
We can approximate the derivative of $J(\Theta)$ with respect to $\Theta_{ij}^{(l)}$ as:

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) \approx \frac{J(\ldots, \Theta_{ij}^{(l)} + \epsilon, \ldots) - J(\ldots, \Theta_{ij}^{(l)} - \epsilon, \ldots)}{2\epsilon}$$

A small value for ϵ such as $\epsilon = 10^{-4}$, guarantees that the math works out properly. If the value for ϵ is too small, we can end up with numerical problems. Then we can check if $gradApprox \approx deltaVector$. Once you have verified once that your backpropagation algorithm is correct, you don't need to compute gradApprox again. The code to compute gradApprox can be very slow.

random initialization: (symmetry breaking)
Initializing all the weights to zero does not work with neural networks, because all hidden units will be completely identical (symmetric) - compute exactly the same function. When we backpropagate, all nodes will update to the same value repeatedly. Instead we can randomly initialize our weights for our Θ matrices using the following method: (here ϵ is a value selected for random initialization)

$$\text{initialize each } \Theta_{ij}^{(l)} \text{ to a random value in } [-\epsilon, +\epsilon]$$

One effective strategy for choosing ϵ is to base it on the number of units in the network: (Xavier normalized initialization)

$$\epsilon^{(l)} = \frac{\sqrt{6}}{\sqrt{s^{(l)} + s^{(l+1)}}}$$

in summary:
First, pick a network architecture; choose the layout of your neural network, including how many hidden units in each layer and how many layers in total you want to have.

- Number of input units = dimension of features $x^{(i)}$
- Number of output units = number of classes
- Number of hidden units per layer = usually more the better (must balance with cost of computation as it increases with more hidden units)
- Defaults: 1 hidden layer. If you have more than 1 hidden layer, then it is recommended that you have the same number of units in every hidden layer.

Training a Neural Network

1. Randomly initialize the weights
2. Implement forward propagation to get $h_\Theta(x^{(i)})$ for any $x^{(i)}$
3. Implement the cost function
4. Implement backpropagation to compute partial derivatives
5. Use gradient checking to confirm that your backpropagation works. Then disable gradient checking.

6. Use gradient descent or a built-in optimization function to minimize the cost function with the weights in theta.

This will minimize our cost function. However, keep in mind that $J(\Theta)$ is not always convex and thus we can end up in a local minimum instead.

4 Evaluating a Learning Algorithm

4.1 Concepts

Possible methods to improve the performance of model:

- Getting more training examples.
- Trying smaller sets of features.
- Trying additional features.
- Trying polynomial features.
- Increasing or decreasing λ.

Break down our dataset into the three sets:

- Training set: 60%. A set of examples used for learning the optimal weights with backpropagation.
- Cross validation set: 20%. A set of examples used to tune the hyper parameters including the number of layers and hidden units, and find a stopping point for the backpropagation algorithm to avoid over-fitting.
- Test set: 20%. A set of examples used only to assess the performance of a fully-trained model.

The estimate of the final model on validation data will be biased (smaller than the true error rate) since the validation set is used to select the final model. So we need to use test set to to assess the performance of the final model.

We can now calculate three separate error values for the three different sets using the following method:

1. Optimize the parameters in Θ using the training set for each polynomial degree.
2. Find the polynomial degree d with the least error using the cross validation set.
3. Estimate the generalization error using the test set with $J_{test}(\Theta^{[d]})$.

This way, the degree of the polynomial d has not been trained using the test set.

The test error:

1. For linear regression:

$$J(\Theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\Theta(x^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

$$J_{train}(\Theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\Theta(x^{(i)}) - y^{(i)})^2$$

$$J_{cv}(\Theta) = \frac{1}{2m_{cv}}\sum_{i=1}^{m_{cv}}(h_\Theta(x_{cv}^{(i)}) - y_{cv}^{(i)})^2$$

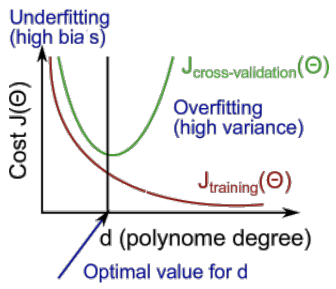
$$J_{test}(\Theta) = \frac{1}{2m_{test}}\sum_{i=1}^{m_{test}}(h_\Theta(x_{test}^{(i)}) - y_{test}^{(i)})^2$$

2. For classification: (misclassification error, aka 0/1 misclassification error)

$$err(h_\Theta(x), y) = \begin{cases} 1, & \text{if } h_\Theta(x) \geq 0.5 \text{ and } y = 0 \text{ or } h_\Theta(x) < 0.5 \text{ and } y = 1 \\ 0, & \text{otherwise} \end{cases}$$

$$\text{test error} = \frac{1}{m_{test}}\sum_{i=1}^{m_{test}} err(h_\Theta(x_{test}^{(i)}), y_{test}^{(i)})$$

Bias and Variance:
The training error will tend to decrease as we increase the degree d of the polynomial. At the same time, the cross validation error will tend to decrease as we increase d up to a point, and then it will increase as d is increased, forming a convex curve.



High bias(underfitting): both $J_{train}(\Theta)$ and $J_{cv}(\Theta)$ will be high. Also, $J_{cv}(\Theta) \approx J_{train}(\Theta)$.
High variance(overfitting): $J_{train}(\Theta)$ will be low and $J_{cv}(\Theta)$ will be high. Also, $J_{cv}(\Theta) \gg J_{train}(\Theta)$.

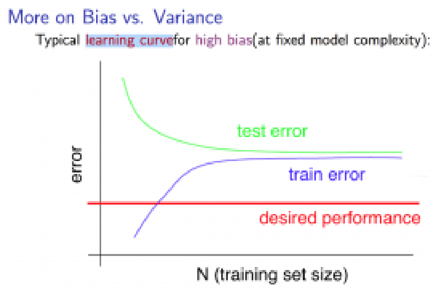
Regularization:
$\lambda = 0$ causes overfitting. λ too large causes underfitting. To find a good λ:

1. Create a list of lambdas (i.e. $\lambda \in \{0, 0.1, 0.2, 0.4, 0.8, ..., 6.4\}$).
2. Create a set of models with different degrees or any other variants.
3. Iterate through the λs and for each λ go through all the models to learn some Θ.
4. Compute the cross validation error using the learned Θ(computed with λ) on the $J_{cv}(\Theta)$.
5. Select the best combo that produces the lowest error on the cross validation set.
6. Using the best combo Θ and λ, apply it on $J_{test}(\Theta)$ to see if it has a good generalization of the problem.

Learning curves:
Training an algorithm on a very few number of data points (such as 1, 2 or 3) will easily have 0 errors because we can always find a quadratic curve that touches exactly those number of points. Hence:

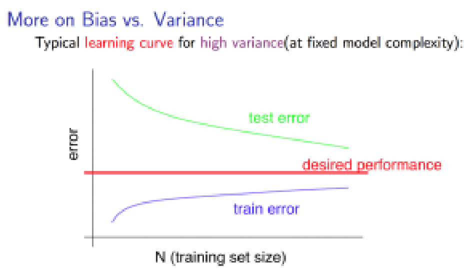
- As the training set gets larger, the error for a quadratic function increases.
- The error value will plateau out after a certain m, or training set size.

When Experiencing high bias:



Low training set size: $J_{train}(\Theta)$ will be low and $J_{cv}(\Theta)$ will be high.
Large training set size: both $J_{train}(\Theta)$ and $J_{cv}(\Theta)$ will be high, with $J_{train}(\Theta) \approx J_{cv}(\Theta)$.
If a learning algorithm is suffering from high bias, getting more training data will not (by itself) help much.

When Experiencing high variance:



Low training set size: $J_{train}(\Theta)$ will be low and $J_{cv}(\Theta)$ will be high.
Large training set size: $J_{train}(\Theta)$ will increase and $J_{cv}(\Theta)$ will decrease, with $J_{train}(\Theta) < J_{cv}(\Theta)$ significantly.
If a learning algorithm is suffering from high variance, getting more training data is likely to help.

Optimizing approaches:

- Getting more training examples: Fixes high variance.
- Trying smaller sets of features: Fixes high variance.
- Adding features: Fixes high bias.
- Adding polynomial features: Fixes high bias.
- Decreasing λ: Fixes high bias.
- Increasing λ: Fixes high variance.

Diagnosing Neural Networks:

- A neural network with fewer parameters is prone to underfitting. It is also computationally cheaper.
- A large neural network with more parameters is prone to overfitting. It is also computationally expensive. In this case you can use regularization (increase λ) to address the overfitting.

Using a single hidden layer is a good starting default. You can train your neural network on a number of hidden layers using your cross validation set. You can then select the one that performs best.

Model Complexity Effects:

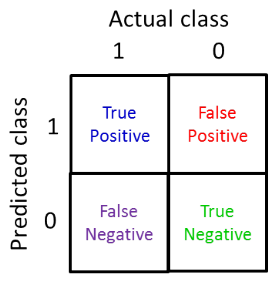
- Lower-order polynomials (low model complexity) have high bias and low variance. In this case, the model fits poorly consistently.
- Higher-order polynomials (high model complexity) fit the training data extremely well and the test data extremely poorly. These have low bias on the training data, but very high variance.

- In reality, we would want to choose a model somewhere in between, that can generalize well but also fits the data reasonably well.

Recommended approach:

- Start with a simple algorithm, implement it quickly, and test it early on your cross validation data.
- Plot learning curves to decide if more data, more features, etc. are likely to help.
- Manually examine the errors on examples in the cross validation set and try to spot a trend where most of the errors were made.

Precision & Recall: (deal with skewed classes)
High $F_{score} \in [0, 1]$ (high precision and high recall) represents a good prediction model.



$$\text{Precision} = \frac{\text{True positives}}{\text{predicted as positive}} = \frac{\text{True positives}}{\text{True positives} + \text{False positives}}$$

$$\text{Recall} = \frac{\text{True positives}}{\text{actual positives}} = \frac{\text{True positives}}{\text{True positives} + \text{False negatives}}$$

$$F_{score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$