CZ 4042 - Neural Networks

Suyash Lakhotia

AY 17/18, Semester 1

Contents

Chapter 1

Neural Networks Basics

1.1 Biological Neurons

Artificial neural networks are inspired by the biological neural networks in the brain that are made up of billions of basic information-processing units called neurons, which consist of:

- **Soma**: Cell body which processes incoming activations and converts them into output activations.
- **Dendrites**: Receptive zones that receive activation signals from other neurons.
- **Axon**: Transmission lines that send activation signals to other neurons.
- **Synapse**: The connection point between two neurons (presynaptic & postsynaptic). Allow weighted signal transmission through diffusion of chemicals.

Each neuron receives electrochemical inputs from other neurons at the dendrites. The soma sums the incoming signals and if the sum is sufficient enough to activate the neuron, it transmits an electrochemical signal along the axon where it is passed to other neurons whose dendrites may be attached at any of the axon terminals.

The neuron either fires or it doesn't and synapses may be excitatory or inhibitory.

1.2 Artificial Neurons

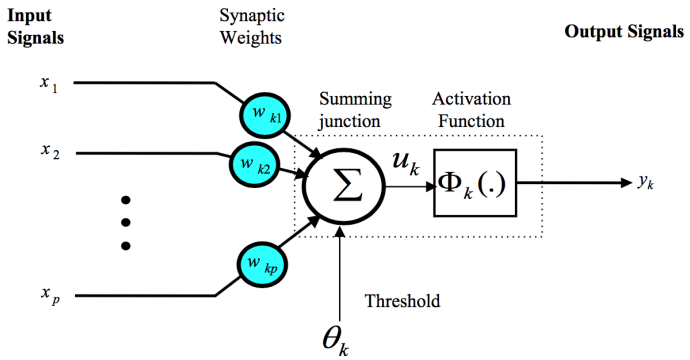

Figure 1.1: Artificial Neuron

An artificial neuron is the basic unit of neural networks. The total synaptic input (u) is given by the sum of the products of the inputs and their corresponding connecting weights minus the threshold (or plus the bias) of the neuron.

$$u = \sum_{i=1}^{n} x_i w_i - \theta = \boldsymbol{x}^T \boldsymbol{w} - \theta$$

The activation function f relates the synaptic input to the activation of the neuron. For a typical artificial neuron, the output y is equal to the result of the activation function (i.e. $f(u)$).

1.2.1 Bias vs. Threshold

The threshold can be considered as the weight for an input of -1. Often, the threshold is represented as a bias that receives a constant input of +1.

$$\text{Threshold: } u = \boldsymbol{x}^T \boldsymbol{w} - \theta$$
$$\text{Bias: } u = \boldsymbol{x}^T \boldsymbol{w} + b$$

1.2.2 Typical ANN Activation Functions

1.2.2.1 Threshold (Unit Step) Activation Function

$$f(u) = \begin{cases} 1 & u > 0 \\ 0 & u \leq 0 \end{cases}$$

1.2.2.2 Linear Activation Function

$$f(u) = \beta u$$

where the slope β is often assumed to be 1.0.

1.2.2.3 Ramp Activation Function

$$f(u) = \max\{0.0, \min\{1.0, u + 0.5\}\}$$

1.2.2.4 Unipolar Sigmoid Activation Function

$$f(u) = \frac{a}{1 + e^{-bu}}$$

where the function ranges from 0 to a and b denotes the slope. When $a = 1.0$ and $b = 1.0$, the unipolar sigmoid function is known as the **logistic function**.

1.2.2.5 Bipolar Sigmoid Activation Function

$$\begin{aligned} f(u) &= a\left(\frac{1 - e^{-bu}}{1 + e^{-bu}}\right) \\ &= a\left(\frac{2}{1 + e^{-bu}} - 1\right) \end{aligned}$$

where the function ranges from $-a$ to $+a$ and b denotes the slope.

Sigmoidal functions are the most pervasive and biologically plausible activation functions. Since sigmoidal functions are differentiable, they lead to mathematically attractive neuronal models.

1.2.2.6 Rectified Linear Unit (ReLU) Activation Function

$$f(u) = \max\{0, u\}$$

ReLU units are increasingly being used to model neurons because of their speed when implementing large networks.

1.3 ANN Architectures

1.3.1 Two-Layer Feedforward Network

- Comprised of an input layer of source units that inject into an output layer of neurons.

1.3.2 Multilayer Feedforward Network

- Comprised of more than one layer of neurons. Layers between input source nodes and output layer are referred to as *hidden layers*.
- Multilayer neural networks can handle more complicated and larger scale problems than single-layer networks.
- However, training a multilayer network may be more difficult and time-consuming.

1.3.3 Recurrent Networks (w/o hidden neurons)

- Recurrent networks consist of a single layer of neurons with each neuron feeding its output signal back to the input layer.
- One unit of delay is assumed during the feedback of output to input.

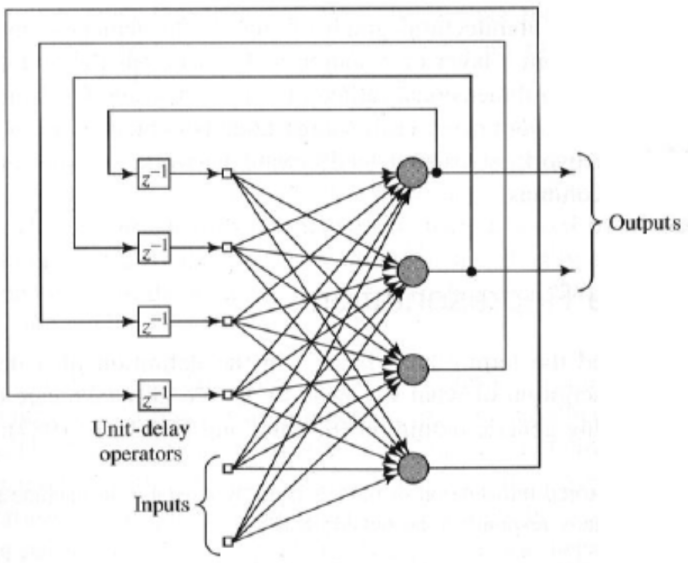

Figure 1.2: Recurrent Network

1.4 ANN Learning

- Neural networks attain their operating characteristics through **learning**.
- During training, the weights or the strengths of connections are gradually adjusted.
- Training may be *supervised* or *unsupervised*.

Supervised Learning: For each training input pattern, the network is presented with the correct target answer (i.e. the desired output).

Unsupervised Learning: For each training input pattern, the network adjusts weights without knowing the correct target. The network self-organizes to classify similar input patterns into clusters.

1.4.1 General Learning Algorithm

If $\boldsymbol{w_k} = (w_{k1}, w_{k2}, \ldots, w_{kn})^T$ is the weight vector of neuron k and the desired output for input $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)^T$ is d, the change Δ of weight can be expressed as:

$$\Delta \boldsymbol{w_k} = \alpha \cdot r(\boldsymbol{w_k}, b_k, \boldsymbol{x}, d)$$

where $r(\boldsymbol{w_k}, b_k, \boldsymbol{x}, d)$ is the learning signal which depends on the current weight vector, bias, input and desired output & α is the learning parameter which spans between 0 and 1. Note that d is zero for unsupervised learning.

Training occurs iteratively for all training patterns and all output nodes until convergence. A cycle of iteration refers to a set of iterations that goes over all the input patterns (i.e. one epoch) and convergence is achieved when there is no change in the values of weights (and bias) with every iteration.

1.4.2 Hebbian Learning Rule

The Hebbian learning rule states that the weight of a connection is altered in proportion to the product of the input and output related to that connection during learning.

So, for a neuron with i connections:

$$\Delta w_i \propto y x_i$$
$$\Delta w_i = \alpha y x_i$$

The Hebbian learning rule is given by:

$$w_i^{new} = w_i^{old} + \alpha y x_i$$
$$b^{new} = b^{old} + \alpha y$$

1.5 Characteristics of ANNs

Neural networks posses many attractive characteristics which surpass some of the limitations in classical information processing systems.

- **Parallel & Distributed Processing:** The information is stored in connections distributed over the network and processed in a large number of neurons connected in parallel. This greatly enhances the speed and efficiency of processing.
- **Adaptiveness:** Neural networks learn from exemplars (training patterns) as they arise in the external world.
- **Generalization:** Networks have the ability to learn the rules and mimic the behavior of a large ensemble of inputs from an adequate set of exemplars.
- **Fault Tolerance:** Since information processing involves a large number of neurons and connections, the loss of a few connections does not necessarily affect the overall performance.
- **Ease of Construction:** A short development time is required to implement a neural-based solution for complex problems.

1.6 ANN Applications

- Optical Character Recognition
- Computer Vision: Object / Face Recognition, Motion Detection
- Pattern Recognition: Defect Detection
- Engineering Process Control

Chapter 2

Perceptron

Fundamental Objective: To provide meaningful categorization for some input information.

A typical pattern recognition system is comprised of two stages:

1. Feature Extraction
2. Classification / Clustering

2.1 Linear Classifier

A linear classifier implements a discriminant function or decision boundary represented by a straight line (hyperplane) in the multidimensional feature space.

Given an input (features), $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)^T$, the decision boundary of a linear classifier is given by a linear discriminant function:

$$g(\boldsymbol{x}) = w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n$$

where $\boldsymbol{w} = (w_1, w_2, \ldots, w_n)^T$ are the coefficients or weights and w_0 is the constant term.

The decision boundary is given by $g(\boldsymbol{x}) = 0$. The challenge is to determine a suitable weight vector that gives the correct results for all inputs from the two classes.

2.1.1 Linear Neuron

$$g(\boldsymbol{x}) = w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n = 0$$
$$w_0 + \boldsymbol{x}^T \boldsymbol{w} = 0$$

Therefore, a two-class linear classifier can be implemented with an artificial neuron that has a threshold activation function.

$$g(\boldsymbol{x}) = \boldsymbol{x}^T \boldsymbol{w} + b$$

Then,

$$g(\boldsymbol{x}) > 0 \implies y = 1 \implies \text{Class 1}$$
$$g(\boldsymbol{x}) \leq 0 \implies y = 0 \implies \text{Class 2}$$

Therefore, the output of the linear neuron represents the label of the class. The bias is needed so that the discriminant boundary does not always go through origin.

2.1.2 Weight Vector

At the decision boundary,

$$g(\boldsymbol{x}) = \boldsymbol{x}^T \boldsymbol{w} + b = \boldsymbol{0}$$

the weight vector is normal to the decision boundary because the dot product for two orthogonal vectors is always zero.

Proof: Considering two points, $\boldsymbol{x_1}$ and $\boldsymbol{x_2}$, on the decision boundary:

$$\boldsymbol{x_1}^T \boldsymbol{w} + b = \boldsymbol{x_2}^T \boldsymbol{w} + b = 0$$
$$\therefore (\boldsymbol{x_1} - \boldsymbol{x_2})^T \boldsymbol{w} = 0$$

Since $(\boldsymbol{x_1} - \boldsymbol{x_2})$ represents the decision boundary and the dot product with $\boldsymbol{w}$ is zero, they are orthogonal.

2.2 Simple (or Discrete) Perceptron

- The simple (or discrete) perceptron is a neuron that has a threshold activation function.
- It classifies input patterns into two (linearly separable) classes.
- It acts as a two-class classifier or dichotomizer.

2.2.1 Simple Perceptron Learning Algorithm

The perceptron learning algorithm is a form of supervised learning.

Assuming that there are P training pairs of the form $(\boldsymbol{x}_p, d_p)$ in the training dataset, where $\boldsymbol{x}_p \in \boldsymbol{R}^n$ is the n-dimensional input and $d_p \in \{0, 1\}$ is the binary desired output of the pth training pattern.

Initially, the weight vector and bias are initialized to small random values and the inputs are presented to the network in a sequence from $\boldsymbol{x}_1$ to $\boldsymbol{x}_P$. The weights and bias are updated for every presentation of an input pattern (if $y_p \neq d_p$) and the training iterations are continued until convergence is achieved. Note that in every epoch, the sequence in which the training patterns will be presented will be randomized.

```
if (y_p == 1 && d_p == 0) {
    w = w - x_p
    b = b - 1
} else if (y_p == 0 && d_p == 1) {
    w = w + x_p
    b = b + 1
}
```

2.2.1.1 Convergence

Convergence is achieved when:

- all outputs of training patterns equal their desired (target) outputs, or
- the weights and biases converge to stable values, or
- the output for every input does not change

If the patterns are linearly separable, the perceptron is able to separate the patterns correctly after convergence.

2.2.2 Modified Simple Perceptron Learning Algorithm

The weights change proportional to the difference between the desired output d and perceptron output y.

For training pattern $(\boldsymbol{x}_p, d_p)$:

$$u_p = \boldsymbol{x}_p^T \boldsymbol{w} + b$$
$$y_p = f(u_p)$$
$$\delta_p = d_p - y_p$$

Learning:

$$\boldsymbol{w} = \boldsymbol{w} + \alpha_1 \delta_p \boldsymbol{x}_p$$
$$b = b + \alpha_1 \delta_p$$

The learning rate $\alpha_1 \in (0.0, 1.0]$ slows down the change in the weights, hence, taking smaller steps towards a stable solution.

Note that $\delta_p \in \{-1, 0, +1\}$ and the simple perceptron algorithm corresponds to the case when $\alpha_1 = 1.0$.

2.3 Continuous Perceptron

A continuous perceptron has a sigmoidal activation function.

Assuming the training dataset $(\boldsymbol{x}_p, d_p)_{p=1}^P$ where $\boldsymbol{x}_p = (x_{p1}, x_{p2}, \ldots, x_{pn})^T \in \boldsymbol{R}^n$ and $d_p \in \boldsymbol{R}$, the continuous perceptron finds the following functional mapping through learning:

$$\phi : \boldsymbol{R}^n \to \boldsymbol{R}$$

The adjustment of the weight vector is resolved by minimizing an error (cost) function by using a gradient descent approach.

2.3.1 Gradient Descent Learning

The cost function $J(\boldsymbol{w}, b)$ of an artificial neuron is typically a multi-dimensional function that depends on the weight vector and bias. The gradient descent approach states that the value of $\boldsymbol{w}$ and b is updated during learning by searching in the direction of and proportional to the negative gradient of the cost function.

$$\Delta \boldsymbol{w} \propto -\frac{\partial J(\boldsymbol{w}, b)}{\partial \boldsymbol{w}}$$
$$\Delta \boldsymbol{w} = -\alpha \frac{\partial J(\boldsymbol{w}, b)}{\partial \boldsymbol{w}}$$

where α is the gradient descent learning rate.

Therefore, the gradient descent learning equations are:

$$\boldsymbol{w} = \boldsymbol{w} - \alpha \frac{\partial J(\boldsymbol{w}, b)}{\partial \boldsymbol{w}}$$
$$b = b - \alpha \frac{\partial J(\boldsymbol{w}, b)}{\partial \boldsymbol{w}}$$

The error function J for the continuous function is defined as the **mean squared error** of the outputs.

$$J(\boldsymbol{w}, b) = \frac{1}{P} \sum_{p=1}^{P} (d_p - y_p)^2$$

The perceptron learning algorithm aims to find $\boldsymbol{w}^*$ such that:

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} J(\boldsymbol{w}, b)$$

The partial derivative is derived as follows (complete derivation in Lecture Notes):

$$\begin{aligned}\frac{\partial J(\boldsymbol{w},b)}{\partial \boldsymbol{w}} &= -\frac{2}{P}\sum_{p=1}^{P}(d_p - y_p)\frac{\partial y_p}{\partial \boldsymbol{w}} \\ &= -\frac{2}{P}\sum_{p=1}^{P}(d_p - y_p)\frac{\partial y_p}{\partial u_p}\frac{\partial u_p}{\partial \boldsymbol{w}} \\ &= -\frac{2}{P}\sum_{p=1}^{P}(d_p - y_p)f'(u_p)\boldsymbol{x}_p \\ &= -\frac{2}{P}((d_1 - y_1)f'(u_1)\boldsymbol{x}_1 + \cdots + (d_P - y_P)f'(u_P)\boldsymbol{x}_P) \\ &= -\frac{2}{P}\begin{pmatrix}\boldsymbol{x}_1 & \cdots & \boldsymbol{x}_P\end{pmatrix}\begin{pmatrix}(d_1 - y_1)f'(u_1) \\ \vdots \\ (d_P - y_P)f'(u_P)\end{pmatrix}\end{aligned}$$

Substituting the following values:

$$\boldsymbol{d} = \begin{pmatrix}d_1 \\ d_2 \\ \vdots \\ d_P\end{pmatrix} \quad \boldsymbol{y} = \begin{pmatrix}y_1 \\ y_2 \\ \vdots \\ y_P\end{pmatrix} \quad f(\boldsymbol{u}) = \begin{pmatrix}f(u_1) \\ f(u_2) \\ \vdots \\ f(u_P)\end{pmatrix} \quad \boldsymbol{X} = \begin{pmatrix}\boldsymbol{x}_1^T \\ \boldsymbol{x}_2^T \\ \vdots \\ \boldsymbol{x}_P^T\end{pmatrix}$$

$$\begin{aligned}\frac{\partial J(\boldsymbol{w},b)}{\partial \boldsymbol{w}} &= -\frac{2}{P}\boldsymbol{X}^T(\boldsymbol{d} - \boldsymbol{y}) \cdot f'(\boldsymbol{u}) \\ &= -\frac{2}{P}\boldsymbol{X}^T\boldsymbol{\delta}\end{aligned}$$

Substituting into the gradient descent learning equation:

$$\boldsymbol{w} = \boldsymbol{w} + \alpha_1 \boldsymbol{X}^T\boldsymbol{\delta}$$

where $\alpha_1 = \frac{2\alpha}{P}$ and the error term $\boldsymbol{\delta} = (\boldsymbol{d} - \boldsymbol{y}) \cdot f'(\boldsymbol{u})$.

Similarly for bias:

$$b = b + \alpha_1 \mathbf{1}_P{}^T\boldsymbol{\delta}$$

where $\mathbf{1}_P = (1 \quad 1 \quad \cdots \quad 1)^T$.

2.3.2 Delta Learning Equations

The below learning equations are known as the delta learning or least mean squares (LMS) learning equations:

$$\begin{aligned}\boldsymbol{\delta} &= (\boldsymbol{d} - \boldsymbol{y}) \cdot f'(\boldsymbol{u}) \\ \boldsymbol{w} &= \boldsymbol{w} + \alpha_1 \boldsymbol{X}^T\boldsymbol{\delta} \\ b &= b + \alpha_1 \mathbf{1}_P{}^T\boldsymbol{\delta}\end{aligned}$$

where α_1 is the delta learning rate.

Since all the training patterns are presented in a batch in one epoch, this learning is known as *batch learning*. Note the synaptic input to the layer for a batch of P patterns:

$$\boldsymbol{u} = \begin{pmatrix}u_1 \\ u_2 \\ \vdots \\ u_P\end{pmatrix} = \begin{pmatrix}\boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \vdots \\ \boldsymbol{x}_P\end{pmatrix}\boldsymbol{w} + b\begin{pmatrix}1 \\ 1 \\ \vdots \\ 1\end{pmatrix} = \boldsymbol{X}\boldsymbol{w} + b\mathbf{1}_P$$

where $\boldsymbol{X}$ is the data matrix where each data point is represented as a row.

2.3.3 Derivatives of Sigmoid Functions

Logistic Function:

$$\begin{aligned}y = f(u) &= \frac{1}{1 + e^{-u}} \\ f'(u) &= y(1 - y)\end{aligned}$$

Bipolar Sigmoidal:

$$\begin{aligned}y = f(u) &= \frac{1 - e^{-u}}{1 + e^{-u}} \\ f'(u) &= \frac{1}{2}(1 - y^2)\end{aligned}$$

2.3.4 Stochastic Gradient Descent

In (batch) gradient descent, in one epoch, all the patterns are applied in a batch and the weights and biases are updated at once. In stochastic gradient descent (SGD), training patterns are applied to the perceptron in a sequence in one epoch and the weights and bias are updated after presenting every individual pattern.

GD	SGD
$(\boldsymbol{X}, \boldsymbol{d})$	$(\boldsymbol{x}_p, d_p)$
$J(\boldsymbol{w}, b) = \frac{1}{P}\sum_{p=1}^{P}(d_p - y_p)^2$	$J(\boldsymbol{w}, b) = (d_p - y_p)^2$
$\boldsymbol{u} = \boldsymbol{X}\boldsymbol{w} + b\mathbf{1}_P$	$u_p = \boldsymbol{x}_p{}^T\boldsymbol{w} + b$
$\boldsymbol{y} = f(\boldsymbol{u})$	$y_p = f(u_p)$
$\boldsymbol{\delta} = (\boldsymbol{d} - \boldsymbol{y}) \cdot f'(\boldsymbol{u})$	$\delta_p = (d_p - y_p)f'(u_p)$
$\boldsymbol{w} = \boldsymbol{w} + \alpha_1 \boldsymbol{X}^T\boldsymbol{\delta}$	$\boldsymbol{w} = \boldsymbol{w} + \alpha_1 \delta_p \boldsymbol{x}_p$
$b = b + \alpha_1 \mathbf{1}_P{}^T\boldsymbol{\delta}$	$b = b + \alpha_1 \delta_p$

Figure 2.1: Stochastic Gradient Descent

2.3.5 Converging Speed & Values

- At a higher learning rate, the number of epochs required for convergence decreases but the convergence may not be stable. A higher learning rate is possible with GD.
- The time for one add/multiply computation is less when patterns are trained in a batch. Usually, time for a weight update versus batch size takes a U-shape curve.
- The optimal learning rate is the largest rate at which learning does not diverge. The optimum learning speed depends on the *learning rate & batch size*.
- In practice, *mini-batch GD* is used.

2.4 Limitations of Perceptrons

- Linear Separability:
  - Linear separability requires that the patterns to be classified must be sufficiently separated from each other to ensure that the decision boundaries are hyperplanes.
  - As long as an ANN consists of a linear combiner followed by a non-linear element, then regardless of the form of non-linearity used, a perceptron can only perform pattern classification on linearly separable patterns. Therefore, more complex functions cannot be approximated by a perceptron.
- Local Minima Problem:
  - The learning algorithm may get stuck in a local minimum of the error function depending on the initial weights.

Chapter 3

Regression

3.1 Linear Regression w/ Linear Neuron

In linear regression, the output (dependent) variable is predicted as a linear combination of input (independent) variables.

Given input $\boldsymbol{x} \in \boldsymbol{R}^n$ and output $y \in \boldsymbol{R}$, linear regression states that:

$$y = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$$

Therefore, linear regression can be implemented with a linear neuron (where $y = f(u) = u$) having a weight vector $\boldsymbol{w} = (w_1 \quad w_2 \quad \cdots w_n)$ and bias b. Then:

$$y = \boldsymbol{w}^T \boldsymbol{x} + b$$

Given a training dataset $\{(\boldsymbol{x}_p, d_p)\}_{p=1}^P$ where $\boldsymbol{x}_p \in \boldsymbol{R}^n$ and $d_p \in \boldsymbol{R}$, a linear neuron finds the linear regression function ϕ by learning from the training data.

$$\phi : \boldsymbol{R}^n \to \boldsymbol{R}$$

3.1.1 GD for Linear Neuron

For a perceptron, the error term is $\boldsymbol{\delta} = (\boldsymbol{d} - \boldsymbol{y}) \cdot f'(u)$. However, for a linear neuron, $f(u) = u$ and $f'(u) = 1$. Therefore, the delta learning equations are:

$$\begin{aligned} \boldsymbol{y} &= \boldsymbol{X}\boldsymbol{w} + b\boldsymbol{1}_P \\ \boldsymbol{\delta} &= \boldsymbol{d} - \boldsymbol{y} \\ \boldsymbol{w} &= \boldsymbol{w} + \alpha_1 \boldsymbol{X}^T \boldsymbol{\delta} \\ b &= b + \alpha_1 \boldsymbol{1}_P{}^T \boldsymbol{\delta} \end{aligned}$$

where $\alpha_1 = \frac{2\alpha}{P}$.

3.1.2 SGD for Linear Neuron

For every training pattern:

$$\begin{aligned} y_p &= \boldsymbol{x}_p{}^T \boldsymbol{w} + b \\ \delta_p &= d_p - y_p \\ \boldsymbol{w} &= \boldsymbol{w} + \alpha_1 \delta_p \boldsymbol{x}_p \\ b &= b + \alpha_1 \delta_p \end{aligned}$$

where $\alpha_1 = 2\alpha$.

3.2 Logistic Regression

A neuron performing logistic regression classifies inputs into two classes ('0' or '1'). The activation (logistic function) gives the probability of the neuron output belonging to class '1':

$$\begin{aligned} P(y = 1|\boldsymbol{x}) = f(u) &= \frac{1}{1 + e^{-u}} \\ P(y = 0|\boldsymbol{x}) = 1 - P(y = 1|\boldsymbol{x}) &= 1 - f(u) \end{aligned}$$

where $u = \boldsymbol{w}^T \boldsymbol{x} + b$.

The output of the neuron:

$$y = \begin{cases} 1 & \text{if } f(u) = P(y = 1|\boldsymbol{x}) > 0.5 \\ 0 & \text{otherwise} \end{cases}$$

Given a training dataset $\{(\boldsymbol{x}_p, d_p)\}_{p=1}^P$ where $\boldsymbol{x}_p \in \boldsymbol{R}^n$ and $d_p \in \{0, 1\}$, the cost function for logistic regression is given by the **cross-entropy** (or negative log-likelihood) over all the training patterns:

$$J(\boldsymbol{w}, b) = -\sum_{p=1}^{P} d_p \log\big(f(u_p)\big) + (1 - d_p) \log\big(1 - f(u_p)\big)$$

where $u = \boldsymbol{x}_p{}^T \boldsymbol{w} + b$.

3.2.1 GD for Logistic Regression

The cost function J is minimized using the gradient descent algorithm. The partial derivative is derived as follows (complete derivation in Lecture Notes):

$$\begin{aligned}
\frac{\partial J(\boldsymbol{w},b)}{\partial \boldsymbol{w}} &= -\sum_{p=1}^{P}\left(\frac{d_p}{f(u_p)} - \frac{(1-d_p)}{1-f(u_p)}\right)\frac{\partial f(u_p)}{\partial \boldsymbol{w}} \\
&= -\sum_{p=1}^{P}\left(\frac{d_p - f(u_p)}{f(u_p)(1-f(u_p))}\right)\frac{\partial f(u_p)}{\partial u_p}\frac{\partial u_p}{\partial \boldsymbol{w}} \\
&= -\sum_{p=1}^{P}\left(\frac{d_p - f(u_p)}{f(u_p)(1-f(u_p))}\right) f(u_p)\big(1-f(u_p)\big)\boldsymbol{x}_p \\
&= -\sum_{p=1}^{P}\big(d_p - f(u_p)\big)\boldsymbol{x}_p \\
&= -\big((d_1 - f(u_1))\boldsymbol{x}_1 + \cdots + (d_P - f(u_P))\boldsymbol{x}_P\big) \\
&= -\begin{pmatrix}\boldsymbol{x}_1 & \cdots & \boldsymbol{x}_P\end{pmatrix}\begin{pmatrix} d_1 - f(u_1) \\ \vdots \\ d_P - f(u_P)\end{pmatrix} \\
&= -\boldsymbol{X}^T\big(\boldsymbol{d} - f(\boldsymbol{u})\big) \\
&= -\boldsymbol{X}^T\boldsymbol{\delta}
\end{aligned}$$

where $\boldsymbol{\delta} = \boldsymbol{d} - f(\boldsymbol{u})$ and,

$$\boldsymbol{d} = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_P\end{pmatrix} \quad f(\boldsymbol{u}) = \begin{pmatrix} f(u_1) \\ f(u_2) \\ \vdots \\ f(u_P)\end{pmatrix}$$

3.2.1.1 Learning Equations

$$\begin{aligned}
\boldsymbol{w} &= \boldsymbol{w} - \alpha\frac{\partial J(\boldsymbol{w},b)}{\partial \boldsymbol{w}} \\
\boldsymbol{w} &= \boldsymbol{w} + \alpha_1\boldsymbol{X}^T\boldsymbol{\delta} \\
b &= b + \alpha_1\boldsymbol{1}_P{}^T\boldsymbol{\delta}
\end{aligned}$$

where $\boldsymbol{\delta} = \boldsymbol{d} - f(\boldsymbol{u})$ and $\alpha_1 = \alpha$ for a logistic neuron. Note that $\boldsymbol{y}$ in the linear regression neuron equations is now replaced with $f(\boldsymbol{u})$ for logistic regression.

3.2.2 Indicator Function

Using the *indicator function* $1(\cdot)$:

$$y = 1(f(u) > 0.5)$$

where:

$$1(a) = \begin{cases} 1 & \text{if } a \text{ is True} \\ 0 & \text{if } a \text{ is False}\end{cases}$$

3.2.3 SGD for Logistic Regression

GD	SGD
$(\boldsymbol{X}, \boldsymbol{d})$	$(\boldsymbol{x}_p, d_p)$
$J(\boldsymbol{w},b) = -\sum_{p=1}^{P} d_p\, log\big(f(u_p)\big) + (1-d_p) log\big(1-f(u_p)\big)$	$J(\boldsymbol{w},b) = -d_p log\big(f(u_p)\big) - (1-d_p) log\big(1-f(u_p)\big)$
$\boldsymbol{u} = \boldsymbol{X}\boldsymbol{w} + b\boldsymbol{1}_P$	$u_p = \boldsymbol{x}_p{}^T\boldsymbol{w} + b$
$f(\boldsymbol{u}) = \frac{1}{1+e^{-\boldsymbol{u}}}$	$f(u_p) = \frac{1}{1+e^{-u_p}}$
$\boldsymbol{\delta} = \boldsymbol{d} - f(\boldsymbol{u})$	$\delta_p = d_p - f(u_p)$
$\boldsymbol{w} = \boldsymbol{w} + \alpha_1\boldsymbol{X}^T\boldsymbol{\delta}$	$\boldsymbol{w} = \boldsymbol{w} + \alpha_1\delta_p\boldsymbol{x}_p$
$b = b + \alpha_1\boldsymbol{1}_P{}^T\boldsymbol{\delta}$	$b = b + \alpha_1\delta_p$

Figure 3.1: Stochastic Gradient Descent

3.3 Neuron GD Summary

$$\begin{aligned}
&(\boldsymbol{X}, \boldsymbol{d}) \\
&\boldsymbol{u} = \boldsymbol{X}\boldsymbol{w} + b\boldsymbol{1}_P \\
&\boldsymbol{w} = \boldsymbol{w} + \alpha_1\boldsymbol{X}^T\boldsymbol{\delta} \\
&b = b + \alpha_1\boldsymbol{1}_P{}^T\boldsymbol{\delta}
\end{aligned}$$

neuron	$f(\boldsymbol{u}), \boldsymbol{y}$	$\boldsymbol{\delta}$
Discrete perceptron	$\boldsymbol{y} = f(\boldsymbol{u}) = \begin{cases}1, & \boldsymbol{u} > 0 \\ 0, & \boldsymbol{u} \le 0\end{cases}$	$\boldsymbol{d} - \boldsymbol{y}$
Logistic regression neuron	$f(\boldsymbol{u}) = \frac{1}{1+e^{-\boldsymbol{u}}}$ $\boldsymbol{y} = \begin{cases}1, & f(\boldsymbol{u}) > 0.5 \\ 0, & f(\boldsymbol{u}) \le 0.5\end{cases}$	$\boldsymbol{d} - f(\boldsymbol{u})$
Linear neuron	$\boldsymbol{y} = f(\boldsymbol{u}) = \boldsymbol{u}$	$\boldsymbol{d} - \boldsymbol{y}$
Continuous perceptron	$\boldsymbol{y} = f(\boldsymbol{u}) = \frac{1}{1+e^{-\boldsymbol{u}}}$	$(\boldsymbol{d} - \boldsymbol{y}) \cdot f'(\boldsymbol{u})$

Figure 3.2: Neuron Gradient Descent Summary

Chapter 4

Neuron Layers

4.1 Softmax Regression

Softmax regression is an extension of logistic regression for multi-class classification problems. It is also known as multinomial logistic regression.

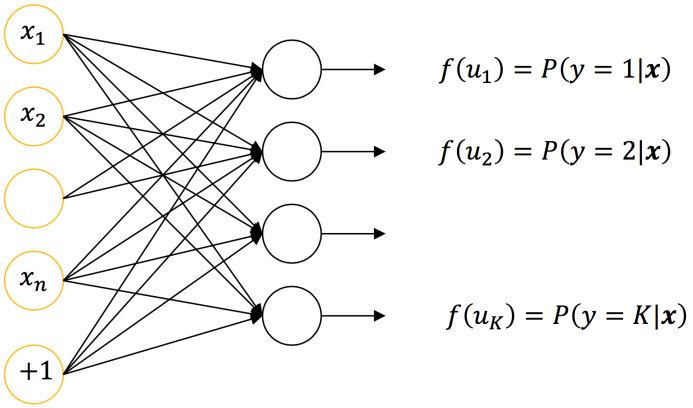

Figure 4.1: Softmax

The softmax regression layer consists of a layer of neurons, each corresponding to one class label. The activation $f(u_k)$ of neuron k gives the probability $P(y = k|\boldsymbol{x})$ and the output y gives class label k.

$$y = \underset{k}{\text{argmax}}\ P(y = k|\boldsymbol{x})$$

The activation of each neuron k estimates the probability that $\boldsymbol{x}$ belongs to the class k (i.e. softmax activation function):

$$P(y = k|\boldsymbol{x}) = f(u_k) = \frac{e^{(u_k)}}{\sum_{k=1}^{K} e^{(u_k)}}$$

where $u_k = \boldsymbol{x}^T \boldsymbol{w}_k + b_k$.

4.1.1 Maximum Synaptic Input

If $u_{\max} = \max_k u_k$ corresponds to the maximum activation, the activation of the neurons can be expressed as:

$$\begin{aligned} P(y = k|\boldsymbol{x}) &= f(u_k - u_{\max}) \\ &= \frac{e^{(u_k - u_{\max})}}{\sum_{k=1}^{K} e^{(u_k - u_{\max})}} \\ &= f(u_k) \end{aligned}$$

Then, if neuron $k^* = \text{argmax}\ u_k$,

$$f(u_k) = \frac{e^{(u_k - u_{\max})}}{1 + \sum_{k=1, k \neq k^*}^{K} e^{(u_k - u_{\max})}}$$

That is, the activation function is given by a logistic function if the synaptic input is subtracted by the maximum synaptic input:

$$u_k = u_k - u_{\max}$$

This can be done for computational stability.

4.1.2 Synaptic Input to Softmax Layer

Synaptic input to kth neuron for input pattern $\boldsymbol{x}_p$:

$$u_{pk} = {\boldsymbol{x}_p}^T \boldsymbol{w}_k + b_k$$

Synaptic input to kth neuron for batch of P patterns:

$$\boldsymbol{u}_k = \boldsymbol{X} \boldsymbol{w}_k + b_k \boldsymbol{1}_P$$

The matrix of synaptic input to the layer is given by:

$$\begin{aligned}\boldsymbol{U} &= \begin{pmatrix}\boldsymbol{u}_1 & \cdots & \boldsymbol{u}_K\end{pmatrix} \\ &= \begin{pmatrix}\boldsymbol{X}\boldsymbol{w}_1 + b_1\mathbf{1}_P & \cdots & \boldsymbol{X}\boldsymbol{w}_K + b_K\mathbf{1}_P\end{pmatrix} \\ &= \boldsymbol{X}\begin{pmatrix}\boldsymbol{w}_1 & \cdots & \boldsymbol{w}_K\end{pmatrix} + \mathbf{1}_P\begin{pmatrix}b_1 & \cdots & b_K\end{pmatrix} \\ &= \boldsymbol{X}\boldsymbol{W} + \boldsymbol{B}\end{aligned}$$

where $\boldsymbol{W} = \begin{pmatrix}\boldsymbol{w}_1 & \cdots & \boldsymbol{w}_K\end{pmatrix}$ is the weight matrix connected to the layer and:

$$\boldsymbol{B} = \begin{pmatrix} b_1 & b_2 & \cdots & b_K \\ b_1 & b_2 & \cdots & b_K \\ \vdots & \vdots & \vdots & \vdots \\ b_1 & b_2 & \cdots & b_K \end{pmatrix}$$

$$\boldsymbol{U} = \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1K} \\ u_{21} & u_{22} & \cdots & u_{2K} \\ \vdots & \vdots & \vdots & \vdots \\ u_{P1} & u_{P2} & \cdots & u_{PK} \end{pmatrix}$$

Rows of $\boldsymbol{U}$ gives the synaptic inputs to the layer, corresponding to each input pattern.

The activation is given by:

$$\begin{aligned} f(\boldsymbol{U}) &= \begin{pmatrix} f(u_{11}) & f(u_{12}) & \cdots & f(u_{1K}) \\ f(u_{21}) & f(u_{22}) & \cdots & f(u_{2K}) \\ \vdots & \vdots & \vdots & \vdots \\ f(u_{P1}) & f(u_{P2}) & \cdots & f(u_{PK}) \end{pmatrix} \\ &= \begin{pmatrix} P(y_1 = 1|\boldsymbol{x}_1) & \cdots & P(y_1 = K|\boldsymbol{x}_1) \\ P(y_2 = 1|\boldsymbol{x}_2) & \cdots & P(y_2 = K|\boldsymbol{x}_2) \\ \vdots & \vdots & \vdots \\ P(y_P = 1|\boldsymbol{x}_P) & \cdots & P(y_P = K|\boldsymbol{x}_P) \end{pmatrix} \end{aligned}$$

Thus, the output of the pth pattern is:

$$y_p = \underset{k}{\text{argmax}}\ \{P(y_p = k|\boldsymbol{x}_p)\}$$

And the output of all the patterns is:

$$\boldsymbol{y} = \underset{k}{\text{argmax}}\ \{P(\boldsymbol{y} = k|\boldsymbol{X})\}$$

4.1.3 GD for Softmax

Given a training dataset $\{(\boldsymbol{x}_p, d_p)\}_{p=1}^P$ where $\boldsymbol{x}_p \in \boldsymbol{R}^n$ and $d_p \in \{1, 2, \cdots, K\}$, the cost function for logistic regression is given by the **multiclass cross-entropy** (or negative log-likelihood):

$$J(\boldsymbol{W}, \boldsymbol{b}) = -\sum_{p=1}^{P}\left(\sum_{k=1}^{K} 1(d_p = k)\log\big(f(u_{pk})\big)\right)$$

where $1(\cdot)$ is the indicator function 1(True) = 1 and 1(False) = 0, and u_{pk} is the synaptic input to the neuron k for input $\boldsymbol{x}_p$.

The cost function can also be written as:

$$J(\boldsymbol{W}, \boldsymbol{b}) = -\sum_{p=1}^{P}\log\big(f(u_{pd_p})\big)$$

Thus, the partial derivative is derived as follows (complete derivation in Lecture Notes):

$$\begin{aligned}\frac{\partial J(\boldsymbol{W}, \boldsymbol{b})}{\partial \boldsymbol{w}_k} &= -\sum_{p=1}^{P}\Big(1(d_p = k) - f(u_{pk})\Big)\boldsymbol{x}_p \\ &= -\boldsymbol{X}^T\big(1(\boldsymbol{d} = k) - f(\boldsymbol{u}_k)\big) \\ &= -\boldsymbol{X}^T\boldsymbol{\delta}_k\end{aligned}$$

where $\boldsymbol{\delta}_k = 1(\boldsymbol{d} = k) - f(\boldsymbol{u}_k)$.

$$1(\boldsymbol{d} = k) = \begin{pmatrix} 1(d_1 = k) \\ 1(d_2 = k) \\ \vdots \\ 1(d_P = k) \end{pmatrix} \quad f(\boldsymbol{u}_k) = \begin{pmatrix} f(u_{1k}) \\ f(u_{2k}) \\ \vdots \\ f(u_{Pk}) \end{pmatrix}$$

Substituting in the GD equation:

$$\begin{aligned}\boldsymbol{w}_k &= \boldsymbol{w}_k - \alpha\frac{\partial J(\boldsymbol{W}, \boldsymbol{b})}{\partial \boldsymbol{w}_k} \\ \boldsymbol{w}_k &= \boldsymbol{w}_k + \alpha_1\boldsymbol{X}^T\boldsymbol{\delta}_k\end{aligned}$$

where $\alpha_1 = \alpha$.

Extending to the weight matrix:

$$\begin{aligned}\boldsymbol{W} &= \begin{pmatrix}\boldsymbol{w}_1 & \cdots & \boldsymbol{w}_K\end{pmatrix} \\ &= \begin{pmatrix}\boldsymbol{w}_1 + \alpha_1\boldsymbol{X}^T\boldsymbol{\delta}_1 & \cdots & \boldsymbol{w}_K + \alpha_1\boldsymbol{X}^T\boldsymbol{\delta}_K\end{pmatrix} \\ &= \begin{pmatrix}\boldsymbol{w}_1 & \cdots & \boldsymbol{w}_K\end{pmatrix} + \alpha_1\boldsymbol{X}^T\begin{pmatrix}\boldsymbol{\delta}_1 & \cdots & \boldsymbol{\delta}_K\end{pmatrix} \\ &= \boldsymbol{W} + \alpha_1\boldsymbol{X}^T\Delta\end{aligned}$$

where:

$$\begin{aligned}\Delta &= \begin{pmatrix}\boldsymbol{\delta}_1 & \cdots & \boldsymbol{\delta}_K\end{pmatrix} \\ &= \begin{pmatrix}1(\boldsymbol{d} = 1) - f(\boldsymbol{u}_1) & \cdots & 1(\boldsymbol{d} = K) - f(\boldsymbol{u}_K)\end{pmatrix} \\ &= \begin{pmatrix}1(\boldsymbol{d} = 1) & \cdots & 1(\boldsymbol{d} = K)\end{pmatrix} - \begin{pmatrix}f(\boldsymbol{u}_1) & \cdots & f(\boldsymbol{u}_K)\end{pmatrix} \\ &= \boldsymbol{K} - f(\boldsymbol{U})\end{aligned}$$

Note that $\boldsymbol{K}$ is a binary matrix, where every row corresponds to an input pattern and has a '1' corresponding to the class of the pattern.

The gradient descent learning for softmax layer is given by:

$$\boldsymbol{W} = \boldsymbol{W} + \alpha_1 \boldsymbol{X}^T \Delta$$
$$\boldsymbol{b}^T = \boldsymbol{b}^T + \alpha_1 \mathbf{1}_P{}^T \Delta$$
$$\boldsymbol{b} = \boldsymbol{b} + \alpha_1 \Delta^T \mathbf{1}_P$$

4.1.4 SGD for Softmax

GD	SGD
$(\boldsymbol{X}, \boldsymbol{D})$	$(\boldsymbol{x}_p, d_p)$
$\boldsymbol{U} = \boldsymbol{XW} + \boldsymbol{B}$	$\boldsymbol{u}_p = \boldsymbol{W}^T \boldsymbol{x}_p + \boldsymbol{b}$
$\boldsymbol{U} = \boldsymbol{U} - u_{max}$	$\boldsymbol{u}_p = \boldsymbol{u}_p - u_{max}$
$f(\boldsymbol{U}) = \frac{e^{\boldsymbol{U}}}{\sum_{k=1}^{K} e^{\boldsymbol{U_k}}}$	$f(\boldsymbol{u}_p) = \frac{e^{\boldsymbol{u}_p}}{\sum_{k=1}^{K} e^{\boldsymbol{u_k}}}$
$\boldsymbol{y} = \underset{k}{\operatorname{argmax}}\, f(\boldsymbol{U})$	$y_p = \underset{k}{\operatorname{argmax}}\, f(\boldsymbol{u}_p)$
$\Delta = \boldsymbol{K} - f(\boldsymbol{U})$	$\boldsymbol{\delta_p} = 1(d_p = \boldsymbol{k}) - f(\boldsymbol{u}_p)$
$\boldsymbol{W} = \boldsymbol{W} + \alpha_1 \boldsymbol{X}^T \Delta$	$\boldsymbol{W} = \boldsymbol{W} + \alpha_1 \boldsymbol{x}_p \boldsymbol{\delta}_p{}^T$
$\boldsymbol{b} = \boldsymbol{b} + \alpha_1 \Delta^T \mathbf{1}_P$	$\boldsymbol{b} = \boldsymbol{b} + \alpha_1 \boldsymbol{\delta}_p$

Figure 4.2: SGD for Softmax

4.2 Perceptron Layer

To achieve multidimensional functional mapping, a layer of continuous perceptrons is required.

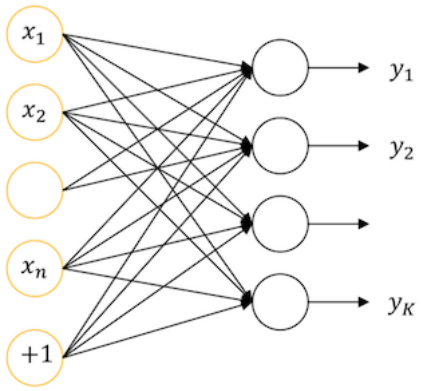

Figure 4.3: Perceptron Layer

A layer of K perceptrons learns a functional mapping:

$$\phi : \boldsymbol{R}^n \rightarrow \boldsymbol{R}^K$$

4.2.1 GD for Perceptron Layer

Consider a training dataset $\{(\boldsymbol{x}_p, \boldsymbol{d}_p)\}_{p=1}^{P}$ where $\boldsymbol{x} \in \boldsymbol{R}^n$ and $\boldsymbol{d}_p \in \boldsymbol{R}^K$. As a batch:

$$\boldsymbol{X} = \begin{pmatrix} \boldsymbol{x}_1^T \\ \boldsymbol{x}_2^T \\ \vdots \\ \boldsymbol{x}_P^T \end{pmatrix} \quad \boldsymbol{D} = \begin{pmatrix} \boldsymbol{d}_1^T \\ \boldsymbol{d}_2^T \\ \vdots \\ \boldsymbol{d}_P^T \end{pmatrix}$$

Let the weight matrix connected to the layer be $\boldsymbol{W} = \begin{pmatrix} \boldsymbol{w}_1 & \cdots & \boldsymbol{w}_K \end{pmatrix}$ and bias be $\boldsymbol{b} = \begin{pmatrix} b_1 & \cdots & b_K \end{pmatrix}^T$.

Synaptic input $\boldsymbol{U} = \boldsymbol{XW} + \boldsymbol{B}$ and output $\boldsymbol{Y} = f(\boldsymbol{U})$, where activation function f is a sigmoid function and $\boldsymbol{B}$ is the bias matrix with bias vector as rows.

Thus, the mean square error cost function is given as:

$$J(\boldsymbol{W}, \boldsymbol{b}) = \frac{1}{P} \sum_{p=1}^{P} \sum_{k=1}^{K} (d_{pk} - y_{pk})^2$$

The gradient with respect to weight $\boldsymbol{w}_k$ of perceptron k (complete derivation in Lecture Notes):

$$\begin{aligned} \frac{\partial J(\boldsymbol{W}, \boldsymbol{b})}{\partial \boldsymbol{w}_k} &= -\frac{2}{P} \sum_{p=1}^{P} (d_{pk} - y_{pk}) \frac{\partial y_{pk}}{\partial \boldsymbol{w}_k} \\ &= -\frac{2}{P} \sum_{p=1}^{P} (d_{pk} - y_{pk}) f'(u_{pk}) \boldsymbol{x}_p \\ &= -\frac{2}{P} \boldsymbol{X}^T (\boldsymbol{d}_k - \boldsymbol{y}_k) \cdot f'(\boldsymbol{u}_k) \\ &= -\frac{2}{P} \boldsymbol{X}^T \boldsymbol{\delta}_k \end{aligned}$$

where $\boldsymbol{\delta}_k = (\boldsymbol{d}_k - \boldsymbol{y}_k) \cdot f'(\boldsymbol{u}_k)$.

Substituting in the GD equation:

$$\boldsymbol{w}_k = \boldsymbol{w}_k - \alpha \frac{\partial J(\boldsymbol{w}, \boldsymbol{b})}{\partial \boldsymbol{w}_k}$$
$$\boldsymbol{w}_k = \boldsymbol{w}_k + \alpha_1 \boldsymbol{X}^T \boldsymbol{\delta}_k$$

where $\alpha_1 = \frac{2\alpha}{P}$.

Extending to a weight matrix:

$$\boldsymbol{W} = \boldsymbol{W} + \alpha_1 \boldsymbol{X}^T \Delta$$

where:

$$\Delta = \begin{pmatrix} \boldsymbol{\delta}_1 & \boldsymbol{\delta}_2 & \cdots & \boldsymbol{\delta}_K \end{pmatrix} = (\boldsymbol{D} - \boldsymbol{Y}) \cdot f'(\boldsymbol{U})$$

Thus, the delta learning equations are given as:

$$\boldsymbol{W} = \boldsymbol{W} + \alpha_1 \boldsymbol{X}^T \Delta$$
$$\boldsymbol{b} = \boldsymbol{b} + \alpha_1 \Delta^T \boldsymbol{1}_P$$

4.2.2 SGD for Perceptron Layer

GD	SGD
$(\boldsymbol{X}, \boldsymbol{D})$	$(\boldsymbol{x}_p, \boldsymbol{d}_p)$
$\boldsymbol{U} = \boldsymbol{X}\boldsymbol{W} + \boldsymbol{B}$	$\boldsymbol{u}_p = \boldsymbol{W}^T \boldsymbol{x}_p + \boldsymbol{b}$
$\boldsymbol{Y} = f(\boldsymbol{U})$	$\boldsymbol{y}_p = f(\boldsymbol{u}_p)$
$\Delta = (\boldsymbol{D} - \boldsymbol{Y}) \cdot f'(\boldsymbol{U})$	$\boldsymbol{\delta}_p = (\boldsymbol{d}_p - \boldsymbol{y}_p) \cdot f'(\boldsymbol{u}_p)$
$\boldsymbol{W} = \boldsymbol{W} + \alpha_1 \boldsymbol{X}^T \Delta$	$\boldsymbol{W} = \boldsymbol{W} + \alpha_1 \boldsymbol{x}_p \boldsymbol{\delta}_p{}^T$
$\boldsymbol{b} = \boldsymbol{b} + \alpha_1 \Delta^T \boldsymbol{1}_P$	$\boldsymbol{b} = \boldsymbol{b} + \alpha_1 \boldsymbol{\delta}_p$

Figure 4.4: SGD for Perceptron Layer

4.3 Initialization of Weights

At initialization, it is desirable that weights:

- are small and near zero to operate in the linear region of the activation function
- preserve the variance of activation and feedback gradients

By considering the above two properties, it has been show that weights should be initialized as follows:

Logistic Sigmoid:

$$w \sim \text{Uniform}\left[-\frac{4\sqrt{6}}{\sqrt{n_{in} + n_{out}}}, \frac{4\sqrt{6}}{\sqrt{n_{in} + n_{out}}} \right]$$

Others:

$$w \sim \text{Uniform}\left[-\frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}} \right]$$

where n_{in} is the number of input neurons and n_{out} is the number of neurons in the layer (i.e. outputs). Uniform is a uniformly distributed number within limits.

4.4 Neurons & Layers Summary

Classification	Perceptron	Regression
Two-class	Discrete perceptron	Logistic regression neuron
Multiclass	Discrete perceptron layer	Softmax neuron layer

Functional approximation	Linear	Non-linear
One dimensional	Linear neuron	Perceptron
Multi-dimensional	Linear neuron layer	Perceptron layer

Figure 4.5: Neurons & Layers Summary

4.5 Neuron Layer GD Summary

$$(\boldsymbol{X}, \boldsymbol{D})$$
$$\boldsymbol{U} = \boldsymbol{X}\boldsymbol{W} + \boldsymbol{B}$$
$$\boldsymbol{W} = \boldsymbol{W} + \alpha_1 \boldsymbol{X}^T \Delta$$
$$\boldsymbol{b} = \boldsymbol{b} + \alpha_1 \Delta^T \boldsymbol{1}_P$$

layer	$f(\boldsymbol{U}), \boldsymbol{Y}$	Δ
Discrete perceptron layer	$\boldsymbol{Y} = f(\boldsymbol{U}) = \begin{cases} 1, \boldsymbol{U} > 0 \\ 0, \boldsymbol{U} \le 0 \end{cases}$	$\boldsymbol{D} - \boldsymbol{Y}$
Linear neuron layer	$\boldsymbol{Y} = f(\boldsymbol{U}) = \boldsymbol{U}$	$\boldsymbol{D} - \boldsymbol{Y}$
Perceptron layer	$\boldsymbol{Y} = f(\boldsymbol{U}) = \frac{1}{1 + e^{-\boldsymbol{U}}}$	$(\boldsymbol{D} - \boldsymbol{Y}) \cdot f'(\boldsymbol{U})$
Softmax layer	$f(\boldsymbol{U}) = \frac{e^{\boldsymbol{U}}}{\sum_{k=1}^{K} e^{\boldsymbol{U}_k}}$ $\boldsymbol{y} = \underset{k}{\text{argmax}}\, f(\boldsymbol{U})$	$\boldsymbol{K} - f(\boldsymbol{U})$

Figure 4.6: Neuron Layer Gradient Descent Summary

Chapter 5

Multilayer Perceptron

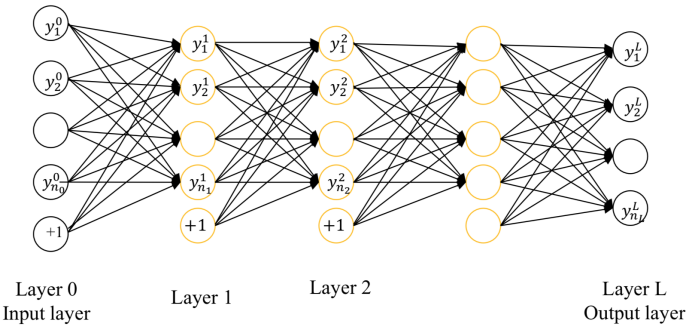

Figure 5.1: Multilayer Perceptron

- A multilayer perceptron (MLP) is a feedforward network with L layers, where the hidden layers of the network consists of perceptrons.
- MLP has been applied successfully to resolve many complex real-world problems consisting of *non-linear decision boundaries*.
- The network is trained in a *supervised* manner with the error backpropagation algorithm based on the gradient descent algorithm.
- Backpropagation learning can be seen as a generalization of the delta learning equations for multilayer networks.

5.1 Three-Layer MLP Network

Given a training dataset $\{(\boldsymbol{x}_p, \boldsymbol{d}_p)\}_{p=1}^{P}$ where $\boldsymbol{x}_p \in \boldsymbol{R}^n$ and $d_p \in \boldsymbol{R}^K$. The three-layer MLP can learn to approximate an arbitrary complex function ϕ:

$$\phi : \boldsymbol{R}^n \to \boldsymbol{R}^K$$

The hidden layer neurons converts input signal into a hidden representation in order to approximate the input-output mapping posed by the training patterns.

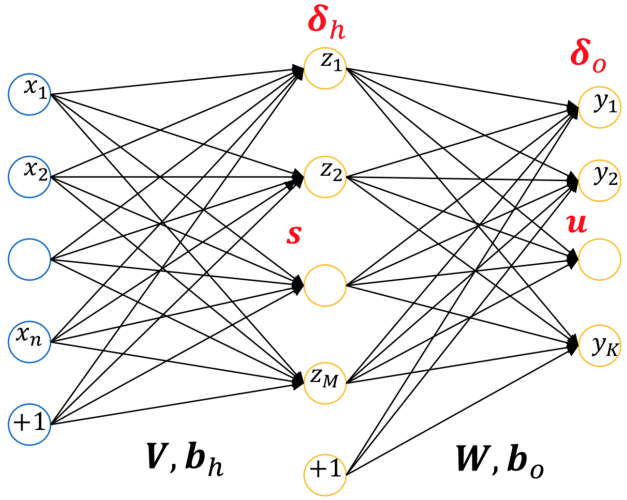

Figure 5.2: Three-Layer MLP Network (neurons have sigmoid activation functions)

For the hidden layer, g is the activation function, synaptic input $\boldsymbol{S} = \boldsymbol{X}\boldsymbol{V} + \boldsymbol{B}_h$ and output $\boldsymbol{Z} = g(\boldsymbol{S})$.

For the output layer, f is the activation function, synaptic input $\boldsymbol{U} = \boldsymbol{Z}\boldsymbol{W} + \boldsymbol{B}_o$ and output $\boldsymbol{Y} = f(\boldsymbol{U})$.

5.1.1 GD for Output Layer

The output layer is a perceptron layer receiving outputs $\boldsymbol{Z}$ from the hidden layer. So we can directly apply equations of a perceptron layer to the output layer:

$$\boldsymbol{U} = \boldsymbol{Z}\boldsymbol{W} + \boldsymbol{B}_o$$
$$\boldsymbol{Y} = f(\boldsymbol{U})$$

Delta learning equations:

$$\Delta_o = (\boldsymbol{D} - \boldsymbol{Y}) \cdot f'(\boldsymbol{U})$$
$$\boldsymbol{W} = \boldsymbol{W} + \alpha_1 \boldsymbol{Z}^T \Delta_o$$
$$\boldsymbol{b}_o = \boldsymbol{b}_o + \alpha_1 \Delta_o^T \boldsymbol{1}_P$$

5.1.2 GD for Hidden Layer

The mean square error function:

$$J = \frac{1}{P}\sum_{p=1}^{P}\sum_{k=1}^{K}(d_{pk} - y_{pk})^2$$

where y_{pk} is the kth neuron output at the output layer.

Gradient with respect to the jth hidden layer neuron weight $\boldsymbol{v}_j$ (complete derivation in Lecture Notes):

$$\begin{aligned}
\frac{\partial J}{\partial \boldsymbol{v}_j} &= -\frac{2}{P}\sum_{p=1}^{P}\sum_{k=1}^{K}(d_{pk} - y_{pk})\frac{\partial y_{pk}}{\partial \boldsymbol{v}_j} \\
&= -\frac{2}{P}\sum_{p=1}^{P}\sum_{k=1}^{K}(d_{pk} - y_{pk})f'(u_{pk})w_{kj}g'(s_{pj})\boldsymbol{x}_p \\
&= -\frac{2}{P}\sum_{p=1}^{P}\sum_{k=1}^{K}\delta_{opk}w_{kj}g'(s_{pj})\boldsymbol{x}_p \\
&= \frac{2}{P}\sum_{p=1}^{P}(\boldsymbol{\delta}_{op}{}^T\boldsymbol{w}_j)\cdot g'(s_{pj})\boldsymbol{x}_p \\
&= -\frac{2}{P}\begin{pmatrix}\boldsymbol{x}_1 & \dots & \boldsymbol{x}_P\end{pmatrix}\begin{pmatrix}(\boldsymbol{\delta}_{o1}{}^T\boldsymbol{w}_j)\cdot g'(s_{1j}) \\ \vdots \\ (\boldsymbol{\delta}_{oP}{}^T\boldsymbol{w}_j)\cdot g'(s_{Pj})\end{pmatrix} \\
&= -\frac{2}{P}\boldsymbol{X}^T\boldsymbol{\delta}_{hj}
\end{aligned}$$

where:

$$\begin{aligned}
\boldsymbol{\delta}_{hj} &= \begin{pmatrix}(\boldsymbol{\delta}_{o1}{}^T\boldsymbol{w}_j)\cdot g'(s_{1j}) \\ \vdots \\ (\boldsymbol{\delta}_{oP}{}^T\boldsymbol{w}_j)\cdot g'(s_{Pj})\end{pmatrix} \\
&= \begin{pmatrix}\boldsymbol{\delta}_{o1}{}^T \\ \vdots \\ \boldsymbol{\delta}_{oP}{}^T\end{pmatrix}\boldsymbol{w}_j \cdot \begin{pmatrix}g'(s_{1j}) \\ \vdots \\ g'(s_{Pj})\end{pmatrix} \\
&= \Delta_o\boldsymbol{w}_j \cdot g'(\boldsymbol{s}_j)
\end{aligned}$$

Substituting in the GD equation for the jth hidden neuron:

$$\begin{aligned}
\boldsymbol{v}_j &= \boldsymbol{v}_j - \alpha\frac{\partial J}{\partial \boldsymbol{v}_j} \\
\boldsymbol{v}_j &= \boldsymbol{v}_j + \alpha_1\boldsymbol{X}^T\boldsymbol{\delta}_{hj}
\end{aligned}$$

where $\alpha_1 = \frac{2\alpha}{P}$.

Therefore, for the hidden layer:

$$\boldsymbol{V} = \boldsymbol{V} + \alpha_1\boldsymbol{X}^T\Delta_h$$

where:

$$\begin{aligned}
\Delta_h &= \begin{pmatrix}\boldsymbol{\delta}_{h1} & \dots & \boldsymbol{\delta}_{hM}\end{pmatrix} \\
&= \begin{pmatrix}\Delta_o\boldsymbol{w}_1 \cdot g'(\boldsymbol{s}_1) & \dots & \Delta_o\boldsymbol{w}_M \cdot g'(\boldsymbol{s}_M)\end{pmatrix} \\
&= \Delta_o\boldsymbol{W}^T \cdot g'(\boldsymbol{S})
\end{aligned}$$

Similarly, for bias:

$$\boldsymbol{b}_h = \boldsymbol{b}_h + \alpha_1\Delta_h{}^T\boldsymbol{1}_P$$

5.1.3 SGD for Three-Layer MLP

GD, (X, D)	SGD, (x_p, d_p)
$\boldsymbol{S} = \boldsymbol{XV} + \boldsymbol{B}_h$	$\boldsymbol{s_p} = \boldsymbol{V}^T\boldsymbol{x}_p + \boldsymbol{b}_h$
$\boldsymbol{Z} = g(\boldsymbol{S})$	$\boldsymbol{z}_p = g(\boldsymbol{s_p})$
$\boldsymbol{U} = \boldsymbol{ZW} + \boldsymbol{B}_o$	$\boldsymbol{u}_p = \boldsymbol{W}^T\boldsymbol{z}_p + \boldsymbol{b}_o$
$\boldsymbol{Y} = f(\boldsymbol{U})$	$\boldsymbol{y}_p = f(\boldsymbol{u_p})$
$\Delta_o = (\boldsymbol{D} - \boldsymbol{Y}) \cdot f'(\boldsymbol{U})$	$\boldsymbol{\delta}_{op} = (\boldsymbol{d}_p - \boldsymbol{y}_p) \cdot f'(\boldsymbol{u}_p)$
$\Delta_h = \Delta_o\boldsymbol{W}^T \cdot g'(\boldsymbol{S})$	$\boldsymbol{\delta}_{hp} = \boldsymbol{W}\boldsymbol{\delta}_{op} \cdot g'(\boldsymbol{s_p})$
$\boldsymbol{W} = \boldsymbol{W} + \alpha_1\boldsymbol{Z}^T\Delta_o$	$\boldsymbol{W} = \boldsymbol{W} + \alpha_1\boldsymbol{z}_p\boldsymbol{\delta}_{op}{}^T$
$\boldsymbol{b}_o = \boldsymbol{b}_o + \alpha_1\Delta_o{}^T\boldsymbol{1}_P$	$\boldsymbol{b}_o = \boldsymbol{b}_o + \alpha_1\boldsymbol{\delta}_{op}$
$\boldsymbol{V} = \boldsymbol{V} + \alpha_1\boldsymbol{X}^T\Delta_h$	$\boldsymbol{V} = \boldsymbol{V} + \alpha_1\boldsymbol{x}_p\boldsymbol{\delta}_{hp}{}^T$
$\boldsymbol{b}_h = \boldsymbol{b}_h + \alpha_1\Delta_h{}^T\boldsymbol{1}_P$	$\boldsymbol{b}_h = \boldsymbol{b}_h + \alpha_1\boldsymbol{\delta}_{hp}$

Figure 5.3: SGD for Three-Layer MLP

5.1.4 Backpropagation of Error

Note that the hidden layer terms Δ_h are computed from output error terms Δ_o. The error can be seen as propagating to the hidden layer from the output layer and therefore learning is known as the *backpropagation algorithm*.

5.2 Functional Approximation with MLP

Multilayer feedforward networks such as MLP that use continuous perceptrons are able to approximate complex multi-dimensional functions. Given a set of training samples, the network finds a smooth function that approximates the actual functional mapping.

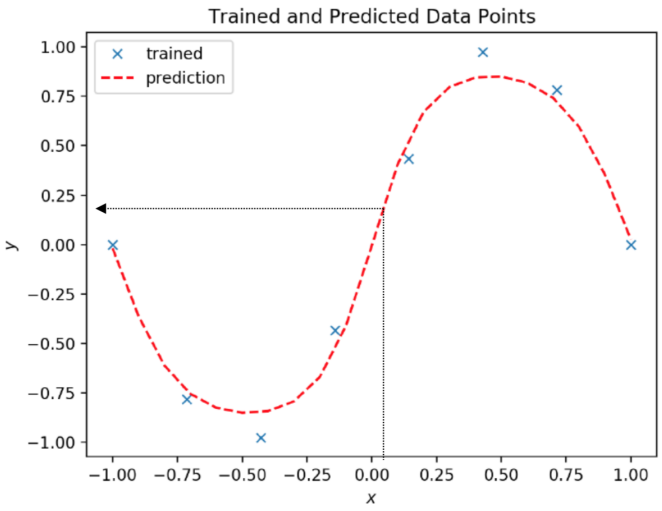

Figure 5.4: MLP Functional Approximation

MLP finds a multidimensional functional mapping:

$$\phi : \boldsymbol{R}^n \rightarrow \boldsymbol{R}^K$$

by minimizing the mean square error cost function:

$$J = \frac{1}{P}\sum_{p=1}^{P} ||\boldsymbol{d}_p - \boldsymbol{y}_p||^2$$

Once the weights and biases are learnt by backpropagation, the predicted value $\hat{\boldsymbol{y}}$ for an input pattern $\boldsymbol{x}$ can be found:

$$\hat{\boldsymbol{y}} = \phi(\boldsymbol{x})$$

5.3 Universal Approximation Theorem

Kolmogorov Theorem: Given any function $\phi : I^n \rightarrow \boldsymbol{R}^k$ where I is the closed unit interval $[0, 1]$, ϕ can be implemented exactly by a three layer neural network with n input nodes, $2n + 1$ hidden layer neurons and K output layer neurons.

The theorem doesn't refer to the algorithms or the amount of training data needed for the approximation. Recent researches find that deep neural networks and a large number of training data are required to learn any arbitrarily complex function.

5.4 Output Layer Activation Function

MLP uses sigmoid as the activation function of neurons. Hence, the output neuron activation function should be scaled and translated to match the space of output.

If output $y \in [y_{min}, y_{max}]$, the output layer activation function:

$$f(u) = \frac{a}{1 + e^{-u}} + b$$

where $a = y_{max} - y_{min}$ and $b = y_{min}$.

5.5 Normalization of Inputs

If inputs have similar variations, a better approximation of inputs is achieved. Mainly, there are two approaches to normalization. If $\widetilde{x}_i$ denotes the normalized input:

I. Scale the inputs such that $x_i \in [0, 1]$:

$$\widetilde{x}_i = \frac{x_i - x_{i,min}}{x_{i,max} - x_{i,min}}$$

II. Normalize the input to have standard normal distributions $x_i \sim N(0, 1)$:

$$\widetilde{x}_i = \frac{x_i - \mu_i}{\sigma_i}$$

5.6 Number of Hidden Neurons

With increasing number of hidden units, the number of parameters of the network increases and therefore, the network attempts to remember the training patterns. That is, the network aims at minimizing the training error but this can increase the errors on test data (i.e. overfitting) after some point.

As the number of hidden units increases, the test error decreases initially but tends to increase at some point. The optimal number of hidden neurons is often determined empirically (i.e. trial and error).

5.7 Deep Feedforward Networks

Total $L+1$ layers.

- Layer l:
  - Number of Neurons: n_l
  - Weight Vector: $\boldsymbol{W}^l$
  - Bias: $\boldsymbol{b}^l$
  - Synaptic Input: $\boldsymbol{U}^l$
  - Output of Layer: $\boldsymbol{Y}^l$
  - Input from Previous Layer: $\boldsymbol{Y}^{l-1}$
- Input Layer ($l=0$):
  - Number of Inputs: $n_0 = n$
  - Input to Network: $\boldsymbol{Y}^0 = \boldsymbol{X}$
- Output Layer ($l=L$):
  - Softmax layer for classification or linear neurons for functional approximation.

5.7.1 GD for Deep Feedforward Network

- Input: $(\boldsymbol{X}, \boldsymbol{D})$
- Input Layer: $\boldsymbol{Y}^0 = \boldsymbol{X}$
- Layers $l = 1, 2, \ldots L$:
  - $\boldsymbol{U}^l = \boldsymbol{Y}^{l-1}\boldsymbol{W}^l + \boldsymbol{B}^l$
  - $\boldsymbol{Y}^l = f(\boldsymbol{U}^l)$
- Output Layer: $\Delta^L = (\boldsymbol{D} - \boldsymbol{Y}^L) \cdot f'(\boldsymbol{U}^L)$
- Layers $l = L-1, L-2, \ldots, 1$:
  - $\Delta^l = \Delta^{l+1}{\boldsymbol{W}^{l+1}}^T \cdot f'(\boldsymbol{U}^l)$
- Layers $l = 1, 2, \ldots L$:
  - $\boldsymbol{W}^l = \boldsymbol{W}^l + \alpha_1 {\boldsymbol{Y}^{l-1}}^T \Delta^l$
  - $\boldsymbol{b}^l = \boldsymbol{b}^l + \alpha_1 {\Delta^l}^T \boldsymbol{1}_P$

5.7.2 SGD for Deep Feedforward Network

- Input: $(\boldsymbol{x}, \boldsymbol{d})$
- Input Layer: $\boldsymbol{y}^0 = \boldsymbol{x}$
- Layers $l = 1, 2, \ldots L$:
  - $\boldsymbol{u}^l = {\boldsymbol{W}^l}^T \boldsymbol{y}^{l-1} + \boldsymbol{b}^l$
  - $\boldsymbol{y}^l = f(\boldsymbol{u}^l)$
- Output Layer: $\boldsymbol{\delta}^L = (\boldsymbol{d} - \boldsymbol{y}^L) \cdot f'(\boldsymbol{u}^L)$
- Layers $l = L-1, L-2, \ldots, 1$:
  - $\boldsymbol{\delta}^l = {\boldsymbol{W}^{l+1}}^T \boldsymbol{\delta}^{l+1} \cdot f'(\boldsymbol{u}^l)$
- Layers $l = 1, 2, \ldots L$:
  - $\boldsymbol{W}^l = \boldsymbol{W}^l + \alpha_1 \boldsymbol{y}^{l-1} {\boldsymbol{\delta}^l}^T$
  - $\boldsymbol{b}^l = \boldsymbol{b}^l + \alpha_1 {\boldsymbol{\delta}^l}^T$

5.8 GD vs. SGD

SGD explores the randomness of individual data points and presentation of data can be randomized. Therefore, this usually improves convergence. However, the number of parameter updates in a training epoch is equal to the number of training patterns in SGD, which makes it very slow in practice.

GD has one update per training epoch. Updates are based on the average loss of error function over the training data, and hence, convergence is smoother. So, a higher learning factor can be used as compared to SGD.

GD can explore the parallelism in batch processing and matrix processing. So, the time for one iteration is less for GD than for SGD. For large training datasets, SGD is computationally prohibitive.

5.8.1 Mini-Batch GD

In practice, gradient descent is performed on mini-batch updates of gradients within a batch or block of data of size B. In mini-batch GD, the data is divided into blocks and the gradients are averaged for each block in an epoch.

- $B = 1$: stochastic (online) gradient descent
- $B = P$: (batch) gradient descent
- $1 < B < P$: mini-batch stochastic gradient descent

When B increases, there are more add-multiply operations per second, taking advantage of parallelism and matrix computation. At the same time, the number of computations per update of parameters increases.

Therefore, the curve of the time for weight update against batch size usually takes a U-shaped curve. There exists an optimal value of B, which depends on the size of the cache of the CPU / GPU.

In order to efficiently sample blocks, the training patterns are shuffled at the beginning of every training epoch and then blocks are sequentially fetched from memory.

Typical batch sizes: 16, 32, 64, 128 and 256.

Chapter 6

Model Selection

- In most pattern recognition techniques, there exist one or more free parameters (i.e. hyperparameters). For example, the learning parameters, number of hidden layers, number of hidden neurons etc.
- Every set of parameters for the network leads to a specific model.
- Therefore, it is necessary to find the set of parameters that leads to the optimum model.

6.1 Performance Estimation

Mean Square Error / Root Mean Square Error for approximation.

$$\text{MSE} = \frac{1}{P}\sum_{p=1}^{P}\sum_{k=1}^{K}(d_{pk} - y_{pk})^2$$

$$\text{RMSE} = \sqrt{\frac{1}{P}\sum_{p=1}^{P}\sum_{k=1}^{K}(d_{pk} - y_{pk})^2}$$

Misclassification Error for classification.

$$\text{ME} = \sum_{p=1}^{P} 1(d_p \neq y_p)$$

where d_p is the target and y_p is the predicted output of pattern p. $1(\cdot)$ is the indicator function.

6.2 Terminology

- **Apparent Error** or training error is the error on the training data, which is what the learning algorithm tries to optimize.
- **True Error** is the error that will be obtained in use (over the whole sample space). This is what we want to optimize but is unknown.
- **Test Error** or out-of-sample error is an estimate of the true error obtained by testing the network on some independent data. Generally, a large test set helps provide a greater confidence on the accuracy of this estimate.

6.3 Validation

While we want to minimize the error on the entire sample population, this is difficult as the entire sample population is often unavailable. Hence, in real applications, we use **validation**, in which a part of the available data, which is known as the *validation set*, is used to select the model and estimate the error rate.

Validation attempts to solve the following problems of using training error:

- The final model tends to *overfit* the training data.
- There is no way of knowing how well the model performs on unseen data.
- The error rate estimate will be overly optimistic (usually lower than the true error rate).

6.3.1 Validation Methods

An effective approach is to split the entire data into subsets i.e. Training / Validation / Testing datasets. There are a few common validation methods:

1. Holdout (1/3 - 2/3 Rule)
2. Re-Sampling Techniques
    1. Random Subsampling
    2. K-Fold Cross Validation
    3. Leave-One-Out Cross Validation
3. Three-Way Data Split

6.4 Holdout Method

Split entire dataset into two sets:

1. Training Set (2/3), which is used to train the classifier.

2. Testing Set (1/3), which is used to estimate the error rate of the trained classifier on unseen data samples.

6.4.1 Drawbacks

- In problems where we have a sparse dataset, it may not be possible to set aside a portion of the dataset for testing.
- Since it is a single train-and-test experiment, the holdout estimate of error rate will be misleading if we happen to make an *unlucky split*.

6.5 Re-Sampling Techniques

The drawbacks of holdout can be overcome using re-sampling techniques.

6.5.1 Random Subsampling (K Data Splits)

- Split entire dataset into K splits where each split randomly selects a (fixed) number of examples.
- For each data split, the classifier is retrained from scratch with the training data.
- Estimate e_i using the test data and compute the average test error:

$$\frac{1}{K}\sum_{i=1}^{K} e_i$$

6.5.2 K-Fold Cross Validation

- Split the dataset into K partitions (or folds).
- For each of the K experiments, use K - 1 folds for training and the remaining one fold for testing.
- The average test error is also known as the cross-validation error.
- The advantage of K-fold cross validation is that all the examples in the dataset are eventually used for training and testing.

6.5.3 Leave-One-Out Cross Validation

- Leave-one-out is the degenerate case of K-fold cross validation, where K is chosen as the total number of examples.
- For a dataset with N examples, N experiments are performed i.e. K = N.
- For each experiment, N - 1 examples are used for training and the remaining one example is used for testing.

6.6 Three-Way Data Split

If model selection and true error estimates are to be computed simultaneously, the data needs to be divided into three disjoint sets:

1. Training Set, which is used for learning.
2. Validation Set, which determines the error J_m of different models m. For multi-layer perceptrons, the validation set is used to estimate the error rate of the model.
3. Training & Validation Set are combined to re-train the optimal model (i.e. model with lowest validation error).
4. Test Set, which is used only to assess the performance of the final optimal model.

Why separate test and validation sets?

- The error estimate of the final model on validation data will be biased (i.e. smaller than the true error rate) since the validation set is also used in the process of model selection.
- To assess the final model, an independent test set is required to estimate the true performance of the model.

6.7 Model Complexity

Complex models are models with many adjustable weights and biases. They will be:

- more likely to be able to solve the required task
- more likely to memorize the training data without solving the task

Simple models can learn from the training data and are more likely to *generalize* over the entire sample space, assuming they don't underfit / under-learn.

6.8 Overfitting

Overfitting occurs when the training error of the network is driven to a very small value at the expense of the test error. The network learns to respond correctly to the training inputs but is unable to generalize to produce correct outputs to novel inputs.

6.8.1 Overcoming Overfitting

6.8.1.1 Early Stopping

The training of the network should be stopped when the test error starts to increase. Early stopping can be used in training with a validation set by stopping when the validation error is minimum.

6.8.1.2 Regularization of Weights

During overfitting, some weights attain large values to reduce the training error, which prevents the model from generalizing. In order to avoid this, a penalty term (i.e. regularization term) is added to the cost function.

$$J_1(\boldsymbol{W}, \boldsymbol{b}) = J(\boldsymbol{W}, \boldsymbol{b}) + \beta_1 \sum_{i,j} |w_{ij}| + \beta_2 \sum_{i,j} (w_{ij})^2$$

where $J(\boldsymbol{W}, \boldsymbol{b})$ is the standard cost function (i.e. m.s.e or cross-entropy) and β_1 & β_2 are known as L_1 & L_2 regularization constants respectively. These penalties discourage weights from attaining large values.

$$L_1 \text{ - norm} = \sum_{i,j} |w_{ij}|$$
$$L_2 \text{ - norm} = \sum_{i,j} (w_{ij})^2$$

6.8.1.3 L2 Regularization

For neural networks, regularization is usually not applied on bias terms and L_2 regularization is most popular.

$$J_1(\boldsymbol{W}, \boldsymbol{b}) = J(\boldsymbol{W}, \boldsymbol{b}) + \beta_2 \sum_{i,j} (w_{ij})^2$$

Gradient of the regularized cost w.r.t the weights:

$$\begin{aligned} \frac{\partial J_1}{\partial \boldsymbol{W}} &= \frac{\partial J}{\partial \boldsymbol{W}} + \beta_2 \frac{\partial (\sum_{i,j} {w_{ij}}^2)}{\partial \boldsymbol{W}} \\ &= \frac{\partial J}{\partial \boldsymbol{W}} + 2\beta_2 \boldsymbol{W} \end{aligned}$$

Gradient descent learning using the regularized cost function:

$$\begin{aligned} \boldsymbol{W} &= \boldsymbol{W} - \alpha \frac{\partial J_1}{\partial \boldsymbol{W}} \\ &= \boldsymbol{W} - \alpha \Big(\frac{\partial J}{\partial \boldsymbol{W}} + \beta \boldsymbol{W} \Big) \end{aligned}$$

where $\beta = 2\beta_2$ and is known as the weight decay parameter.

6.8.1.4 Dropouts

Deep neural networks with a large number of parameters are powerful learning machines. However, overfitting is a serious problem in deep neural networks.

The key idea of *dropouts* is to randomly drop neurons (along with their connections) from the network during training. This prevents neurons from co-adapting. The neurons are *present* with a probability p during training and always present during testing.

Training is similar to standard networks, however, for mini-batch training, the training is applied on a *thinned network* with dropped-out units. The gradients of each parameter are averaged over the training cases in each mini-batch.

Chapter 7

Deep Convolutional Neural Networks

Convolutional neural networks (CNNs) are biologically inspired variants of MLP. The animal visual cortex is the most powerful visual processing system and CNNs emulate its behavior:

- Each neuron in the visual cortex is responsive to small subregions of the visual field, known as receptive fields.
- The subregions are tiled together to cover the entire visual field.
- The neurons act as local filters over the input space and exploit the strong local correlations in natural images.

Thus, two types of cells (i.e. neurons) are identified: *simple cells* and *complex cells*. Simple cells respond maximally to specific edge-like patterns within their respective receptive fields whereas complex cells have larger receptive fields and are locally invariant to the exact position of the patterns.

7.1 Locally Connected Networks

- Backpropagation works well for signals and images that are low in resolution. Deep CNNs were developed to process more realistic signals and images that are of larger dimensions.
- Fully connected networks are not feasible for signals of large resolutions as their feedforward and backpropagation computations are computationally expensive.
- One solution is to restrict the number of connections from one layer to another to a smaller subset. The idea is to connect only a contiguous (local) set of nodes to the next layer.

7.1.1 Sparse Local Connectivity

CNNs exploit spatially local correlations by enforcing local connectivity between neurons of adjacent layers. Thus, the receptive fields of the neurons are limited.

7.1.2 Shared Connections

In CNNs, each filter is replicated across the entire visual field. These replicated units share the same parameterization (i.e. weights & biases) and form a feature map. In the image below, weights of the same color are shared.

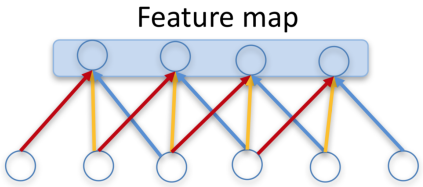

Figure 7.1: Feature Map

Replicating weights in this way allows for features to be detected regardless of the position in the visual field. Additionally, weight sharing increases learning efficiency by greatly reducing the number of free parameters.

7.2 Convolutional Layer

CNNs are usually made of alternating convolutional and pooling layers.

- Features of natural images are invariant at every location of the image. Thus, filters can be replicated and used to cover the entire visual field.
- The convolutional layer learns the features over small patches of the image and the learned features are then convolved with the image to obtain feature activations.
- The weights can be learned using feature detectors or backpropagation.
- The region of the input layer that is connected to a convolutional layer is referred to as the *receptive field*. The weights learned are known as *filters* or *kernels*.
- The output activation learned by a particular filter is known as a feature map.

7.2.1 Input → Output

Synaptic input at location (i, j) of the first hidden layer due to a kernel $\boldsymbol{w} = \{w(l, m)\}_{l,m=-L/2,-M/2}^{L/2,M/2}$ is given by:

$$u(i, j) = \sum_l \sum_m x(i + l, j + m)w(l, m) + b$$

The output of the neuron at (i, j) of the convolution layer is:

$$y(i, j) = f(u(i, j))$$

where f is the sigmoid activation function.

Note that one kernel (filter) creates one feature map.

7.2.2 Handling Boundaries

There are two ways to handle the boundary:

1. Apply the filter wherever it completely overlaps with the input. (default)
2. Apply the filter wherever it partially overlaps the input.

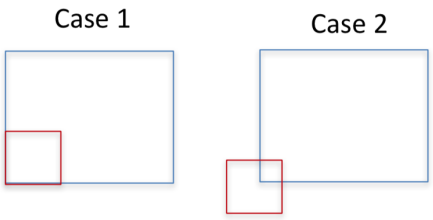

Figure 7.2: Handling Boundaries

If we have k kernels, input image is $I \times J$ and kernel size is $L \times M$, then the size of the convolution layer:

Case 1: $k \times (I - L + 1) \times (J - M + 1)$

Case 2: $k \times (I + L - 1) \times (J + M - 1)$

Stride is the factor with which the output is subsampled. The default stride is 1.

7.3 Pooling Layer

- To reduce the dimensions of the convolutional layer output, pooling of the activation is performed at the pooling layer. Either max or average pooling is used at the pooling layer. Pooling *downsamples* the input.
- The convolved features are divided into disjoint regions and pooled by either taking the maximum or the mean. Pooled features are *translational invariant*. The default stride is equal to the pooling width.

7.3.1 Input → Output

Consider pooling with non-overlapping windows $\{(p, q)\}_{p,q=-P/2,-Q/2}^{P/2,Q/2}$ of size $P \times Q$.

The max pooling output is the maximum of the activation inside the pooling window:

$$z(i, j) = \max_{p,q}\{y(i + p, j + q)\}$$

The mean pooling output is the mean of activations in the pooling window:

$$z(i, j) = \frac{1}{P \times Q} \sum_p \sum_q y(i + p, j + q)$$

7.4 Fully Connected Layer

- The final layers of a deep CNN consist of one or more fully connected layers.
- The output layer is usually a softmax regression layer for classification.

For example, with three filters (of size 5×5) followed by pooling (of 2×2 regions):

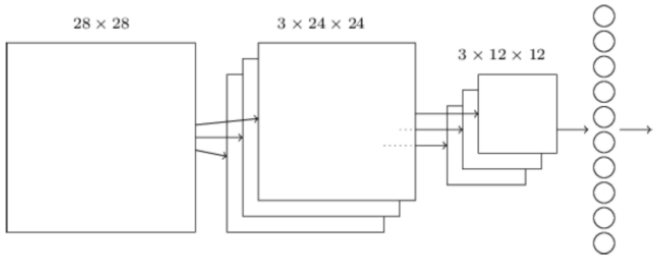

Figure 7.3: Deep CNN

7.5 Backpropagation for CNNs

During forward propagation, activations are downsampled at the pooling layer. Thus, during error backpropagation, the error terms need to be upsampled at the pooling layer.

In MLP, the error Δ^{l+1} is propagated to layer l as:

$$\Delta^l = \Delta^{l+1}{\boldsymbol{W}^{l+1}}^T \cdot f'(\boldsymbol{U}^l)$$

For deep CNNs, the transfer of error terms involves upsampling of errors at the pooling layer:

$$\Delta^l = \text{upsample}(\Delta^{l+1}\boldsymbol{W}^{l+1^T}) \cdot f'(\boldsymbol{U}^l)$$

7.6 GD w/ Momentum

The method of momentum is designed to accelerate learning, especially in the face of high curvature, small but consistent gradients or noisy gradients. The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction.

When the error function has the form a shallow ravine leading to the optimum and steep walls on the side, stochastic gradient descent algorithm tends to oscillate near the optimum. This leads to very slow converging rates. The problem is typical in deep learning architecture.

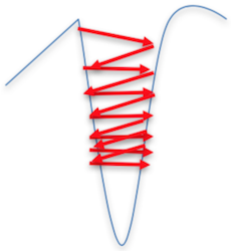

Figure 7.4: GD Oscillation

Momentum is one method of speeding the convergence along a narrow ravine. The momentum update is given by:

$$\boldsymbol{V} = \gamma \boldsymbol{V} - \alpha \frac{\partial J}{\partial \boldsymbol{W}}$$
$$\boldsymbol{W} = \boldsymbol{W} + \boldsymbol{V}$$

where $\boldsymbol{V}$ is known as the velocity term and has the same dimension as the weight vector. The momentum parameter $\gamma \in [0, 1]$ indicates how much of the previous gradients are incorporated into the current update.

Often, γ is initially set to 0.1 until learning stabilizes and increased to 0.9 thereafter.

7.7 GD w/ Annealing

Often, stochastic gradient descent is employed in a few training samples or a mini-batch. This reduces variance of the individual patterns and achieves stable convergence but at the expense of the true minima.

The learning rate in the online (stochastic) version is usually much less than the learning rate in a batch version. When online or mini-batch versions of gradient descent is used, it is not easy to determine the value of the learning parameter because of the variance in individual patterns. One way to overcome this is to use an annealing schedule, that is, to start with a large learning factor and then gradually reduce it.

A possible annealing schedule (t is the iteration count):

$$\alpha(t) = \frac{a}{b+t}$$

where a & b are two positive constants, $\alpha(0) = a/b$ and $\alpha(\infty) = 0$.

7.8 RMSProp Algorithm

Adaptive learning rates with annealing usually works well with convex cost functions. However, the learning trajectory of a neural network minimizing a non-convex cost function passes through many different structures before eventually arriving at a region that is locally convex.

RMSProp uses an exponentially decaying average to discard the history from the extreme past so that it can converge rapidly after finding a convex region.

$$\boldsymbol{r} = \rho \boldsymbol{r} + (1-\rho)\Big(\frac{\partial J}{\partial \boldsymbol{W}}\Big)^2$$
$$\boldsymbol{W} = \boldsymbol{W} - \frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \cdot \Big(\frac{\partial J}{\partial \boldsymbol{W}}\Big)$$

where ρ, ϵ and δ are parameters to be determined empirically.

RMSProp has been shown to be an effective and practical optimization algorithm for deep neural networks.

7.9 Examples of CNN Architectures

See Lecture Notes for details on **LeNet** & **AlexNet**.

Chapter 8

Autoencoders

An autoencoder is a neural network that is trained to attempt to copy its input to its output. Its hidden layer describes a *code* that represents the input. The network consists of two parts, an encoder and a decoder.

Given an input $\boldsymbol{x}$, the hidden layer performs the encoding function $\boldsymbol{h} = \varphi(\boldsymbol{x})$ and the decoder ϕ produces the reconstruction $\boldsymbol{y} = \phi(\boldsymbol{h})$.

If the encoder succeeds,

$$\boldsymbol{y} = \phi(\varphi(\boldsymbol{x})) = \boldsymbol{x}$$

Note that autoencoders are designed to be unable to make exact copies and only prioritize the aspects of the input that are learned to be useful by the hidden layer during training.

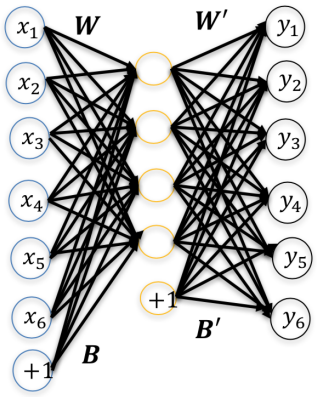

Figure 8.1: Autoencoder w/ One Hidden Layer

$$\boldsymbol{H} = f(\boldsymbol{X}\boldsymbol{W} + \boldsymbol{B})$$
$$\boldsymbol{Y} = f(\boldsymbol{H}\boldsymbol{W}' + \boldsymbol{B}')$$

Reverse mapping may be optionally constrained such that $\boldsymbol{W}' = \boldsymbol{W}^T$.

8.1 Cost Function

The cost function can be measured by many ways, depending on the appropriate distributional assumptions on the input.

If the data is assumed to be continuous and Gaussian distributed, the mean squared error cost is usually used:

$$J_{\text{mse}}(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{b}') = \frac{1}{P}\sum_{p=1}^{P} ||\boldsymbol{y}_p - \boldsymbol{x}_p||^2$$

If the inputs are interpreted as bit vectors or vectors of bit probabilities, cross-entropy of the reconstruction can be used:

$$J_{\text{ce}}(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{b}') = -\sum_{p=1}^{P}(\boldsymbol{x}_p \cdot \log \boldsymbol{y}_p + (1 - \boldsymbol{x}_p) \cdot \log(1 - \boldsymbol{y}_p))$$

8.2 Denoising Autoencoders

A denoising autoencoder (DAE) receives corrupted data points as inputs and is trained to predict the original uncorrupted data points as its output. In order to achieve this, the DAE's hidden layer discovers more robust features instead of simply learning the identity.

In other words, DAEs attempt to preserve information about the input in order to undo the effect of the corruption process applied to the input.

For training a DAE:

$$\boldsymbol{X} \to \widetilde{\boldsymbol{X}}$$
$$\boldsymbol{Y} \to \boldsymbol{X}$$

where $\widetilde{\boldsymbol{X}}$ is the corrupted version of data. The corruption process simulates the distribution of data.

8.2.1 Corrupting Inputs

To obtained corrupted version of input data, each input x_i is added with additive or multiplicative noise.

Additive Noise (usually used for continuous data):

$$\widetilde{x_i} = x_i + \epsilon$$

where the noise ϵ is Gaussian distributed (i.e. $\epsilon \sim N(0, \sigma^2)$) and σ is the standard deviation that determines the signal to noise ratio.

Multiplicative Noise (usually used for binary data):

$$\widetilde{x_i} = \epsilon x_i$$

where the noise ϵ could be binomially distributed (i.e. $\epsilon \sim$ Binomial(p), where p is the probability of ones and $1 - p$ is the probability of zeroes i.e. noise).

8.3 Undercomplete Autoencoders

- The hidden layer has a lower dimension than the input in undercomplete autoencoders.
- Learning an undercomplete representation forces the autoencoder to capture the most salient features.
- By limiting the number of hidden neurons, interesting (hidden) structures of input data can be inferred from autoencoders. For example, correlations among input variables, principal components of data etc.
- By learning to approximate n-dimensional inputs with M ($< n$) number of hidden units, a lower dimensional representation of the input signals is obtained. The network reconstructs the input signals from this hidden representation.

8.4 Overcomplete Autoencoders

- The hidden layer has a higher dimension than the input in overcomplete autoencoders.
- When the hidden dimensions are large, one could explore interesting structures of inputs by introducing other constraints such as the 'sparsity' of input data.

Both, undercomplete and overcomplete autoencoders, fail to learn anything useful if the encoder and decoder are given too much capacity. Some constraints are required to make them useful.

8.5 Regularizing Autoencoders

Regularized autoencoders provide the ability to train any autoencoder architecture successfully, choosing the code dimension and the capacity of the encoder and decoder based on the complexity of the distribution to be modeled.

Rather than limiting the model capacity by keeping the encoder and decoder shallow and the code size small, regularized autoencoders use a loss function that encourages the model to have other properties besides the ability to copy its input to its output.

Regularized autoencoders add an appropriate penalty function Ω to the cost function:

$$J_1(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{b}') = J(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{b}') + \beta\Omega(\boldsymbol{H})$$

A regularized autoencoder can be nonlinear and overcomplete but still learn something useful about the data distribution, even if the model capacity is large enough to learn a trivial identity function.

8.6 Sparse Autoencoders

A sparse autoencoder (SAE) is simply an autoencoder whose training criterion involves the sparsity penalty Ω_{sparsity} at the hidden layer:

$$J_1(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{b}') = J(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{b}') + \beta\Omega_{\text{sparsity}}(\boldsymbol{H})$$

The sparsity penalty term makes the features (weights) learned by the hidden layer to be sparse.

Consider a sigmoid activation function, we say that the neuron is "active" when its output is close to 1 and the neuron is "inactive" when its output is close to 0. With the sparsity constraint, the neurons at the hidden layer(s) are constrained to be inactive most of the time.

8.6.1 Sparsity Constraint

The average activation ρ_j at neuron j of the hidden layer is given by:

$$\rho_j = \frac{1}{P}\sum_{p=1}^{P} h_{pj} = \frac{1}{P}\sum_{p=1}^{P} f(\boldsymbol{x}_p^T \boldsymbol{w}_j + b_j)$$

We would like to enforce the constraint $\rho_j = \rho$ where the sparsity parameter ρ is set to a small value close to zero (say 0.05). That is, the hidden neuron activations are maintained only 5% of the time on average.

8.6.2 Kullback-Leibler Divergence

To achieve sparse activations at the hidden layer, we introduce Kullback-Leibler (KL) divergence as the sparsity constraint. KL divergence measures the deviation of the distribution of activations at the hidden layer from uniform distribution.

$$D(\boldsymbol{h}) = \sum_{j=1}^{M} \rho \log \frac{\rho}{\rho_j} + (1-\rho) \log \frac{1-\rho}{1-\rho_j}$$

where M is the number of hidden layer neurons and ρ is the sparsity parameter.

Note that KL divergence is minimum when $\rho = \rho_j$ for all j (i.e. when average activations are uniform and equal to a very low value ρ).

The cost function for the sparse autoencoder (SAE) is given by:

$$J_1(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{b}') = J(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{b}') + \beta D(\boldsymbol{H})$$

For gradient descent (complete derivation in Lecture Notes):

$$\frac{\partial J_1(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{b}')}{\partial \boldsymbol{W}} = \frac{\partial J(\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{b}')}{\partial \boldsymbol{W}} + \beta \frac{\partial D(\boldsymbol{H})}{\partial \boldsymbol{W}}$$

$$\frac{\partial D(\boldsymbol{H})}{\partial \boldsymbol{w}_j} = \frac{1}{P} \left(\frac{\rho}{\rho_j} + \frac{1-\rho}{1-\rho_j} \right) \sum_p f'(u_{pj}) \boldsymbol{x}_p$$

8.7 Deep Stacked Autoencoders

Deep autoencoders can be built by stacking autoencoders. After training the first level autoencoder, the resulting representation is used to train a second level encoder. The second hidden layer attempts to reconstruct the output of the first hidden layer. This process is repeated and a deep stacked autoencoder is realized.

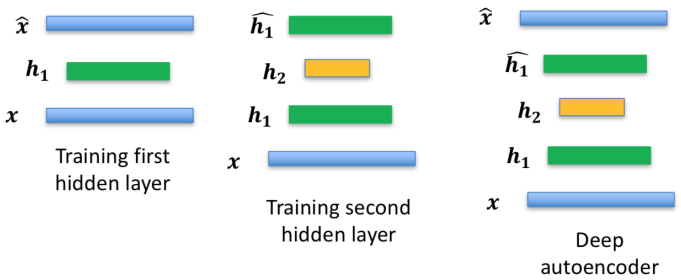

Figure 8.2: Training Stacked Autoencoders

8.7.1 Fine-Tuning Deep Network for Classification

After training a stacked autoencoder, an output layer (softmax or linear layer) may be added on the top of the stack. The parameters of the entire system are fine-tuned to minimize the error in predicting the supervised target by supervised gradient descent learning.

Chapter 9

Restricted Boltzmann Machines

9.1 Energy-Based Models (EBM)

Energy-based models associate a scalar *energy* $E(\boldsymbol{x})$ to each configuration of the variables $\boldsymbol{x}$ of interest. They define the probability distribution through an energy function:

$$P(\boldsymbol{x}) = \frac{e^{-E(\boldsymbol{x})}}{Z_1}$$

where the normalizing constant Z_1 is called the *partition function* and is given by:

$$Z_1 = \sum_{\boldsymbol{x}} e^{-E(\boldsymbol{x})}$$

The summation in the partition function is taken over the sample space (all the configurations) of $\boldsymbol{x}$.

The learning of energy-based models corresponds to modifying the energy function such that its shape reaches desirable properties such as minimum energy.

9.1.1 EBM Learning

An EBM is learned by performing (stochastic) gradient descent on the empirical negative log-likelihood of the training data. Thus, the cost function $J(\boldsymbol{\theta})$ is defined as:

$$J(\boldsymbol{\theta}) = -\sum_{p} \log(P(\boldsymbol{x}_p))$$

Parameters $\boldsymbol{\theta}$ (i.e. biases and weights) are found by computing the gradient $\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$ and performing gradient descent.

9.1.2 EBM w/ Hidden Units

In many cases of interest, we do not observe $\boldsymbol{x}$ fully or we want to include non-observed variables such as variables of hidden units.

Let $\boldsymbol{x}$ be the observed part (usually the input) and $\boldsymbol{h}$ be the non-observed part (i.e. hidden variables). Then, the joint probability of both observed and non-observed variables can be written as:

$$P(\boldsymbol{x}, \boldsymbol{h}) = \frac{e^{-E(\boldsymbol{x},\boldsymbol{h})}}{Z}$$

where

$$Z = \sum_{\boldsymbol{x},\boldsymbol{h}} e^{-E(\boldsymbol{x},\boldsymbol{h})}$$

Then, the probability of data $P(\boldsymbol{x})$ is obtained by marginalizing $P(\boldsymbol{x}, \boldsymbol{h})$ over the hidden variables $\boldsymbol{h}$:

$$P(\boldsymbol{x}) = \sum_{\boldsymbol{h}} P(\boldsymbol{x}, \boldsymbol{h}) = \frac{\sum_{\boldsymbol{h}} e^{-E(\boldsymbol{x},\boldsymbol{h})}}{Z} \quad \text{(A)}$$

For energy-based models with hidden units, the notion of *free energy* $F(\boldsymbol{x})$ is used such that:

$$P(\boldsymbol{x}) = \frac{e^{-F(\boldsymbol{x})}}{Z} \quad \text{(B)}$$

where $Z = \sum_{\boldsymbol{x}} e^{-F(\boldsymbol{x})}$.

Thus, by comparing (A) & (B):

$$F(\boldsymbol{x}) = -\log\left(\sum_{\boldsymbol{h}} e^{-E(\boldsymbol{x},\boldsymbol{h})}\right)$$

9.1.2.1 Gradient Descent

The gradient is defined as (complete derivation in Lecture Notes):

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \frac{\partial F(\boldsymbol{x})}{\partial \boldsymbol{\theta}} - \sum_{\tilde{\boldsymbol{x}}} p(\tilde{\boldsymbol{x}}) \frac{\partial F(\tilde{\boldsymbol{x}})}{\partial \boldsymbol{\theta}}$$

where the LHS of the minus sign is the positive phase and the RHS is the negative phase. The positive phase gradients are generated by the training data and increase the cost while

the negative phase gradients decrease the cost. Note that the samples $\tilde{\boldsymbol{x}}$ in the negative phase are generated by the network (i.e. drawn from the model).

From above:

$$J(\boldsymbol{\theta}) = F(\boldsymbol{x}) - \sum_{\tilde{\boldsymbol{x}}} p(\tilde{\boldsymbol{x}}) F(\tilde{\boldsymbol{x}})$$

If N samples are drawn and equally likely:

$$J(\boldsymbol{\theta}) = F(\boldsymbol{x}) - \frac{1}{N} \sum_{\tilde{\boldsymbol{x}}} F(\tilde{\boldsymbol{x}})$$

Usually, $N = P$ (i.e. no. of training samples):

$$J(\boldsymbol{\theta}) = F(\text{positive samples}) - F(\text{negative samples})$$

That is, the positive phase of the cost function corresponds to the free energy computed over the training data (positive samples) and the negative phase corresponds to the free energy computed over the samples generated by the model (negative samples). The cost is given by the difference of the two computed free energies.

9.2 Restricted Boltzmann Machines

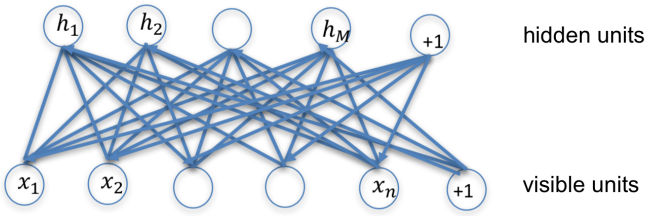

Figure 9.1: Restricted Boltzmann Machine

A RBM is an EBM, defined by the energy function:

$$E(\boldsymbol{x}, \boldsymbol{h}) = -\boldsymbol{b}^T \boldsymbol{x} - \boldsymbol{c}^T \boldsymbol{h} - \boldsymbol{h}^T \boldsymbol{W}^T \boldsymbol{x}$$

- $\boldsymbol{x} = (x_1, x_2, \cdots x_n)^T$ are the activations of visible units (i.e. inputs).
- $\boldsymbol{h} = (h_1, h_2, \cdots h_M)^T$ are the activations of hidden units.
- $\boldsymbol{W} = \{w_{ij}\}_{n \times M}$ is the weight matrix connecting the inputs to the hidden units.
- $\boldsymbol{b}$ is the bias vector connected to the input layer (top down).
- $\boldsymbol{c}$ is the bias vector connected to the hidden layer (bottom up).

The energy of the RBM can also be written as:

$$E(\boldsymbol{x}, \boldsymbol{h}) = -\sum_i b_i x_i - \sum_j c_j h_j - \sum_i \sum_j h_j w_{ij} x_i$$

Note that the weights w_{ij} are bidirectional.

Synaptic input $\boldsymbol{s}$ to the hidden units:

$$\boldsymbol{s} = \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{c}$$

Synaptic input to the jth hidden unit:

$$s_j = \boldsymbol{w}_j{}^T \boldsymbol{x} + c_j$$

Synaptic input $\boldsymbol{u}$ to the visible (input) units from hidden units:

$$\boldsymbol{u} = \boldsymbol{W}\boldsymbol{h} + \boldsymbol{b}$$

Synaptic input to the ith input unit:

$$u_i = \boldsymbol{w'}_i{}^T h + b_i$$

9.2.1 Conditional Probabilities

The conditional probability of hidden activations given the input is given by:

$$P(\boldsymbol{h}|\boldsymbol{x}) = \frac{P(\boldsymbol{h}, \boldsymbol{x})}{P(\boldsymbol{x})} = \frac{P(\boldsymbol{h}, \boldsymbol{x})}{\sum_{\boldsymbol{h}} P(\boldsymbol{h}, \boldsymbol{x})}$$

After substituting $P(\boldsymbol{x}, \boldsymbol{h}) = \frac{e^{-E(\boldsymbol{x}, \boldsymbol{h})}}{Z}$ (complete derivation in Lecture Notes):

$$P(\boldsymbol{h}|\boldsymbol{x}) = \prod_j P(h_j|\boldsymbol{x})$$

Similarly,

$$P(\boldsymbol{x}|\boldsymbol{h}) = \prod_i P(x_i|\boldsymbol{h})$$

That is, because of the specific structure of RBMs, visible and hidden units are conditionally independent given one another, where the conditional probabilities are given by:

$$P(h_j|\boldsymbol{x}) = \frac{e^{-E(\boldsymbol{x}, h_j)}}{\sum_{h_j} e^{-E(\boldsymbol{x}, h_j)}}$$

$$P(x_i|\boldsymbol{h}) = \frac{e^{-E(x_i, \boldsymbol{h})}}{\sum_{x_i} e^{-E(x_i, \boldsymbol{h})}}$$

and $E(\boldsymbol{x}, h_j) = -h_j(\boldsymbol{x}^T \boldsymbol{w}_j + c_j)$ and $E(x_i, \boldsymbol{h}) = -x_i(\boldsymbol{h}^T \boldsymbol{w'}_i + b_i)$.

9.2.2 Sampling in RBM

Samples of $P(\boldsymbol{x})$ are obtained by running a *Markov chain* to convergence, using *Gibbs sampling* as the transition operator.

A *Markov chain* is a sequence of states, which satisfies the Markov property — future states are determined only by the current state.

Gibbs sampling of a joint distribution of N random variables $\boldsymbol{s} = (s_1, s_2, \cdots s_N)$ is done through N sampling subsets of the form $s_i \sim P(s_i | s_{-i})$ where s_{-i} contains $N-1$ variables in $\boldsymbol{s}$ excluding the variable s_i.

Gibbs sampling forms a Markov chain and when executed long enough, converges to the samples from the true distribution.

In a RBM, the set of random variables consists of the set of hidden units and visible units. However, since they are conditionally independent, one can perform block Gibbs sampling. In this setting, visible units are sampled given the fixed values of hidden units (i.e. from $P(\boldsymbol{x}|\boldsymbol{h})$) and similarly, hidden units are sampled simultaneously given the values of visible units (i.e. from $P(\boldsymbol{h}|\boldsymbol{x})$).

For t Gibbs sampling steps starting from the training samples (i.e. $\hat{P}(\boldsymbol{x})$):

$$
\begin{aligned}
\boldsymbol{x}^{(1)} &\sim \hat{P}(\boldsymbol{x}) \\
\boldsymbol{h}^{(1)} &\sim P(\boldsymbol{h}|\boldsymbol{x}^{(1)}) \\
\boldsymbol{x}^{(2)} &\sim P(\boldsymbol{x}|\boldsymbol{h}^{(1)}) \\
\boldsymbol{h}^{(2)} &\sim P(\boldsymbol{h}|\boldsymbol{x}^{(2)}) \\
\boldsymbol{x}^{(3)} &\sim P(\boldsymbol{x}|\boldsymbol{h}^{(2)}) \\
&\vdots \\
\boldsymbol{x}^{(t+1)} &\sim P(\boldsymbol{x}|\boldsymbol{h}^{(t)})
\end{aligned}
$$

It makes sense to start with training samples $\hat{P}(\boldsymbol{x})$ because as the model becomes better at capturing the structure, model distribution P and $\hat{P}$ become similar.

As $t \to \infty$, samples $(\boldsymbol{x}^{(t)}, \boldsymbol{h}^{(t)})$ are guaranteed to be samples from $P(\boldsymbol{x}, \boldsymbol{h})$.

In theory, each parameter update in the learning process requires running a chain for convergence (to generate negative samples). This is very computationally prohibitive. As such, several algorithms have been devised in order to efficiently sample from $P(\boldsymbol{x}, \boldsymbol{h})$ during learning.

9.2.3 RBM w/ Binary Units

RBMs are mostly studied with binary units. That is, $x_i, h_i \in \{0, 1\}$.

For this case (complete derivation in Lecture Notes):

$$P(h_j = 1|\boldsymbol{x}) = \text{logistic}(\boldsymbol{x}^T \boldsymbol{w}_j + c_j)$$

where $s_j = \boldsymbol{x}^T \boldsymbol{w}_j$ + c_j.

Similarly,

$$P(x_i = 1|\boldsymbol{h}) = \text{logistic}(\boldsymbol{h}^T \boldsymbol{w'}_i + b_i)$$

where $u_i = \boldsymbol{h}^T \boldsymbol{w'}_i$ + b_i.

Outputs are sampled by drawing bits from binomial distributions with the above probabilities:

$$
\begin{aligned}
\boldsymbol{h} &= \text{Binomial}\big(p = \text{logistic}(\boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{c})\big) \\
\boldsymbol{x} &= \text{Binomial}\big(p = \text{logistic}(\boldsymbol{W}\boldsymbol{h} + \boldsymbol{b})\big)
\end{aligned}
$$

The t-th step in the Markov chain becomes:

$$
\begin{aligned}
\boldsymbol{h}^{(t+1)} &\sim \text{logistic}(\boldsymbol{W}^T \boldsymbol{x}^{(t)} + \boldsymbol{c}) \\
\boldsymbol{x}^{(t+1)} &\sim \text{logistic}(\boldsymbol{W}\boldsymbol{h}^{(t)} + \boldsymbol{b})
\end{aligned}
$$

Since the outputs of hidden and visible units are binary, the samples are drawn from binomial distributions with probabilities p given by the corresponding activations:

$$
\begin{aligned}
\boldsymbol{h}^{(t+1)} &= \text{Binomial}\big(p = \text{logistic}(\boldsymbol{W}^T \boldsymbol{x}^{(t)} + \boldsymbol{c})\big) \\
\boldsymbol{x}^{(t+1)} &= \text{Binomial}\big(p = \text{logistic}(\boldsymbol{W}\boldsymbol{h}^{(t)} + \boldsymbol{b})\big)
\end{aligned}
$$

9.2.4 Free Energy of RBM

Substituting energy $E(\boldsymbol{x}, \boldsymbol{h})$ for RBM into the equation for free energy $F(\boldsymbol{x})$:

$$F(\boldsymbol{x}) = -\boldsymbol{b}^T \boldsymbol{x} - \sum_j \log \sum_{h_j} e^{h_j(\boldsymbol{x}^T \boldsymbol{w}_j + c_j)}$$

Since $E(\boldsymbol{x}, h_j) = -h_j(\boldsymbol{x}^T \boldsymbol{w}_j + c_j)$, free energy is given by:

$$F(\boldsymbol{x}) = -\boldsymbol{b}^T \boldsymbol{x} - \sum_j \log \sum_{h_j} e^{E(\boldsymbol{x}, h_j)}$$

For RBM with binary units:

$$F(\boldsymbol{x}) = -\boldsymbol{b}^T \boldsymbol{x} - \sum_j \log \left(1 + e^{(\boldsymbol{x}^T \boldsymbol{w}_j + c_j)}\right)$$

Note that the cost function is computed as the difference of free energies of positive (training) samples and negative (model) samples:

$$J(\boldsymbol{\theta}) = F(\boldsymbol{x}) - \frac{1}{N}\sum_{\tilde{\boldsymbol{x}}} F(\tilde{\boldsymbol{x}})$$

9.3 Contrastive Divergence (CD-k)

Contrastive Divergence uses two tricks to speed up the sampling process:

1. Since we eventually want $P(\boldsymbol{x}) = \hat{P}(\boldsymbol{x})$ i.e. to be close to the training process, we initialize the Markov chain with a training example.
2. CD does not wait for the chain to converge and samples are taken after k steps of Gibbs sampling. In practice, $k = 1$ has been shown to work surprisingly well.

9.4 Persistent CD

Persistent CD uses another approximation for sampling from $P(\boldsymbol{x}, \boldsymbol{h})$. It relies on a single Markov chain, which has a persistent chain (i.e. not restarting the chain for each observed sample). For each parameter update, we extract new samples by simply running the chain for k steps. The state of the chain is then preserved for subsequent updates.

The general intuition is that if parameter updates are small enough compared to the mixing rate of the chain, the Markov chain should be able to 'catch up' to the changes of the model.

9.5 Tracking Progress of Training

RBMs are particularly tricky to train because of the partition function Z, the log likelihood $\log P(\boldsymbol{x})$ cannot be estimated during training. Therefore, we need some useful indicators to determine hyperparameters.

Inspection of Negative Samples: Negative samples generated by the model can be visualized. As the training progresses, the negative samples should look like the training samples.

Visual Inspection of Filters: Filters (weights) learned by the model can be visualized (as gray level images, for example). They should pick up useful features of data.

Proxies to Likelihood:

1. Reconstruction Error (Cross-Entropy)
2. Pseudo-Likelihood (for Persistent CD)

9.5.1 Reconstruction Error

For input $\boldsymbol{x}$,

Synaptic input at hidden units $\boldsymbol{s} = \boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{c}$.

Hidden layer activations: $P(\boldsymbol{h} = 1|\boldsymbol{x}) = f(\boldsymbol{s}) = \frac{1}{1+e^{-\boldsymbol{s}}}$

Hidden layer output: $\boldsymbol{h} = \text{Binomial}(p = f(\boldsymbol{s}))$

Synaptic input to input layer $\boldsymbol{u} = \boldsymbol{W}\boldsymbol{h} + \boldsymbol{b}$

Input layer activations: $P(\boldsymbol{x} = 1|\boldsymbol{h}) = f(\boldsymbol{u}) = \frac{1}{1+e^{-\boldsymbol{u}}}$

Sampled input: $\boldsymbol{x} = \text{Binomial}(p = f(\boldsymbol{u}))$

Sampling is done k times...

$$\text{Cross-Entropy } = \sum_p \sum_{pi} x_{pi} f(u_{pi}) + (1 - x_{pi})(1 - f(u_{pi}))$$

9.5.2 Pseudo-Likelihood for RBM

Pseudo-likelihood is less expensive to compute as it assumes that all the elements (i.e. the dimensions of input) are independent.

PL is defined as:

$$\text{PL}(\boldsymbol{x}) = \prod_i P(x_i|x_{-i})$$

or

$$\log \text{PL}(\boldsymbol{x}) = \prod_i \log P(x_i|x_{-i})$$

Note that x_{-i} denotes the set of bits in $\boldsymbol{x}$ except the bit x_i. Therefore, the PL is the sum of log probabilities of each bit x_i, given the condition of the state of all other bits.

However, for high-dimensional data, summing over the RHS would be rather expensive. Therefore, the following approximation is used:

$$\log \text{PL}(\boldsymbol{x}) = \frac{1}{m} \sum_{i \sim \text{Uniform}[1,n]} n \log P(x_i|x_{-i})$$

where n is the number of visible units and the summation is taken over m visible units uniformly chosen from $[1, n]$. Often, m is chosen to be 1.

$$\log \text{PL}(\boldsymbol{x}) \approx n \log P(x_i|x_{-i}) = n \log \frac{e^{-F(\boldsymbol{x})}}{\sum_{\boldsymbol{x}} e^{-F(\boldsymbol{x})}}$$

Let $\overline{\boldsymbol{x}_i}$ be the $\boldsymbol{x}$ with x_i being flipped. Then, for binary units:

$$\log \text{PL}(\boldsymbol{x}) \approx n \log \left(\text{logistic}(F(\overline{\boldsymbol{x}_i}) - F(\boldsymbol{x}))\right)$$

Chapter 10

Recurrent Neural Networks

10.1 Computational Graphs

The computations in a neural network can be described in a *computational graph*. Each *node* in the graph indicates a variable that could be a scalar, vector, matrix, tensor or a variable of another type. A *directed edge* denotes an *operation* executed on one or more variables that results in another variable. A set of allowable operations defines the language of the computational graph. *Functions* are described by a set of operations together.

10.2 Recurrent Neural Networks (RNNs)

RNNs are designed to process sequential information (i.e. data represented as sequences). The next data point in a sequence is usually dependent on the current data point and RNNs attempt to capture this dependency.

RNNs are called *recurrent* because they perform the same task for every element of a sequence, with the output depending on the previous computations. RNNs have memory that captures information about what has been processed so far. It achieves this by using feedback connections that enable learning of sequential (temporal) information of sequences.

10.2.1 Types of RNNs

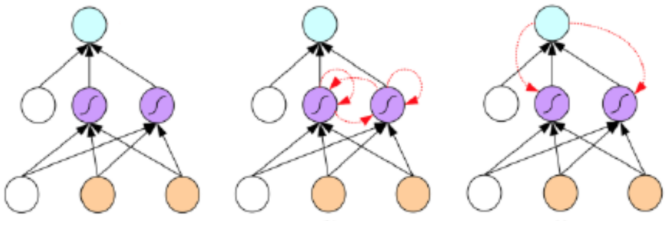

Figure 10.1: Types of RNNs

From left to right:

- Feedforward Neural Network
- RNN w/ Hidden Recurrence (Elman-type)
- RNN w/ Top-Down Recurrence (Jordan-type)

10.3 RNN w/ Hidden Recurrence

Recurrent networks that produce an output at each time step and have recurrent connections between hidden units.

In Elman-type RNNs, the output at the hidden layer at time $t-1$ is kept and fed to the hidden layer at time t together with the raw input $x(t)$ for time t.

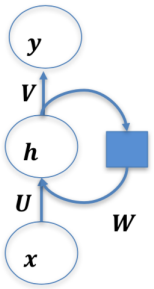

Figure 10.2: Elman-type RNN

Note that time t is assumed to be discretized and activations are updated at each time instance t. The delay unit (blue box) is added to indicate that the activation is held until the next time instance.

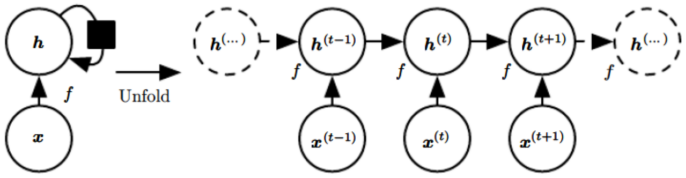

Figure 10.3: RNN w/ Hidden Recurrence (Unfolded)

By considering the unfolded recurrence:

$$\begin{aligned}\boldsymbol{h}(t) &= g^t(\boldsymbol{x}(t), \boldsymbol{x}(t-1), \cdots, \boldsymbol{x}(2), \boldsymbol{x}(1)) \\ &= f(\boldsymbol{h}(t-1), \boldsymbol{x}(t))\end{aligned}$$

where the function g^t takes the whole past sequence as input and produces the hidden layer activation. The unfolded recurrent structure allows us to factorize g^t into repeated applications of a function f.

The unfolding process has two major advantages:

1. Regardless of the sequence length, the learned model always has the same size rather than specified in terms of a variable-length history of states.
2. It is possible to use the same transition function f with the same parameters at every time step.

10.3.1 Forward Propagation

Let $\boldsymbol{x}(t)$, $\boldsymbol{y}(t)$ and $\boldsymbol{h}(t)$ be the input, output and hidden output of the network at time t. Activation of the three-layer Elman-type RNN is given by:

$$\begin{aligned}\boldsymbol{h}(t) &= \phi(\boldsymbol{U}^T\boldsymbol{x}(t) + \boldsymbol{W}^T\boldsymbol{h}(t-1) + \boldsymbol{b}) \\ \boldsymbol{y}(t) &= \sigma(\boldsymbol{V}^T\boldsymbol{h}(t) + \boldsymbol{c})\end{aligned}$$

- $\boldsymbol{U}$ - Weight vector that transforms raw inputs to the hidden layer activations.
- $\boldsymbol{W}$ - Recurrent weight vector connecting previous hidden layer outputs to the current hidden layer input.
- $\boldsymbol{V}$ - Weight vector connecting the hidden layer to the output layer.
- $\boldsymbol{b}$ - Bias connected to the hidden layer.
- $\boldsymbol{c}$ - Bias connected to the output layer.
- ϕ is the tanh activation function (for hidden layer) and σ is the output layer activation function.

Note that the output layer activation function is softmax for classification and linear/sigmoidal for regression.

Typically, the hidden layer activation function is given by the tanh function:

$$\begin{aligned}\phi(u) = \tanh(u) = \frac{e^u - e^{-u}}{e^u + e^{-u}} = \frac{1 - e^{-2u}}{1 + e^{-2u}} \\ \phi'(u) = 1 - \phi(u)^2\end{aligned}$$

10.3.2 Deep RNN w/ Hidden Recurrence

First hidden layer, $l = 1$:

$$\boldsymbol{h}^1(t) = \phi\Big({\boldsymbol{U}^1}^T\boldsymbol{x}(t) + {\boldsymbol{W}^1}^T\boldsymbol{h}^1(t-1) + \boldsymbol{b}^1\Big)$$

For hidden layers $l = 2, \cdots, L-1$:

$$\boldsymbol{h}^l(t) = \phi\Big({\boldsymbol{U}^l}^T\boldsymbol{h}^{l-1}(t) + {\boldsymbol{W}^l}^T\boldsymbol{h}^l(t-1) + \boldsymbol{b}^l\Big)$$

For output layer, $l = L$:

$$\boldsymbol{h}^L(t) = \sigma\Big({\boldsymbol{U}^L}^T\boldsymbol{h}^{L-1}(t) + \boldsymbol{b}^L\Big)$$

10.4 RNN w/ Top-Down Recurrence

In Jordan-type RNNs, the output of the output layer at time $t-1$ is kept and fed to the hidden layer at time t together with the raw input $x(t)$ for time t.

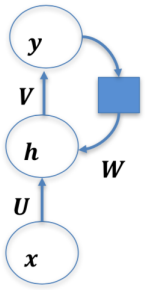

Figure 10.4: RNN w/ Top-Down Recurrence

Note that time t is assumed to be discretized and activations are updated at each time instance t. The delay unit (blue box) is added to indicate that the activation is held until the next time instance.

10.4.1 Forward Propagation

Let $\boldsymbol{x}(t)$, $\boldsymbol{y}(t)$ and $\boldsymbol{h}(t)$ be the input, output and hidden output of the network at time t. Activation of the three-layer Jordan-type RNN is given by:

$$\begin{aligned}\boldsymbol{h}(t) &= \phi(\boldsymbol{U}^T\boldsymbol{x}(t) + \boldsymbol{W}^T\boldsymbol{y}(t-1) + \boldsymbol{b}) \\ \boldsymbol{y}(t) &= \sigma(\boldsymbol{V}^T\boldsymbol{h}(t) + \boldsymbol{c})\end{aligned}$$

where $\boldsymbol{W}$ represents the recurrent weight matrix connecting the previous output to the current hidden input.

10.4.2 Deep RNN w/ Top-Down Recurrence

First hidden layer, $l = 1$:

$$\boldsymbol{h}^1(t) = \phi\Big({\boldsymbol{U}^1}^T \boldsymbol{x}(t) + {\boldsymbol{W}^1}^T \boldsymbol{h}^2(t-1) + \boldsymbol{b}^1\Big)$$

For hidden layers $l = 2, \cdots, L-2$:

$$\boldsymbol{h}^l(t) = \phi\Big({\boldsymbol{U}^l}^T \boldsymbol{h}^{l-1}(t) + {\boldsymbol{W}^l}^T \boldsymbol{h}^{l+1}(t-1) + \boldsymbol{b}^l\Big)$$

For hidden layer, $l = L-1$:

$$\boldsymbol{h}^{L-1}(t) = \phi\Big({\boldsymbol{U}^{L-1}}^T \boldsymbol{h}^{L-2}(t) + {\boldsymbol{W}^{L-1}}^T \boldsymbol{y}(t-1) + \boldsymbol{b}^{L-1}\Big)$$

For output layer, $l = L$:

$$\boldsymbol{h}^L(t) = \sigma\Big({\boldsymbol{U}^L}^T \boldsymbol{h}^{L-1}(t) + \boldsymbol{b}^L\Big)$$

10.5 Chain Rule in Multidimensions

If $x \in \boldsymbol{R}$, $x \in \boldsymbol{R}$ and $x \in \boldsymbol{R}$, and $y = f(x)$ and $z = g(y)$, the chain rule of differentiation says:

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$

If $\boldsymbol{x} = (x_i) \in \boldsymbol{R}^n$, $\boldsymbol{y} = (y_k) \in \boldsymbol{R}^K$ and $z \in \boldsymbol{R}$, and $\boldsymbol{y} = f(\boldsymbol{x})$ and $z = g(\boldsymbol{y})$, the chain rule of differentiation in multidimensions is:

$$\frac{\partial z}{\partial x_i} = \sum_k \frac{\partial z}{\partial y_k}\frac{\partial y_k}{\partial x_i}$$

This can be rewritten as:

$$\frac{\partial z}{\partial x_i} = \left(\frac{\partial \boldsymbol{y}}{\partial x_i}\right)^T \frac{\partial z}{\partial \boldsymbol{y}}$$

$$\frac{\partial \boldsymbol{y}}{\partial x_i} = \begin{pmatrix} \frac{\partial y_1}{\partial x_i} \\ \vdots \\ \frac{\partial y_K}{\partial x_i} \end{pmatrix} \quad \frac{\partial z}{\partial \boldsymbol{y}} = \begin{pmatrix} \frac{\partial z}{\partial y_1} \\ \vdots \\ \frac{\partial z}{\partial y_K} \end{pmatrix}$$

This can be further rewritten as:

$$\frac{\partial z}{\partial \boldsymbol{x}} = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^T \frac{\partial z}{\partial \boldsymbol{y}}$$

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_K}{\partial x_1} \\ \vdots & \vdots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \cdots & \frac{\partial y_K}{\partial x_n} \end{pmatrix}$$

$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ is called the *Jacobian* of the transformation $\boldsymbol{y} = f(\boldsymbol{x})$.

10.6 Gradient w/ Respect to a Tensor

Consider a scalar function z and an m-rank tensor $\boldsymbol{X}$. Let the gradient of z with respect to $\boldsymbol{X}$ be denoted by $\nabla_{\boldsymbol{X}} z$:

$$\frac{\partial z}{\partial \boldsymbol{X}} = \nabla_{\boldsymbol{X}} z$$

Assume that a vector with index j maps all dimensions in m ranks of the tensor. Then the gradient of z with respect to the jth element X_j of the tensor is denoted by:

$$(\nabla_{\boldsymbol{X}} z)_j = \frac{\partial z}{\partial X_j}$$

If $\boldsymbol{Y} = f(\boldsymbol{X})$ and $z = g(\boldsymbol{Y})$, then from the chain rule of differentiation:

$$\nabla_{\boldsymbol{X}} z = \sum_j (\nabla_{\boldsymbol{X}} Y_j)\frac{\partial z}{\partial Y_j}$$

10.7 Neuron Layers Revisited

Figure 10.5: Computational Graph of a Neuron Layer

For a softmax layer:

$$J = -\sum_k 1(d = \boldsymbol{k}) \log f(u_k)$$
$$\frac{\partial J}{\partial \boldsymbol{u}} = -(1(d = \boldsymbol{k}) - f(\boldsymbol{u}))$$

For a perceptron layer:

$$J = (\boldsymbol{d} - f(\boldsymbol{u}))^2$$
$$\frac{\partial J}{\partial \boldsymbol{u}} = -2(\boldsymbol{d} - f(\boldsymbol{u}))$$

The synaptic input is given by:

$$\boldsymbol{u} = \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{b}$$

Jacobian:

$$\frac{\partial \boldsymbol{u}}{\partial \boldsymbol{x}} = \boldsymbol{W}$$

Therefore,

$$\frac{\partial J}{\partial \boldsymbol{x}} = \left(\frac{\partial \boldsymbol{u}}{\partial \boldsymbol{x}}\right)^T \frac{\partial J}{\partial \boldsymbol{u}} = \boldsymbol{W}^T \frac{\partial J}{\partial \boldsymbol{u}}$$

That is, the error terms at the synaptic inputs can be transformed to the input by multiplying by the transpose of the Jacobian.

Therefore,

$$\begin{aligned}\nabla_{\boldsymbol{W}} J &= \sum_k (\nabla_{\boldsymbol{W}} u_k) \frac{\partial J}{\partial u_k} \\ &= \boldsymbol{x}\left(\frac{\partial J}{\partial \boldsymbol{u}}\right)^T \\ \nabla_{\boldsymbol{b}} J &= \left(\frac{\partial J}{\partial \boldsymbol{u}}\right)\end{aligned}$$

10.8 Backpropagation Revisited

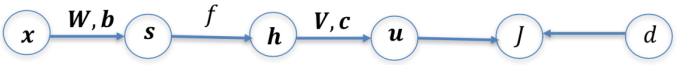

Figure 10.6: Computational Graph of Feedforward Network

$$\frac{\partial J}{\partial \boldsymbol{h}} = \left(\frac{\partial \boldsymbol{u}}{\partial \boldsymbol{h}}\right)^T \frac{\partial J}{\partial \boldsymbol{u}} = \boldsymbol{V}^T \frac{\partial J}{\partial \boldsymbol{u}}$$
$$\frac{\partial J}{\partial \boldsymbol{s}} = \left(\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{s}}\right)^T \frac{\partial J}{\partial \boldsymbol{h}}$$

where $\boldsymbol{h} = f(\boldsymbol{s})$ and $\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{s}} = \text{diag}(f'(\boldsymbol{s}))$. $\text{diag}(f'(\boldsymbol{s}))$ is a diagonal matrix with diagonal elements $f'(s_i)$.

Substituting,

$$\nabla_{\boldsymbol{W}} J = \boldsymbol{x}\left(\frac{\partial J}{\partial \boldsymbol{s}}\right)^T = \boldsymbol{x}\left(\frac{\partial J}{\partial \boldsymbol{u}}\right)^T \boldsymbol{V}\,\text{diag}(f'(s))$$
$$\nabla_{\boldsymbol{b}} J = \frac{\partial J}{\partial \boldsymbol{s}} = \text{diag}(f'(\boldsymbol{s}))\boldsymbol{V}^T \frac{\partial J}{\partial \boldsymbol{u}}$$

10.9 Backpropagation Through Time (BPTT) for RNN w/ Hidden Reccurence

The gradient computation involves performing a forward propagation pass moving left to right through the unfolded graph of the RNN, followed by a backpropagation pass moving right to left through the graph.

The runtime is $O(\tau)$ where τ is the length of the input sequence and cannot be reduced by parallelization because the forward propagation is inherently sequential.

Hence, a network with recurrence between hidden units is very powerful but also expensive to train.

Forward Propagation Equations:

$$\boldsymbol{h}(t) = \tanh(\boldsymbol{U}^T \boldsymbol{x}(t) + \boldsymbol{W}^T \boldsymbol{h}(t-1) + \boldsymbol{b})$$
$$\boldsymbol{u}(t) = \boldsymbol{V}^T \boldsymbol{h}(t) + \boldsymbol{c}$$

For classification, $\boldsymbol{y}(t) = \text{softmax}(\boldsymbol{u}(t))$.

For regression, $\boldsymbol{y}(t) = \boldsymbol{u}(t)$.

Total Error:

$$J = \sum_{t=1}^{\tau} J(t)$$

Note that $\frac{\partial J}{\partial J(t)} = 1$.

Error Gradients at Output:

$$\nabla_{\boldsymbol{u}(t)} J = \nabla_{\boldsymbol{u}(t)} J(t) = \begin{cases} -(1(\boldsymbol{k} = d(t)) - f(\boldsymbol{u})) & \text{for softmax} \\ -2(\boldsymbol{d}(t) - f(\boldsymbol{u})) & \text{for linear regression} \end{cases}$$

The gradients propagate backward, starting from the end of the sequence. At the final time step τ, $\boldsymbol{h}(t)$ has only one descendant. Hence, for $t = \tau$:

$$\nabla_{\boldsymbol{h}(\tau)} J = \boldsymbol{V}^T \nabla_{\boldsymbol{u}(\tau)} J$$

We can then iterate backward in time to backpropagate gradients. For $t < \tau$, $\boldsymbol{h}(t)$ has descendants both $\boldsymbol{h}(t+1)$ and $\boldsymbol{u}(t)$. Therefore, for $t = 1, \cdots, \tau - 1$:

$$\nabla_{\boldsymbol{h}(t)} J = \left(\frac{\partial \boldsymbol{h}(t+1)}{\partial \boldsymbol{h}(t)}\right)^T \nabla_{\boldsymbol{h}(t+1)} J + \left(\frac{\partial \boldsymbol{u}(t)}{\partial \boldsymbol{h}(t)}\right)^T \nabla_{\boldsymbol{u}(t)} J$$

where:

$$\frac{\partial \boldsymbol{h}(t+1)}{\partial \boldsymbol{h}(t)} = \frac{\partial \boldsymbol{h}(t+1)}{\partial \boldsymbol{s}(t+1)} \frac{\partial \boldsymbol{s}(t+1)}{\partial \boldsymbol{h}(t)} = \text{diag}(1 - \boldsymbol{h}^2(t+1))\boldsymbol{W}$$
$$\frac{\partial \boldsymbol{u}(t)}{\partial \boldsymbol{h}(t)} = \boldsymbol{V}$$

Substituting,

$$\nabla_{\boldsymbol{h}(t)} J = \boldsymbol{W}^T \text{diag}(1 - \boldsymbol{h}^2(t+1)) \nabla_{\boldsymbol{h}(t+1)} J + \boldsymbol{V} \nabla_{\boldsymbol{u}(t)} J$$

Using above error gradients, we can calculate the gradients with respect to the parameters $\boldsymbol{\theta} = \{\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{W}, \boldsymbol{b}, \boldsymbol{c}\}$:

$$\nabla_{\boldsymbol{c}} J = \sum_t \frac{\partial J(t)}{\partial \boldsymbol{c}} = \sum_t \left(\frac{\partial \boldsymbol{u}(t)}{\partial \boldsymbol{c}} \right)^T \nabla_{\boldsymbol{u}(t)} J = \sum_t \nabla_{\boldsymbol{u}(t)} J$$

$$\nabla_{\boldsymbol{b}} J = \sum_t \left(\frac{\partial \boldsymbol{h}(t)}{\partial \boldsymbol{b}} \right)^T \nabla_{\boldsymbol{h}(t)} J = \sum_t \text{diag}(1 - \boldsymbol{h}(t)^2) \nabla_{\boldsymbol{h}(t)} J$$

$$\nabla_{\boldsymbol{V}} J = \sum_t \sum_i \frac{\partial \boldsymbol{J}}{\partial u_i(t)} \nabla_{\boldsymbol{V}} u_i(t) = \sum_t \boldsymbol{h}(t) (\nabla_{\boldsymbol{u}(t)} J)^T$$

$$\nabla_{\boldsymbol{W}} J = \sum_t \sum_i \frac{\partial \boldsymbol{J}}{\partial h_i(t)} \nabla_{\boldsymbol{W}} h_i(t) = \sum_t \boldsymbol{h}(t-1) (\nabla_{\boldsymbol{h}(t)} J)^T \text{diag}(1 - \boldsymbol{h}(t)^2)$$

$$\nabla_{\boldsymbol{U}} J = \sum_t \sum_i \frac{\partial \boldsymbol{J}}{\partial h_i(t)} \nabla_{\boldsymbol{U}} h_i(t) = \sum_t \boldsymbol{x}(t) (\nabla_{\boldsymbol{h}(t)} J)^T \text{diag}(1 - \boldsymbol{h}(t)^2)$$

Chapter 11

Gated RNNs

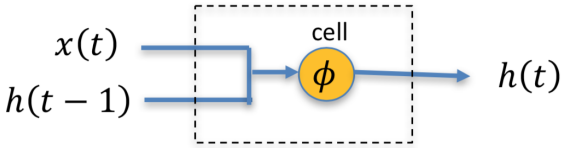

Figure 11.1: RNN Cell

The memory cell of a RNN is characterized by:

$$\boldsymbol{h}(t) = \phi(\boldsymbol{U}^T \boldsymbol{x}(t) + \boldsymbol{W}^T \boldsymbol{h}(t-1) + \boldsymbol{b})$$

where ϕ is the tanh activation function and the RNN cell is referred to as a tanh unit. RNNs built with simple tanh units are known as vanilla RNNs.

11.1 Vanishing & Exploding Gradients

Though RNNs have been proven to solve sequential problems, it is difficult to train them to learn long-term dynamics.

During gradient backpropagation, the gradient can end up being multiplied a large number of times (as many as the number of time steps) by the weight matrix associated with the connections between the neurons of the recurrent hidden layer. Note that each time the activations are forward propagated in time, the activations are multiplied by $\boldsymbol{W}$ and each time the gradients are backpropagated, the gradients are multiplied by $\boldsymbol{W}^T$.

If the weights in this matrix are small, it can lead to a situation called *vanishing gradients* where the gradient signal gets so small that learning either becomes very slow or stops working altogether.

Conversely, if the weights in this matrix are large, it can lead to a situation where the gradient signal is so large that it can cause learning to diverge. This is often referred to as *exploding gradients*.

Vanishing and exploding gradients in gradient backpropagation learning make it difficult for RNNs to learn long-term dependencies.

11.2 Long Short-Term Memory Unit

LSTMs provide a solution to the vanishing & exploding gradient problem by incorporating memory units that allow the network to learn when to forget previous hidden states and when to update hidden states given new information.

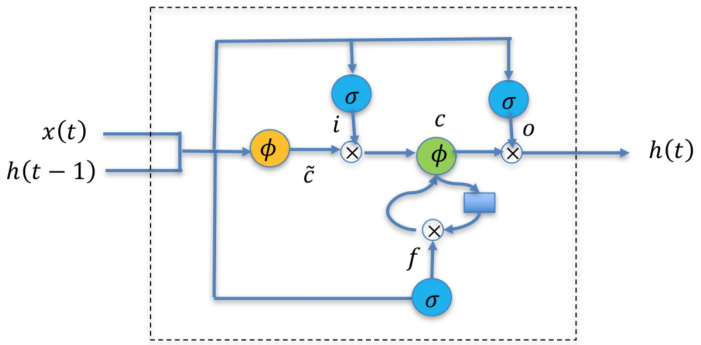

Figure 11.2: LSTM Unit

LSTMs introduce a gated cell where the information flow can be controlled. They have an internal recurrence (a self-loop) in addition to the outer recurrence of the RNN. The self-loop inside the cell is able to produce paths where the gradient can flow for a long duration. The weights on this self-loop are conditioned on the context rather than being fixed.

By making the weight of the self-loop gated, the time scale of integration can be changed dynamically. In this way, LSTMs help preserve error terms that can be propagated through many layers and time steps.

The gates serve to modulate the interactions between the memory cell and its environment. The self-recurrent connection has a weight of 1.0 and ensures that, barring any outside

interference, the state of a memory cell can remain constant from one time step to another.

The input gate can allow the incoming signal to alter the state of the memory cell or block it. On the other hand, the output gate can allow the state of the memory cell to have an effect on other neurons or prevent it. Finally, the forget gate can modulate the memory cell's self-recurrent connection allowing the cell to remember or forget its previous state, as needed.

11.2.1 Equations

$$\boldsymbol{i}(t) = \sigma(\boldsymbol{U}_i{}^T \boldsymbol{x}(t) + \boldsymbol{W}_i{}^T \boldsymbol{h}(t-1) + \boldsymbol{b}_i)$$
$$\boldsymbol{f}(t) = \sigma(\boldsymbol{U}_f{}^T \boldsymbol{x}(t) + \boldsymbol{W}_f{}^T \boldsymbol{h}(t-1) + \boldsymbol{b}_f)$$
$$\boldsymbol{o}(t) = \sigma(\boldsymbol{U}_o{}^T \boldsymbol{x}(t) + \boldsymbol{W}_o{}^T \boldsymbol{h}(t-1) + \boldsymbol{b}_o)$$
$$\tilde{\boldsymbol{c}}(t) = \phi(\boldsymbol{U}_c{}^T \boldsymbol{x}(t) + \boldsymbol{W}_c{}^T \boldsymbol{h}(t-1) + \boldsymbol{b}_c)$$

$$\boldsymbol{c}(t) = \tilde{\boldsymbol{c}}(t) \odot \boldsymbol{i}(t) + \boldsymbol{c}(t-1) \odot \boldsymbol{f}(t)$$
$$\boldsymbol{h}(t) = \phi(\boldsymbol{c}(t)) \odot \boldsymbol{o}(t)$$

where σ is the sigmoid activation function, ϕ is the tanh activation function and $\odot$ is the element-wise product. $\tilde{\boldsymbol{c}}(t)$ is the candidate state value and $\boldsymbol{c}(t)$ is the cell's new state at time t.

11.3 LSTM Recurrent Network

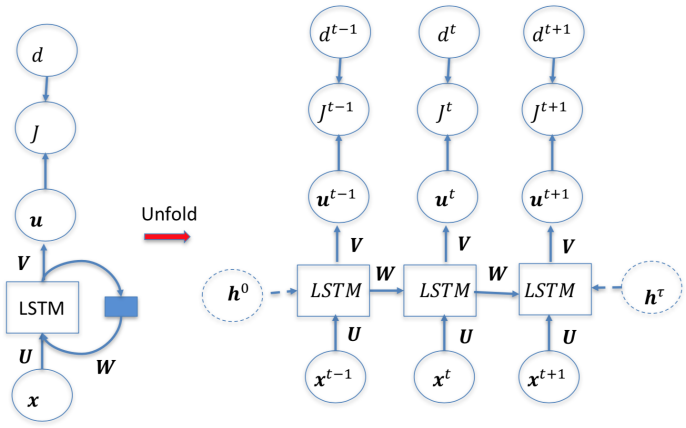

Figure 11.3: LSTM Recurrent Network

$$\boldsymbol{W} = \{\boldsymbol{W}_i, \boldsymbol{W}_f, \boldsymbol{W}_o, \boldsymbol{W}_c\}$$
$$\boldsymbol{U} = \{\boldsymbol{U}_i, \boldsymbol{U}_f, \boldsymbol{U}_o, \boldsymbol{U}_c\}$$
$$\boldsymbol{b} = \{\boldsymbol{b}_i, \boldsymbol{b}_f, \boldsymbol{b}_o, \boldsymbol{b}_c\}$$

The most important component is the state unit $s_i(t)$ which has an inner self-loop. The self-loop weight is controlled by a forget gate unit $f_i(t)$ (for time step t and cell i), which sets this weight to a value between 0 and 1 via a sigmoid unit.

11.4 Gated Recurrent Unit (GRU)

Like LSTM units, gated recurrent units (GRU) attempt to create paths through time that have gradients which either vanish or explode. The main difference with the LSTM is that a single gating unit simultaneously controls the forgetting factor and the decision to update the hidden unit.

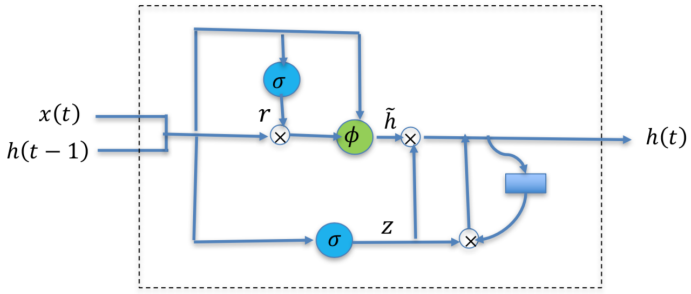

Figure 11.4: Gated Recurrent Unit

11.4.1 Equations

$$\boldsymbol{r}(t) = \sigma(\boldsymbol{U}_r{}^T \boldsymbol{x}(t) + \boldsymbol{W}_r{}^T \boldsymbol{h}(t-1) + \boldsymbol{b}_r)$$
$$\boldsymbol{z}(t) = \sigma(\boldsymbol{U}_z{}^T \boldsymbol{x}(t) + \boldsymbol{W}_z{}^T \boldsymbol{h}(t-1) + \boldsymbol{b}_z)$$
$$\tilde{\boldsymbol{h}}(t) = \phi(\boldsymbol{U}_h{}^T \boldsymbol{x}(t) + \boldsymbol{W}_h{}^T (\boldsymbol{r}(t) \odot \boldsymbol{h}(t-1)) + \boldsymbol{b}_h)$$

$$\boldsymbol{h}(t) = (1 - \boldsymbol{z}(t)) \odot \boldsymbol{h}(t-1) + \boldsymbol{z}(t) \odot \tilde{\boldsymbol{h}}(t)$$

where $\boldsymbol{r}$ is the reset gate and $\boldsymbol{z}$ is the update gate inputs. $\tilde{\boldsymbol{h}}(t)$ is the candidate cell state.

11.5 Gated Recurrent Neural Network (GRNN)

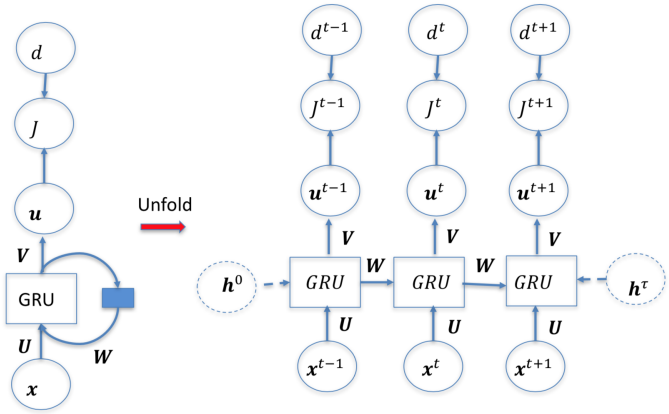

Figure 11.5: Gated Recurrent Neural Network

$$\boldsymbol{W} = \{\boldsymbol{W}_r, \boldsymbol{W}_z, \boldsymbol{W}_h\}$$
$$\boldsymbol{U} = \{\boldsymbol{U}_r, \boldsymbol{U}_z, \boldsymbol{U}_h\}$$
$$\boldsymbol{b} = \{\boldsymbol{b}_r, \boldsymbol{b}_z, \boldsymbol{b}_h\}$$

11.6 Deep LSTM & Deep GRUs

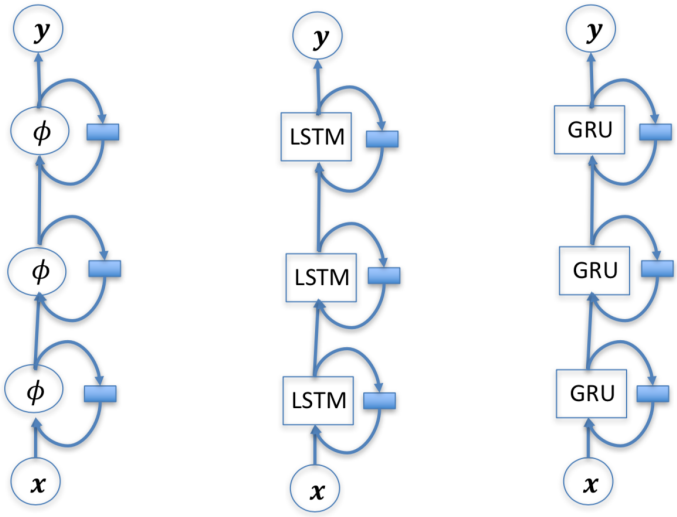

Figure 11.6: Deep Vanilla RNN, LSTM Network & GRNN