Deep Feedforward Networks

Han Shao, Hou Pong Chan, and Hongyi Zhang
Deep Feedforward Networks

- Goal: approximate some function $f^*$
  - e.g., a classifier, $y = f^*(x)$ maps input $x$ to a class $y$
  - Defines a mapping $y = f(x; \theta)$ and learns the value $\theta$ that results in the best approximation
Neuron

- Takes $n$ inputs and produce a single output

$$\sum_{i=1}^{n} w_i x_i + b$$
Neuron

\[
\text{output} = \sum \left( w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b \right)
\]

inputs \quad weights \quad sum \quad activation function

\[ g \]
Neuron

output = \sum_{i=1}^{n} x_i w_i
Neuron

output = \sum_{i=1}^{n} x_i w_i + b
Neuron

output = \( g(\sum_{i=1}^{n} x_i w_i + b) \)
Neuron

output = \( g(w^T x + b) \)

\[ x = [x_1, \ldots, x_n]^T \]
\[ w = [w_1, \ldots, w_n]^T \]
Common Activation Functions

\[ g(z) = \tanh(z) \]

\[ g(z) = \sigma(z) = \max\{0, z\} \]

**Sigmoid**
\[
\sigma(z) = \frac{1}{1 + e^{-z}}
\]

**Hyperbolic**
\[ g(z) = \tanh(z) \]

**Rectified-Linear (ReLu)**
\[ g(z) = \max\{0, z\} \]

Figure 6.3: The rectified linear activation function. This activation function is the default activation function recommended for use with most feedforward neural networks. Applying this function to the output of a linear transformation yields a nonlinear transformation. However, the function remains very close to linear, in the sense that it is a piecewise linear function with two linear pieces. Because rectified linear units are nearly linear, they preserve many of the properties that make linear models easy to optimize with gradient-based methods. They also preserve many of the properties that make linear models generalize well. A common principle throughout computer science is that we can build complicated systems from minimal components. Much as a Turing machine’s memory needs only to be able to store 0 or 1 states, we can build a universal function approximator from rectified linear functions.

Figure from Deep Learning Book
Two-layer Neural Networks

- Two-layer neural networks model linear classifiers
- e.g., logistic regression

However, many real-world problems are non-linear!
Example: Learning the XOR

• XOR function:
  • Operation on two binary values, $x_1$ and $x_2$
  • If exactly one of them is 1, returns 1
  • Else, returns 0

• Goal: Learn a function that correctly performs on

$$X = \{ [0, 0]^T, [0, 1]^T, [1, 0]^T, [1, 1]^T \}$$
Example: Learning the XOR

- Cannot use a linear model to fit the data
- Need a three-layer neural network
Example: Learning the XOR

- Define a three-layer neural network (one hidden layer)

\[ h = g(U^T x + c) \]
\[ y = w^T h + b \]

Use ReLu
\[ g(z) = \max\{0, z\} \]

Perform linear regression on the learned space
Example: Learning the XOR

- Can use a linear model to fit the data in the learned space.
Deep Feedforward Network

- Add more hidden layers to build a deep architecture
- The word “deep” means many layers
- Why going “deep”?

![Deep Feedforward Network Diagram](image-url)
Shallow Architecture

• A feedforward network with a single hidden layer can approximate any function

• But the number of hidden units required can be very large

  • $O(N)$ parameters are needed to represent $N$ regions

• e.g., represent the following k-NN classifier
Deep Architecture

- Greater expressive power
  
  A feedforward network with piece-wise linear activation functions (e.g., ReLu) can represent functions with a number of regions that is exponential in the depth of the network [Montufar et al. 2014]

- Better generalization
  
  Empirically results show that greater depth results in better generalization for a wide variety of tasks
Better Generalization with Greater Depth

- Transcribe multi-digit numbers from photographs of addresses [Goodfellow et al. 2014d]

Figure 6.6: Empirical results showing that deeper networks generalize better when used to transcribe multi-digit numbers from photographs of addresses. Data from Goodfellow et al. (2014d). The test set accuracy consistently increases with increasing depth. See figure 6.7 for a control experiment demonstrating that other increases to the model size do not yield the same effect.

Another key consideration of architecture design is exactly how to connect a pair of layers to each other. In the default neural network layer described by a linear transformation via a matrix $W$, every input unit is connected to every output unit. Many specialized networks in the chapters ahead have fewer connections, so that each unit in the input layer is connected to only a small subset of units in the output layer. These strategies for reducing the number of connections reduce the number of parameters and the amount of computation required to evaluate the network, but are often highly problem-dependent. For example, convolutional networks, described in chapter 9, use specialized patterns of sparse connections that are very effective for computer vision problems. In this chapter, it is difficult to give much more specific advice concerning the architecture of a generic neural network. Subsequent chapters develop the particular architectural strategies that have been found to work well for different application domains.
Large Shadow models overfit more

- Transcribe multi-digit numbers from photographs of addresses [Goodfellow et al. 2014d]

![Figure 6.7: Deeper models tend to perform better. This is not merely because the model is larger. This experiment from Goodfellow et al. (2014d) shows that increasing the number of parameters in layers of convolutional networks without increasing their depth is not nearly as effective at increasing test set performance. The legend indicates the depth of network used to make each curve and whether the curve represents variation in the size of the convolutional or the fully connected layers. We observe that shallow models in this context overfit at around 20 million parameters while deep ones can benefit from having over 60 million. This suggests that using a deep model expresses a useful preference over the space of functions the model can learn. Specifically, it expresses a belief that the function should consist of many simpler functions composed together. This could result either in learning a representation that is composed in turn of simpler representations (e.g., corners defined in terms of edges) or in learning a program with sequentially dependent steps (e.g., first locate a set of objects, then segment them from each other, then recognize them).](Figure from Deep Learning Book)
Training

• Commonly used loss functions:

  • Squared loss:  
    \[ l(\theta) = \frac{1}{2} \mathbb{E}_{x,y \sim \hat{P}_{data}} \| x - f(x; \theta) \|^2 \]

    Empirical distribution

  • Cross-entropy loss:  
    \[ l(\theta) = -\mathbb{E}_{x,y \sim \hat{P}_{data}} \log f(x; \theta) \]

    Use it when the output is a probability distribution

• Use gradient-based optimization algorithms to learn the parameters
Output Units

• Suppose the network provides us hidden features $h$

• Linear Units:
  
  • $y = w^T h + b$

  • No activation function

  • Usually used to produce the mean of a conditional Gaussian

  • Do not saturated, good for gradient based algorithm
Output Units

• Sigmoid Units
  
  • \( y = \sigma(\mathbf{w}^T \mathbf{h} + b) \)

  • Usually used to predict a Bernoulli distribution
    
    • e.g., binary classification, output \( P(\text{class} = 1 | x) \)

  • Saturated when \( y \) is close to 1 or 0 because it is exponentiated

  • Should use cross-entropy loss as training loss

\[
l(\theta) = -\mathbb{E}_{x, y \sim \hat{P}_{data}} \log f(x; \theta)
\]

Undergoes the exp in the sigmoid
Output Units

- **Softmax Units**
  - \( y = \text{softmax}(W^T h + b), \ y \in \mathbb{R}^k, \ W \in \mathbb{R}^{d \times k} \)
  - Output a probability distribution over a discrete variable with \( k \) possible values
  - \( \text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \)
  - Softmax is a generalisation of sigmoid
    - Squashes the values of a \( k \)-dimensional vector
    - Suffers from saturation, should use cross-entropy loss
Hidden Units

- Rectified-Linear Units
  - \( h = g(U^T x + c) \)
  - \( g(z) = \max\{0, z\} \)
- Excellent default choices
- The derivative remains 1 whenever the unit is active
- Easy to optimise by gradient-based algorithms
- Drawback: cannot take gradient when activation is 0
Hidden Units

- Generalization of ReLU
  - \( g(\alpha, z) = \max(0, z) + \alpha \min(z, 0) \)

- Leaky ReLu [Maas et al. 2013]
  - Fixes \( \alpha = 0.01 \), \( g(z) = \max(0, z) + 0.01\min(z, 0) \)

- Parametric ReLu [He et al. 2015]
  - Treat \( \alpha \) as a learnable parameter

- Occasionally performs better than ReLu
Hidden Units

- Sigmoid Units
  \[ y = \sigma(U^T x + c) \]
- Hyperbolic Tangent Units
  \[ y = \tanh(U^T x + c) \]
- Both of them have widespread saturation
- Use them as hidden units in feedforward network are discouraged
Demo

- Task - digit recognition (a classification task)
- Dataset - notMNIST

Setup

- Training set - 200000 pics
- Validation set - 10000 pics
- Test set - 18724 pics

Measurement - accuracy