Gradient Descent Learning (I): Regression

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

Linear regression

Solving linear regression problems using ANNs?

Gradient Descent Learning

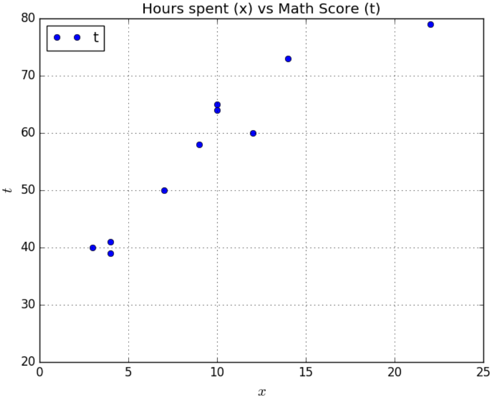
An motivating example

- We collected the the following data of 10 students as they prepared for and took a Math examination:

Student ID	1	2	3	4	5	6	7	8	9	10
Hours spent	4	9	10	14	4	7	12	22	3	10
Math Score	39	58	65	73	41	50	60	79	40	64

- Is there a relationship between Math scores and the number of hours spent studying for the test?
- We can use simple linear regression to reveal the relationship

An motivating example



Linear Regression

- **Linear Regression** (function approximation): approximating an underlying linear function from a set of noisy data
- **Problem definition**: we have P training samples, i.e., $\{\mathbf{x}^p, t^p\}$, $p = 1, \cdots, P$, where
  - $\mathbf{x}^p$: data, which is a M dimensional vector $\mathbf{x}^p = \{x_1^p, x_2^p, \cdots, x_M^p\}^{\mathrm{T}}$, e.g., hours spent $x_1^p = \{4, 9, \cdots 10\}$
  - t^p: corresponding output, e.g., math score
- **Aim**: Approximate a linear regression model to predict the output $y^p = w_0 + \sum_{i=1}^{M} w_i x_i^p$, where w_0 is the the interception and w_i is the weight of the ith dimensional data.
- Let $x_0 = 1$, we have $y^p = w_0 x_0 + \sum_{i=1}^{M} w_i x_i^p = \sum_{i=0}^{M} w_i x_i^p = \mathbf{w}^{\mathrm{T}} \mathbf{x}^p$, where $\mathbf{w}$ is a vector $\mathbf{w} = \{w_0, w_1, \cdots, w_M\}^{\mathrm{T}}$ and $\mathbf{x} = \{x_0, x_1, \cdots, x_M\}^{\mathrm{T}}$

How to estimate linear regression model

- **Ordinary Least Square (OLS)**: the simplest and thus most common estimator
- **Objective function**: minimizes the sum of squared residuals, or sum squared error (SSE):

$$SSE = \frac{1}{2}\sum_{p=1}^{P}(t^p - y^p)^2 = \frac{1}{2}\sum_{p=1}^{P}(t^p - \mathbf{w}^{\mathrm{T}}\mathbf{x}^p)^2$$

- **Optimisation problem**:

$$\hat{\mathbf{w}} = \frac{1}{2}\arg\min_{\mathbf{w}}\sum_{p=1}^{P}(t^p - \mathbf{w}^{\mathrm{T}}\mathbf{x}^p)^2$$

How to estimate linear regression model

- Rewrite the equation in the matrix form and let it be 0:

$$SSE = \frac{1}{2}(\mathbf{t} - \mathbf{w}^{\mathrm{T}}\mathbf{X})^2 = 0,$$

where $\mathbf{t} \in \mathbb{R}^P$ is a vector $\mathbf{t} = \{t^1, t^2, \cdots, t^P\}$ and $\mathbf{X}$ is a matrix $\mathbf{X} \in \mathbb{R}^{M \times P}$

- We can use **matrix pseudo-inversion** to obtain its closed-form solution

$$\hat{\mathbf{w}} = (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{t},$$

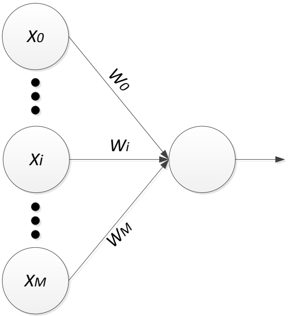
which is a standard and efficient method to solve small OLS problems such as our example.

Solving Linear Regression using ANNs?

- How to solve this simple linear regression 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

Solving linear regression problems using a percetron?

- Network topology: Single Layer Regression Networks



- Activation function: simple linear activation function $f(x) = x$
- The output of a single layer regression network is:

$$y_j^p = \sum_{i=1}^{M} x_i^p w_{ij} - \theta = \sum_{i=0}^{M} x_i^p w_{ij} = \mathbf{w}^{\mathrm{T}} \mathbf{x}^p$$

Learning process of Single Layer Regression Networks

- **Essence of supervised learning process**: adjusting the network weights w_{ij} to minimise a cost function
- **Cost Function:** Sum Squared Error (SSE):

$$SSE = \frac{1}{2}\sum_{p=1}^{P}\sum_{j=1}^{N}(t_j^p - y_j^p)^2 = \frac{1}{2}\sum_{p=1}^{P}\sum_{j=1}^{N}(t_j^p - \mathbf{w}^{\mathrm{T}}\mathbf{x}^p)^2,$$

  where N is the number of output nodes and P is the number of samples
- **Aim**: to minimise the cost function Sum Squared Error (SSE) by updating the weights (learning process):

$$SSE = \frac{1}{2}\sum_{p=1}^{P}\sum_{j=1}^{N}(t_j^p - \mathbf{w}^{\mathrm{T}}\mathbf{x}^p)^2$$

Learning process of Single Layer Regression Networks

- Assuming we have only one output node, we have

$$SSE = \frac{1}{2}\sum_{p=1}^{P}(t^p - y^p)^2 = \frac{1}{2}\sum_{p=1}^{P}(t^p - \mathbf{w}^{\mathrm{T}}\mathbf{x}^p)^2,$$

  which is exactly the same as the objective function of OLS.
- **Question**: Can we use standard matrix pseudo-inversion to obtain weights?

Learning process of Single Layer Regression Networks

- **Question**: Can we use standard matrix pseudo-inversion to obtain weights?
- **Answer**: Yes, but only for **small-scale linear regression problems** (hence small single layer regression networks).
  - **Example**: Suppose we have a problem where the data $\mathbf{X} \in \mathbb{R}^{M \times P}$ is extremely large, e.g., $M = 10,000$ and $P = 10,000$.
  - Using matrix pseudo-invesion to obtain $\hat{\mathbf{w}}$:

$$\hat{\mathbf{w}} = (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{t}$$

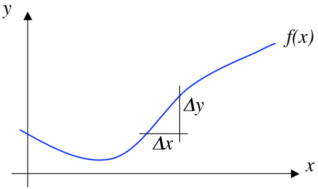
  - Suppose $(\mathbf{X}^{\mathrm{T}}\mathbf{X})$ is a dense matrix, i.e., a matrix with a much higher percentage of nonzero entries, the storage 100,000 by 100,000 element requires 1×10^{10} floating point numbers, approximately 80 GB.
- We need a **general and efficient** learning algorithm

Gradient Descent Learning

- **Aim**: to develop a learning algorithm that minimises a cost function (such as Sum Squared Error) by making appropriate **iterative adjustments** to the weights w_{ij}.
- **Idea**: to apply a series of small updates to the weights $w_{ij}^{t+1} = w_{ij}^{t} + \Delta w_{ij}$ until the cost $E(w_{ij})$ is "small enough".
- **Question**: how to obtain Δw_{ij}

Gradients and Derivatives

- Consider a function $y = f(x)$
- The gradient of $y = f(x)$, at a particular value of x, is the rate of change of $f(x)$ as we change x, and that can be approximated by $\frac{\Delta y}{\Delta x}$ for small Δx, which can be written as:



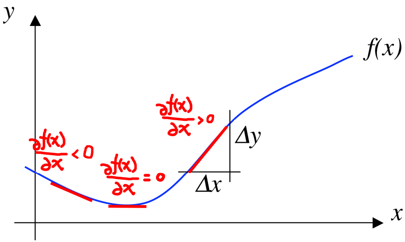
$$\lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

  - The above equation is known as the partial derivative of $f(x)$ with respect to x, i.e., $\frac{\partial f(x)}{\partial x}$, read as "partial $f(x)$ by partial x"

Gradients and Derivatives

Question: how to decrease (minimise) $f(x)$ by chaning x?

- **Observations**: three cases
  - $\frac{\partial f(x)}{\partial x} > 0$: $f(x)$ increases as x increase, so we should decrease x
  - $\frac{\partial f(x)}{\partial x} < 0$: $f(x)$ decreases as x increase, so we should increase x
  - $\frac{\partial f(x)}{\partial x} = 0$: $f(x)$ is at a maximum or minimum



- In summary, we can decrease $f(x)$ by changing x by the amount:

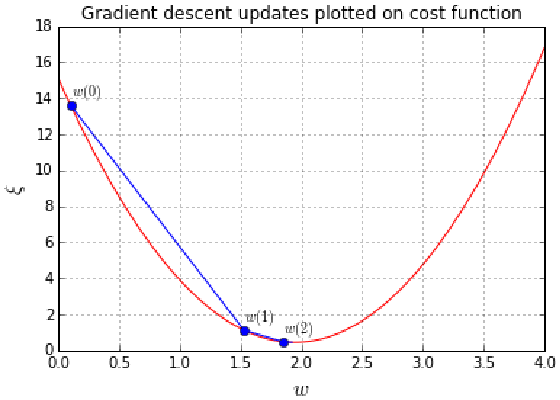
$$\Delta x = x_{new} - x_{old} = -\eta \frac{\partial f(x)}{\partial x},$$

where η is a small positive constant specifying how much we change x by, and the derivative $\frac{\partial f(x)}{\partial x}$ is the direction.

Gradient Descent Learning

- **Aim**: to develop a learning algorithm that minimises a cost function (such as Sum Squared Error) by making appropriate **iterative adjustments** to the weights w_{ij}.
- **Idea**: to apply a series of small updates to the weights $w_{ij}^{t+1} = w_{ij}^{t} + \Delta w_{ij}$ until the cost $E(w_{ij})$ is "small enough".
- **Question**: how to obtain Δw_{ij}
- **Answer**: $\Delta w_{ij} = -\eta \frac{\partial E(w_{ij})}{\partial w_{ij}}$
- **Explanation**: we repeatedly adjust the weights by small steps against the gradient, we will move through weight space, descending along the gradients towards a minimum of the cost function.

Gradient Descent Learning



The Gradient Descent Learning for linear regression

- Recall the liner regression problem, there is only one output, i.e., $N = 1$, so

$$E(\mathbf{w}) = \frac{1}{2}\sum_{p=1}^{P}(t^p - \mathbf{w}^{\mathrm{T}}\mathbf{x}^p)^2$$

- Gradient descent weight updates:

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

The Gradient Descent Learning for Single Layer Regression Networks

- Gradient descent weight updates:

$$\Delta \mathbf{w} = -\eta \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} = -\eta \sum_{p=1}^{P} \frac{\partial \frac{1}{2}(t^p - y^p)^2}{\partial y^p} \cdot \frac{\partial y^p}{\partial \mathbf{w}}$$

$$= \eta \sum_{p=1}^{P} (t^p - y^p) \cdot \frac{\partial \mathbf{w}^{\mathrm{T}} \mathbf{x}^p}{\partial \mathbf{w}} = \eta \sum_{p=1}^{P} (t^p - y^p) \cdot \mathbf{x}^p$$

The Delta Rule for Single Layer Regression Networks

- The gradient for each sample p can be written as:

$$\Delta \mathbf{w} = \eta (t^p - \mathbf{w}^{\mathrm{T}} \mathbf{x}^p) \mathbf{x}^p$$

- In the batch processing, we just add up all the gradients for each sample:

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

Putting them together

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 w_{ij}

Generate random initial weights

repeat

for each training sample p :

For each weight w_{ij}, set $w_{ij}^{t+1} = w_{ij}^t + \eta(t^p - \mathbf{w}^T\mathbf{x}^p)\mathbf{x}^p$

end for

until $E(w_{ij}) < \epsilon$

end

Conclusion

- We learned simple linear regression
- We learned how to compute the gradients/derivatives that enable us to minimise error using efficient general iterative optimisation algorithm, i.e., gradient descent optimisation
- We learned how they how neural network weight learning could be put into the form of minimising an appropriate cost function using gradient descent optimisation
- Reading list
  - Bishop: Sections 3.1, 3.2, 3.3, 3.4, 3.5
  - Haykin-1999: Sections 2.2, 2.4, 3.3, 3.4, 3.5