Deep Learning

Deep Feedforward Networks

Lecturer: Duc Dung Nguyen, PhD.
Contact: nddung@hcmut.edu.vn

Faculty of Computer Science and Engineering
Hochiminh city University of Technology

Contents

1. Deep Networks
2. Gradient Based Learning
3. Hidden Units
4. Architecture Design
5. Back-Propagation

Deep Networks

Deep feedforward networks (**multilayer perceptrons (MLPs)**)

- The quintessential deep learning models
- Goal: approximate some function f^*
- Information flow through the function being evaluated
- No feedback connection

Deep Networks: Deep Feedforward Networks

- Extreme importance to machine learning practitioners
- Form the basis of many important commercial applications
- A conceptual stepping stone on the path to recurrent networks

Linear models

- Logistic regression, linear regression
- Can be fit efficiently and reliably
- Can obtain closed form solution or with convex optimization
- Limitation: capacity is limited to linear functions
    - Can not understand the interaction between any two input variables

Gradient Based Learning

Gradient Based Learning

- Training a neural network is not much different from training any other machine learning model with gradient descent
- Cost function: choose how to represent the output of the model

Cost function

- More or less the same as those for other parametric models, such as linear models
- The total cost function used to train a neural network will often combine **one of the primary cost functions** with a **regularization term**

- Most modern neural networks are trained using maximum likelihood
- The cost function is simply the negative log-likelihood, equivalently described as the cross-entropy between the training data and the model distribution.

$$J(\theta) = -\mathbb{E}_{x,y \sim \hat{p}_{\text{data}}} \log p_{\text{model}}(\mathbf{y}|\mathbf{x}). \tag{1}$$

- Cost function changes from model to model, depending on the specific form of $\log p_{\text{model}}$
- If $p_{\text{model}}(\mathbf{y}|\mathbf{x}) = \mathcal{N}(\mathbf{y}; f(\mathbf{x};\theta), \mathbf{I})$ then

$$J(\theta) = \frac{1}{2}\mathbb{E}_{x,y \sim \hat{p}_{\text{data}}} \|\mathbf{y} - f(\mathbf{x};\theta)\|^2 + C$$

Gradient Based Learning: Learning Conditional Statistics

- A sufficiently powerful neural network: be able to represent any function from a wide class of functions
- **Learning**: choosing a function rather than merely choosing a set of parameters
- **Mean squared error** and **mean absolute error** often lead to poor results when used with *gradient-based optimization*
  - Some output units that saturate produce very small gradients when combined with these cost functions.
  - **Cross-entropy** cost function is more popular

1. Linear Units for Gaussian Output Distributions

- Given features h, a layer of linear output units produces a vector $\hat{y} = \mathbf{W}^\top \mathbf{h} + b$
- The mean of a conditional Gaussian distribution:

$$p(y|\mathbf{x}) = \mathcal{N}(y; \hat{y}, \mathbf{I}). \tag{2}$$

- Maximizing the log-likelihood is then equivalent to minimizing the mean squared error

2. Sigmoid Units for Bernoulli Output Distributions

- Task: predicting the value of a binary variable y
- The neural net needs to predict only $P(y = 1|x)$
- **What if:**

$$P(y = 1|x) = \max\left\{0, \min\left\{1, \mathbf{w}^\top \mathbf{h} + b\right\}\right\}. \tag{3}$$

Gradient Based Learning: Output Units

- It is better to ensure that there is always a strong gradient whenever the model has the wrong answer
- Sigmoid output:

$$\hat{y} = \sigma\left(\mathbf{w}^\top \mathbf{h} + b\right) \tag{4}$$

- We may see this output as a combination of linear transformation $z = \mathbf{w}^\top \mathbf{h} + b$ and an activation function $\sigma(z)$

Gradient Based Learning: Output Units

- The sigmoid activation function saturates when z becomes very negative or very positive
- The gradient can shrink too small to be useful for learning
- Maximum likelihood is almost always the preferred approach to training sigmoid output units

3. Softmax Units for Multinoulli Output Distributions

- **Softmax functions**: are most often used as the output of a classifier, to represent the probability distribution over n different classes
- Linear layer predicts unnormalized log probability:

$$\mathbf{z} = \mathbf{W}^\top \mathbf{h} + \mathbf{b} \tag{5}$$

- Softmax function:

$$\text{softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \tag{6}$$

- Many objective functions other than the log-likelihood do not work as well with the softmax function (e.g. squared error)
    - Vanishing gradient

Gradient Based Learning: Output Units

- Log-likelihood can undo the exp of the softmax

$$\log \text{softmax}(\mathbf{z})_i = z_i - \log \sum_j \exp(z_j) \tag{7}$$

- Softmax output is invariant to adding scalar

$$\begin{aligned} \text{softmax}(\mathbf{z}) &= \text{softmax}(\mathbf{z} + c) \\ \text{softmax}(\mathbf{z}) &= \text{softmax}(\mathbf{z} - \max_i z_i) \end{aligned} \tag{8}$$

Hidden Units

Hidden Units

- How to choose the type of hidden units

Hidden Units: ReLU

- **Rectified linear units (ReLU)**: use activation function $g(z) = \max\{0, z\}$
- The gradient is useful for learning (no second-order effect)
- ReLU is typically used on top of an affine transformation

$$\tilde{h} = g(\mathbf{W}^\top \mathbf{x} + \mathbf{b}) \tag{9}$$

- Initialization is important!

- **Drawback**: cannot learned via gradient-based methods on examples for which their activation is 0
- Generalization:

$$h_i = g(\mathbf{z}, \alpha)_i = \max(0, z_i) + \alpha_i \min(0, z_i) \tag{10}$$

  - Absolute value rectification: fix $\alpha_i = -1$
  - Leaky ReLU (Maas et al., 2013)
  - Parametric ReLU (PReLU) (He et al., 2015)

Hidden Units: Maxout units

- **Maxout units (Goodfellow et al., 2013)**: generalize ReLU
  - Divide $\mathbf{z}$ into groups of k values
  - Each maxout unit outputs the maximum element of one of these groups

$$g(\mathbf{z})_i = \max_{j \in \mathbb{G}^{(i)}} z_j \tag{11}$$

  - $\mathbb{G}^{(i)}$ is the set of indices for group i
- A maxout unit can learn a pieacewise linear, convex function with up to k pieces
- *Learning the activation function itself*

Hidden Units: Logistic Sigmoid & Hyperbolic Tangent

- Hyperbolic tangent activation function:

$$g(z) = tanh(z) = 2\sigma(2z) - 1 \tag{12}$$

- The widespread saturation of sigmoid unit can make gradient-based learning very difficult
- Tangent activation function typically performs better than logistic sigmoid (resemble the identity function more closely)

Architecture Design

- The **architecture** refers to the overall structure of the network:
  - How many units it should have
  - How these units should be connected to each other
- Most NN are organized into groups of units called **layers**
- *Chain structure*

- **Linear model**: represent only linear functions.
  - Easy to train: many loss functions result in convex optimization problems when applied to linear models
- **The universal approximation theorem**: regardless of what function we are trying to learn, we know that a large MLP will be able to represent this function.
  - We are not guaranteed that the training algorithm will be able to learn that function

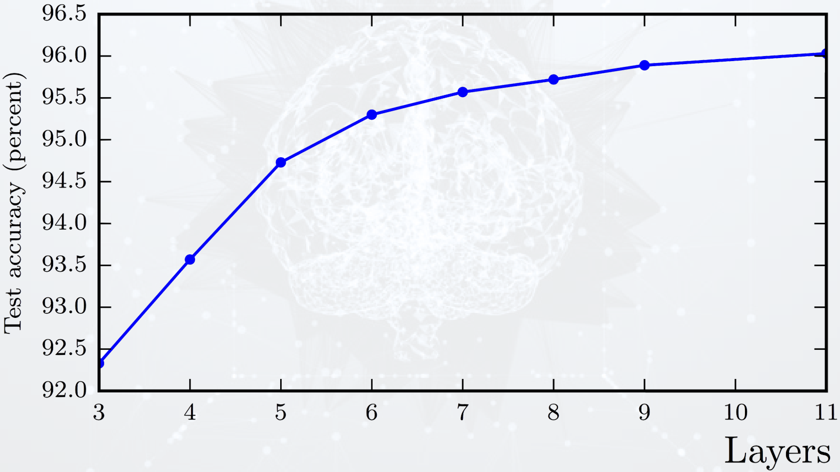
Learning can fail for two different reasons

- The optimization algorithm used for training may not be able to find the value of the parameters that corresponds to the desired function
- The training algorithm might choose the wrong function due to overfitting

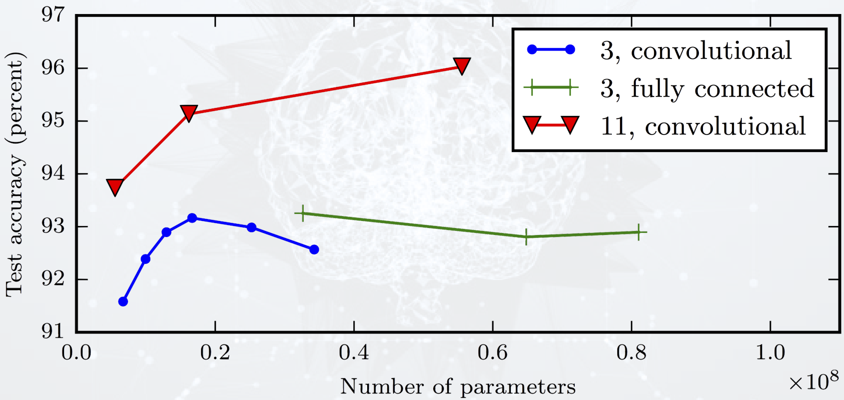
Depth

- **A feedforward network** with a **single layer** is sufficient to represent any function
  - The layer may be **infeasibly large** and may fail to learn and generalize correctly
- Deeper models can *reduce the number of units* required to represent the desired function and can *reduce the amount of generalization error*

Architecture Design: Other Architectural Considerations



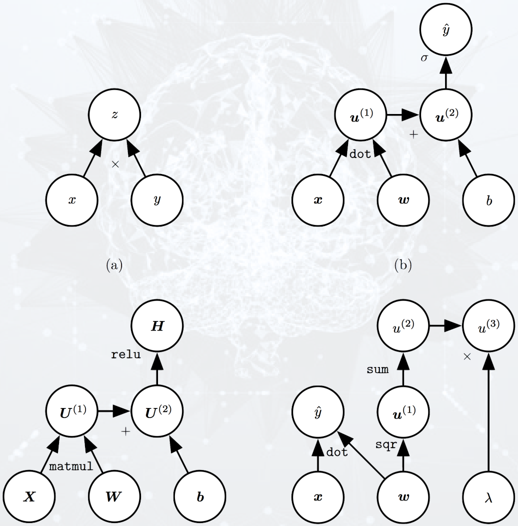
Architecture Design: Other Architectural Considerations



Back-Propagation

- **Feed-forward neural network**: information flows forward through the network.
- **Forward propagation**: the inputs x provide the initial information that then propagates up to the hidden units at each layer and finally produces $\hat{y}$.
- **The back-propagation algorithm**: allows the information from the cost flow backwards through the network, in order to compute the gradient.

Back-Propagation



Chain Rule of Calculus Let x be a real number, and let f and g both be functions mapping from a real number to a real number. Suppose that $y = g(x)$, $z = f(g(x)) = f(y)$.

- **The chain rule**

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} \tag{13}$$

- Generalization: $\mathbf{x} \in \mathbb{R}^m, \mathbf{y} \in \mathbb{R}^n$, $g : \mathbb{R}^m \to \mathbb{R}^n$, and $f : \mathbb{R}^n \to \mathbb{R}$

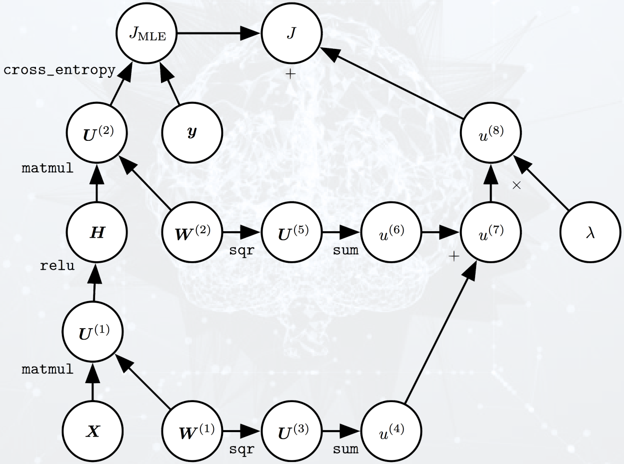
$$\frac{\partial \mathbf{z}}{\partial x_i} = \sum_j \frac{\partial \mathbf{z}}{\partial y_i}\frac{\partial y_i}{\partial x_i} \tag{14}$$

- **Vector notation**:

$$\Delta_x \mathbf{z} = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^\top \Delta_y \mathbf{z} \tag{15}$$

  where $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ is the $n \times m$ Jacobian matrix of g

- The gradient of a variable $\mathbf{x}$ can be obtained by multiplying a Jacobian matrix $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ by a gradient $\Delta_y \mathbf{z}$



J_{MLE}
J
cross_entropy
+
$U^{(2)}$
y
$u^{(8)}$
matmul
×
H
$W^{(2)}$
$U^{(5)}$
$u^{(6)}$
$u^{(7)}$
λ
sqr
sum
+
relu
$U^{(1)}$
matmul
X
$W^{(1)}$
$U^{(3)}$
$u^{(4)}$
sqr
sum