# COMP4211

Machine Learning

Larry128

A summary notes for revision

Fall 2024-2025

# Contents

**1 Linear Regression** **2**
1.1 Basic Ideas of Regression . . . 2
1.2 Linear Regression Function . . . 2
1.3 Loss Function . . . 2
1.4 Non-linear Extensions . . . 4
1.5 Model Over-fitting . . . 5
1.6 Regularization . . . 5

**2 Feedforward Neural Network** **7**
2.1 Artificial Neural Network . . . 7
2.2 Layered Extension of Linear or Logistic Regression . . . 7
2.3 Universal Approximation . . . 7
2.4 An illustrative example: a 3-layer network . . . 8
2.5 Activation Functions . . . 8
2.6 Loss Functions . . . 9
2.7 Back-propagation Learning Algorithm . . . 10

**3 Deep Neural networks** **15**
3.1 Challenges of trainign deep neural networks . . . 15
3.2 Non-saturating activation functions . . . 15
3.3 Choosing activation functions . . . 17
3.4 Weight initialization . . . 17
3.5 Batch Normalisation . . . 18
3.6 Dropout . . . 19
3.7 Data Augmentation . . . 19
3.8 Optimizers . . . 20

# 1 Linear Regression

## 1.1 Basic Ideas of Regression

1. Given a training set $S = \{(x^{(l)}, y^{(l)})\}_{l=1}^{N}$ of $N$ labelled examples of input-output pairs.
2. A **Regression Function** $f(\mathbf{x}; \mathbf{w})$ uses $S$ such that the predicted output $f(\mathbf{x^{(l)}}; \mathbf{w})$ for each input $\mathbf{x}^l$ such that $f(\mathbf{x}^{(l)}; \mathbf{w}) \approx \mathbf{y}^l$.
3. (multi-output regression) When the output $\mathbf{y}$ is a vector, it's a multi-output regression.
4. We denote the output by $y$ if the output is univariate.
5. The input $\mathbf{x} = (x_1, \cdots, x_d)^T$ is $d$-dimensional.

## 1.2 Linear Regression Function

1. If the regression function is linear, then

$$\begin{aligned} f(\mathbf{x}; \mathbf{w}) &= w_0 + w_1 x_1 + \cdots + w_d x_d \\ &= \begin{bmatrix} w_0 & w_1 & \cdots & w_d \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_d \end{bmatrix} \qquad = \begin{bmatrix} 1 & x_1 & \cdots & x_d \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_d \end{bmatrix} \\ &= \mathbf{w}^T \tilde{\mathbf{x}} \qquad\qquad\qquad\qquad\qquad = \tilde{\mathbf{x}}^T \mathbf{w} \end{aligned}$$

2. $w_0$ is the *bias* term which serves as an offset.
3. The learning problem is to find the best $\mathbf{w}$ according to performance measure on $S$.

## 1.3 Loss Function

1. A common way to learn the parameter $\mathbf{w}$ of $f(\mathbf{x}; \mathbf{w})$ is to define a loss function $L(\mathbf{w}; S)$
2. The most common loss function is the **squared loss**

$$\begin{aligned} L(\mathbf{w}; S) &= \sum_{l=1}^{N} (f(\mathbf{x}^{(l)}; \mathbf{w}) - \mathbf{y}^{(l)})^2 \\ &= \sum_{l=1}^{N} (w_0 + w_1 x_1^{(l)} + \cdots + w_d x_d^{(l)} - y^{(l)})^2 \end{aligned}$$

3. We may also define the loss function by **mean** rather than the sum

$$L(\mathbf{w}; S) = \frac{1}{N} \sum_{l=1}^{N} (f(\mathbf{x}^{(l)}; \mathbf{w}) - \mathbf{y}^{(l)})^2$$

4. A special case ($d = 1$)
   Squared loss:

$$L(\mathbf{w}; S) = \sum_{l=1}^{N} (w_0 + w_1 x_1^{(l)} - y^{(l)})^2$$

   We can find the unique optimal solution $\tilde{\mathbf{w}} = \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}$ that minimizes $L(\mathbf{w}; S)$ using the method of least squares.

First, we take the derivatives of $L(\mathbf{w}; S)$ with respect to $w_0$ and $w_1$ and set them to 0.

$$\frac{\partial L}{\partial w_0} = 2\sum_{l=1}^{N}(w_0 + w_1 x_1^{(l)} - y^{(l)}) = 0 \iff \sum_{l=1}^{N}(w_0 + w_1 x_1^{(l)}) = \sum_{l=1}^{N} y^{(l)} \iff Nw_0 + \sum_{l=1}^{N} w_1 x_1^{(l)} = \sum_{l=1}^{N} y^{(l)}$$

$$\frac{\partial L}{\partial w_1} = 2\sum_{l=1}^{N}(w_0 + w_1 x_1^{(l)} - y^{(l)})x_1^{(l)} = 0 \iff w_0\sum_{l=1}^{N} x_1^{(l)} + w_1\sum_{l=1}^{N}(x_1^{(l)})^2 = \sum_{l=1}^{N} x_1^{(l)} y^{(l)}$$

Then, we have a system of linear equations of two unknown $w_0$, $w_1$. We can write it in matrix form.

$$\mathbf{Aw} = \begin{bmatrix} N & \sum_l x_1^l \\ \sum_l x_1^l & \sum_l (x_1^l)^2 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \end{bmatrix} = \begin{bmatrix} \sum_l y^{(l)} \\ \sum_l x_1^{(l)} y^{(l)} \end{bmatrix} = \mathbf{b}$$

Assuming $\mathbf{A}$ is invertible, the least squares estimate is

$$\tilde{\mathbf{w}} = \mathbf{A}^{-1}\mathbf{b}$$

5. General case ($d \geq 1$)

   (a) (First approach) We express the input and output of $N$ examples as follows

$$\mathbf{X} = \begin{bmatrix} 1 & x_1^l & \cdots & x_d^1 \\ 1 & x_1^2 & \cdots & x_d^2 \\ \vdots & & & \\ 1 & x_1^N & \cdots & x_d^N \end{bmatrix}, \qquad \mathbf{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(N)} \end{bmatrix}$$

   Then we can express the matrix form as follows (proof skipped)

$$\mathbf{Aw} = \mathbf{X^T X w} = \mathbf{X^T y} = \mathbf{b}$$

   Therefore, the least squares estimate is

$$\tilde{\mathbf{w}} = (\mathbf{X^T X})^{-1}\mathbf{X^T y}$$

   , assuming $\mathbf{X^T X}$ is invertible

   (b) (Second approach) First write $\mathbf{Xw} - \mathbf{y}$ as

$$\begin{aligned} \mathbf{Xw} - \mathbf{y} &= \begin{bmatrix} 1 & x_1^l & \cdots & x_d^1 \\ 1 & x_1^2 & \cdots & x_d^2 \\ \vdots & & & \\ 1 & x_1^N & \cdots & x_d^N \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ y_d \end{bmatrix} - \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(N)} \end{bmatrix} \\ &= \begin{bmatrix} w_0 + w_1 x_1^{(1)} + \cdots + w_d x_d^1 - y^{(1)} \\ w_0 + w_1 x_1^{(2)} + \cdots + w_d x_d^2 - y^{(2)} \\ \vdots \\ w_0 + w_1 x_1^{(N)} + \cdots + w_d x_d^N - y^{(N)} \end{bmatrix} \end{aligned}$$

   Then the squared loss is just the square of **L-2 norm** of $\mathbf{Xw} - \mathbf{y}$

$$L(\mathbf{w}; S) = ||\mathbf{Xw} - \mathbf{y}||^2$$

   We can further write the squared loss as

$$\begin{aligned} L(\mathbf{w}; S) &= ||\mathbf{Xw} - \mathbf{y}||^2 \\ &= (\mathbf{Xw} - \mathbf{y})^T(\mathbf{Xw} - \mathbf{y}) \\ &= (\mathbf{w^T X^T} - \mathbf{y^T})(\mathbf{Xw} - \mathbf{y}) \\ &= \mathbf{w^T X^T X w} - \mathbf{2 y^T X w} + \mathbf{y^T y} \end{aligned}$$

After that, we can take the derivative with respect to $\mathbf{w}$

$$\begin{aligned} & \frac{\partial L}{\partial \mathbf{w}} = 2\mathbf{X^T X w} - 2\mathbf{X^T y} = 0 \\ \iff & \mathbf{X^T X w} = \mathbf{X^T y} \\ \iff & \tilde{\mathbf{w}} = (\mathbf{X^T X})^{-1}\mathbf{X^T y} \end{aligned}$$

(c) Complexity considerations
To compute $\tilde{\mathbf{w}}$, we need to invert $\mathbf{X^T X} \in \mathbb{R}^{(d+1)\times(d+1)}$. *LeGall* is the fastest algorithm to compute that with $O(n^{2.3728639})$, instead of $O(n^3)$ for Cholesky, LU, Gaussian elimination.

### 1.4 Non-linear Extensions

(a) For solving more complicated problems, non-linear regression function are needed.

(b) Different approaches for non-linear extension are:

i. Explicitly adding more input dimensions (which depend non-linearly on the original input dimensions) and applying linear regress to the expanded input. Here is an example

$$f(\mathbf{x}; \mathbf{w}) = w_0 \cdot 1 + w_1 x_1 + w_2 x_2 + \cdots + w_d x_d + w_{d+1} x_1^2 x_8^3 + w_{d+2} x_{10} x_{19}$$

In this case,

$$\tilde{\mathbf{x}} = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_d \\ x_1^2 x_8^3 \\ x_{10} x_{19} \end{bmatrix}, \mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_d \\ w_{d+1} \\ w_{d+2} \end{bmatrix}$$

ii. Applying an explicit defined non-linear regression function to the original input. For example

$$f(\mathbf{x}; \mathbf{w}) = w_0 + w_1 x_1 x_2^2 + w_2 x_3 x_5^8 + \cdots$$

iii. Applying an implicitly defined non-linear transformation to the original input and then a linear model to the transformed input

$$U \mapsto V, \mathbf{x} \in U, \mathbf{z} \in V, f(\mathbf{z}; \mathbf{w})$$

(c) We will consider the first approach here and leave the other two for some later topics

(d) Advantage of the first approach is that linear regression can still be used and the weights in a linear regression model have **text interpretability** (the larger the magnitude of a weight, the more significant is the corresponding input feature).

(e) (Polynomial Regression). One common approach is to introduce *higher-order terms* as additional input dimensions, e.g., $x_i^2, x_i x_j, x_i x_j^2 x_k$

$$\begin{aligned} f(\mathbf{x}; \mathbf{w}) &= w_0 + w_1 x + \cdots + w_m x^m \\ &= \begin{bmatrix} w_0 & w_1 & \cdots & w_m \end{bmatrix} \begin{bmatrix} 1 \\ x \\ \vdots \\ x^m \end{bmatrix} \\ &= \mathbf{w}^T \tilde{\mathbf{x}} \end{aligned}$$

Remark: Although $f$ is non-linear in $\tilde{\mathbf{x}}$, it is linear in the optimization variable $\mathbf{w}$.
Very often, **feature engineering** that uses domain knowledge to define application-specific features is applied, two of the original features are body weight and body height, we may define the body mass index (BMI)

## 1.5 Model Over-fitting

1. If the training set $S = \{(\mathbf{x}^l, \mathbf{y}^l)\}_{l=1}^N$ is small compared to the number of parameters in the linear regression function $f(\mathbf{x}; \mathbf{w})$, overfitting may occur.

2. When overfitting occurs, it is common to find large magnitudes in at least some of the parameters. This is because a large search space is needed for the (overly complex) model to fit the data exactly.

3. One common solution to the overfitting problem is to prevent the parameters from growing excessively large in magnitude.

$$\begin{aligned} \text{Overfitting} &\implies \text{Large magnitudes in some weights} \\ \equiv \text{Not large magnitudes in some weights} &\implies \text{Not overfitting} \end{aligned}$$

   To attain so, we will do **Regularization**.

## 1.6 Regularization

1. Regularization is an approach which modifies the original loss function by adding one or more penalty terms, called regularizers, that penalize large parameter magnitudes.

2. For example, Regularized loss function based on $L_2$ regularization (a.k.a. Tikhonov regularization)

$$\begin{aligned} L_\lambda(\mathbf{w}; S) &= L(\mathbf{w}; S) + \lambda||\mathbf{w}||^2 \\ &= ||\mathbf{X}\mathbf{w} - \mathbf{y}||^2 + \lambda||\mathbf{w}||^2 \\ &= (\mathbf{X}\mathbf{w} - \mathbf{y})^T(\mathbf{X}\mathbf{w} - \mathbf{y}) + \lambda\mathbf{w}^T\mathbf{w} \\ &= \mathbf{w}^T\mathbf{X}^T\mathbf{X}\mathbf{w} - 2\mathbf{y}^T\mathbf{X}\mathbf{w} + \mathbf{y}^T\mathbf{y} + \lambda\mathbf{w}^T\mathbf{w} \end{aligned}$$

   where $\lambda > 0$ is called the regularization parameter which controls how strong the regularization is. Its value can be determined as part of the validation process.

3. In practice, not regularizing the biase term $w_0$ usually gives better result since overfitting is caused by the data.

4. To compute the closed-form solution with $L_2$ Regularization, we will follow a few steps.

   (a) Differentiate $L_\lambda(\mathbf{w}; S)$ with respect to $\mathbf{w}$ and set the derivative to $\vec{0}$

$$\begin{aligned} 2\mathbf{X^T}\mathbf{X}\mathbf{w} - 2\mathbf{X^T}\mathbf{y} + 2\lambda\mathbf{w} &= 0 \\ (\mathbf{X^T}\mathbf{X} + \lambda\mathbf{I})\mathbf{w} &= \mathbf{X^T}\mathbf{y} \end{aligned}$$

   (b) The least squares estimate can also be obtained in closed form:

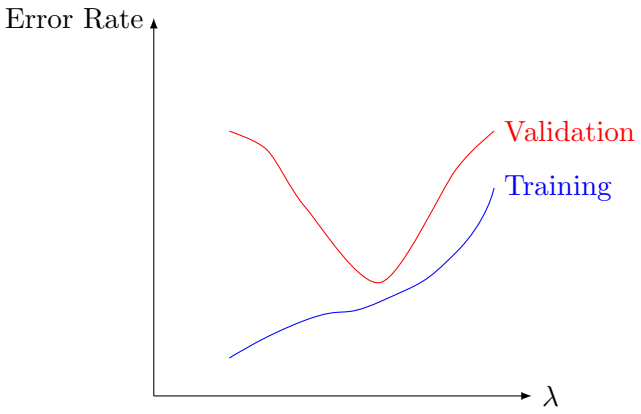
$$\tilde{\mathbf{w}} = (\mathbf{X^T}\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X^T}\mathbf{y}$$

   (c) Note that $\mathbf{X^T}\mathbf{X}$ is positive semi-definite and $\mathbf{X^T}\mathbf{X} + \lambda\mathbf{I}$ is positive definite. Therefore, $\mathbf{X^T}\mathbf{X} + \lambda\mathbf{I}$ is always invertible for any $\lambda > 0$.

   (d) Remark: linear regression with $L_2$ regularization degenerates to the ordinary linear regression (without regularization) when $\lambda = 0$.

5. (Choice of $\lambda$). Although not the only method, **cross validation** (or, more correctly, called holdout validation) is commonly used by training a model on a `training set` and validating the trained model on a separate `validation set` (which mimics the `test set`).

6. Typical Training and validation error curves




The validation process is to look for the sweet spot in the validation error curve.

7. Other Regularizers
   (a) Many other regularizers can also be defined.
   (b) For Example, instead of using $L_2$ norm, the $L_p$ norm for some other value of $p$ has also been used.
   (c) Linear regression with $L_1$ regularization, also called **LASSO** (least absolute shrinkage and selection operator), favors sparse solutions with all but a small number of dimensions equal to 0.
   (d) Although the $L_1$ norm is also convex like $L_2$ norm, there is no closed-form solution for LASSO since it's not differentiable everywhere. Iterative algorithms are needed for estimating the parameters.

8. Mean Squared Error
   (a) A common performance metric for regression problems is the mean square error (MSE)

$$MSE = \frac{1}{N}\sum_{l=1}^{N}(f(\mathbf{x^{(l)}};\mathbf{w}) - y^{(l)})^2$$

   , which is similar to the squared loss but with two differences:
      i. MSE can be used for the validation set and test set in addition to the training set.
      ii. MSE measures the mean over all the examples in the set, not the sum.
   (b) Instead of MSE, it is more common to use the root mean squared error (RMSE).

9. $R^2$ Score
   (a) Another commonly used performance metric for regression is the coefficient of determination or $R^2$ score

$$R^2 = 1 - \frac{\sum_{l=1}^{N} f(\mathbf{x^{(l)};w} - y^{(l)})^2}{\sum_{l=1}^{N}(\bar{y} - y^{(l)})^2} = 1 - \frac{\frac{1}{N}\sum_{l=1}^{N} f(\mathbf{x^{(l)};w} - y^{(l)})^2}{\frac{1}{N}\sum_{l=1}^{N}(\bar{y} - y^{(l)})^2} = 1 - \frac{\text{MSE}}{\text{variance}}$$

   , where $\bar{y} = \frac{1}{N}\sum_{l=1}^{N} y^{(l)}$
   (b) The best possible $R^2$ score is 1 when the corresponding MSE is 0.
   (c) When the model always predicts the mean value of $y$, the $R^2$ score will be equal to 0.
   (d) Negative values are also possible because the model can have arbitrarily large MSE.

# 2 Feedforward Neural Network

## 2.1 Artificial Neural Network

1. Early research in artificial neural networks was inspired by findings from **neuronscience**, but subsequent development has mostly been guided by mathematical and computational considerations.

2. Machine learning researchers and practitioners regard artificial neural networks as computational models for machine learning.

3. There are **two** types of artificial neural networks:

   (a) Feedforward neural networks: networks without loops

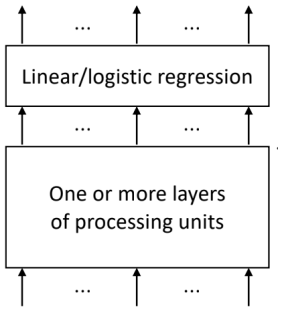
   (b) Recurrent neural networks: networks with loops

## 2.2 Layered Extension of Linear or Logistic Regression

1. A feedforward neural network (a.k.a. multi-layer perceptron MLP), may be considered as an extension of linear or logistic regression.

2. The input is transformed by one or more layers of processing units (a.k.a. neurons) before it is fed into the linear or logistic regression model which corresponds to the output layer of the feedforward neural network.

3. Consequently, feedforward neural networks can be regarded as nonlinear generalizations




## 2.3 Universal Approximation

1. Analogous to **Turing machine** as a universal mathematical model of computation for today's digital computers, one would also like to know the universal mathematical model for feedforward neural networks.

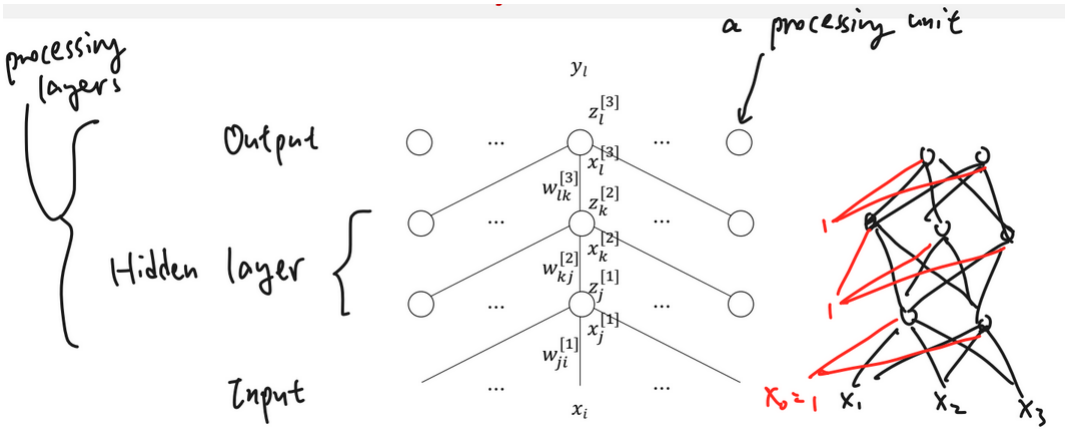
2. An informal way of stating the universal approximation theorem (proved in the late 1980s) is that, a feedforward neural network with **sufficiently many sigmoid hidden units in only one layer** can approximate any well-behaved function to arbitrary precision. However, this theorem might have misled people not to put too much effort into exploring deeper neural networks.

3. Nevertheless, using more than one hidden layer may give a network that can approximate the same function using (exponentially) **fewer parameters** due to the high non-linearly that a deeper network can induce. This higher parameter efficiency also makes deeper networks much **faster to train**.

4. Deeper networks also mimic better the **hierarchical organization** of data in many real-world applications.

## 2.4 An illustrative example: a 3-layer network

1. Each processing unit is shown as a circle. No processing is done in the input layer so no processing units are needed.




2. There are **bias terms** for all the processing layers.

3. We consider here an illustrative example with 2 hidden layers to simply the notation. However, we still call it a 3-layer network as the output layer also count as a layer with processing units.

4. The superscripts, e.g., $[1], [2]$, refer to the corresponding network layers.

5. For each processing unit,

   (a) its (summed) input is denoted by $x^{[*]}$;
   e.g., $x_j^{[1]}$ means the input to $j$-th unit in the first hidden layer.

   (b) its output is denoted by $z^{[*]}$;
   e.g., $z_j^{[1]}$ means the output from $j$-th unit in the first hidden layer.

6. The input layer may also be denoted using the superscript $[0]$.

## 2.5 Activation Functions

1. The function relating the input and output of a processing unit is called an activation function of the unit, e.g.,
$$z_j^{[1]} = g_j^{[1]}(x_j^{[1]})$$

2. All the activation functions are **non-linear**, except for those in the output layer.

3. Note that if the activation functions of the 2 hidden layers are **linear**, then these 2 layers can be merge into one.

   (a) Input layer to first hidden layer:
   Input can be computed by
   $$x_j^{[1]} = \sum_i w_{ji}^{[1]} x_i$$
   or in matrix form
   $$\mathbf{x}^{[1]} = \mathbf{W}^{[1]}\mathbf{x}^{[0]}$$
   ; Output can be computed by
   $$z_j^{[1]} = g_j^{[1]}(x_j^{[1]})$$
   or in matrix form
   $$\mathbf{z}^{[1]} = g^{[1]}(\mathbf{x}^{[1]})$$

(b) First hidden layer to second hidden layer:
Input can be computed by

$$x_j^{[2]} = \sum_j w_{kj}^{[2]} z_j$$

or in matrix form

$$\mathbf{x}^{[2]} = \mathbf{W}^{[2]}\mathbf{z}^{[1]}$$

; Output can be computed by

$$z_k^{[2]} = g_k^{[2]}(x_k^{[2]})$$

or in matrix form

$$\mathbf{z}^{[2]} = g^{[2]}(\mathbf{x}^{[2]})$$

(c) Second hidden layer to output layer:
Input can be computed by

$$x_l^{[3]} = \sum_k w_{lk}^{[3]} z_k$$

or in matrix form

$$\mathbf{x}^{[3]} = \mathbf{W}^{[3]}\mathbf{z}^{[2]}$$

; Output can be computed by

$$z_l^{[3]} = g_l^{[3]}(x_l^{[3]})$$

or in matrix form

$$\mathbf{z}^{[3]} = g^{[3]}(\mathbf{x}^{[3]})$$

Sequentially, we can write

$$\begin{aligned}
\mathbf{z}^{[3]} &= g^{[3]}(\mathbf{x}^{[3]}) \\
&= g^{[3]}(\mathbf{W}^{[3]}\mathbf{z}^{[2]}) \\
&= g^{[3]}(\mathbf{W}^{[3]}g^{[2]}(\mathbf{x}^{[2]})) \\
&= g^{[3]}(\mathbf{W}^{[3]}g^{[2]}(\mathbf{W}^{[2]}\mathbf{z}^{[1]})) \\
&= g^{[3]}(\mathbf{W}^{[3]}g^{[2]}(\mathbf{W}^{[2]}g^{[1]}(\mathbf{x}^{[1]}))) \\
&= g^{[3]}(\mathbf{W}^{[3]}g^{[2]}(\mathbf{W}^{[2]}g^{[1]}(\mathbf{W}^{[1]}\mathbf{x}^{[0]})))
\end{aligned}$$

If $g^{[1]}$, $g^{[2]}$, and $g^{[3]}$ are linear, we can write

$$\mathbf{z}^{[3]} = \mathbf{W}^{[3]}\mathbf{W}^{[2]}\mathbf{W}^{[1]}\mathbf{x}^{[0]} = \mathbf{W}'\mathbf{x}^{[0]}$$

, that is, we can merge 3 layers in to one.

4. One exception is in an autoencoder in which a linear function may be used in the hidden units.
5. The logistic function for logistic regression can also seen as an activation function.

### 2.6 Loss Functions

Recall these loss functions:

1. Squared loss function for regression problems

$$L(\mathbf{W};\mathcal{S}) = \frac{1}{2}\sum_{q=1}^{N}\sum_l (z_l^{[3](q)} - y_l^{(q)})^2 = \frac{1}{2}\sum_{q=1}^{N}\sum_l (x_l^{[3](q)} - y_l^{(q)})^2$$

The constant $\frac{1}{2}$ is introduced to simplify the subsequent derivation.

2. Cross-entropy loss function for classification problems

$$L(\mathbf{W};\mathcal{S}) = -\sum_{q=1}^{N}\sum_l y_l^{(q)} \log z_l^{[3](q)} = -\sum_{q=1}^{N}\sum_l y_l^{(q)} \log \texttt{softmax}(x_l^{[3](q)})$$

## 2.7 Back-propagation Learning Algorithm

1. **Gradient descent** based on the gradients computed recursively in the backward direction starting from the output layer:

$$\Delta w_{lk}^{[3]} \propto -\frac{\partial L}{\partial w_{lk}^{[3]}}, \Delta w_{kj}^{[2]} \propto -\frac{\partial L}{\partial w_{kj}^{[2]}}, \Delta w_{ji}^{[1]} \propto -\frac{\partial L}{\partial w_{ji}^{[1]}}$$

2. This recursive way of gradient computation for gradient descent is called the back-propagation (BP) learning algorithm.

3. Strictly speaking BP is not a recursive algorithm in the usual programming language sense. They share the same spirit though, e.g., computing $n!$ requires doing something (multiplying by $n$) to the result of $(n-1)!$.

4. More advanced gradient-based learning algorithms may also make use of the gradients computed this way.

5. Algorithm Sketch for (Batch) BP Learning

**Algorithm 1** Batch BP Learning

```
Initialize variables
repeat
    for each training example x do
        predicted-output ← neural-network-output(x)        ▷ Forward propagation
        actual-output ← label(x)
        Compute error terms at output units by a loss function

        Compute weights changes for last layers of weights (Δw[3])      ▷ Backward propagation
        Compute weights changes for second layers of weights (Δw[2])
        Compute weights changes for first layers of weights (Δw[1])
    end for
    Update network weights for all layers
until some stopping criterion is satisfied
```

6. Gradients computations
   We first let

$$L^{(q)} = \begin{cases} \frac{1}{2}\sum_l (z_l^{[3](q)} - y_l^{(q)})^2 \text{ if regression} \\ -\sum_l y_l^{(q)} \log z_l^{[3](q)} \text{ if classification} \end{cases}$$

   (a) Gradients of the last layer of weights $w_{lk}^{[3]}$

$$\begin{aligned} \frac{\partial L}{\partial w_{lk}^{[3]}} &= \sum_{q=1}^{N} \frac{\partial L^{(q)}}{\partial w_{lk}^{[3]}} \\ &= \sum_{q=1}^{N} \frac{\partial L^{(q)}}{\partial x_l^{[3](q)}} \frac{\partial x^{[3](q)}}{\partial w_{lk}^{[3]}} \\ &= -\sum_{q=1}^{N} \delta^{[3](q)} \frac{\partial x^{[3](q)}}{\partial w_{lk}^{[3]}} \end{aligned}$$

   So, to compute $\frac{\partial L}{\partial w_{lk}^{[3]}}$, we need both $\delta^{[3](q)}$ and $\frac{\partial x^{[3](q)}}{\partial w_{lk}^{[3]}}$

   i. Compute $\delta^{[3](q)}$:

(for regression)

$$\begin{aligned}\delta^{[3](q)} &= -\frac{\partial L^{(q)}}{\partial x_l^{[3](q)}} \\ &= -\sum_m \frac{\partial L^{(q)}}{\partial z_m^{[3](q)}} \frac{\partial z_m^{[3](q)}}{\partial x_l^{[3](q)}} \\ &= -\frac{\partial L^{(q)}}{\partial z_l^{[3](q)}} \frac{\partial z_l^{[3](q)}}{\partial x_l^{[3](q)}} \\ &= y_l^{(q)} - z_l^{[3](q)}\end{aligned}$$

(for classification)

$$\begin{aligned}\delta^{[3](q)} &= -\frac{\partial L^{(q)}}{\partial x_l^{[3](q)}} \\ &= -\sum_m \frac{\partial L^{(q)}}{\partial z_m^{[3](q)}} \frac{\partial z_m^{[3](q)}}{\partial x_l^{[3](q)}} \\ &= \sum_m \frac{y_m^{(q)}}{z_m^{[3](q)}} z_m^{[3](q)} (\delta_{ml} - z_l^{[3](q)}) \\ &= \sum_m y_m^{(q)} (\delta_{ml} - z_l^{[3](q)}) \\ &= y_l^{(q)} - z_l^{[3](q)}\end{aligned}$$

*Remarks: the terms $\delta_l^{[3](q)}$ can be regarded as the error terms computed at the output layer.*

ii. Compute $\frac{\partial x^{[3](q)}}{\partial w_{lk}^{[3]}}$:

$$\frac{\partial x^{[3](q)}}{\partial w_{lk}^{[3]}} = z_k^{[2](q)}$$

(b) Gradients of the second layer of weights $w_{kj}^{[2]}$

$$\begin{aligned}\frac{\partial L}{\partial w_{kj}^{[2]}} &= \sum_{q=1}^{N} \frac{\partial L^{(q)}}{\partial w_{kj}^{[2]}} \\ &= \sum_{q=1}^{N} \frac{\partial L^{(q)}}{\partial x_k^{[2](q)}} \frac{\partial x_k^{[2](q)}}{\partial w_{kj}^{[2]}} \\ &= -\sum_{q=1}^{N} \delta_k^{[2](q)} \frac{\partial x_k^{[2](q)}}{\partial w_{kj}^{[2]}}\end{aligned}$$

, where

$$\begin{aligned}\delta_k^{[2](q)} &= -\frac{\partial L^{(q)}}{\partial x_k^{[2](q)}} \\ &= -\sum_l \frac{\partial L^{(q)}}{\partial x_l^{[3](q)}} \frac{\partial x_l^{[3](q)}}{\partial z_k^{[2](q)}} \frac{\partial z_k^{[2](q)}}{\partial x_k^{[2](q)}} \\ &= \sum_l \delta_l^{[3](q)} w_{lk}^{[3]} g_k^{[2]\prime}(x_k^{[2](q)})\end{aligned}$$

*Remarks: the error terms $\delta_k^{[2](q)}$ of the second hidden layer are computed based on $\delta_l^{[3](q)}$;* and

$$\frac{\partial x^{[2](q)}}{\partial w_{kj}^{[2]}} = z_j^{[1](q)}$$

(c) Gradients of the first layer of weights $w_{ji}^{[1]}$

$$\begin{aligned}\frac{\partial L}{\partial w_{ji}^{[1]}} &= \sum_{q=1}^{N} \frac{\partial L^{(q)}}{\partial w_{ji}^{[1]}} \\ &= \sum_{q=1}^{N} \frac{\partial L^{(q)}}{\partial x_j^{[1](q)}} \frac{\partial x_j^{[1](q)}}{\partial w_{ji}^{[1]}} \\ &= -\sum_{q=1}^{N} \delta_j^{[1](q)} \frac{\partial x_j^{[1](q)}}{\partial w_{ji}^{[1]}}\end{aligned}$$

, where

$$\begin{aligned}\delta_k^{[1](q)} &= -\frac{\partial L^{(q)}}{\partial x_j^{[1](q)}} \\ &= -\sum_l \frac{\partial L^{(q)}}{\partial x_k^{[2](q)}} \frac{\partial x_k^{[2](q)}}{\partial z_j^{[1](q)}} \frac{\partial z_j^{[1](q)}}{\partial x_j^{[1](q)}} \\ &= \sum_k \delta_k^{[2](q)} w_{kj}^{[2]} g_j^{[1]'}(x_j^{[1](q)})\end{aligned}$$

*Remarks: the error terms $\delta_j^{[1](q)}$ of the first hidden layer are computed based on $\delta_k^{[2](q)}$;* and

$$\frac{\partial x^{[1](q)}}{\partial w_{ji}^{[1]}} = x_i^{[1](q)}$$

7. Weights Update Rules

(a) Last layer:

$$\begin{aligned}\delta_l^{[3](q)} &= y_l^{(q)} - z_l^{[3](q)} \\ \Delta w_{lk}^{[3]} &= -\eta \frac{\partial L}{\partial w_{lk}^{[3]}} = \eta \sum_{q=1}^{N} \delta_l^{[3](q)} z_k^{[2](q)}\end{aligned}$$

(b) Second layer:

$$\begin{aligned}\delta_k^{[2](q)} &= \sum_l \delta_l^{[3](q)} w_{lk}^{[3]} g_k^{[2]'}(x_k^{[2](q)}) \\ \Delta w_{kj}^{[2]} &= -\eta \frac{\partial L}{\partial w_{kj}^{[2]}} = \eta \sum_{q=1}^{N} \delta_k^{[2](q)} z_j^{[1](q)}\end{aligned}$$

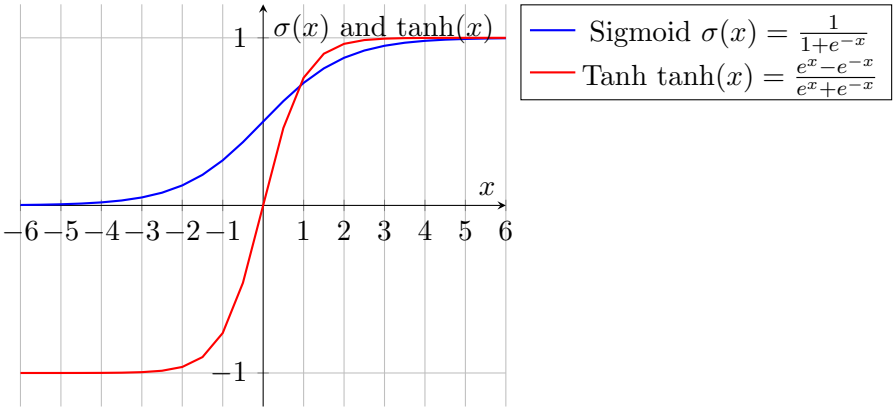
(c) First layer:

$$\begin{aligned}\delta_j^{[1](q)} &= \sum_k \delta_k^{[2](q)} w_{kj}^{[2]} g_j^{[1]'}(x_j^{[1](q)}) \\ \Delta w_{ji}^{[1]} &= -\eta \frac{\partial L}{\partial w_{ji}^{[1]}} = \eta \sum_{q=1}^{N} \delta_j^{[1](q)} x_i^{(q)}\end{aligned}$$

8. Common Activation Functions

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

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




*Remarks:*

i. Relationship between the two activation functions

$$\begin{aligned}\tanh(x) &= \frac{e^x - e^{-x}}{e^x + e^{-x}} \\ &= 2 \cdot \sigma(2x) - 1 \\ &= \frac{2}{1 + e^{-2x}} - 1\end{aligned}$$

ii. Between these two activation functions, tanh should be a better choice for the hidden layers since it can take both positive and negative values (also zero) as its output for more comprehensive representation.

9. Vanishing Gradient Problem
When we differentiate the activation functions $\sigma(x)$ and $\tanh(x)$, we get

$$\begin{aligned}\sigma'(x) &= \frac{e^{-x}}{(1+e^{-x})^2} \\ &= \frac{e^x}{(e^x+1)^2}\end{aligned} \qquad \begin{aligned}\tanh'(x) &= 2 \cdot \sigma'(2x) \\ &= \frac{2e^{2x}}{(e^{2x}+1)^2}\end{aligned}$$

We can see that when $x$ is large in magnitude, both the gradients of $\sigma(x)$ and $\tanh(x)$ are very close to 0. That is, their output does not chnage much when the input becomes very large in magnitude. We call them *Saturating Activation Functions*.

If the activation function $g$ used in the neurons is saturating, the vanishing gradient problem may arise (especially when there are many hidden layers).

$$\begin{aligned}\delta_k^{[2](q)} &= \sum_l \delta_l^{[3](q)} w_{lk}^{[3]} g_k^{[2]\prime}(x_k^{[2](q)}) \\ \delta_j^{[1](q)} &= \sum_k \delta_k^{[2](q)} w_{kj}^{[2]} g_j^{[1]\prime}(x_j^{[1](q)})\end{aligned}$$

The weight changes $\delta_k^{[2](q)}$ and $\delta_j^{[1](q)}$ will be very small, and thus the weight will not be updated probably. The techniques for overcomming the vanishing gradient problem in deep neural networks will be discussed in the next topic.

10. Stochastic Gradient Descent

(a) For feedforward neural network with one or more hidden layers, the loss function is no longer convex, i.e., local minima exist.

(b) While (batch) gradient descent computes the gradients by summing over all $N$ examples in the training set, Stochastic Gradient Descent (SGD) sums over a (usually much smaller) mini-batch of the trainign examples at a time.

(c) SGD can be regarded as a Stochastic Approximation of (batch) gradient descent with faster convergence.

(d) SGD is a more favorable alternative when the training set is large.

(e) SGD is also good at avoiding being trapped in a local minimum because SGD has more randomness than (batch) gradient descent, making SGD more likely to jump out of a local minimum.

11. Regularization

(a) Regularized loss function based on $L_2$ regularization:

$$L_\lambda(\mathbf{W};\mathcal{S}) = L(\mathbf{W};\mathcal{S}) + \frac{\lambda}{2} \sum_{w \text{ except bias terms}} w^2$$

(b) Weight update rule for $w$ (except the bias term)

$$\Delta w = -\eta \frac{\partial L_\lambda}{\partial w} = \eta \frac{\partial L}{\partial w} - \eta\lambda w$$

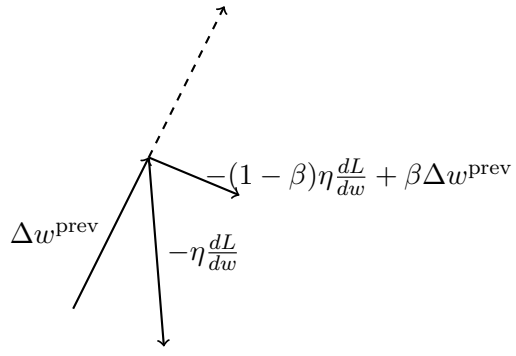
, where the second term is called the *weight decay* term because it moves the weight towards zero.

12. Momentum
We used it to stablise the SGD learning process.

(a) A *momentum* term can be added to the weight update rule for $w$ to improve the speed of convergence

$$\Delta w = -(1-\beta)\eta \frac{\partial L}{\partial w} + \eta \Delta w^{\text{prev}}$$

, where $L$ refers to the loss for a mini-batch, the momentum parameter $\beta$ is generally taken to be between 0.5 and 1 and $\Delta w^{\text{prev}}$ refers to the previous weight update.




(b) The momentum term is more crucial for SGD to make the learning process more stable by smoothing the weight changes over time.

(c) It has been shown mathematically that the momentum term plays a role similar to the *mass* in damped harmonic oscillators by bringing the system closer to critical damping.

# 3 Deep Neural networks

## 3.1 Challenges of trainign deep neural networks

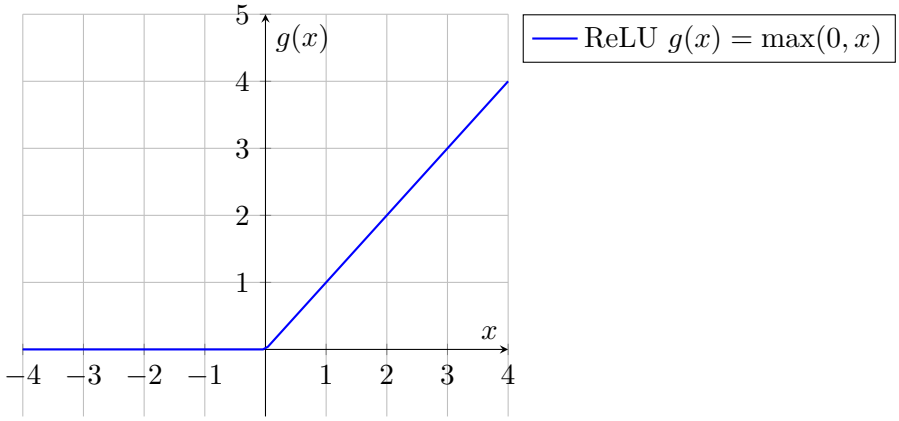
1. The vanishing gradient problem makes the lower layers (i.e., the layers closer to the input layer) of a deep neural network very difficult to train, because the gradient of the often get smaller and smaller in magnitude as the BP algorithm progresses down to the lower layer.
2. The many network weights in a deep neural network make it easy to overfit the training data.
3. It would be extremely time consuming to train a large network especially when simple optimizers are used.

## 3.2 Non-saturating activation functions

1. Rectifier Linear Unit (ReLU)
   ReLU is a processing unit that uses the rectifier

$$g(x) = \max(0, x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

   as the activation function. ReLU alleviates the vanishing gradient problem because it does not saturate for positive input values.




   (a) Another advantage of using the rectifier activation is that its derivative is easy to compute.

$$g(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases} \qquad g'(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

   Meanwhile, computation of derivative of sigmoid function requires floating point computing.

   (b) However, the vanishing gradient problem is not completely addressed because the gradient is 0 for negative inpute values. When this happens, the output is 0 and the gradient is 0, making the *dying ReLU* unlikely to come back to life again.

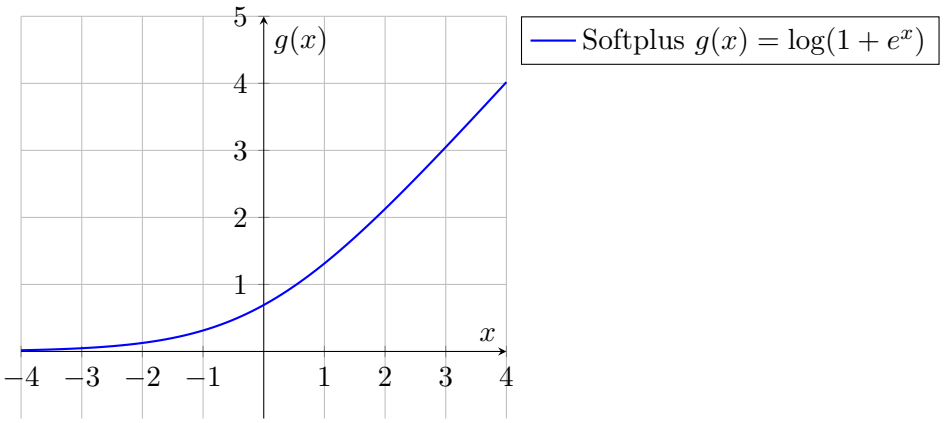
   (c) Also, although the function is continous, it is not differentiable when the input is 0. Nevertheless, this usually does not cause any problem in practice as we can use some programming tricks to solve this problem.

   (d) A smooth approximation to the rectifier is the softplus function

$$g(x) = \log(1 + e^x)$$

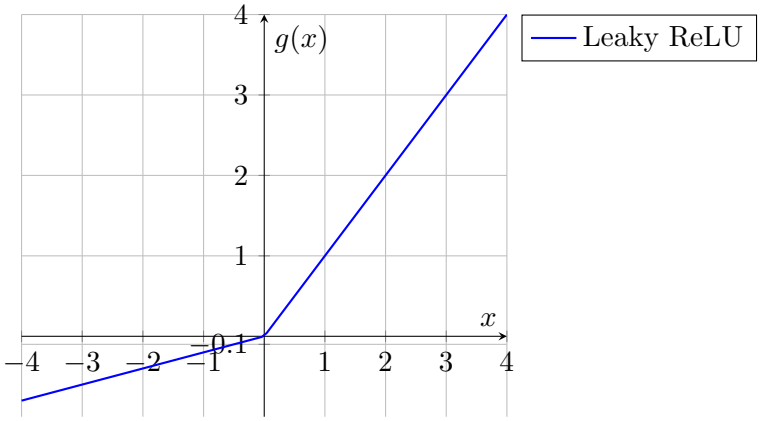
   . Note that the derivative of softplus is the logistic function

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




2. Leaky ReLU
   Leaky ReLU improves ReLU by allowing a small positive gradient for negative input values (e.g., $\alpha = 0.01$).

$$g(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{otherwise} \end{cases} \qquad g'(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ \alpha & \text{otherwise} \end{cases}$$




3. Exponential Linear Unit (ELU)
   Exponential linear unit (ELU) has the following activation function:

$$g(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha(e^x - 1) & \text{otherwise} \end{cases}$$

   , where $\alpha > 0$ is a hyperparameter (e.g., initialised to 1). Note that the derivative of ELU is

$$g'(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ \alpha e^x = g(x) + \alpha & \text{otherwise} \end{cases}$$

   .

   (a) An advantage of ELU is that its mean activation value is close to 0, which can be proved to enable faster learning. Since the softplus function always gives a positive value, it is not as good as ELU.

   (b) Unlike ReLu and its variants, the activation function of ELU is *smooth everywhere*, include around $x = 0$, which speed up gradient descent because it does not bounce as much left and right of $x = 0$.

   (c) The main drawback of the ELU activation funciton is that it is slower to compute than ReLU and its variants due to its use of exponential function. This is not too much of a problem during training because the slower computation is compensated by the faster convergence, but testing it will be considerably slower than ReLU.

(d) There is a variant of ELU, called scaled ELU (SELU), which has an additional hyperparameter $\lambda$ to achieve internal normalization.

$$\text{SELU} = \lambda \text{ELU}(x) = \begin{cases} \lambda x & \text{if } x \geq 0 \\ \lambda\alpha(e^x - 1) & \text{otherwise} \end{cases}$$

### 3.3 Choosing activation functions

1. If runtime performance is NOT a concern, in general the relative performance of different choice is: ELU > leaky ReLU (and its variants) > ReLU > tanh > sigmoid
2. If runtime is an important factor to consider, leaky ReLU will be a better choice than ELU.
3. The hyperparameter in leaky ReLU and ELU may be learned to further boost the performance if the training set is sufficiently large (or else overfitting may occur).

### 3.4 Weight initialization

1. Observations (which are consequences of using non-saturating activation function such as ReLU)
   (a) If the weights in a network start too small, then the signal shrinks as it passes through each layer until it is too tiny to be useful.
   (b) If the weights in a network start too large, then the signal grows as it passes through each layer until it is too massive to be useful.
2. Xavier Initialisation
   (a) The weight intialization strategy called Xavier initialization (named after the author's first name) makes sure the weights in a reasonable range of values so that the signals can reach deep into network.
   (b) The main idea is to let the distribution governing the intialization of weights depend on the number of input connections and the number of output connections for the layer whose weights are being initialised.
   (c) Let $\begin{cases} n_{\text{in}} & := \text{number of input connections in a layer} \\ n_{\text{out}} & := \text{number of output connections in a layer} \end{cases}$, the weights may be initialised randomly using a normal distribution

$$\mathcal{N}(0, \frac{2}{n_{\text{in}} + n_{\text{out}}})$$

   or a uniform distribution

$$\mathcal{U}(-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, +\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}})$$

   .
3. He Initialisation
   (a) The He initialisation (named after the author's last name), also called Kaiming initialisation (named after the author's first name), is for ReLU and its variants, including ELU.
   (b) Similar to Xavier Initialisation, the weights may be intialsed using a normal distribution

$$\mathcal{N}(0, \frac{4}{n_{\text{in}} + n_{\text{out}}})$$

   or a uniform distribution

$$\mathcal{U}(-\sqrt{\frac{12}{n_{\text{in}} + n_{\text{out}}}}, +\sqrt{\frac{12}{n_{\text{in}} + n_{\text{out}}}})$$

   .

### 3.5 Batch Normalisation

1. Internal Covariate Shift

   (a) Although using a carefully designed weight initialisation strategy along with a non-saturating activation function such as leacky ReLU or ELU can significantly alleviate the vanishing/ exploring gradient problem at the beginning of training, this problem may still arise later during training.

   (b) The internal covariate shift problem refers that the distribution of the inputs of each layer changes during training as the network weights of the previous layers change.

2. Overviews of batch normalisation
   Batch normalisation is a technique for addressing the vanishing/ exploding gradient problem by addressing the more general internal covariate shift problem.

   (a) In a way, batch normalisation can be seen as extending the idea of normalising the raw input to the upper layers of the network

   (b) Before applying the activation function of each layer, the mean and variance of the inputs over the current mini-batch are evaluated to zero-center and normalise the inputs by the evaluated variance. (Hence, the name "batch" normalisation).

   (c) Two new parameters $\gamma, \beta$, one for scaling and the other for shifting, are learnt for each layer to restore its representation power.

   (d) Unlike normalising the input which are done for the whole batch, batch normalisation does it for each mini-batch separately.

   (e) While batch normalisation is generally quite effective, the true reason why and how it works is still not fully understood.

3. Transform with Learning Parameters

   (a) Batch normalising transform applied over a mini-batch

**Algorithm 2** Batch Normalising Transform

**Input**: Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1\cdots m}\}$
**Output**: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$
$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$ ▷ mini-batch mean
$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$ ▷ mini-batch variance
$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}}$ ▷ normalise
$y_i \leftarrow \gamma\hat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$ ▷ scale and shift

   (b) The transform can be implemented by introducing an additional layer with learnable parameters $\gamma, \beta$.

   (c) With batch normalisation, the vanishing gradient problem can be significantly reduced even when saturating activation functions such as the sigmoid and hyperbolic tangent functions are used without relying on carefully designed weight initialisation and regularisation.

4. Batch normalisation during prediction

   (a) Like dropout, batch normalisation has different computation results in training and prediction modes.

   (b) During training, normalisation is based on the mini-batch statistics.

   (c) During prediction, normalisation is based on the dataset statistics (i.e., the whole dataset).

### 3.6 Dropout

1. Dropout is a regularisation technique to prevent overfitting.
2. At each training step, each hidden unit or input has a probability $p$ of being temporarily dropped out (i.e., ignored during this training step, but it may become active again during the next step).
3. The hyperparameter $p$, called the dropout rate, is typically set to 0.5 (but other values are also possible, even with different values for different layers). Sometimes it is more convenient to refer to the keep probability with is equal to $1 - p$.
4. In general, a higher dropout rate is used if more serious overfitting is observed and for layers with more units (and hence more parameters to estimate).
5. Dropout is *only applied during training* but not during testing.
6. Dropout can also be applied to connections (i.e., weights), but we will focus on dropping out units here.
7. Like L1/ L2 regularisation, dropout is only applied to the non-bias units.
8. Justifications of dropout
   (a) Since hidden units and inputs are dropped out randomly, the network need to learn to be more robust (i.e., less sensitive to slight changes in the inputs) by replying on more incoming units or inputs rather than just a few, effective spreading out the weights to more units.
   (b) An alternative way is to view dropout as *emsemble learning*, in that each training step "generates" one of $2^N$ possible networks, where $N$ is the total number of droppable units or inputs, and trains the network on one example from the training set. The resulting neural network is essentially an emsemble of a large number of smaller networks.
9. Since dropout is not applied during testing, a hidden unit will on average be connected to $\frac{1}{1-p}$ times as many inputs as it was during training. There are two alternative methods to deal with this problem.
   (a) Multiply the input weights of each unit by the keep probability $1 - p$ after training.
   (b) (Inverted dropout) Divide the output of each unit by the keep probability during training.

   Strictly speaking, these two methods are not equivalent but they both work well. Specifically, nothing needs to be done during testing and hence inference will not be slowed down.

### 3.7 Data Augmentation

1. Overfitting can be reduced by having more training data, but it is not always possible to get more labelled training examples.
2. Data augmentation is a regularisation technique that generates new, synthetic training instances from existing ones to increase the size of the training set effectively.
3. Data augmentation aims to explicitly let the model know the allowed invariances.
4. Instead of wasting storage space and network bandwidth, it is preferable to generate new training instances *on the fly* during training.
5. Example: Image Data
   (a) Some commonly used operations:
      i. Translation, rotation, or rotation by various amounts

ii. Lighting with varous contrasts
iii. Cropping
iv. Flipping horizontally (but note that it is not suitable for some image-based applications such as character recognition).

(b) Some deep learning frameworks such as `TensorFlow` provide image manipulation operations for generating new training instances efficiently on the fly.

(c) Some generative models (such as GAN) and generative AI tools may also be used for data augmentation.

### 3.8 Optimizers

1. AdaGrad

   (a) Recall the ordinary Stochastic Gradient Descent (SGD):

   $$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} L$$

   , where $\nabla$ denotes the vector differential operator.

   (b) AdaGrad improves SGD using an adaptive learning rate which decays faster for steeper dimensions.

   $$\mathbf{s} \leftarrow \mathbf{s} + \nabla_{\mathbf{w}} L \circ \nabla_{\mathbf{w}} L$$
   $$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} L \oslash \sqrt{\mathbf{s} + \epsilon}$$

   , where $\circ$ denotes the Hadamard product (a.k.a. elementwise multiplication), $\oslash$ denotes the Hadamard division (a.k.a. elementwise division), and $\epsilon$ is a smoothing term with a small value (e.g., $10^{-10}$) for all dimensions to avoid division by 0.

   (c) The first step of AdaGrad accumulates the square of the gradients into vector $\mathbf{s}$: a dimension of $\mathbf{s}$ will become larger and larger if the loss function is steep along that dimension.

   (d) The second step is similar to SGD except that there is a scaling factor $\sqrt{\mathbf{s} + \epsilon}$: a dimension will decay its learning rate faster if the loss function is steep along that dimension.

   (e) With the adaptive learning rate (without requiring manual training), AdaGrad helps point the result updates more directly towards the global optimum.

   (f) Although AdaGrad performs well for simple quadratic problems, it often stops too early before reaching the global optimum when used for training neural networks.

2. RMSProp

   (a) To overcome the problem of AdaGrad which decays too fast, RMSProp does not accumulate all the gradient from the beginning but only from the recent iterations by introducing exponential decay in the first step.

   $$\mathbf{s} \leftarrow \beta \mathbf{s} + (1 - \beta) \nabla_{\mathbf{w}} L \circ \nabla_{\mathbf{w}} L$$
   $$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} L \oslash \sqrt{\mathbf{s} + \epsilon}$$

   , where the decay rate $\beta$ is typically set to 0.9.

   (b) If we explicitly add a time step as subscript and use $\mathbf{g}$ to denote $\nabla_{\mathbf{w}} L \circ \nabla_{\mathbf{w}} L$, the first step of RMSProp can be expressed as the recurrence relation $\mathbf{s}_t = \beta \mathbf{s}_{t-1} + (1 - \beta)\mathbf{g}_t$. Iterating this recurrence yields the following:

   $$\mathbf{s}_t = \beta^t + (1 - \beta)(\beta^{t-1}\mathbf{g}_1 + \beta^{t-2}\mathbf{g}_2 + \cdots + \beta \mathbf{g}_{t-1} + \mathbf{g}_t) = \beta^t + \sum_{i=1}^{t} \beta^{t-i} \mathbf{g}_i$$

3. Adam (adaptive moment estimation)

(a) Adam combines the idead of momentum and RMSProp.

i. Like momentum, Adam keeps track of an exponential decaying average of the past gradients.

ii. Like RMSProp, it keeps track of an exponential decaying average of the past squared gradients.

$$\begin{aligned}
\mathbf{\Delta w} &\leftarrow \beta_1 \mathbf{\Delta w} - (1 - \beta_1 \nabla_{\mathbf{w}} \mathbf{L}) \\
\mathbf{s} &\leftarrow \beta_2 \mathbf{s} + (1 - \beta_2) \nabla_{\mathbf{w}} L \circ \nabla_{\mathbf{w}} L \\
\mathbf{\Delta w} &\leftarrow \frac{\mathbf{\Delta w}}{1 - \beta_1} \qquad\qquad \mathbf{s} \leftarrow \frac{\mathbf{s}}{1 - \beta_2} \\
\mathbf{w} &\leftarrow \mathbf{w} + \eta \mathbf{\Delta w} \oslash \sqrt{\mathbf{s} + \epsilon}
\end{aligned}$$

, where $t$ is the iteration number. The third step is for bias correction because the running averages are initialized to 0.

*Remarks*:

(a) Although Adam generally works well and is faster than other methods, a study showed that adaptive optimization methods including AdaGrad, RMSProp and Adam can give solutions that generalize poorly on some data sets.

(b) So there is no single method that is always the best.