Deep Learning Chap 6

지동준

Introduction

- Deep Feedforward Network = Feedforward Neural Networks = Multilayer Perceptrons
- To approximate $f^*(x)$, mapping $y = f(x; \Theta)$ and learns value of Θ for best approximation
- Output of model are not fed back to itself
- Associated with a directed acyclic graph (no loop) to represent function composition
- $f(x) = f3(f2(f1(x)))$

Introduction

- Each function is layer
- Training data specify what output layer suppose to produce
- Extending linear model to non linear by considering transformation of data

Gradient Based Learning

- Non linear model causes loss functions to become nonconvex
- For cost functions we use negative log-likelihood (cross entropy loss)
- Often, we want to learn just statistic instead of full probability distribution
- View the cost function as a functional (mapping from function to real numbers)
-

Gradient Based Learning

- Any kind of unit that can be used as an output can be used as hidden unit
- Feedforward network provides set of hidden features and output layer transform these features further.

Output Units

- Linear output layers
- Logistic Sigmoid
- Softmax

Hidden units

- How to choose hidden unit
- Accept a vector of input and transform it.
- Rectified linear units

Rectified Linear Unit

- Max(0,z)
- Consistent derivative

Logistic Sigmoid and Hyperbolic Tangent

- They saturate so not recommended

Architecture Design

- How many units and how these units are connected to each other
- Arrange each layer in chain

Universal Approximation Properties and Depth

- Can Approximate any Borel Measurable Funciton
- Can fail by Optimization Algorithm
- Wrong function as a result of overfitting(?)

- $Y \sim N(x^t\beta(x), 1)\ with\ error\ |y - x^t\check{\beta}(x)|$ then output of neural network
- $Y \sim N(f(x), 1)$ with error $|y - f(x)$ then output of neural network

Backpropagation

- Computing gradient is not difficult
- Can be computationally expensive
- Backpropagation is algorithm to effectively compute gradient
- $\nabla_{\theta} J(\theta)$ is what we want to evaluate
- When would we have multiple output loss?

Computational Graphs

- Each graph node indicates a variable
- Operation is simple function
- Edge represents operation

Symbol to number

- $u^1, \dots, u^{(n)}$ with m inputs
- $\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:child(j)} \frac{\partial u^{(n)}}{\partial u^{(j)}} \frac{\partial u^{(j)}}{\partial u^{(i)}}$ considering computational graph
- We start from n to go down to 1
- The amount of computation required scales linearly with the number of edges
- If we compute naively it, scales exponentially
- Backpropagation avoids this by storing the gradient that is used in calculation.

Symbol to Symbol Derivatives

- Construct another computational graph that represent gradient
- Can evaluate subset of graph using specific numerical values at a later time.

Implementation of general back-propagation

- Each node in the graph corresponds to a variable V
- Software implementation provide both the operations and their bprop methods (so us built in simple operations)
- get_operation(V) : returns operation that computes V(incoming edge)
- Get_consumer(V,G) return child get_inputs(V,G) returns parents
- Op.bprop(inputs,X,G) returns corresponding gradient of operation

Complication

- Return multiple outputs
- Handle various data types
- Large Tensor Product
- It is just one algorithm to evaluate gradient