Gradient Descent Learning: Explanation and Implementation

Shan He

School for Computational Science
University of Birmingham

Module 06-27818 and 27819:
Introduction to Neural Computation (Level 4/M)
Neural Computation (Level 3)

Outline of Topics

- A tale of two learning algorithms
- Python for neural computation

A tale of two learning algorithms

- In the past 3 weeks, we have learned two learning algorithms for perceptrons
  - Perceptron Learning Rule
  - Gradient Descent Learning

Perceptron Learning Rule: Linear Decision Boundaries

- For simple logic gate problems, what the single layer neural network (perceptron) is doing is to form **decision boundaries** between classes
- Recall for a two dimensional problem, the perceptron output is:

$$y = step(w_1 \cdot x_1 + w_2 \cdot x_2 - \theta)$$

- The decision boundary between $out = 0$ and $out = 1$ is at

$$w_1 \cdot x_1 + w_2 \cdot x_2 - \theta = 0$$

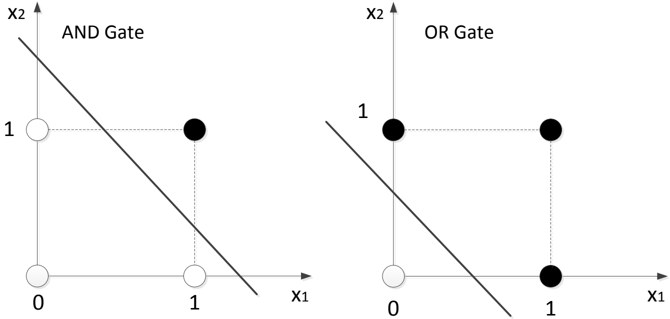
  i.e., along the straight line:

$$x_2 = -\frac{w_1}{w_2} \cdot x_1 + \frac{\theta}{w_2}$$

- So, for perceptrons in two dimensions the decision boundaries are always straight lines

Perceptron Learning

- **Perceptrons learning for classification**: a process of shifting around the decision boundary until each training pattern is classified correctly.



- AND gate: $w_1 = 1, \; w_2 = 1, \; \theta = 1.5$
- OR gate: $w_1 = 1, \; w_2 = 1, \; \theta = 0.5$

Perceptron Learning

- **Perceptrons learning for classification**: a process of shifting around the decision boundary until each training pattern is classified correctly using a step function.
- **Question**: how to formalised "shifting around" into a systematic algorithm that can easily be implemented on a computer.
- **Solutions**: Split up "shifting around" into a number of small steps:
  - If the network weights at time t are $w_{ij}(t)$, then the shifting process corresponds to moving them by a small amount $\Delta w_{ij}(t)$, so that at time $t+1$ we have weights

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t)$$

Formulating the Weight Changes

- Let's do a brute force derivation of such an iterative learning algorithm for simple Perceptrons
- Suppose the target output of unit j is t_j and the actual output is $y_j = step\left(\sum x_i w_{ij}\right)$, where x_i are the inputs.
- We go through all the possibilities
  - $y_j = t_j$
  - $y_j = 1$ and $t_j = 0$
  - $y_j = 0$ and $t_j = 1$
- We then work out an appropriate set of small weight changes for each possibility

Formulating the Weight Changes

- We then work out an appropriate set of small weight changes for each possibility:
  - $y_j = t_j$ $(t_j - y_j = 0)$: do nothing, so

$$w_{ij}(t+1) = w_{ij}(t)$$

  - $y_j = 1$ and $t_j = 0$ $(t_j - y_j = -1\ )$: $\sum x_i w_{ij}$ is too large:
    - when $x_i = 1$, decrease w_{ij}, so

$$w_{ij}(t+1) = w_{ij}(t) - \eta x_i$$

    - when $x_i = 0$, w_{ij} does NOT matter, so

$$w_{ij}(t+1) = w_{ij}(t) - \eta x_i$$

  - $y_j = 0$ and $t_j = 1$ $(t_j - y_j = 1)$: $\sum x_i w_{ij}$ is too small:

Formulating the Weight Changes

- Continue from last slides:
  - $y_j = 0$ and $t_j = 1$ $(t_j - y_j = 1)$: $\sum x_i w_{ij}$ is too small:
    - when $x_i = 1$, increase w_{ij}, so

$$w_{ij}(t+1) = w_{ij}(t) + \eta x_i$$

    - when $x_i = 0$, w_{ij} does NOT matter, so

$$w_{ij}(t+1) = w_{ij}(t) + \eta x_i$$

Reminder: Perceptron Learning Rule

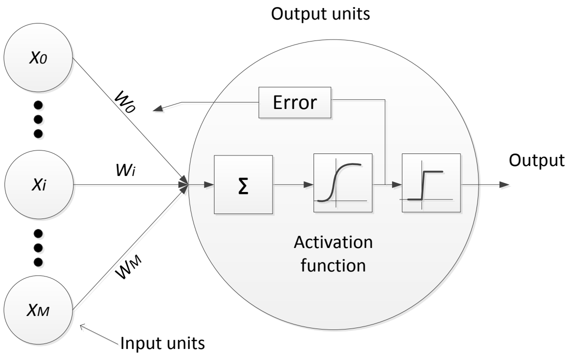
- Shift the linear decision boundary by Perceptron Learning Rule:

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \Delta\mathbf{w}$$

$$\Delta\mathbf{w} = \eta \sum_{p=1}^{P} (t^p - y^p)\mathbf{x}^p$$

Reminder: Gradient Descent Learning

- **Purpose of NN learning**: minimise the output errors on a particular set of training data by adjusting the network weights **w**:



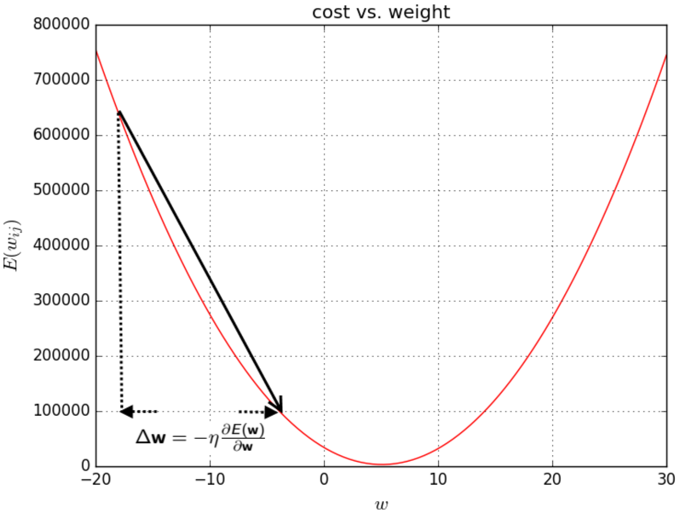
Reminder: Gradient Descent Learning

- **Error or Cost Function** $E(\mathbf{w})$: "measures" how far the current network is from the desired (correctly trained) one.
- **Process to reduce error**:
  - **Gradients**: to obtain direction to move in weight space to reduce the error, which is given by partial derivatives of the error function $\partial E(\mathbf{w})/\partial \mathbf{w}$.
  - **Learning rate** η: specifies the step sizes we take in weight space for each iteration of the weight update equation.
- We keep stepping through weight space until the errors are "small enough":

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \Delta \mathbf{w},$$

where $\Delta \mathbf{w} = -\eta \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}$

Reminder: Gradient Descent Learning



Reminder: Gradient Descent Learning

- For **single layer Perceptrons** (networks), the Gradient Descent Learning algorithms we have derived:
  - Linear regression (with sum squared error cost function and linear activation function):

$$\Delta \mathbf{w} = \eta \sum_{p=1}^{P} (t^p - y^p)\mathbf{x}^p$$

  - Linear separable classification (with cross entropy cost function and logistic activation function):

$$\Delta \mathbf{w} = \eta \sum_{p=1}^{P} (t^p - y^p)\mathbf{x}^p$$

Gradient Desecent Learning vs. Perceptron Learning Rule

- **Question**: Are Gradient Desecent Learning and Perceptron Learning Rule the same???!!!

Gradient Desecent Learning: general update equation

- General update equation:

$$\Delta \mathbf{w}^p = -\eta \frac{\partial E^p(\mathbf{w})}{\partial \mathbf{w}} = -\eta \frac{\partial E^p(\mathbf{w})}{\partial y^p} \frac{\partial y^p}{\partial \mathbf{w}} = -\eta \frac{\partial E^p(\mathbf{w})}{\partial y^p} \frac{\partial f(\mathbf{w}^{\mathrm{T}}\mathbf{x}^p))}{\partial \mathbf{w}}$$

$$= -\eta \frac{\partial E^p(\mathbf{w})}{\partial y^p} \frac{\partial f(\mathbf{w}^{\mathrm{T}}\mathbf{x}^p))}{\partial \mathbf{w}^{\mathrm{T}}\mathbf{x}^p} \frac{\partial \mathbf{w}^{\mathrm{T}}\mathbf{x}^p}{\partial \mathbf{w}} = -\eta \frac{\partial E^p(\mathbf{w})}{\partial y^p} f'(\mathbf{w}^{\mathrm{T}}\mathbf{x}^p)\mathbf{x}^p$$

- For linear regression problem:
  - Activation function is a linear function $f(x) = x$
  - Cost function: Sum Squared Error: $E_{sse}(\mathbf{w}) = \frac{1}{2}\sum_{p=1}^{P}(t^p - y^p)^2$
- For binary classification problem:
  - Activation function: logistic (sigmoid) function $f(x) = \sigma(x)$
  - Cost function: Cross Entropy cost function $E_{ce} = -\sum_p [t^p \log(y^p) + (1 - t^p)\log(1 - y^p)]$

Gradient Desecent Learning vs. Perceptron Learning Rule

- Apart from the fact that Gradient Desecent Learning is more general, there are other differences:
  - **Theoretical starting points**
    - **Perceptron Learning Rule**: considered how we should shift around the decision hyper-planes for **step function outputs**
    - **Gradient Desecent Learning**: emerged from a gradient descent minimisation of some cost function
  - **Convergence**:
    - **Perceptron Learning Rule**: converge to zero error if the problem is linearly separable, but otherwise the weights will keep oscillating (not converging).
    - **Gradient Desecent Learning**: always converge to a set of weights for which the error is a minimum

Installing Python for programming neural networks

- Python: 2.7 or 3.0+? That's a question.
- What are the differences?
- My suggestion: use 2.7. Here is why.
- Download and install the latest Python 2.7.12
- Download and install NumPy
- Download and install matplotlib

A toy example: binary classification

- We generated target classes from two distributions: blue ($t = 1$) and red ($t = 0$).
- Samples (data) from both classes are two dimensional data sampled from their respective distributions, which are two dimensional normal distributions with the same standard deviation but different means:

$$\mathbf{x} = \{x_1, x_2\} \ \sim \ \mathcal{N}_2(\boldsymbol{\mu}_t, \mathbf{\Sigma}),$$

  where $\mathbf{\Sigma} = \{1.2, 1.2\}$
  - Red ($t = 0$): $\boldsymbol{\mu}_1 = \{-1, 0\}$,
  - Blue ($t = 1$): $\boldsymbol{\mu}_2 = \{1, 0\}$,

Solving binary classification problem using ANNs

- Task: build a 2-class classification neural network with two input dimensions

Solving binary classification problem using ANNs

- How to solve this simple binary classification using neural networks.
- We need three things to define a neural networks:
  - **Network topology:** to define how neurons are connected by weights
  - **Activation function:** to convert a neuron's weighted input to its output activation
  - **Learning process:** to update the weights

General Gradient Learning Algorithm

Inputs: Training samples $\{\mathbf{x}, \mathbf{y}\}^p$, where $p = 1, \cdots, P$, $\mathbf{x} \in \mathbb{R}^M$ and $\mathbf{y} \in \mathbb{R}^N$

begin

Set up the network with M input units fully connected to N output units via connections with weights $\mathbf{w}$

Generate random initial weights

repeat

for each training sample p :

Set $\mathbf{w}(t+1) = \mathbf{w}(t) + \eta(t^p - y^p)\mathbf{x}^p$

end for

until $E(\mathbf{w}) < \epsilon$

end

General Gradient Learning Algorithm

Inputs: Training samples $\{\mathbf{x}, \mathbf{y}\}^p$, where $p = 1, \cdots, P$, $\mathbf{x} \in \mathbb{R}^M$ and $\mathbf{y} \in \mathbb{R}^N$

begin

Set up the network with M input units fully connected to N output units via connections with weights $\mathbf{w}$

Generate random initial weights

Set $i = 0$

repeat

$i = i + 1$

for each training sample p :

Set $\mathbf{w}(t+1) = \mathbf{w}(t) + \eta(t^p - y^p)\mathbf{x}^p$

end for

until $i > i_{\text{max}}$

end

Python demonstration

Wait!

- How to solve the Iris classification problem?
- Reminder: three species of Iris:
  - Iris-setosa
  - Iris-virginica
  - Iris-versicolor
- Reminder what we learned: Perceptrons for binary classification problem:
  - **Network topology:** one output nodes
  - **Activation function:** logistic function
  - **Learning process:** adjusting the network weights $\mathbf{w}$ to minimise th **cross entropy cost function**
- We shall see how to solve multiple classes classification problem.