Deep Learning

Author: Wenxiao Yang

Institute: Department of Mathematics, University of Illinois at Urbana-Champaign

Date: 2022

All models are wrong, but some are useful.

Contents

Chapter 1 Learning Basics

1.1 Parameters and Hyperparameters

Definition 1.1

Parameters *are values learned by the model given the data.* ♣

e.g. β, W, b, θ.

Definition 1.2

Hyperparameters *are values supplied to tune the model and cannot be learned from data.* ♣

e.g. Number of Hidden Layers, Neurons, and Epochs to Train. Learning Rate, and Batch Size.

1.2 Neural Network: Back Propagation Algorithm

Neural Network

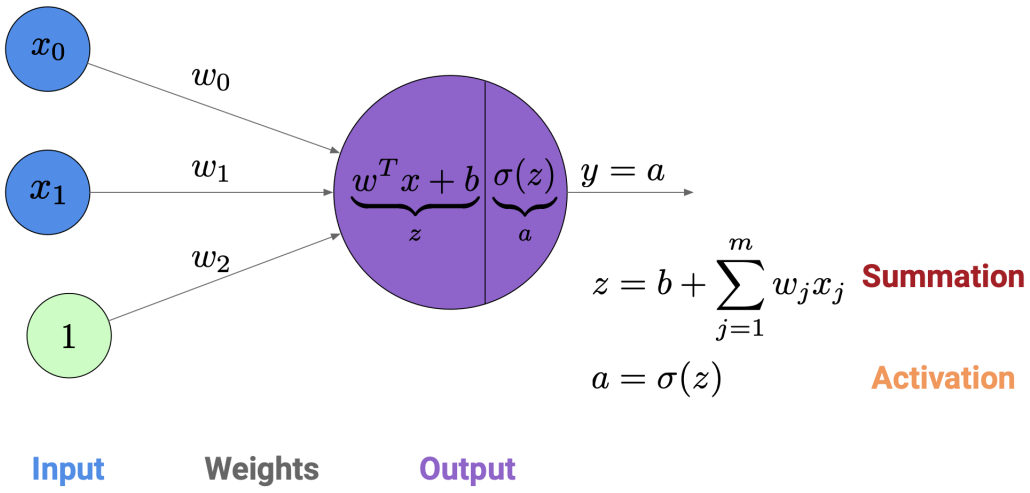


Figure 1.1: Simple Neural Network

Given a vector input x, we need to find the best estimator $\hat{y}$ which minimizes lost function. In the figure that has only one layer and one pathway, we find the parameter (ω, b) to form an input $\omega^T x + b$ to neuron σ (activation function). Then, the final output (estimator) of the network is $\hat{y} = \sigma(\omega^T x + b)$.

1.2.1 Activations

Definition 1.3

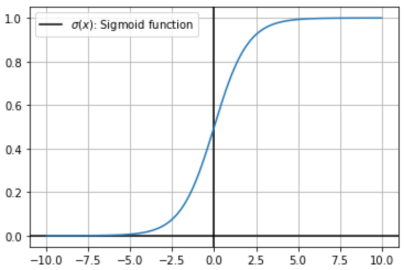
Activation functions are element-wise gates for letting information propagate to future layers either transformed with non-linearities or left as-is.

♣

Example of activation function:

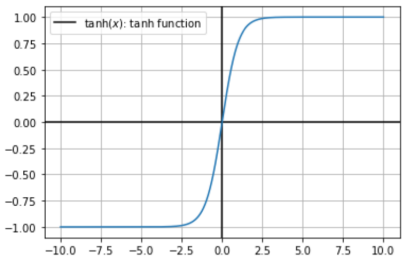
Sigmoid

$$\frac{1}{1+\exp(-x)}$$



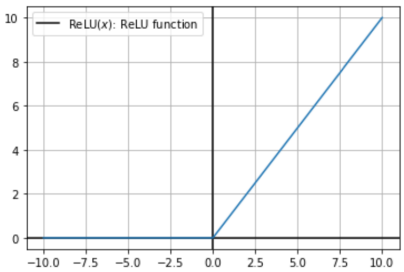
TanH

$$\frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$$



ReLU

$$\max(0,x)$$



Leaky ReLU

$$\max(0.1x,x)$$

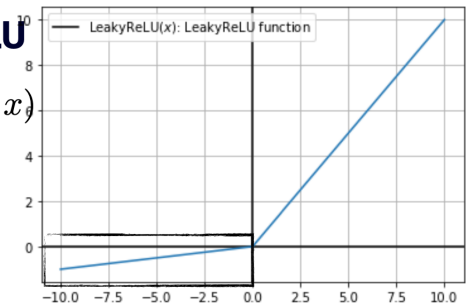


Figure 1.2: Activations

(1) **Identity:**

$$\text{identity}(x) = I(x) = x$$

(2) **Binary:**

$$\text{binary}(x) = \text{step}(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

(3) **Sigmoid:**

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\sigma(-x) = 1-\sigma(x);\ \frac{d\sigma(x)}{dx} = \sigma(x)\cdot(1-\sigma(x))$$

$$\frac{\partial \sigma(\vec{x})}{\partial \vec{x}} = \begin{bmatrix} \sigma(x_1)(1-\sigma(x_1)) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \sigma(x_n)(1-\sigma(x_n)) \end{bmatrix} = \text{diag}(\sigma(\vec{x}) \cdot (1-\sigma(\vec{x})))$$

(4) **TanH:**

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

(TanH is a rescaled sigmoid)

$$tanh(x) = \frac{e^{2x}-1}{e^{2x}+1} = 1 - 2\sigma(-2x) = 2\sigma(2x) - 1$$

(5) **ReLU:**

$$g(x) = \max(0, x)$$

(6) **Leaky ReLU:**

$$g(x) = \max(0.1x, x)$$

(7) **Softmax:** $S_j(\vec{x}) = \frac{e^{x_j}}{\sum_{i=1}^n e^{x_i}}$,

$$S(\vec{x}) = \left[\frac{e^{x_1}}{\sum_{i=1}^n e^{x_i}}, \frac{e^{x_2}}{\sum_{i=1}^n e^{x_i}}, \cdots, \frac{e^{x_n}}{\sum_{i=1}^n e^{x_i}}\right]^T$$

$$\frac{\partial S_j(\vec{x})}{\partial x_j} = S_j(\vec{x})[1 - S_j(\vec{x})]$$

1.2.2 Multilayer Neural Network

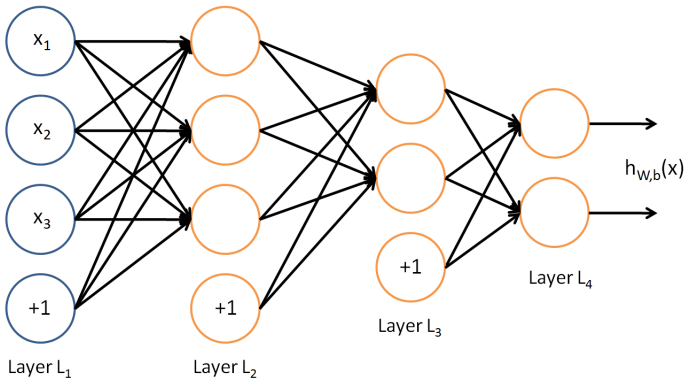


Figure 1.3: Multilayer Neural Network

- Number of neurons in each layer can be different.
- All weights on edge connecting layers $m-1$ and m is matrix $W^{(m)}$, with $w_{ij}^{(m)}$ being the weight connecting output j of layer $m-1$ with neuron i of layer m.

- Input to network is vector x; output of layer m is vector $y^{(m)}$

$$y_i^{(1)} = \sigma(x_i^{(1)}), \text{ with } x_i^{(1)} = \sum_j w_{ij}^{(1)} x_j + b_i^{(1)}$$

$$y^{(1)} = \sigma(x^{(1)}), \text{ with } x^{(1)} = W^{(1)}x + b^{(1)}$$

$$y^{(2)} = \sigma(x^{(2)}), \text{ with } x^{(2)} = W^{(2)}y^{(1)} + b^{(2)}$$

$$\vdots$$

$$y^{(M)} = \sigma(x^{(M)}), \text{ with } x^{(M)} = W^{(M)}y^{(M-1)} + b^{(M)}$$

We want to find the weights $W^{(1)}, \cdots, W^{(M)}, b^{(1)}, \cdots, b^{(M)}$ so that the output of last layer

$$\hat{y} = y^{(M)} \approx f^*(x) = y$$

$f^*(x)$ is the unknown thing we need to predict.

We use labelled training data, i.e.

$$(x[1], y[1]), (x[2], y[2]), \cdots (x[N], y[N])$$

Minimize the "empirical" loss on training data.

$$J = \sum_{i=1}^{N} L(y[i], \hat{y}[i])$$

where $\bar{y}[i]$ is the output of NN whose input is $x[i]$.

- L is the function of $W^{(1)}, \cdots, W^{(M)}, b^{(1)}, \cdots, b^{(M)}$ to measure the loss. e.g. the square loss

$$L(y, \hat{y}) = (y - \hat{y})^2$$

- We wish to minimize J using a gradient descent procedure.
- To compute gradient we need:

$$\frac{\partial L}{\partial w_{ij}^{(l)}} \text{ for each } l, i, j; \quad \frac{\partial L}{\partial b_i^{(l)}} \text{ for each } l, i.$$

1.2.3 A Simple Example of Back Propagation Algorithm

We can consider a simple example (one layer, two pathways)

$W^{(1)} = [w_{1,1}^{[1]}, w_{2,1}^{[1]}]^T$, $b^{(1)} = [0, 0]^T$, $\sigma_1(x) = x$.
$W^{(2)} = [w_{1,1}^{[2]}, w_{1,2}^{[2]}]$, $b^{(2)} = 0$, $\sigma_2(x) = x$.

$$[a_1^{[1]}, a_2^{[1]}]^T = \sigma_1(W^{(1)}x_1 + b^{(1)}) = [w_{1,1}^{[1]}x_1, w_{2,1}^{[1]}x_1]^T$$

$$\hat{y} = \sigma_2(W^{(2)}[a_1^{[1]}, a_2^{[1]}]^T + b^{(2)}) = (w_{1,1}^{[1]}w_{1,1}^{[2]} + w_{2,1}^{[1]}w_{1,2}^{[2]})x_1$$

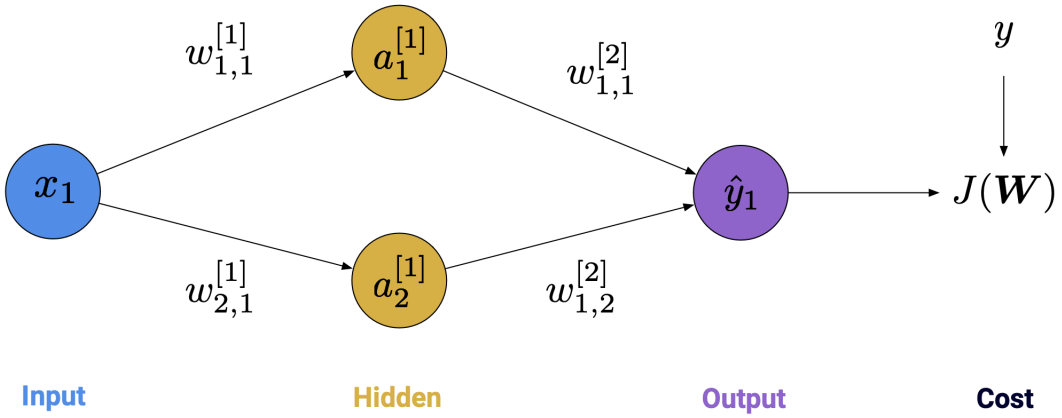


Figure 1.4: Two Independent Pathways

$$\frac{\partial J(\hat{y})}{\partial w_{2,1}^{[1]}} = \frac{\partial J(\hat{y})}{\partial \hat{y}} \cdot \frac{\hat{y}}{\partial a_2^{[1]}} \cdot \frac{a_2^{[1]}}{\partial w_{2,1}^{[1]}}$$

1.2.4 Back Propagation Algorithm

Recall $y_i^{(m)} = \sigma(x_i^{(m)})$, $x_i^{(m)} = \sum_j w_{ij}^{(m)} y_j^{(m-1)} + b_i^{(m)}$

$$\frac{\partial L}{\partial w_{ij}^{(m)}} = \frac{\partial L}{\partial y_i^{(m)}} \cdot \frac{\partial y_i^{(m)}}{\partial w_{ij}^{(m)}} = \frac{\partial L}{\partial y_i^{(m)}} \cdot \frac{\partial y_i^{(m)}}{\partial x_i^{(m)}} \cdot \frac{\partial x_i^{(m)}}{\partial w_{ij}^{(m)}}$$

$$\frac{\partial L}{\partial b_i^{(m)}} = \frac{\partial L}{\partial y_i^{(m)}} \cdot \frac{\partial y_i^{(m)}}{\partial x_i^{(m)}} \cdot \frac{\partial x_i^{(m)}}{\partial b_i^{(m)}}$$

For large M,

- $\frac{\partial L}{\partial y_i^{(M)}}$ is easy to compute.
- $\frac{\partial y_i^{(M)}}{\partial x_i^{(M)}} = \frac{\partial \sigma(x_i^{(M)})}{\partial x_i^{(M)}} = \sigma'(x_i^{(M)})$, (assuming σ differentiable).
- $\frac{\partial x_i^{(M)}}{\partial w_{ij}^{(M)}} = y_j^{(M-1)}$

Thus,

$$\frac{\partial L}{\partial w_{ij}^{(M)}} = \frac{\partial L}{\partial y_i^{(M)}} \cdot \sigma'(x_i^{(M)}) \cdot y_j^{(M-1)}$$

Similarly,

$$\frac{\partial L}{\partial b_i^{(M)}} = \frac{\partial L}{\partial y_i^{(M)}} \cdot \frac{\partial y_i^{(M)}}{\partial x_i^{(M)}} \cdot \frac{\partial x_i^{(M)}}{\partial b_i^{(M)}} = \frac{\partial L}{\partial y_i^{(M)}} \cdot \sigma'(x_i^{(M)})$$

For $1 \leq m < M$, in this situation $\frac{\partial L}{\partial y_i^{(m)}}$ is not easy to compute. Note that $x^{(m+1)} = W^{(m+1)} y^{(m)} + b^{(m+1)}$.

$$\begin{aligned}
\frac{\partial L}{\partial y_i^{(m)}} &= \sum_k \frac{\partial L}{\partial x_k^{(m+1)}} \cdot \frac{\partial x_k^{(m+1)}}{\partial y_i^{(m)}} \\
&= \sum_k \frac{\partial L}{\partial y_k^{(m+1)}} \cdot \frac{\partial y_k^{(m+1)}}{\partial x_k^{(m+1)}} \cdot \frac{\partial x_k^{(m+1)}}{\partial y_i^{(m)}} \\
&= \sum_k \frac{\partial L}{\partial y_k^{(m+1)}} \cdot \sigma'(x_k^{(m+1)}) \cdot w_{ki}^{(m+1)}
\end{aligned}$$

Then use this form to compute,

(We can set $\delta^{(m)} = \frac{\partial L}{\partial y_i^{(m)}} \cdot \sigma'(x_i^{(m)})$ to avoid duplicate computation.)

$$\begin{aligned}
\frac{\partial L}{\partial w_{ij}^{(m)}} &= \frac{\partial L}{\partial y_i^{(m)}} \cdot \frac{\partial y_i^{(m)}}{\partial x_i^{(m)}} \cdot \frac{\partial x_i^{(m)}}{\partial w_{ij}^{(m)}} \\
&= \frac{\partial L}{\partial y_i^{(m)}} \cdot \sigma'(x_i^{(m)}) \cdot y_j^{(m-1)} \\
&= \delta^{(m)} \cdot y_j^{(m-1)}
\end{aligned}$$

Similarly,

$$\begin{aligned}
\frac{\partial L}{\partial b_i^{(m)}} &= \frac{\partial L}{\partial y_i^{(m)}} \cdot \frac{\partial y_i^{(m)}}{\partial x_i^{(m)}} \cdot \frac{\partial x_i^{(m)}}{\partial b_i^{(m)}} \\
&= \frac{\partial L}{\partial y_i^{(m)}} \cdot \sigma'(x_i^{(m)}) \\
&= \delta^{(m)}
\end{aligned}$$

Summary

1. Compute $\frac{\partial L}{\partial y_i^{(M)}}$.
2. Use
$$\frac{\partial L}{\partial y_i^{(m)}} = \sum_k \frac{\partial L}{\partial y_k^{(m+1)}} \cdot \sigma'(x_k^{(m+1)}) \cdot w_{ki}^{(m+1)}$$
compute $\frac{\partial L}{\partial y_i^{(m)}}$ for $m = 1, 2..., M-1$.
3. Compute
$$\frac{\partial L}{\partial b_i^{(m)}} = \frac{\partial L}{\partial y_i^{(m)}} \cdot \sigma'(x_i^{(m)}) = \delta^{(m)}$$
for $m = 1, 2..., M$.
4. Compute
$$\frac{\partial L}{\partial w_{ij}^{(m)}} = \frac{\partial L}{\partial y_i^{(m)}} \cdot \sigma'(x_i^{(m)}) \cdot y_j^{(m-1)} = \delta^{(m)} \cdot y_j^{(m-1)}$$
for $m = 1, 2..., M$.

1.2.5 Other Methods

Stochastic Gradient Descent (SGD)

Subgradient Method

1.3 Perceptron Algorithm

Definition 1.4

Binary linear classifiers distinguish between two categories through a linear function of the inputs.

Definition 1.5

Linearly separable refers to a line that can be drawn to perfectly split the two classes.

The Perceptron algorithm is an efficient algorithm for learning a **linear separator** in $d-$dimensional space, with a mistake bound that depends on the margin of separation of the data.

1.3.1 General Idea

Given the training data

$$D = \left\{ \left\langle x^{(i)}, y^{(i)} \right\rangle, i = 1, ..., n \right\} \in (\mathbb{R}^m \times \{0, 1\})^n$$

we want to know the exact value of $y \in \{0, 1\}$.

$$\hat{y} = g(b + \boldsymbol{w}^T \boldsymbol{X})$$

Output Activation Bias Weights Input

Summation

Figure 1.5: Perceptron Output

General idea:

- If the perceptron correctly predicts ($\hat{y} = y$):
  - Do nothing
- If the perceptron yields an incorrect prediction ($\hat{y} \neq y$):
  - If the prediction is 0 and truth is 1 ($\hat{y} = 0 | y = 1 \Rightarrow e = y - \hat{y} = 1$), add feature vector to weight vector.
  - If the prediction is 1 and truth is 0 ($\hat{y} = 1 | y = 0 \Rightarrow e = y - \hat{y} = -1$), subtract feature vector from the weight vector.

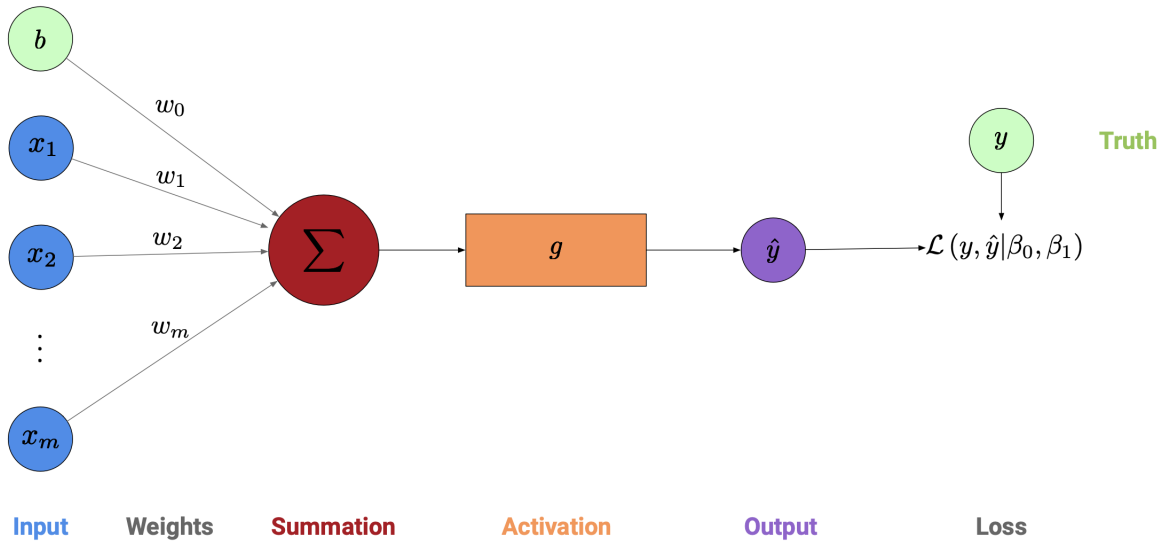


Figure 1.6: Perceptron

Since we want the prediction to be either 0 or 1, we usually use binary function as the activation function in perceptron.

1.3.2 Algorithm

Perceptron Algorithm:

- Initialize weights (including a bias term) to zero, e.g. $W = [w, b] = 0^{m+1}$.
- Under each training epoch: Compute for each sample $\langle x^{(i)}, y^{(i)} \rangle \in D$
  - A prediction $\hat{y}^{(i)} = g({x^{(i)}}^T W)$
  - Prediction error $e^{(i)} = y^{(i)} - \hat{y}^{(i)}$
  - Weighted update $W = W + \eta e^{(i)} x^{(i)}$

1.3.3 Limitations

(1) Only provides a linear classifier boundary.

(2) Only allows for **binary classifier** between two classes.

(3) **No convergence possible** if classes are not linearly separable.

(4) Perceptron will yield **multiple boundary/"optimal" solutions**.

(5) Boundaries found may **not** perform **equally well**.

1.4 ADAptive LInear NEuron (ADALINE)

1.4.1 General Idea

Except the activation function in perceptron, we can add a threshold function.

In perceptron, we generate the estimation $\hat{y}$ (after binary function) to help update weight $\{w_i\}_{i=0}^m$. However, in ADALINE, we minimize MSE $z = x^T W$ to update weight $\{w_i\}_{i=0}^m$ before output estimation $\hat{y}$ (before binary function).

Before entering threshold (binary function), we want to minimize a mean- squared error (MSE) loss function to estimate weights.

e.g. suppose $g(x) = x$, let $z = x^T W$ be the input of threshold, for each y,

$$W^* = \underset{W}{\operatorname{argmin}}\, L(z, y) = (y - z)^2$$

$$\frac{\partial L(z, y)}{\partial w_i} = -2(y - z)\frac{\partial z}{\partial w_i} = -2(y - z)x_i$$

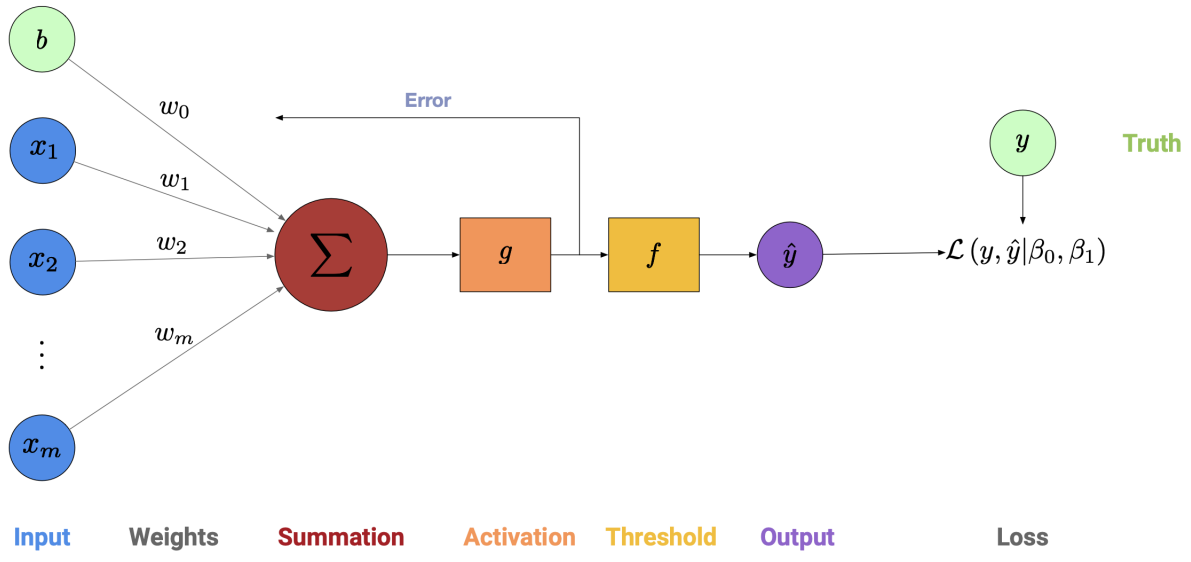


Figure 1.7: ADALINE

1.4.2 Widrow-Hoff Delta Rule

(Gradient Descent Rule for ADALINE)

- **Original:**

$$W = W + \eta(y^{(j)} - z)x^{(j)}$$

- **Unit-norm:**

$$W = W + \eta(y^{(j)} - z)\frac{x^{(j)}}{\|x^{(j)}\|}$$

where $\|x\| = \sqrt{x_1^2 + x_2^2 + \cdots + x_m^2}$

The Perceptron and ADALINE use variants of the delta rule!

(1) **Perceptron**: Output used in delta rule is $\hat{y} = g(x^T W)$; $W = W + \eta(y^{(j)} - \hat{y}^{(j)})x^{(j)}$

(2) **ADALINE**: Output used to estimate weights is $z = x^T W$. $W = W + \eta(y^{(j)} - z)x^{(j)}$

1.5 Logistic Regression (Binary-class Output)

1.5.1 Generative and Discriminative Classifiers

The most important difference between naive Bayes and logistic regression is that logistic regression is a **discriminative classifier** while naive Bayes is a **generative classifier**.

Suppose we want to classify class A (dogs) and class B (cats) (More genearl form: assign a class $c \in C$ to a document $d \in D$):

(1) **Generative model:** A generative model would have the goal of understanding what dogs look like and what cats look like. You might literally ask such a model to 'generate', i.e., draw, a dog.

e.g. **naive Bayes:** we do not directly compute the probability that the document d belongs to each class c, $P(c|d)$. We compute likelihood $P(d|c)$ and prior probability $P(c)$ to generate best estimation $\hat{c}$. (i.e., we want to know what should the distribution of a document d in class c be like.)

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} P(d|c)P(c)$$

(2) **Discriminative model:** A discriminative model, by contrast, is only trying to learn to distinguish the classes (perhaps without learning much about them). That is we want to directly computing $P(c|d)$.

1.5.2 Sigmoid function

The goal of binary logistic regression is to train a classifier that can make a binary decision about the class of a new input observation.

The input observation is $x = [x_1, ..., x_m]^T$ and the output y is either 1 or 0. Instead of using the optimal weights of each feature x_i and binary activation function (**threshold**: $\hat{y} = 1$ if $z \geq 0$ and $\hat{y} = 0$ otherwise) to estimate in Perceptron and ADALINE, **we want to estimate the probability $P(y = 1|x)$.**

However, the weighted sum $z = x^T W = \sum_{i=1}^{m} w_i x_i + b$ ranges $-\infty$ to ∞. We want to **force the z to be a legal probability**, that is, to lie between 0 and 1.

The **sigmoid function** $\sigma(z) = \frac{1}{1+e^{-z}}$ can be used as acitivation for this purpose, $P(y = 1|x) = \sigma(x^T W)$. Since $1 - \sigma(x) = \sigma(-x)$, $P(y = 0|x) = \sigma(-x^T W)$.

1.5.3 Cross-entropy Loss Function

We choose the parameters W that maximize the log probability of the true y labels in the training data given the observations x. The conditional probability

$$p(y|x) = \begin{cases} \hat{y}, & y = 1 \\ 1 - \hat{y}, & y = 0 \end{cases} = \hat{y}^y (1 - \hat{y})^{1-y}$$

To maximize the probability, we log both sides:

$$\log p(y|x) = y \log \hat{y} + (1 - y) \log(1 - \hat{y})$$

Then, we want the $\hat{y}$ to maximize the probability (also the logarithm of the probability):

$$\begin{aligned} \hat{y}^* &= \underset{\hat{y} \in [0,1]}{\operatorname{argmax}} \log p(y|x) \\ &= \underset{\hat{y} \in [0,1]}{\operatorname{argmin}} - \log p(y|x) \\ &= \underset{\hat{y} \in [0,1]}{\operatorname{argmin}} -(y \log \hat{y} + (1 - y) \log(1 - \hat{y})) \end{aligned}$$

The right hand side is exactly the **cross-entropy loss function**:

$$L(y, \hat{y}) = -(y \log \hat{y} + (1 - y) \log(1 - \hat{y}))$$

where $\hat{y}^{(i)} = \sigma(w^T x^{(i)} + b)$

$$\frac{\partial L(y^{(i)}, \hat{y}^{(i)})}{\partial w_j} = \left(\sigma(w^T x^{(i)} + b) - y^{(i)}\right) x_j^{(i)} = (\hat{y}^{(i)} - y^{(i)}) x_j^{(i)}$$

The risk (Binary Cross-Entropy Cost) of a weight W is

$$J(W) = -\frac{1}{n} \sum_{i=1}^{n} (y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}))$$

$$\frac{\partial J(w,b)}{\partial w_j} = \frac{1}{n}\sum_{i=1}^{n}\left(\sigma(w^T x^{(i)} + b) - y^{(i)}\right) x_j^{(i)}$$

1.5.4 Algorithm

- Initialize weights (including a bias term) to zero, e.g. $W = [w, b] = 0^{m+1}$.
- Under each training epoch: Compute for each sample $\langle x^{(i)}, y^{(i)} \rangle \in D$
  - A prediction $\hat{y}^{(i)} = g({x^{(i)}}^T W)$
  - Prediction error $e^{(i)} = y^{(i)} - \hat{y}^{(i)}$
  - Weighted update $W = W + \eta e^{(i)} x^{(i)} = W - \eta \nabla L(W)$

1.6 Softmax Regression (Multi-class Output)

1.6.1 Multi-Class Classification and Multi-Label Classification

Definition 1.6

Multi-Class Classification is a process for assigning each sample exactly one class. In this case, classes are considered mutually exclusive (no intersection). ♣

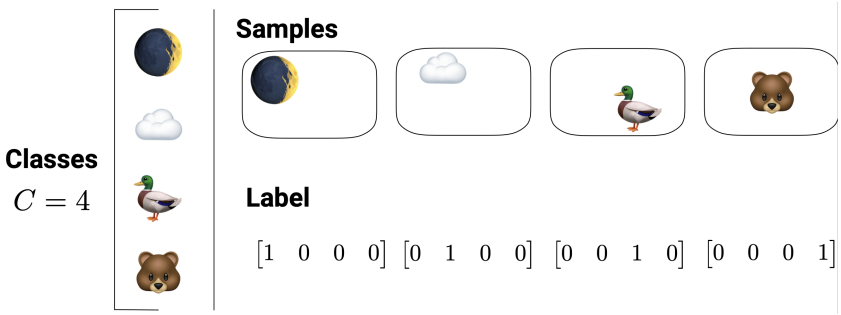


Figure 1.8: Multi-Class Classification

Definition 1.7

Multi-Label Classification or annotation allows for each sample to have 1 or more classes assigned to it. In this case, classes are mutually non-exclusive (one common element). ♣

We can show some examples of activation layer and loss choice in different probelms.

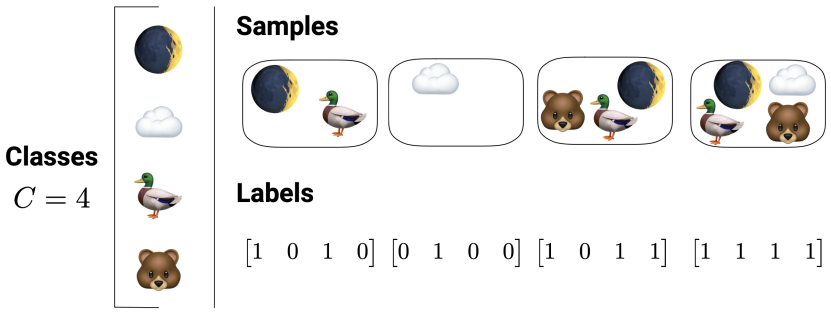


Figure 1.9: Multi-Label Classification

	Output Activation	Loss Function
Binary Classification	Sigmoid	Binary Cross-Entropy
Multi-Class Classification	Softmax	Categorical Cross-Entropy
Multi-Label Classification	Sigmoid	Binary Cross-Entropy
Linear Regression	Identity	Mean Squared Error
Regression to values in [0, 1]	Sigmoid	Mean Squared Error or Binary Cross-Entropy

Figure 1.10: Examples of Activation Layer and Loss Choice

1.6.2 One-hot Encoding

Definition 1.8

One-hot encoding is the process of assigning a single location within a vector to represent a given category.

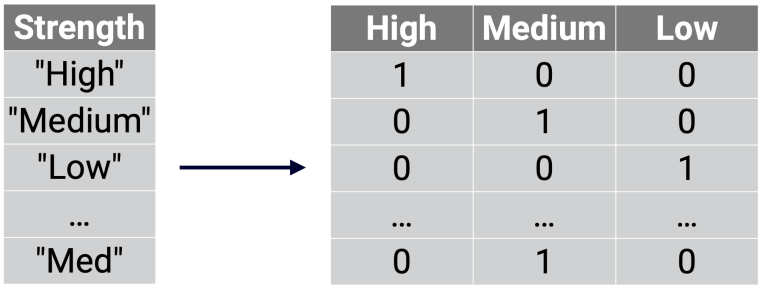


Figure 1.11: One-hot encoding

Examples:

1. $\mathbf{z} = [4]_{1\times1} \rightarrow \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \end{bmatrix}_{1\times5}$

2. $\mathbf{y} = \begin{bmatrix} 3 & 0 & 2 \end{bmatrix}_{1\times 3} \rightarrow \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}_{3\times 4}$

3. $\mathbf{v} = \begin{bmatrix} 5 & 0 & 4 & 4 & 3 \end{bmatrix}_{1\times 5} \rightarrow \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}_{5\times 6}$

Usefulness of Encodings

1. Close to a traditional design matrix for linear regression that uses a codified dummy variable structure. e.g. FALSE (0) or TRUE (1)
2. Reduce the size of data stored by using numbers instead of strings.
3. Poor if there are too many unique values (e.g. text messages on a phone.)

1.6.3 Softmax function

Definition 1.9

Softmax function or normalized exponential function converts between real valued numbers to values between 0 and 1

♣

Softmax: $S_j(\vec{x}) = \frac{e^{x_j}}{\sum_{i=1}^{n} e^{x_i}}$,

$$S(\vec{x}) = \left[\frac{e^{x_1}}{\sum_{i=1}^{n} e^{x_i}}, \frac{e^{x_2}}{\sum_{i=1}^{n} e^{x_i}}, \cdots, \frac{e^{x_n}}{\sum_{i=1}^{n} e^{x_i}} \right]^T$$

We can show the softmax function has the following properties:

(1) First-derivative:

$$\frac{\partial S_i(\vec{x})}{\partial x_i} = S_i(\vec{x})[1 - S_i(\vec{x})]; \ \frac{\partial S_i(\vec{x})}{\partial x_j} = -S_i(\vec{x})S_j(\vec{x})$$

(2) Stabilizing softmax:

$$S_j(\vec{x} + c) = \frac{e^{x_j + c}}{\sum_{i=1}^{n} e^{x_i + c}} = \frac{e^{x_j}}{\sum_{i=1}^{n} e^{x_i}} = S_j(\vec{x})$$

We can minus $\max_i x_i$ to avoid overflow in softmax function. (numerical issue) i.e., $S_j(\vec{x} - (\max_i x_i)) = S_j(\vec{x})$

1.6.4 Categorical Cross-entropy Loss Function

Definition 1.10

Categorical Cross-entropy Loss is a way to quantify the difference between a "true" values $\{y_c\}_{c\in C}$ *and an estimated* $\{\hat{y}_c\}_{c\in C}$ *across* C *categories.*

♣

Note: y needs one-hot encoding firstly, $\hat{y}$ are estimated probability.

$$L(y, \hat{y}) = -\sum_{c\in C} (y_c \cdot \log(\hat{y}_c))$$

Definition 1.11

Categorical Cross-entropy Cost is a way of quantifying the cost over multiple points from different categories.

♣

$$J(W) = \frac{1}{n}\sum_{i=1}^{n} L(y_i, \hat{y}_i) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{c\in C} (y_{i,c} \cdot \log(\hat{y}_{i,c}))$$

1.7 Deep Feedforward Networks

1.7.1 Definition

In any neural network, a dense layer is a layer that is deeply connected with its preceding layer which means the neurons of the layer are connected to *every neuron* of its preceding layer.

Definition 1.12

Deep feedforward networks, feedforward neural networks, multilayer perceptrons (MLPs), or dense neural networks form the foundations of deep learning models. Learning occurs in only one direction: ***forward****. There are* ***no feedback*** *connections in whichoutputs of the model are fed back into itself. There are no cycles or loops present. Information must flow from the input layer, through one or more hidden layers, before reaching the output.*

♣

A deep neural network contains *Input Layer*, *Hidden Layer*, and *Output Layer*.

1.7.2 Universal Approximation Theorem

Universal Approximation Theorem, in its lose form, states that a **feed-forward network** with a **single hidden layer** containing a finite number of neurons **can approximate any continuous function**. (Which is also

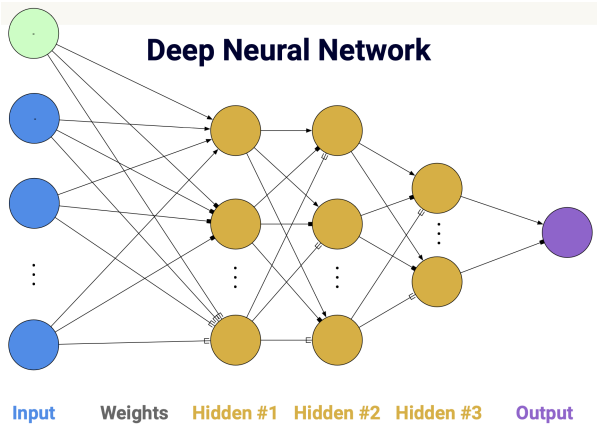


Figure 1.12: Deep Neural Network

equivalent to having a nonpolynomial activation function)

Theorem 1.1 (Universal approximation theorem)

Fix a continuous function $\sigma : \mathbb{R} \to \mathbb{R}$ *(activation function) and positive integers* $d, D \in \mathbb{Z}^+$. *The function* σ *is not a polynomial* $\Leftrightarrow$ *for every continuous function* $f : \mathbb{R}^d \to \mathbb{R}^D$ *(target function), every compact subset* K *of* $\mathbb{R}^d$, *and every* $\epsilon > 0$ *there exists a continuous function* $f_\epsilon : \mathbb{R}^d \to \mathbb{R}^D$ *(the layer output) with representation*

$$f_\epsilon = W_2 \circ \sigma \circ W_1$$

where W_2, W_1 *are composable affine maps and* o *denotes component-wise composition, such that the approximation bound*

$$\sup_{x \in K} \|f(x) - f_\epsilon(x)\| < \varepsilon$$

holds for any ϵ *arbitrarily small (distance from* f *to* f_ϵ *can be infinitely small).* ♡

1.8 Mini-batch Optimization

1.8.1 Stochastic Gradient Descent (SGD) and Batch Gradient Descent (BGD)

Stochastic Gradient Descent (SGD)

1. Start with a random guess.
2. For n epochs:
   1) Reorder data
   2) Retrieve an observation $i = 1, 2, ...$ one by one in reordered data:
      (1) Compute gradient on single data point i: $\frac{\partial J_i(W)}{\partial W}$

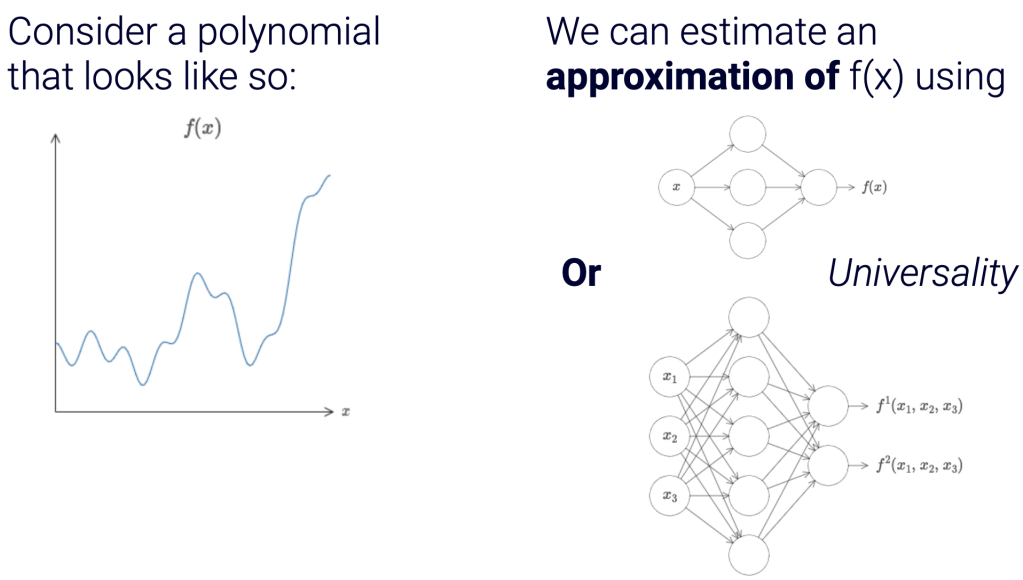


Figure 1.13: Universal Approximation Theorem

(2) Update parameters: $W = W - \alpha \frac{\partial J_i(W)}{\partial W}$

3. Output parameters

Note: ”On-line”/”Stochastic” **Single** Observation Updates

Batch Gradient Descent (BGD)

1. Start with a random guess.
2. For n epochs:
   1) Compute gradients on **all the data**: $\frac{\partial J(W)}{\partial W}$
   2) Update parameters: $W = W - \alpha \frac{\partial J(W)}{\partial W}$
3. Output parameters

Note: **All** data used in update

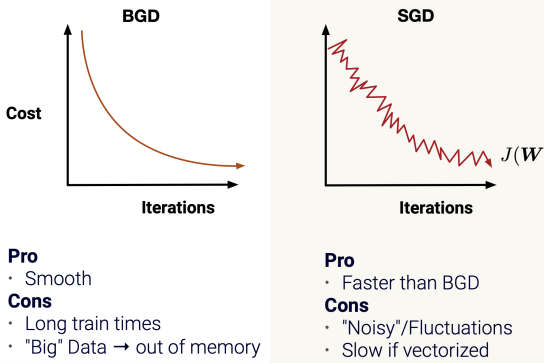


Figure 1.14: BGD and SGD

1.8.2 Mini-Batch Gradient Descent (MBGD)

We want a middle ground between SGD and BGD.

Mini-Batch Gradient Descent (MBGD)

1. Start with a random guess.
2. For n epochs:
    1) Reorder data and retrieve a subet of reordered data with size b (batch size)
    2) Compute gradient on subset: $\frac{\partial J(W)}{\partial W}$
    3) Update parameters: $W = W - \alpha \frac{\partial J(W)}{\partial W}$
3. Output parameters

If $b = n$, the algorithm is exactly BGD; If $b = 1$, the algorithm is exactly SGD.

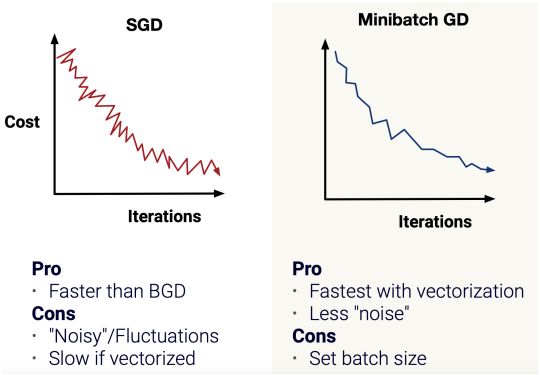


Figure 1.15: SGD and MBGD

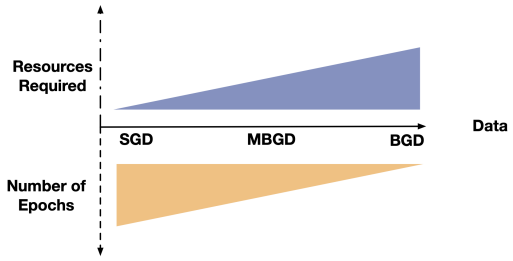


Figure 1.16: Comparison of Approaches

1.9 Weight Initialization

1.9.1 Xavier Initialization

Normal distribution with a scale variance by weights. Used on layers where either $TanH$ or $Sigmoid$ is present.

$$\text{Initialize weights for layer } l \text{ with: } W^{[l]} = W^{[l]} \sqrt{\frac{1}{n^{[l-1]}}}$$

where $n^{[l-1]}$ is the number of weights in the last layer. Each weight is sampled by $W_{j,i}^{[l]} \sim \mathcal{N}(0, 1)$

1.9.2 He Activation

Weight initialization for $ReLU$-powered network.

$$\text{Initialize weights for layer } l \text{ with: } W^{[l]} = W^{[l]} \sqrt{\frac{2}{n^{[l-1]}}}$$

where $n^{[l-1]}$ is the number of weights in the last layer. Each weight is sampled by $W_{j,i}^{[l]} \sim \mathcal{N}(0, 1)$

Chapter 2 Adaptive Optimization

2.1 Exponentially Weighted Moving Averages

How can we get an average across time?

1. (Simple) (weighted) Moving Average ((S)MA):

$$\bar{x}_{MA} = \frac{x_m + x_{m-1} + \cdots + x_{m-(n-1)}}{n} = \frac{1}{n}\sum_{i=0}^{n-1} x_{m-i}$$

2. Exponentially (weighted) Moving Averages (EMA):

$$EMA_t = \begin{cases} Y_1, & t = 1 \\ \beta \cdot Y_t + (1-\beta) \cdot EMA_{t-1}, & t > 1 \end{cases}$$

EMA is quicker to react: focuses more on recent events; SMA is slower to react: focuses on long series events.

2.2 Adaptive Learning Rates

Adaptive Learning Rate is a change to the learning rate while training a model to reduce the training time and improve output

2.2.1 Momentum

For a gradient descent with form:

$$\theta_{t+1} := \theta_t - v_t$$

where v_t is the **velocity** which is amplified gradient speed.

1. **SGD:**

$$v_t = \alpha \nabla_\theta J(\theta_t)$$

2. **SGD + Momentum:**

$$v_t = \rho v_{t-1} + \alpha \nabla_\theta J(\theta_t)$$

   where ρ is the **friction or momentum** which dampens the amount of the previous gradient included. Default: 0.9 for ~ 10 gradients.

With momentum, the learning rate is **decreased** if gradient direction changes and **increased** if gradient direction stays on same path.

2.2.2 Root Mean Square Propagation (RMSProp)

Decrease learning rate by EMA using squared gradient.

$$g_0 = 0 \quad \text{(Initial "gain")}$$

$$g_t = \alpha g_{t-1} + (1-\alpha)\nabla_\theta J(\theta_t)^2 \quad \text{(MA over gradient squared)}$$

$$\theta_t := \theta_{t-1} - \frac{\varepsilon}{\sqrt{g_t} + 1 \times 10^{-5}} v_t \quad (1 \times 10^{-5} \text{ is used to avoid division by } 0)$$

2.2.3 Adaptive Moment Estimation (ADAM)

Merges the momentum and RMSProp paradigms.

Novelty is a bias correction of 1st/2nd moments.

Focus is on learning rate annealing (start fast, decrease).

(1) **Initial:** $v_0 = 0, g_t = 0$.

(2) **Momentum-variant:** $v_t = \beta_1 v_{t-1} + (1-\beta_1)\nabla_\theta J(\theta_t)$

(3) **RMSProp:** $g_t = \beta_2 g_{t-1} + (1-\beta_2)\nabla_\theta J(\theta_t)^2$

(4) **Bias Correction:** $v'_t = \frac{v_t}{1-\beta_1^t}$, $g'_t = \frac{g_t}{1-\beta_2^t}$ (where β_i^t is the t^{th} power of β_i)

(5) **RMSProp + Momentum:** $\theta_t := \theta_{t-1} - \frac{\varepsilon}{\sqrt{g'_t}+1\times10^{-5}} v'_t$

Pytorch Code

```
torch.optim.Adam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08,
                 weight_decay=0, amsgrad=False, *, foreach=None,
                 maximize=False, capturable=False,
                 differentiable=False, fused=False)
```

Pseudocode

---

input : γ (lr), β_1, β_2 (betas), θ_0 (params), $f(\theta)$ (objective)
λ (weight decay), *amsgrad*, *maximize*
initialize : $m_0 \leftarrow 0$ (first moment), $v_0 \leftarrow 0$ (second moment), $\widehat{v_0}^{max} \leftarrow 0$

---

for $t = 1$ **to** $\ldots$ **do**
  if *maximize* :
    $g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1})$
  else
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  if $\lambda \neq 0$
    $g_t \leftarrow g_t + \lambda \theta_{t-1}$
  $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
  $\widehat{m_t} \leftarrow m_t / \big(1 - \beta_1^t\big)$
  $\widehat{v_t} \leftarrow v_t / \big(1 - \beta_2^t\big)$
  if *amsgrad*
    $\widehat{v_t}^{max} \leftarrow \max(\widehat{v_t}^{max}, \widehat{v_t})$
    $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \big(\sqrt{\widehat{v_t}^{max}} + \epsilon\big)$
  else
    $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \big(\sqrt{\widehat{v_t}} + \epsilon\big)$

---

return θ_t

Figure 2.1: Pseudocode of ADAM

Chapter 3 Convolutional Neural Network (CNN)

3.1 Convolution and Cross-correlation

Cross-correlation and convolution are both operations applied to images. Cross-correlation means sliding a kernel (filter) across an image. Convolution means sliding a flipped kernel across an image. **Most convolutional neural networks in machine learning libraries are actually implemented using cross-correlation**, but it doesn't change the results in practice because if convolution were used instead, the same weight values would be learned in a flipped orientation.

We have some *source pixels*, we use a *convolution kernel* (filter) to process the *source pixels*. The output is *destination pixels*.

Given the *source pixels* $\{\textbf{Image}(x, y) : x \in [-d_X, d_X], y \in [-d_Y, d_Y]\}$ and *convolution kernel* (filter) $\{K(i, j) : i, j \in [-d, d]\}$. $n_X = 2d_X + 1, n_Y = 2d_Y + 1$ are image dimension, $f = 2d + 1$ is the filter dimension. We can generate destination pixels in $(n_X - f + 1) \times (n_Y - f + 1) = (n_X - 2d) \times (n_Y - 2d)$

For $x \in [d + 1, n_X - d], y \in [d + 1, n_Y - d]$

1. **Convolution:**

$$\textbf{CONV}(x, y) = \sum_{i=-d}^{d} \sum_{j=-d}^{d} \textbf{Image}(x - i, y - j)K(i, j)$$

2. **Cross-Correlation:**

$$\textbf{CrossCorrelation}(x, y) = \sum_{i=-d}^{d} \sum_{j=-d}^{d} \textbf{Image}(x + i, y + j)K(i, j)$$

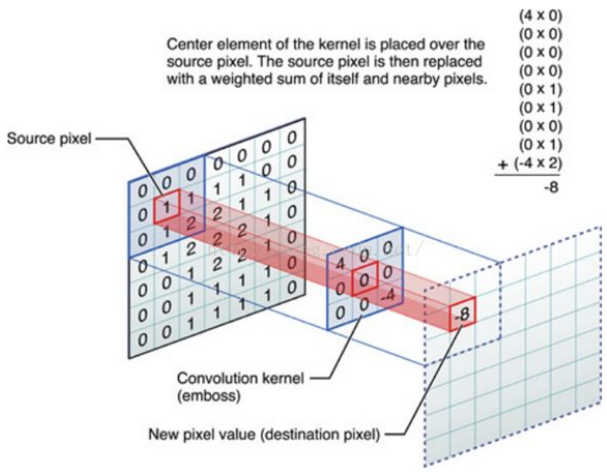


Figure 3.1: Convolution Used in CNN (actually cross-correlation)

Kernel

1. **Edge Detection:** vertical $\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$; horizontal $\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$
2. Blur Pixel: $\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$; Sharpen: $\begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix}$; Identity: $\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$; Shift Pixel: $\begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$

3.2 Padding (cover the border)

Definition 3.1

Padding refers to the extension of the input image by adding a border of pixels the image.

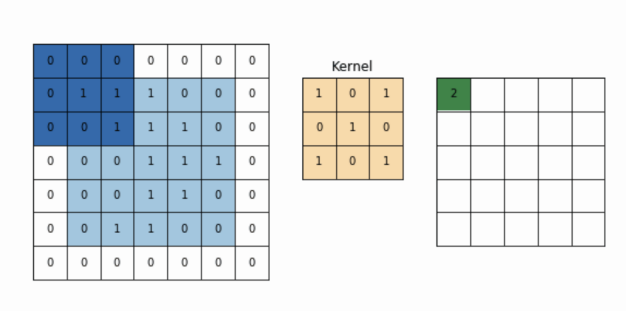


Figure 3.2: Padding: $p = 1$

For image $n \times n$ with filter $f \times f$ and padding p, the output has dimension

$$(n + 2p - f + 1) \times (n + 2p - f + 1)$$

In order for the output dimensions to be equivalent to the image dimension the padding value must be $p = \frac{f-1}{2}$

3.3 Stride

Definition 3.2

Stride refers to the sliding distance of the filter/kernel over spatial locations.

The default stride or strides in two dimensions is (1,1) for the height and the width movement.

For example, the stride can be changed to (2,2). This has the effect of moving the filter two pixels right for each horizontal movement of the filter and two pixels down for each vertical movement of the filter when creating the feature map.

The new dimension of the output would be

$$\left\lfloor \frac{n_X + 2p - f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n_Y + 2p - f}{s} + 1 \right\rfloor$$

3.4 Other Layer Types

3.4.1 Pooling

Definition 3.3

Pooling refers to the process of downsampling features by aggregating values at places in the feature map.

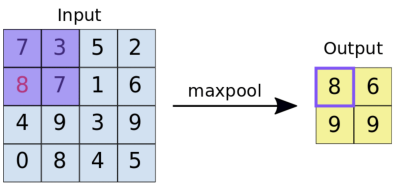


Figure 3.3: Example: maxpool

3.4.2 Unpooling

Definition 3.4

Unpooling refers to the process of upsampling features by recreating the dimensions of feature map pooled and placing the pooled values into their original location.

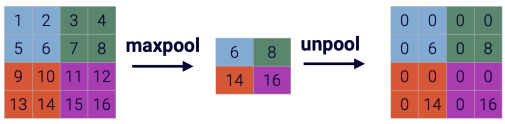


Figure 3.4: Example: maxpool+unpool

3.5 3D Convolution

Note the default filter has dimension $f \times f \times n'$ (n' is the number of filters in the previous layer)

SL.No		Activation Shape	Activation Size	# Parameters
1.	Input Layer:	(32, 32, 3)	3072	0
2.	CONV1 (f=5, s=1)	(28, 28, 8)	6272	608
3.	POOL1	(14, 14, 8)	1568	0
4.	CONV2 (f=5, s=1)	(10, 10, 16)	1600	3216
5.	POOL2	(5, 5, 16)	400	0
6.	FC3	(120, 1)	120	48120
7.	FC4	(84, 1)	84	10164
8.	Softmax	(10, 1)	10	850

Figure 3.5: 3D Convolution

Parameters Computing:

1. **CONV layer:** (shape of width of the filter * shape of height of the filter * number of filters in the previous layer+1)*number of filters

   (added 1 because of the bias term for each filter.)

$$(5 \times 5 \times 3 + 1) \times 8 = 608$$

2. **Fully Connected Layer (FC):** (current layer neurons number * previous layer neurons number)+1* current layer neurons number

$$120 \times 400 + 1 \times 120 = 48120$$

Chapter 4 Generative model

4.1 Autoencoders

Definition: *Self-supervised learning (SSL)* is a machine learning process where the model trains itself to learn one part of the input from another part of the input. It is a technique similar in scope to how humans learn to classify objects. SSL relies on unlabeled data to solve a task by splitting the task into at least two halves:

1. A decomposition into pseudo-labels by withholding some training data (self-supervised task/pretext task); and,
2. Reconstruction using either supervised or unsupervised learning.

For example, in natural language processing, if we have a few words, using self-supervised learning we can complete the rest of the sentence. Similarly, in a video, we can predict past or future frames based on available video data.

4.1.1 Basics

Definition 4.1

Autoencoders *are designed to take the input data, say x, and, then, predict the very same input x! In other words, the network trains itself to imitate its input so that its output is the same.* ♣

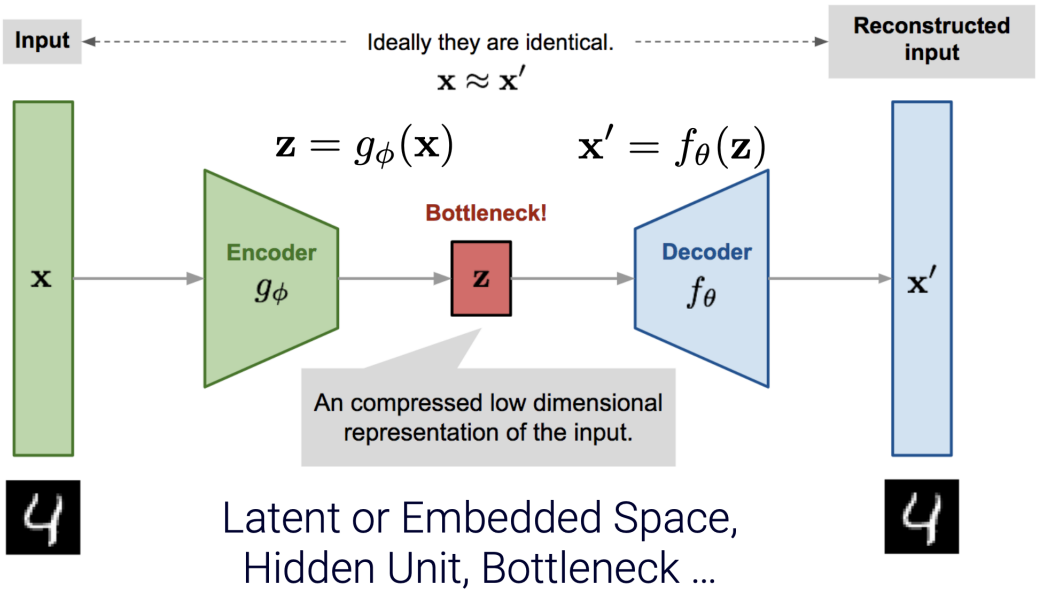


Figure 4.1: Autoencoders

Undercomplete autoencoders are defined as having a bottleneck layer dimension that is less than that of the input. e.g.

$$\dim(z) < \dim(x)$$

The lower dimention of bottleneck (hidden unit) avoids overfitting.

If the dimension is equal, the x will be completely transformed into x' which is often the same as the model learns nothing and may also be overfitted. Therefore, some other conditions are usually added to make the model only approximate its input, so that the model can really learn the hidden vector expression of the input samples and prevent overfitting.

4.1.2 PCA and Autoencoders

Principal Component Analysis (PCA) is using orthogonal basis to reduce dimensionality.

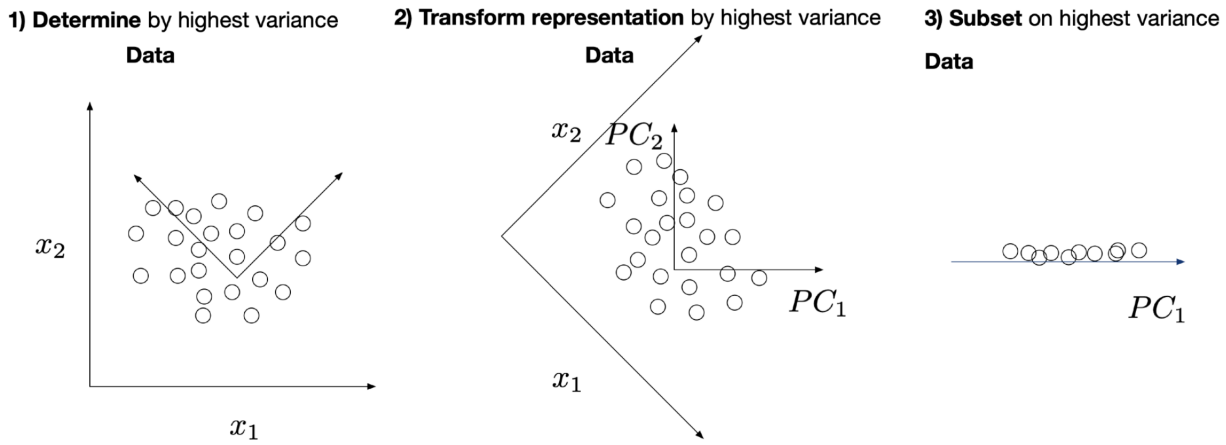


Figure 4.2: Principal Component Analysis (PCA)

Consider a single hidden layer linear autoencoder network with linear activations and MSE loss:

$$\vec{z} = W^{[1]}\vec{x} + \vec{b}^{[1]}$$

$$\vec{x}' = W^{[2]}\vec{z} + \vec{b}^{[2]}$$

$$L(\vec{x}, \vec{x}') = \|\vec{x} - \vec{x}'\|_2^2$$

If we have the $dim(m) < dim(n)$, then the problem will be a PCA without an orthogonality restriction on the weights.

Why bother with autoencoders if PCA exists? Dimensional Reductions (Addressing curse of dimensionality)

- Rarely are we seeking to use an auto encoder for solely a dimension reduction.

- In such cases, we probably would be better off with:

- Principal Component Analysis (PCA): If linear and desire global structure under a deterministic algorithm.
- T-distributed stochastic neighbour embedding (t-SNE): If non-linear and desire local structure under a randomized algorithm with dense structures.
- Uniform Manifold Approximation and Projection (UMAP): If non-linear and desire local structure under a randomized algorithm with sparse structures.

4.1.3 Transposed Convolutions (upscale method)

Definition: *Transposed Convolutions*, *uncov*, or *fractionally striped convolution* is a technique to upscale the feature map so that it matches in dimension with the input feature map.
(Sometimes erroneously called ”deconvolution”.)

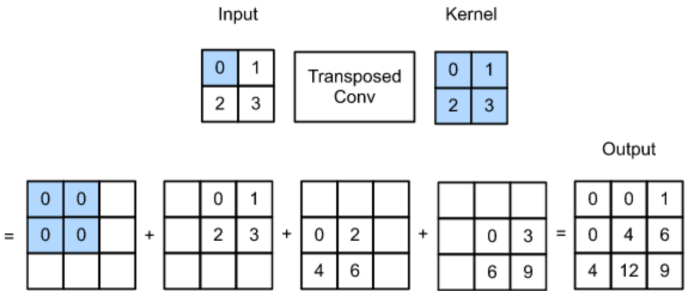


Figure 4.3: Transposed Convolutions

Comparison between Convolution and Transposed Convolution

Transposed Convolution with Stride

For a image $n \times n$ with filter $f \times f$, padding p and stride s, the dimension of the output is

$$(s(n-1)+f-2p) \times (s(n-1)+f-2p)$$

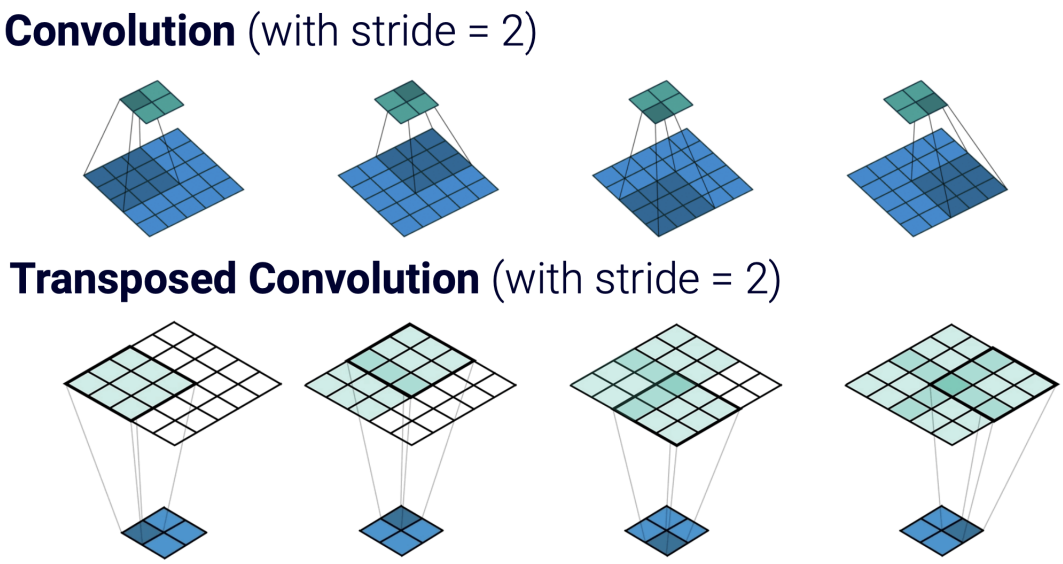


Figure 4.4: Comparison

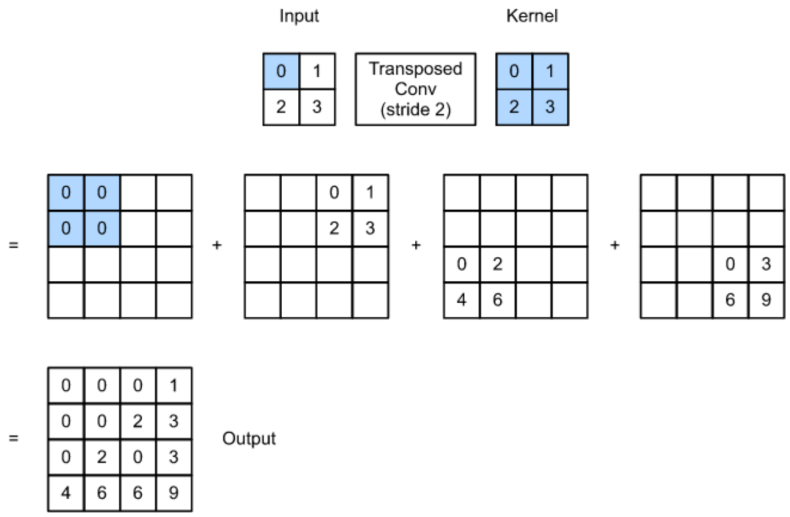


Figure 4.5: Transposed Convolution with Stride