Deeplearning summary

BUI TRUNG NGUYEN

Sorbonne Paris Nord University

November 10, 2024

Contents

Chapter 1

Neural network

1.1 Foward and Backward pass

In this section, we will step-by-step brake down every step in a Multi-Layer Neural Network to understand their role in the model as well as figuring the imperative computation of parameters must be accomplised in these steps.

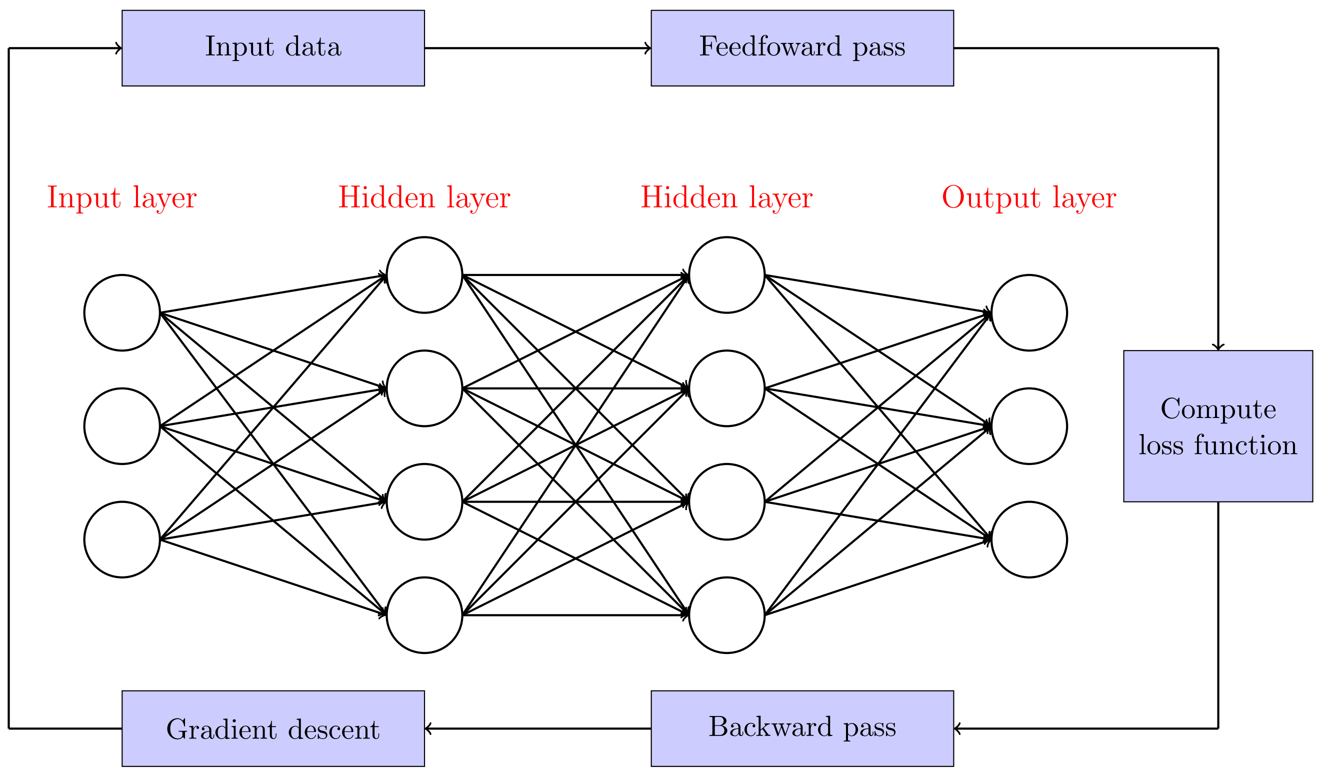

Figure 1.1: Procedure to find the optimal solution in Multi-layer Neural Network

Generally, a basic Multi-layer Neural Network consists of 5 steps:

- **Load input data:** data is fed into the network. Depend on the optimization strategy, the number of data input can be different.
- **Feedforward pass:** calculate the activation output of neurons for all layers based on *non-linear* operation.

- **Compute loss function:** the loss function is computed in the output layer to quantify the performance of the model's optimization.
- **Backward pass:** calculate the gradient of loss function w.r.t the parameters weight matrix w^l and bias vector b^l
- **Gradient descend:** the parameters w^l,b^l are updated by using gradient of cost function to improve the performace. Unless the model completely loads every element in the dataset, the new input unit will be continuously fed into the model.

Once the **Backward pass** is reached, a new data will be fed to feedforward pass again to redo the exact procedure until the model either achieves favoriable performance or fails to solve the problem.

1.1.1 Forward Pass

For the forward pass of neuron network, or so also known as a perceptron, we define the weighted input to a neuron l^{th} by:

$$z^l_j = \Sigma_k w^l_{kj} a^{l-1}_k + b^l_j \tag{1.1}$$

where:
a^{l-1}_k is the input from the neuron k^{th} of layer $(l-1)^{th}$
w^l_{kj} is the weight corresponding to the link between neuron k^{th} of layer $(l-1)^{th}$ and neuron j^{th} of layer l^{th}
b^l_j is the bias term of the neuron j^{th} of the layer l^{th}

The weighted input to neuron z^k_j is passed through an activation function $\sigma()$ (which will be carefully declared in chapter 2). The result of activation function will either become the next layer's neurons input or the output of the network. Then we obtain the expectated output of the neuron j^{th} of layer l^{th} as follow:

$$a^l_j = \sigma\left(\Sigma_k w^l_{kj} a^{l-1}_k + b^l_j\right) \tag{1.2}$$

In fact, this representation (1.2) only well describe the output of a single neuron. In practice, we would like utilize the power of linear algebra in order to avoid looping so many time to calculate for each neuron in the layer. Because of this, we would like to rewrite the (1.2) in the matrix form:

$$a^l = \sigma\left(w^l a^{l-1} + b^l\right) \tag{1.3}$$

Remark 1:the activation function for (1.3) is **element-wise** function.
Remark 2:some textbooks may write (1.3) as $a^l = \sigma\left((w^l)^T a^{l-1} + b^l\right)$ in which the weight matrix w^l is transposed.This unsynchronization is dued to the difference in notation of the weight matrix component w^l_{jk} in (1.2) where we use the inverse order of relationship between 2 connected neurons, e.g the "suppose" to be correct description for this parameter is "the weight between neuron k^{th}-layer $(l-1)^{th}$ connects to neuron j^{th}-layer l^{th}", so the precise notation should have been w^l_{kj}. But once we use the reverse notation, it allows us to get rid of the transpose operation for matrix representation!

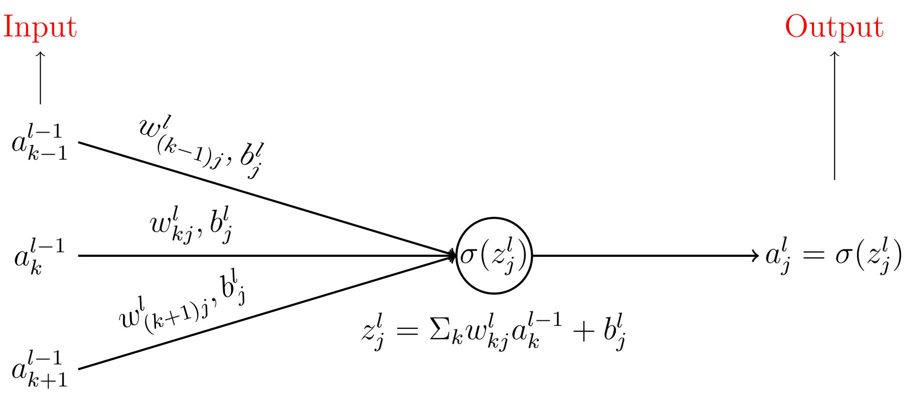

Figure 1.2: Illustration input and output of a neuron

1.1.2 Backward pass (Backpropagation)

Objective: calculate the rate of change of loss function w.r.t the weight and the bias of each layer. Unfortunately, we don't have direct formulas to accomplish this job, but we can still calculate these unknowns through 2 supporting equations that will be derived as follow.

Calculating the error in the output layer

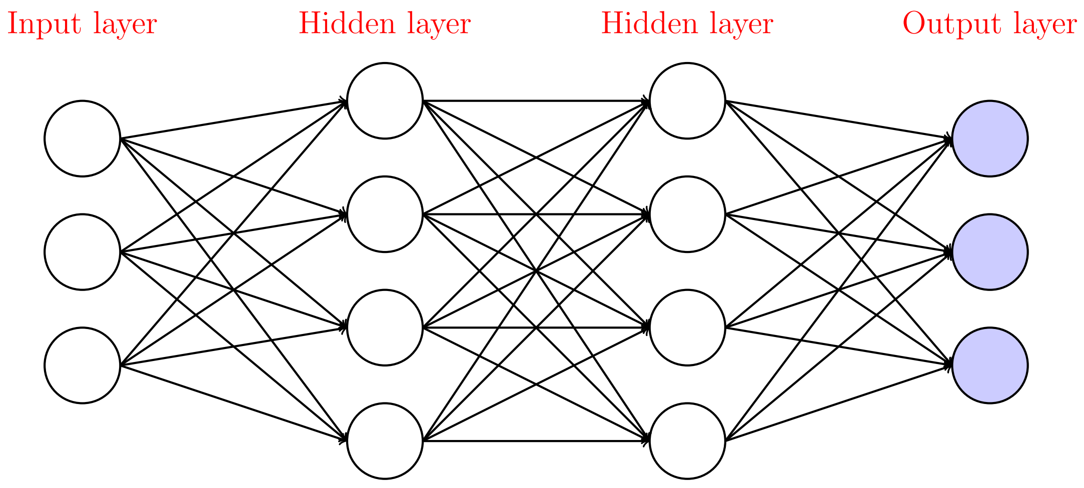

Figure 1.3: Calculating the error of output layer

The error of neuron j^{th} of the output layer L is calculated by

$$\delta^L_j = \frac{\partial \mathcal{C}}{\partial a^L_j} \sigma^{'}(z^L_j) \tag{1.4}$$

or in matrix form:

$$\delta^L = \nabla_a \mathcal{C} \odot \sigma^{'}(z^L) \tag{1.5}$$

where $\frac{\partial \mathcal{C}}{\partial a^L_j}$ indicates how fast the cost function changes w.r.t the output layer activation of neuron j^{th} and $\sigma^{'}(z^L_j)$ measures the rate of change of activation function σ w.r.t the z^L_j. In the matrix form, $\odot$ is called Hadamard product. The result of uv is the element-wise product u_i and v_i of u and v respectively. Moreover, ∇ is called "nabla" - the notation for vector differential operation.

Interpretation: this factor allows us to know the dependency of the cost function C on the output neuron j^{th}. If the neuron j^{th} of the output layer doesn't contribute much to the cost function result, then the error δ^L should be small

For example: $\mathcal{C}$ is MSE function $\mathcal{C} = ||a^L - y||_2$ then the error δ^L is calculated by: $\delta^L = (a^L - y) \odot \sigma^{'}(z^L)$

Calculate the error δ^l in terms of the error in the next layer δ^{l+1}

$$\delta_k^l = (w_{jk}^{l+1})(\delta_j^{l+1})\sigma^{'}(z_k^l) \tag{1.6}$$

or in matrix form:

$$\delta^l = ((w^{l+1})^T\delta^{l+1}) \odot \sigma^{'}(z^l) \tag{1.7}$$

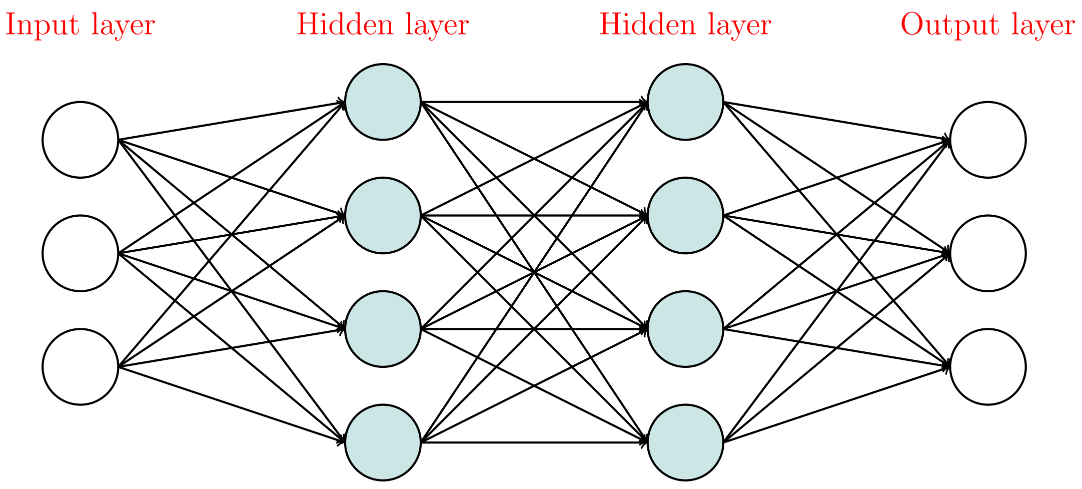

Figure 1.4: Calculating the error of hidden layers

where δ^{l+1},w^{l+1} are the error and weight terms of the next layer or the right next layer according to the figure 1.4
Interpretation: initially once we compute the (1.7), we then have enough "inertia" to go backward from the output layer back to input layer direction in order to calculate the error term for each hidden layer.

Now with the support from (1.5) and (1.7), we are confident to say that the goal of backpropagation is feasible to achieve. The formulas to estimate the rate of change of the loss function C w.r.t the weight and bias of each layer are:

Calculate the rate of change of the cost w.r.t any bias in the network

$$\frac{\partial \mathcal{C}}{\partial b_j^l} = \delta_j^l \tag{1.8}$$

The equation is straightforward ! The rate of change of loss fucntion w.r.t the bias is <u>the error of each layer</u>.

Calculate the rate of change of the cost w.r.t any weight in the network

$$\frac{\partial \mathcal{C}}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \tag{1.9}$$

the rate of change of the cost w.r.t the weight is depended on the activation output of neuron k^{th} of layer $(l-1)^{th}$ and the error term of neuron l^{th} of layer l^{th} itself.

Evaluation: What can we perceive from the result of (1.8) **and** (1.9) **?**
Answer: the $\frac{\partial C}{\partial w}$ is <u>small</u> meaning that the weight tend to update their weight <u>slower</u>. There are factors that can cause this result: (1) when the activation ouput $a^{l-1}_k \approx 0$, (2) when the error $\delta^l_j \approx 0$
For 1^{th} factor to the design of activation function that can force the weight input to low value. For 2^{th} factor we must go backward to the error of output layer at (1.5). When the rate of change of activation of output layer $\sigma^{'}(z^L_j)$ is small, it means that the output neuron had stepped to saturated region where z^L_J is not likely to vary its value. Then the error of output neuron j^{th} is small and makes the error term δ^l_k is small as well according to (1.7).
The explaination for the bias gradient hold the same idea as above. The phenomenom when gradient of weight $\frac{\partial \mathcal{C}}{w}$ and gradient of bias $\frac{\partial \mathcal{C}}{b}$ are becoming very small <u>during the training</u> (when the loss function is not yet minimized) is called **vanishing gradient**. The original source of such phenomenom is from the degradation of activation gradient $\sigma^{'}(z^l)$ which triggers a chain effect to $\frac{\partial \mathcal{C}}{b}, \frac{\partial \mathcal{C}}{w}$. Depend on the usage of activation function, we are going to analyze clearer the way to get rid of vanishing gradient in section 3

1.2 Optimizer

1.2.1 Gradient Descent and its variants

Stochastic Gradient Descent (SGD)

This is the most naive way to update the weight matrix as this optimizer repeats the update sequentially for every 1 input sample. *Weight update formula for any layer in SGD*

$$w^l := w^l - \alpha \nabla_{w^l} \mathcal{C} \tag{1.10}$$

where $\alpha > 0$ is the learning rate

Batch Gradient Descent

Instead of considering 1 input only, the idea of batch GD is to feed the entire dataset to the neural network. What is changed in batch GD formula compares to SGD formula is the update subtrahende term as it becomes the average of all subtrahends corresponding to each data sample. We adjust batch formula based on the computation of the loss function for the whole dataset with A samples:

$$\mathcal{C}_w = \frac{1}{N} \sum_{i=1}^{A} \mathcal{C}_w(x_i, y_i) \tag{1.11}$$

Then the batch GD becomes:

$$w^l := w^l - \alpha \frac{1}{A} \sum_{i=1}^{N} \nabla_{w^l} \mathcal{C}(x_i, y_i) \tag{1.12}$$

Remind that we will implement vectorization for these formula, so if the dataset has A samples, then the dimension of $\nabla_{w^l}C$ matrix is (A,N,M) but the dimension of w^l is still (N,M) (N,M are the height and width of weight matrix of each layer).
One reason that batch GD is not so commonly used is because the size of dataset is usually very large nowadays such that it is no longer economical to upgrade the hardware for loading the entire dataset. Moreover, the efficiency of this optimizer is not always assured since the weight matrix is updated only 1 time which may not bring the best converged result for the model.

Mini-Batch Gradient Descent

Mini-Batch has the same idea as batch GD except the fact this strategy split the dataset into many batchs and feed them to the model one-by-one. That means, the weight will be updated more frequently and still be able to ultize the parallelization computational power of the hardware. Given a dataset that is split into B batchs, each with C samples, then the formula to update the weight for 1 batch is:

$$w^l := w^l - \alpha \frac{1}{A} \sum_{i=1}^{C} \nabla_{w^l} \mathcal{C}(x_i, y_i) \tag{1.13}$$

Mini-batch is one of the widely used strategy for optimization the neural network.

Stochastic Gradient Descent with Momentum

The SGD with momentum offers a velocity terms to speed up the optimizating step, especially when the neural network model get stuck at a local minimum. In that case, if the *learning rate* is set small, the model will accept the local minimum as general solution hence unable to optimize the model. On the contrary, if we would like to get rid of this aannoying issue by simply setting the *learning rate* to a bigger value, then probally we can accidentally "jump" over the global minimum. These problems are indicated in figure 1.5.

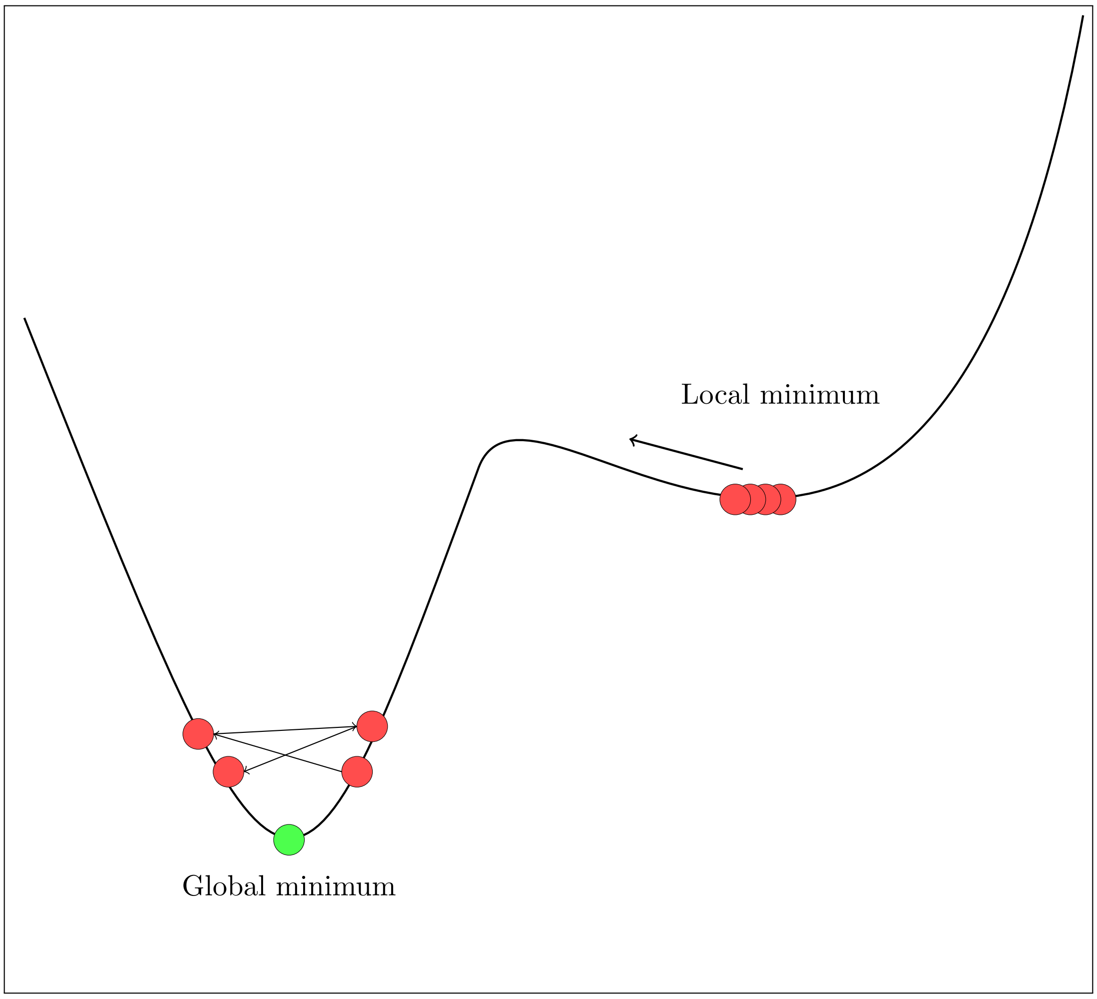

Figure 1.5: Illustration for big *learning rate* and small *learning rate* situation. In the former case, the red points are the attemps of neural network model to approach optimized point (green point) but large LR occurs the next update value keep jumpng around the global bottom point. In the later case, we can see multiple attemps to pass the local maxima but because of small LR, the updated effort is very slow.

Hence a new variable v which defines the velocity (speed and direction) for the gradient descent algorithm upgrading purpose. The SGD with momentum algorithm now contains 2 formulas; Given a dataset that is split into B batchs, each with C samples, then the formula to update the weight for 1 batch is:

$$\begin{aligned} v &= \alpha v - \epsilon \nabla_\theta \left(\frac{1}{A} \sum_{i=1}^{C} \nabla_{w^l} \mathcal{C}(x_i, y_i) \right) \\ w^l &= w^l + v \end{aligned} \tag{1.14}$$

where $\alpha \in [0, 1)$ is hyperparameter which controls the rate of previous velocity.

1.2.2 Nesterov Accelerated Gradient

Given a dataset that is split into B batchs, each with C samples, then the formula to update the weight for 1 batch is:

$$\begin{aligned} v &= \alpha v - \epsilon \nabla_\theta \left(\frac{1}{A} \sum_{i=1}^{C} \mathcal{C}(f(x_i; w^l + \alpha v), y_i) \right) \\ w^l &= w^l + v \end{aligned} \tag{1.15}$$

1.2.3 Adaptive Gradient Algorithm (Adagrad)

1.2.4 Root Mean Square Propagation (RMSProp)

1.2.5 Adaptive Moment Estimation (Adam) and its variants

Adaptive Moment Estimation (Adam)

Adam with Weight Decay (AdamW)

AdamMax

Nesterov-accelerated Adam (Nadam)

1.3 Activation function

1.3.1 Sigmoid function

$$f(x) = \frac{1}{1+e^{-x}} \tag{1.16}$$

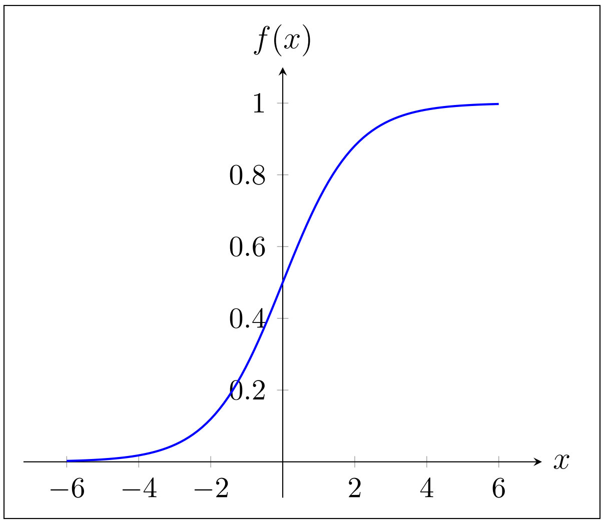

Figure 1.6: Sigmoid function

1.3.2 Softmax function

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^j} \tag{1.17}$$

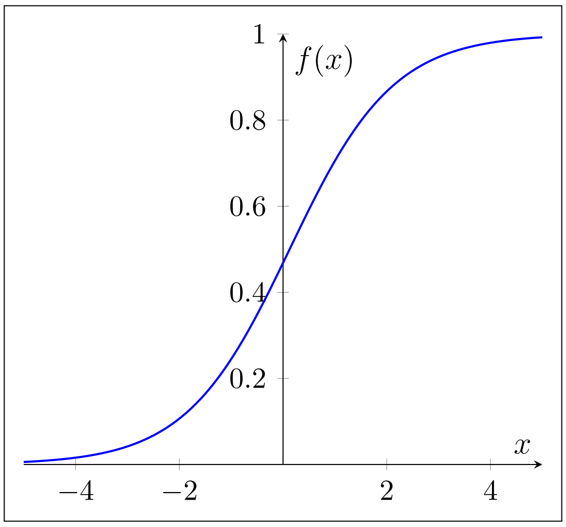

Figure 1.7: Softmax function

1.3.3 Hyperbolic Tangent (Tanh) function

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{1.18}$$

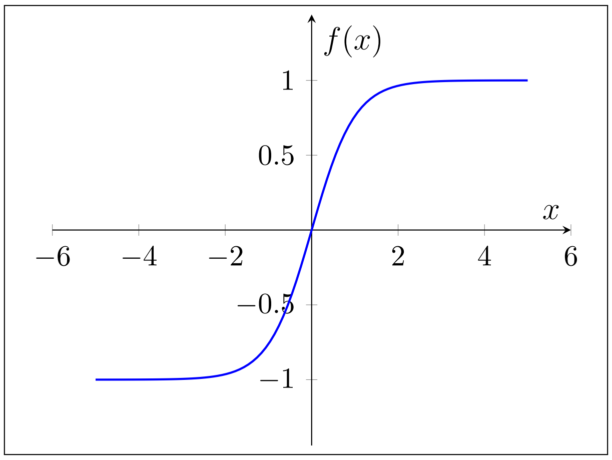

Figure 1.8: TanH function

1.3.4 Rectifier Unit (ReLU) function

$$f(x) = \begin{cases} x & \text{if x } \geq 0, \\ 0 & \text{if x } < 0 \end{cases} \tag{1.19}$$

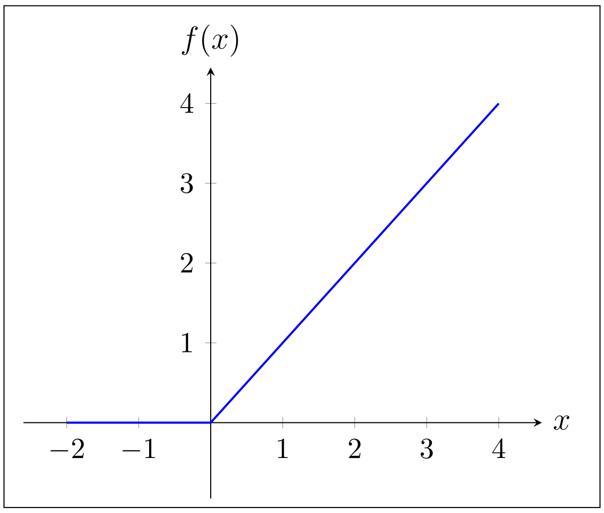

Figure 1.9: ReLU function

1.3.5 Leaky Rectifier Unit (LeakyReLU) function

$$f(x) = \begin{cases} x & \text{if x } \geq 0, \\ \alpha x & \text{if x } < 0 \end{cases} \tag{1.20}$$

where $\alpha > 0$

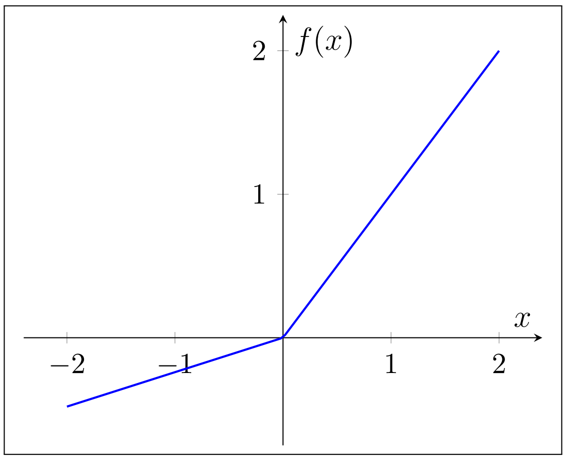

Figure 1.10: Leaky ReLU function

1.3.6 Exponential Linear Unit (ELU) function

$$f(x) = \begin{cases} x & \text{if x} \geq 0 \\ \alpha(e^x - 1) & \text{if x } < 0 \end{cases} \tag{1.21}$$

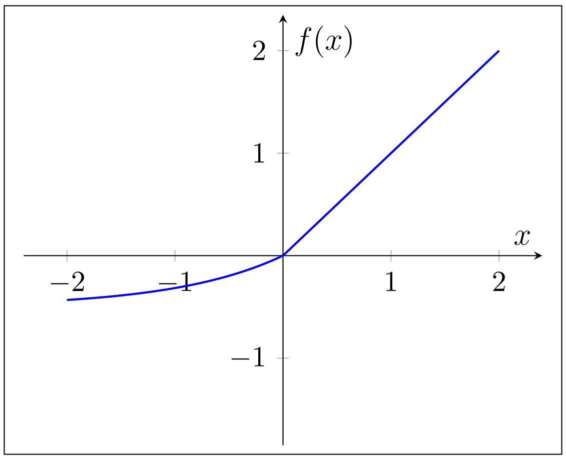

Figure 1.11: ELU function

1.3.7 Gaussian Error Linear Unit (GELU) function

$$\begin{aligned} f(x) =& x\Phi(x) \\ =& \frac{x}{2}\left(1 + erf\left(\frac{x}{\sqrt{2}}\right)\right) \end{aligned} \tag{1.22}$$

where $erf()$ function is...

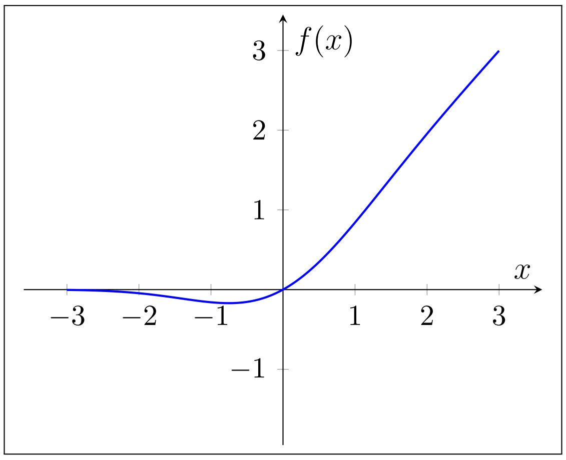

Figure 1.12: GELU function

1.3.8 Swish function

$$\begin{aligned} f(x) =& x\sigma(\beta x) \\ =& \frac{x}{1 + e^{-\beta x}} \end{aligned} \tag{1.23}$$

where $\sigma()$ function is *sigmoid* function

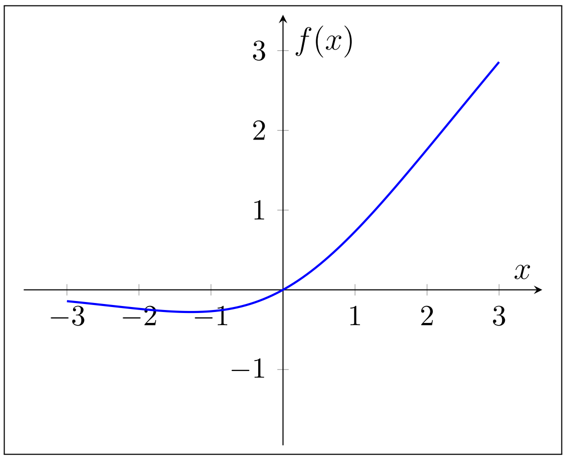

Figure 1.13: GELU function

1.3.9 Maxout function

$$f(x) = MAX(w_1^T x + b_1, w_2^T x + b_2, ..., w_k^T x + b_k) \tag{1.24}$$

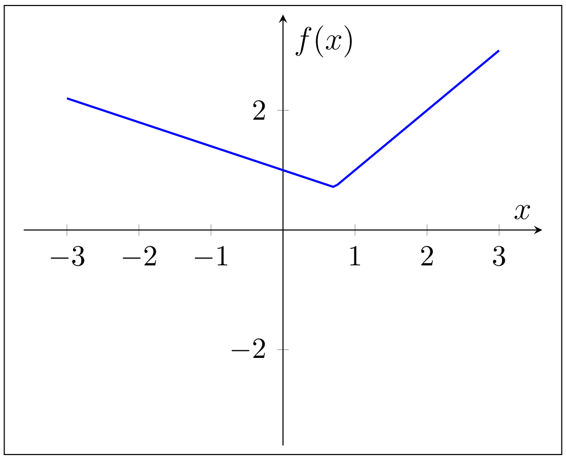

Figure 1.14: Maxout function

1.4 Loss function

1.4.1 Mean Squared Error (MSE) loss

$$\begin{aligned}\mathcal{L}(x, \hat{x}) =& ||x - \hat{x}||_2 \\ =& \frac{1}{N}\sum_i^N (x_i - \hat{x}_i)^2\end{aligned} \tag{1.25}$$

1.4.2 Mean Absolute Error (MSA) loss

$$\begin{aligned}\mathcal{L}(x, \hat{x}) =& ||x - \hat{x}|| \\ =& \frac{1}{N}\sum_i^N (x_i - \hat{x}_i)\end{aligned} \tag{1.26}$$

1.4.3 Huber loss

$$\mathcal{L}_\delta(x, \hat{x}) = \begin{cases} 0.5(x - \hat{x})^2 & \text{if } |x - \hat{x}| < \delta \\ \delta(|x - \hat{x}| - 0.5\delta) & \text{otherwise} \end{cases} \tag{1.27}$$

1.4.4 Cross-Entropy loss and its variants

Cross-Entropy loss

Categorical Cross-Entropy loss

Sparse Categorical Cross-Entropy loss

1.4.5 Dice loss

1.4.6 Focal loss

1.4.7 Triplet loss

1.4.8 Tversky loss

1.5 Proved of equation in backpropagation

Chapter 2

Convolutional Neural Network (CNN)

2.1 Convolutional Neural Network

2.1.1 Vectorized convolution by Im2col and col2im

The convolution operation can be implemented by many loops because the kernel will travel over the image. In terms of complexity, this is the easiest way to program but regard to timing, this idea is not efficient since it can only perform a convolution calculation sequentially. Now, with the highly support of hardward (explicitly GPU) that allows us to accelerate even more with parallel computation. In this part, we would like to break down 2 approaches in order to analyse their advantages and downsides.

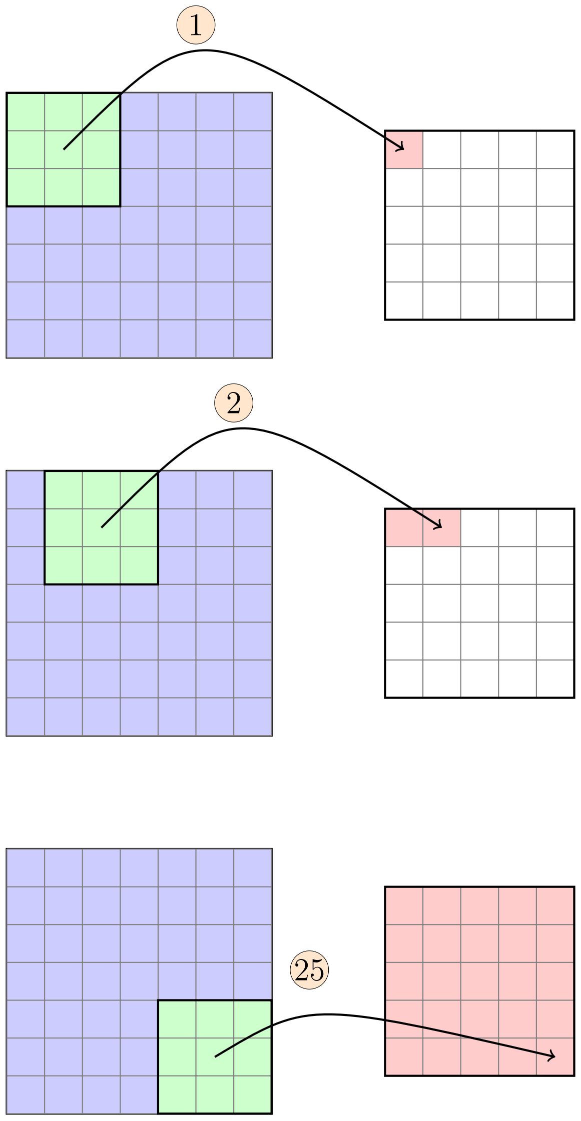

Figure 2.1: Convolution operation between 7x7 input and 3x3 kernel filter with $stride = 1$. If we use a loop then each iteration we slide over a 3x3 subgrid of input, we have to do9 multiplications and 8 addition. The number operations we have to do is $9x8x25 = 1800$ operations for 1 convolution. Remind that with this looping implementation, we can only do $9x8 = 72$ operations sequentially.