

Deeplearning summary

BUI TRUNG NGUYEN

Sorbonne Paris Nord University

December 25, 2024

Contents

1	Neural network	3
1.1	Foward and Backward pass	3
1.1.1	Forward Pass	4
1.1.2	Backward pass (Backpropagation)	5
1.2	Optimizer	7
1.2.1	Gradient Descent and its variants	7
1.2.2	Nesterov Accelerated Gradient	8
1.2.3	Adaptive Gradient Algorithm (Adagrad)	9
1.2.4	Root Mean Square Propagation (RMSProp)	9
1.2.5	Adaptive Moment Estimation (Adam) and its variants	9
1.3	Activation function	9
1.3.1	Sigmoid function	9
1.3.2	Softmax function	10
1.3.3	Hyperbolic Tangent (Tanh) function	10
1.3.4	Rectifier Unit (ReLU) function	11
1.3.5	Leaky Rectifier Unit (LeakyReLU) function	11
1.3.6	Exponential Linear Unit (ELU) function	12
1.3.7	Gaussian Error Linear Unit (GELU) function	12
1.3.8	Swish function	13
1.3.9	Maxout function	13
1.4	Loss function	14
1.4.1	Mean Squared Error (MSE) loss	14
1.4.2	Mean Absolute Error (MSA) loss	14
1.4.3	Huber loss	14
1.4.4	Cross-Entropy loss	14
1.4.5	Dice loss	16
1.4.6	Focal loss	18
1.4.7	Triplet loss	19
1.4.8	Tversky loss	19
1.5	Metrics	19
1.5.1	Metrics for classification	19
1.6	Proved of equation in backpropagation	19
2	Convolutional Neural Network (CNN)	20
2.1	Convolutional Neural Network	20
2.1.1	Vectorized convolution by Im2col and col2im	20
3	Configure deep learning model	24
3.1	Introduce common datatype in deep learning	24
3.2	Weight quantization	25

3.2.1	Post-training quantization	25
3.2.2	Quantization-aware training	27

Chapter 1

Neural network

1.1 Forward and Backward pass

In this section, we will step-by-step break 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 accomplished in these steps.

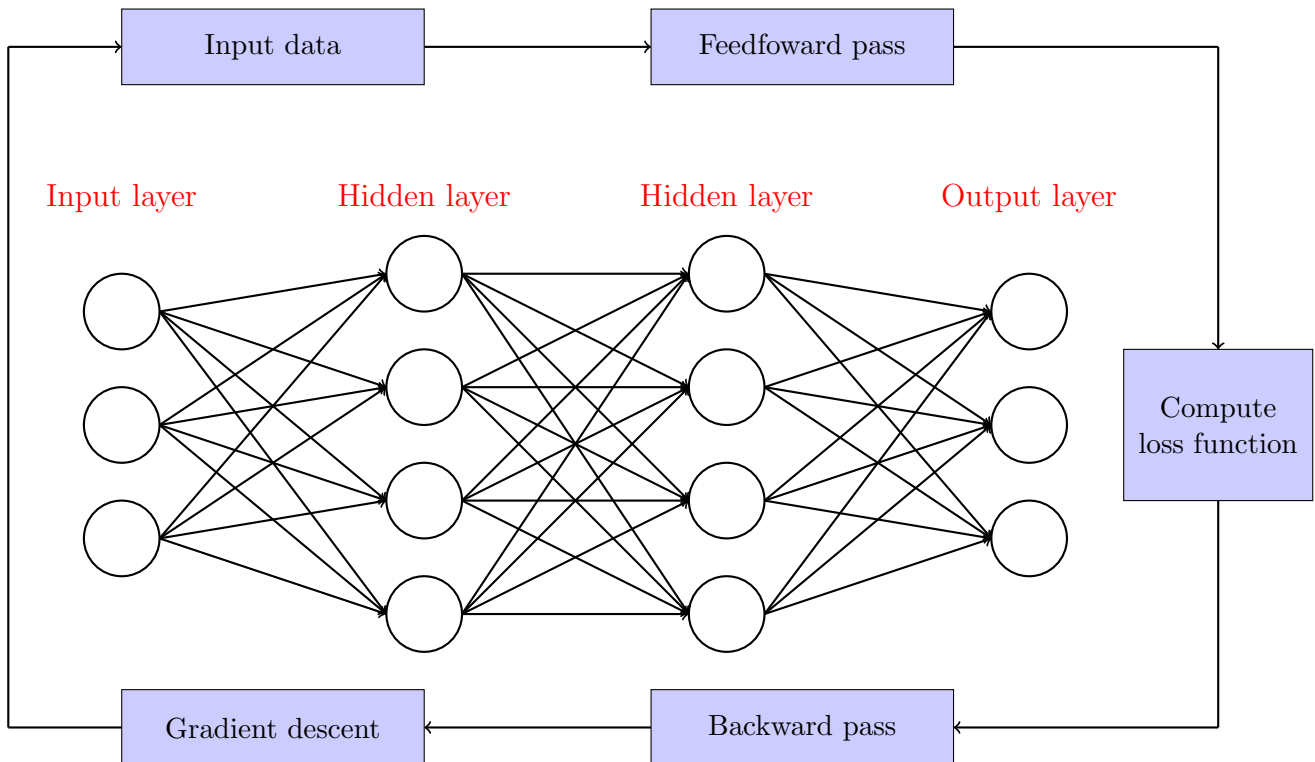


Figure 1.1: Procedure to find the optimal solution in Multi-layer Neural Network

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 performance. 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 favorable 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:

$$z_j^l = \sum_k w_{kj}^l a_k^{l-1} + b_j^l \quad (1.1)$$

where:

a_k^{l-1} is the input from the neuron k^{th} of layer $(l-1)^{th}$

w_{kj}^l 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_j^l is the bias term of the neuron j^{th} of the layer l^{th}

The weighted input to neuron z_j^k 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 expected output of the neuron j^{th} of layer l^{th} as follow:

$$a_j^l = \sigma(\sum_k w_{kj}^l a_k^{l-1} + b_j^l) \quad (1.2)$$

u 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:

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

Remark 1:the activation function for (1.3) is **element-wise** function.

Remark 2:some textbooks may write (1.3) as $a^l = \sigma((w^l)^T a^{l-1} + b^l)$ in which the weight matrix w^l is transposed. This unsynchronization is due to the difference in notation of the weight matrix component w_{jk}^l 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_{kj}^l . But once we use the reverse notation, it allows us to get rid of the transpose operation for matrix representation!

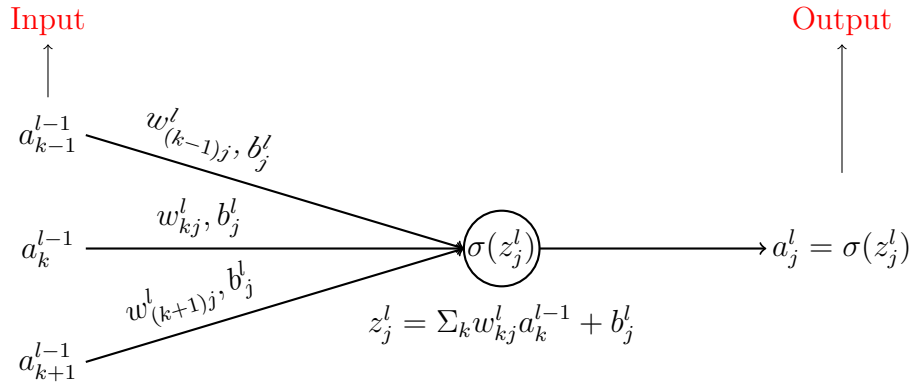


Figure 1.2: Illustration input and output of a neuron

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

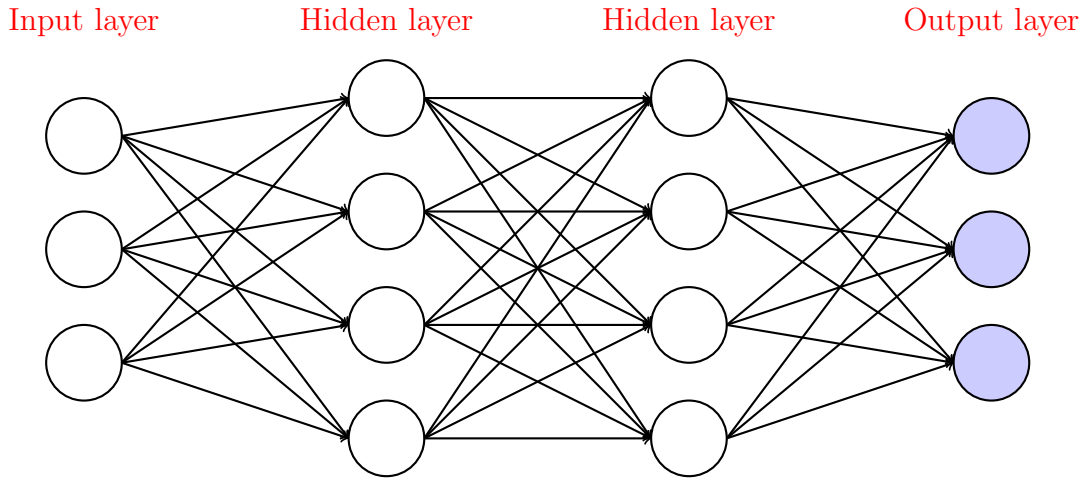


Figure 1.3: Calculating the error of output layer

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

$$\delta_j^L = \frac{\partial \mathcal{C}}{\partial a_j^L} \sigma'(z_j^L) \quad (1.4)$$

or in matrix form:

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

where $\frac{\partial \mathcal{C}}{\partial a_j^L}$ indicates how fast the cost function changes w.r.t the output layer activation of neuron j^{th} and $\sigma'(z_j^L)$ measures the rate of change of activation function σ w.r.t the z_j^L . In the matrix form, \odot is called Hadamard product. The result of $u \odot v$ 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}

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

or in matrix form:

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

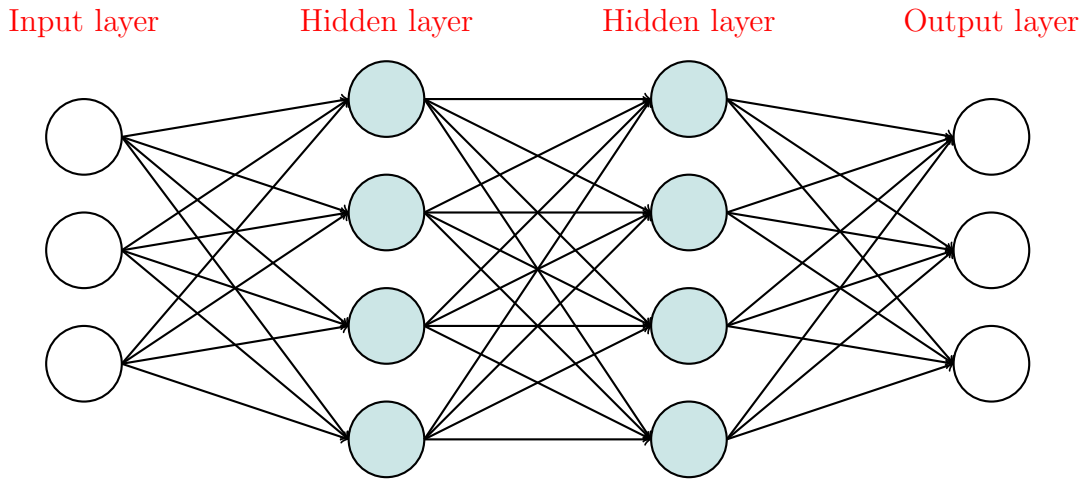


Figure 1.4: Calculating the error of hidden layers

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

$$\frac{\partial \mathcal{C}}{\partial b_j^l} = \delta_j^l \quad (1.8)$$

The equation is straightforward! The rate of change of loss function w.r.t the bias is the error of each layer.

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

$$\frac{\partial \mathcal{C}}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \quad (1.9)$$

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 $\frac{\partial \mathcal{C}}{\partial w}$ is small meaning that the weight tend to update their weight slower. There are factors that can cause this result: (1) when the activation output $a_k^{l-1} \approx 0$, (2) when the error $\delta_j^l \approx 0$

For 1th factor to the design of activation function that can force the weight input to low value. For 2th factor we must go backward to the error of output layer at (1.5). When the rate of change of activation of output layer $\sigma'(z_j^L)$ is small, it means that the output neuron had stepped to saturated region where z_j^L is not likely to vary its value. Then the error of output neuron j^{th} is small and makes the error term δ_k^l is small as well according to (1.7).

The explanation for the bias gradient hold the same idea as above. The phenomenon when gradient of weight $\frac{\partial \mathcal{C}}{\partial w}$ and gradient of bias $\frac{\partial \mathcal{C}}{\partial b}$ are becoming very small during the training (when the loss function is not yet minimized) is called **vanishing gradient**. The original source of such phenomenon is from the degradation of activation gradient $\sigma'(z^l)$ which triggers a chain effect to $\frac{\partial \mathcal{C}}{\partial b}, \frac{\partial \mathcal{C}}{\partial w}$. 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*

$$w^l := w^l - \alpha \nabla_{w^l} \mathcal{C} \quad (1.10)$$

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:

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

Then the batch GD becomes:

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

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:

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

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 optimizing 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.

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:

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

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:

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

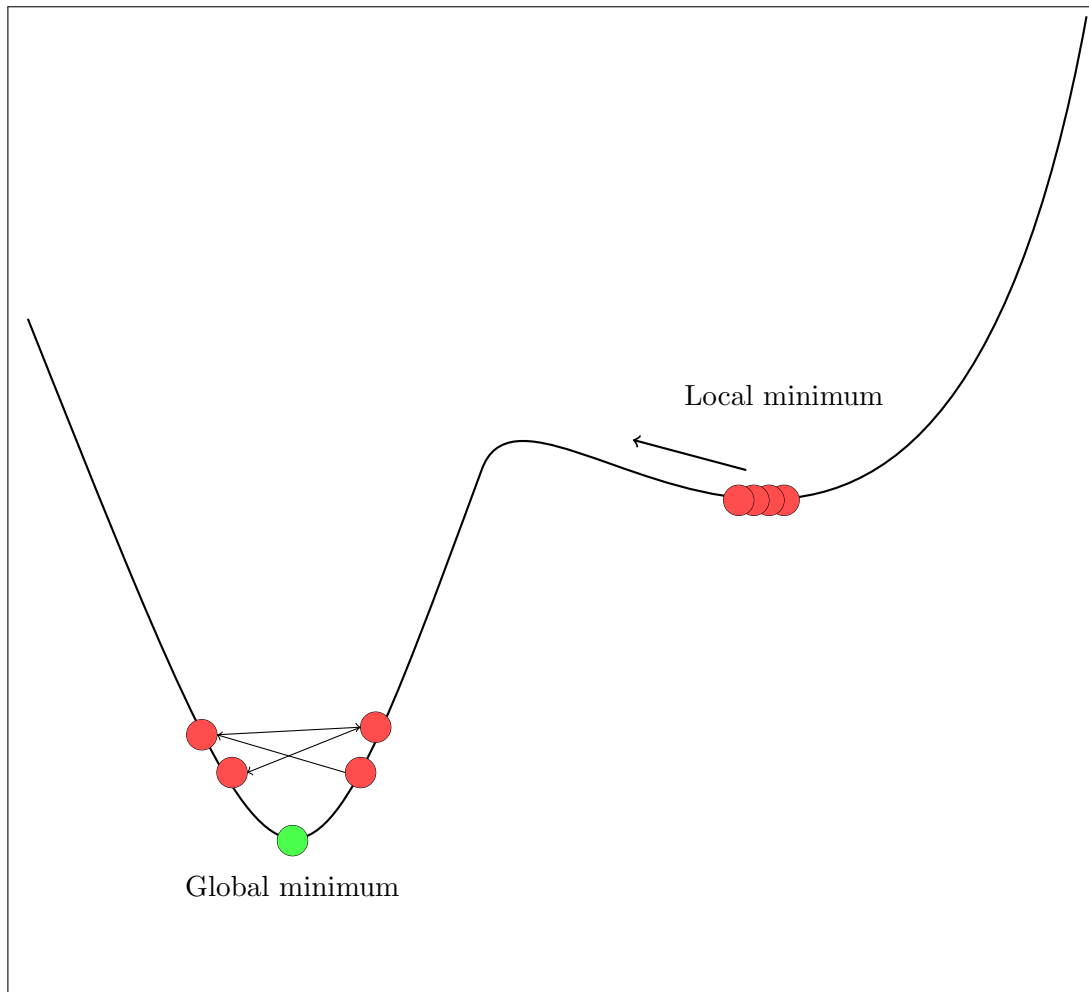


Figure 1.5: Illustration for big *learning rate* and small *learning rate* situation. In the former case, the red points are the attempts of neural network model to approach optimized point (green point) but large LR occurs the next update value keep jumping around the global bottom point. In the later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

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

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

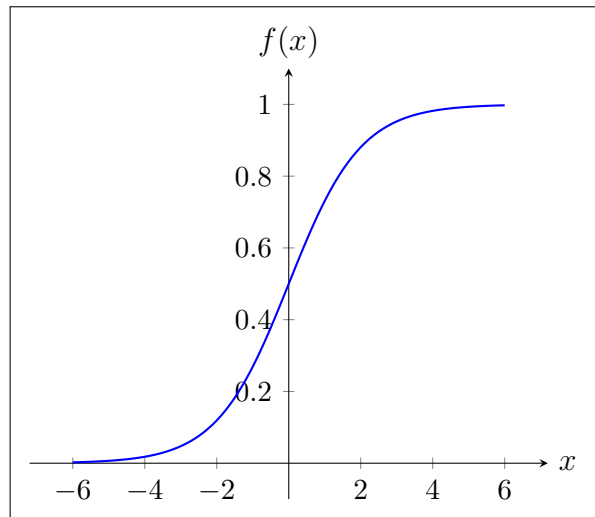


Figure 1.6: Sigmoid function

1.3.2 Softmax function

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

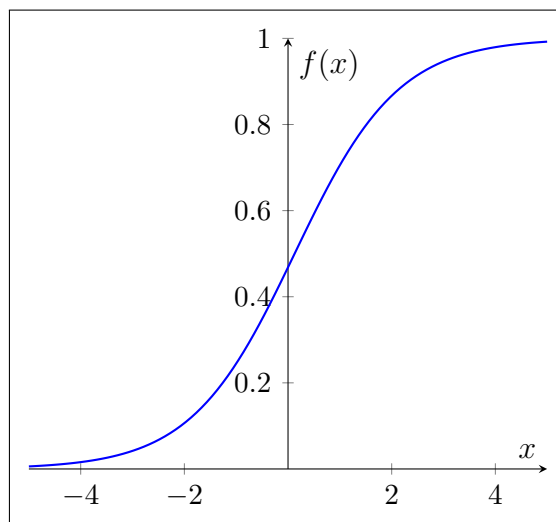


Figure 1.7: Softmax function

1.3.3 Hyperbolic Tangent (Tanh) function

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

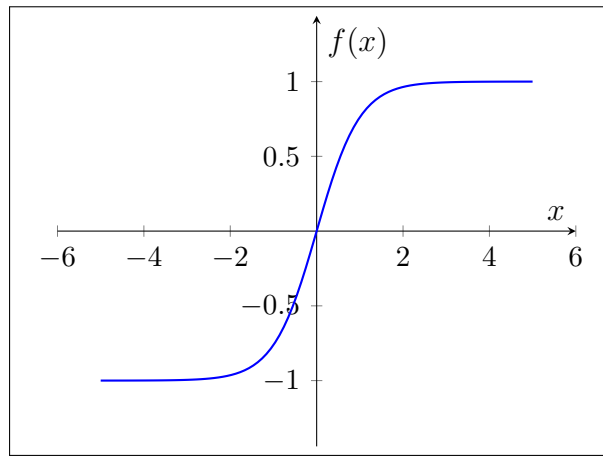


Figure 1.8: TanH function

1.3.4 Rectifier Unit (ReLU) function

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

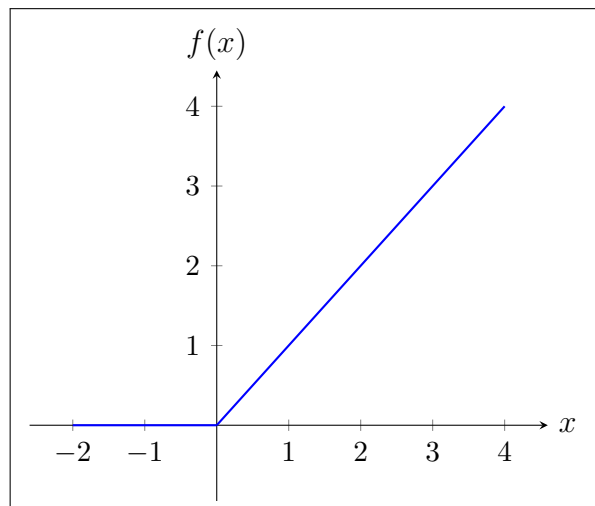


Figure 1.9: ReLU function

1.3.5 Leaky Rectifier Unit (LeakyReLU) function

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

where $\alpha > 0$

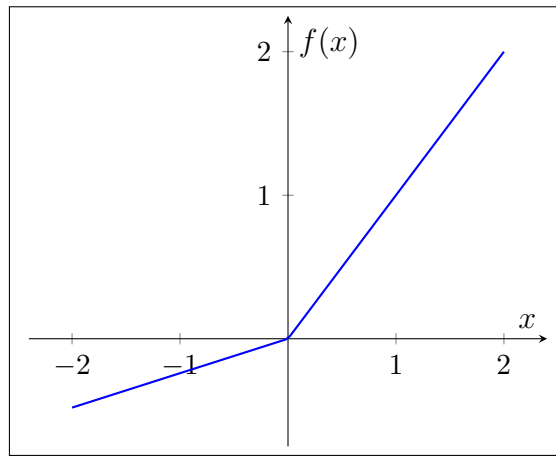


Figure 1.10: Leaky ReLU function

1.3.6 Exponential Linear Unit (ELU) function

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

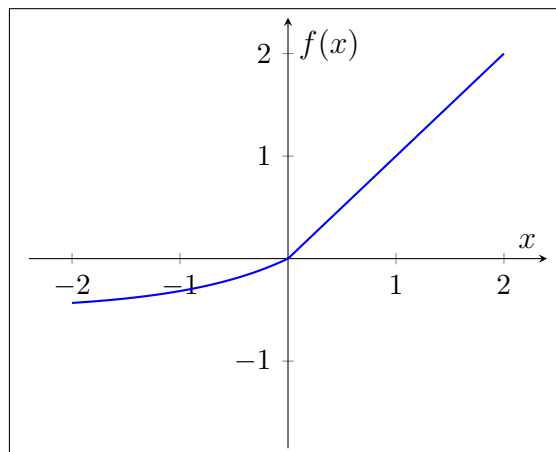


Figure 1.11: ELU function

1.3.7 Gaussian Error Linear Unit (GELU) function

$$f(x) = x\Phi(x) = \frac{x}{2} \left(1 + \operatorname{erf} \left(\frac{x}{\sqrt{2}} \right) \right) \quad (1.22)$$

where $\operatorname{erf}()$ function is

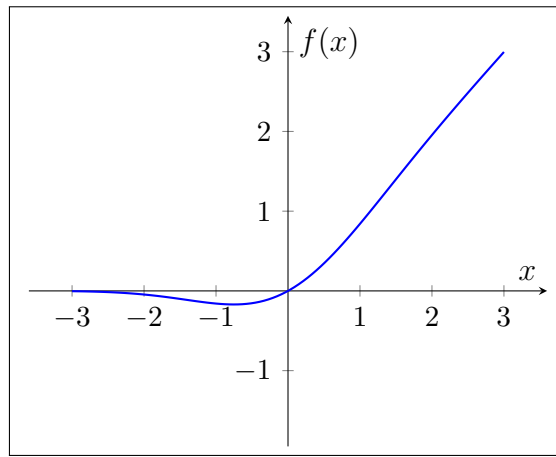


Figure 1.12: GELU function

1.3.8 Swish function

$$f(x) = x\sigma(\beta x) = \frac{x}{1 + e^{-\beta x}} \quad (1.23)$$

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

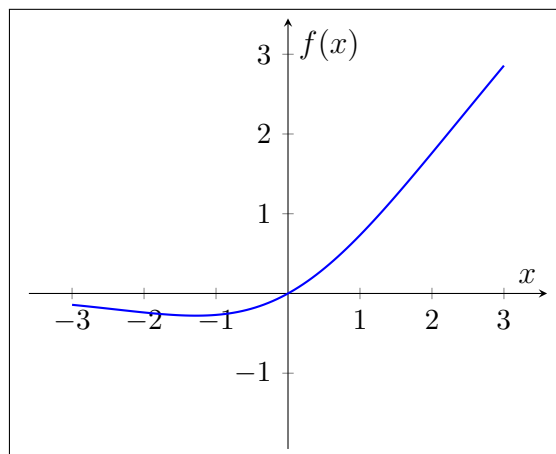


Figure 1.13: GELU function

1.3.9 Maxout function

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

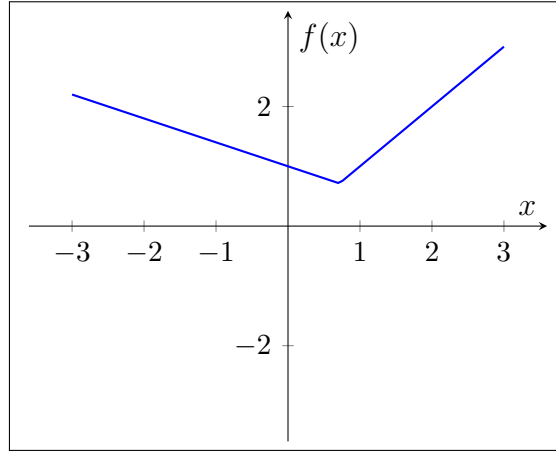


Figure 1.14: Maxout function

1.4 Loss function

1.4.1 Mean Squared Error (MSE) loss

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

1.4.2 Mean Absolute Error (MSA) loss

$$\begin{aligned}\mathcal{L}(y, \hat{y}) &= \|y - \hat{y}\|_1 \\ &= \frac{1}{N} \sum_i^N (y_i - \hat{y}_i)\end{aligned}\tag{1.26}$$

1.4.3 Huber loss

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

1.4.4 Cross-Entropy loss

Cross-Entropy loss is widely used for classification problem. The name of this loss function associates to terminology in *information theory* "Entropy" in which it measures the average information per symbol with the given source. In particular, the smaller entropy is, the more information is provided by the source. The idea of Cross-Entropy function implies the same manner as we would like to minimize the value of Cross-Entropy to optimize the model. Given the label \hat{y} formatted in binary values 0,1 and prediction $p(y)$ formatted as the probability output of softmax function, the Cross-Entropy is defined by:

$$\mathcal{L}(y, \hat{y}) = -\frac{1}{N} \sum_{i=1}^N \hat{y}_i \log(p(y_i))\tag{1.28}$$

Although the Cross-Entropy formula is transparent enough, it is better to check out an example to see how this loss function is explicitly calculated.

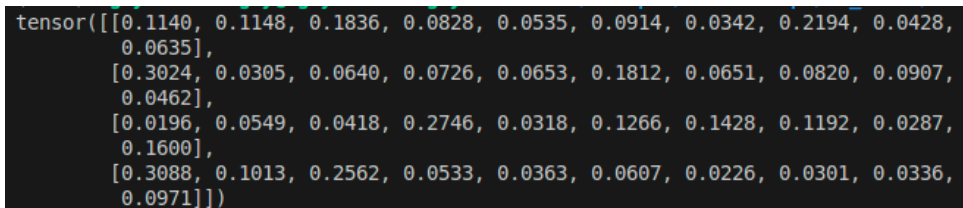
For example, we assume that the in 1 mini-batche has 4 samples corresponding to 4 predictions. There are 10 different classes to classify in this dataset so we can see in the code 1.1, e.g the **predicts** variable is a (4,10) matrix with 4 predictions consist of logits represent for 10 classes classification. In order to obtain the probabilities for each class to guess the sample input's class, we need to feed these logits to the softmax function (refer 1.15), in which the total sum of probabilities for 10 classes is 1. Then the way equation 1.29 is applied can be interpreted by taking the labels¹ to map the corresponding probability of the softmax output, then take the log and the mean of them. Remark that we can also transform the integer labels to the one-hot encoding style but it would take $4 \times 10 = 40$ vectors to implement it.

```

1 import numpy as np
2 import torch
3 import torch.nn.functional as F
4
5
6 g_cpu = torch.Generator()
7 g_cpu.manual_seed(12)
8 predicts = torch.randn(4,10,generator=g_cpu)
9 labels = torch.LongTensor([2,3,0,1])
10
11 # Calculate the softmax of logits
12 output_probs = F.softmax(predicts,dim=1)
13 print(output_probs)
14 print(labels)
15
16 # Gather the true class probability
17 true_class_probs = output_probs.gather(1,labels.view(-1,1))
18 print(true_class_probs)
19 # Calculate the loss
20 loss = torch.mean(-torch.log(true_class_probs))

```

Listing 1.1: Example for calculating cross entropy loss

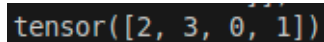


```

tensor([[0.1140, 0.1148, 0.1836, 0.0828, 0.0535, 0.0914, 0.0342, 0.2194, 0.0428,
         0.0635],
        [0.3024, 0.0305, 0.0640, 0.0726, 0.0653, 0.1812, 0.0651, 0.0820, 0.0907,
         0.0462],
        [0.0196, 0.0549, 0.0418, 0.2746, 0.0318, 0.1266, 0.1428, 0.1192, 0.0287,
         0.1600],
        [0.3088, 0.1013, 0.2562, 0.0533, 0.0363, 0.0607, 0.0226, 0.0301, 0.0336,
         0.0971]])

```

Figure 1.15: The **predicts** mini-batch has 4 samples, each prediction has 10 classes. This figure indicates the predictions after applying softmax function which are stored in **output_probs** variables.



```

tensor([2, 3, 0, 1])

```

Figure 1.16: Ground truth index by the **labels**

¹normally in classification, we mark the ground truth for the sample in either integer or one-hot encoding. In the code 1.1, we use integer


```
tensor([[0.1836],
        [0.0726],
        [0.0196],
        [0.1013]])
```

Figure 1.17: Match the probabilities with the ground truth indices by **true_class_probs**. Remind that when the obtained values are also the ones that have the highest probability then they are the correct prediction, elsewhere they would be the failed prediction.

```
Cross-Entropy Loss: 0.3788231611251831
```

Figure 1.18: Cross entropy loss

Cross-Entropy in Imbalance dataset

In some extreme cases, there exists a class that dominate others in terms of quantity within a dataset, the loss function will behave biasly toward the one who got the most direct influence to the numeric result but not the desired goal that we want, i.e in tumor classification, normally, the benign case will be much more popular than malignant case but the prediction must not gravitate toward predicting all the inputs as benign although doing so will yield a good evaluation metric (explicit for accuracy). Therefore, in order to leverage the Cross-Entropy loss in Imbalance data, we usually multiply a weight term to penalize more the minor class and relief the impact if the major class got the correct prediction.

$$\mathcal{L}(y, \hat{y}) = -\frac{1}{N} \sum_{i=1}^N \mathbf{w} \hat{y}_i \log(p(y_i)) \quad (1.29)$$

where \mathbf{w} is the weight term corresponding to each class, in which:

$$\mathbf{w} = \begin{cases} \frac{\text{num_major}}{\text{num_total}} & \text{for major class} \\ \frac{\text{num_minor}}{\text{num_total}} & \text{for minor class} \end{cases} \quad (1.30)$$

1.4.5 Dice loss

For Dice loss:

$$\mathcal{L}(y, \hat{y}) = 1 - \frac{1}{N} \sum_{i=1}^N \frac{2 \times \sum(y_i \times \hat{y}_i)}{\sum(y_i) + \sum(\hat{y}_i) + \epsilon} \quad (1.31)$$

where ϵ uses to avoid null denominator. In the numerator is the operation **intersect** the predictions and ground truths while at the denominator is the **union** of predictions and ground truths. If we take account of the part after the subtraction by 1, we call this term as **Dice coefficient**. The advantage of Dice loss is that it focus on checking over the performance of classifier if the predictions and true labels are overlapped. Moreover, Dice loss doesn't be affected much by the imbalance dataset since the denominator term has already normalized the numerator term, in which for example the background class dominant the whole image, then the prediction likely to bias over the major class (which is background in this case) will not make a great impact the overall Dice loss since the union of predictions and true labels is big as well. On the other hand, if a prediction of a minor class is excellent then its contribution to the Dice loss is as big as major classes.

Example: A 200×200 medical image is used to predict the tumor of human brain. The ground truth has 3500 pixels of background class and 500 pixels of tumor class. The prediction yield 3700 pixels of background class (which 3400 are True Positive) and 300 pixels of tumor class (which 250 are True Positive). Mark background class as $c = 0$ and tumor as $c = 1$ then the Dice loss given by equation 1.31 can be calculated as follow:

$$\text{Dice coef } (c = 0) = \frac{2 \times 3400}{3500 + 3700} = 0.944$$

$$\text{Dice coef } (c = 1) = \frac{2 \times 250}{500 + 300} = 0.625$$

$$\Rightarrow \text{Dice loss} = 1 - \frac{0.944 + 0.625}{2} = 0.2155$$

```

1 import numpy as np
2 import torch
3 import torch.nn.functional as F
4
5
6 g_cpu = torch.Generator()
7 g_cpu.manual_seed(12)
8 predicts = torch.randn(4,10,generator=g_cpu)
9 labels = torch.LongTensor([2,3,0,1])
10 num_classes = 10
11 eps = 1e-7
12
13 # Calculate the softmax of logits
14 output_probs = F.softmax(predicts,dim=1)
15 preds = torch.argmax(output_probs,dim=1)
16 print(preds)
17 preds_flat = preds.view(-1)
18 dice_score=[]
19
20 for i in range(num_classes):
21     true_classes = (labels == i).float()
22     print(true_classes)
23     preds_classes = (preds_flat == i).float()
24     print(preds_classes)
25
26     intersection = torch.sum(true_classes*preds_classes)
27     union = torch.sum(true_classes) + torch.sum(preds_classes)
28
29     dice = (2.0 * intersection) / (union + eps)
30     dice_score.append(dice)
31
32 dice_loss = 1-torch.tensor(dice_score).mean()
33 print(dice_loss)

```

Listing 1.2: Multi classes Dice loss calculation

1.4.6 Focal loss

First, with the estimated probability of the model for the class $y = 1$, we denote the general case for any $y \in 0, 1$ as follow:

$$p_t = \begin{cases} p & , \text{if } \hat{y} = 1 \\ 1 - p & , \text{if } \hat{y} = 0 \end{cases} \quad (1.32)$$

Focal loss is then defined by:

$$\mathcal{L}(p_t) = -(1 - p_t)^\gamma \log(p_t) \quad (1.33)$$

where $(1 - p_t)^\gamma$ is the modulating factor, $\gamma \geq 0$ is the focusing parameter.

The design of Focal loss is relatively associate with the cross-entropy loss function in case of imbalanced dataset Cross-Entropy as 2 ideas both address the imbalance by multiply the Cross-Entropy with a weight factor (it's the modulating factor in Focal loss) but the Focal loss proposed 2 properties:

- When the \hat{y} is misclassified, p_t can't be maximum but rather to be small thus lead to $(1 - p_t)$ near 1 \Rightarrow The loss is unaffected
- For easy example, the $p_t \rightarrow 1$, therefore $\gamma \in [0, 5]$ will down-weighted the $(1 - p_t)^\gamma$ and reduce the loss contribution for easy case

For instance: an easy example has $p_t = 0.9$; we set $\gamma = 2$, then the focal loss is: $l_f = -1(1 - 0.9)^2 \log(0.9) = 0.000475 < l_{ce} = -\log(0.9) = 0.04575$. It's 100 times scale down

while a misclassified example has $p_t = 0.4$; we set $\gamma = 2$, then the focal loss is: $l_f = -1(1 - 0.4)^2 \log(0.4) = 0.1432 < l_{ce} = -\log(0.4) = 0.3979$. It's 4 times scale down.

We observe that for the hard case, there is more room for the loss to continuously optimize (from $0.1432 \rightarrow 0$) compares to the easy case (from $0.000475 \rightarrow 0$) during backpropagation process. Or it can be understood that the gradient yielded from a very small loss will barely makes a significant update to the weight regarding the easy examples. The illustrating code for focal loss is presented at Code 1.3

```

1 import numpy as np
2 import torch
3 import torch.nn.functional as F
4
5
6 g_cpu = torch.Generator()
7 g_cpu.manual_seed(12)
8 predicts = torch.randn(4,10,generator=g_cpu)
9 labels = torch.LongTensor([2,3,0,1])
10 gamma = 2 # focusing factor
11 alpha = 0.25 # Balancing factor
12
13
14 # Calculate the softmax of logits
15 output_probs = F.softmax(predicts,dim=1)
16 print(output_probs)
17 print(labels)
18
19 # Gather the true class probability
20 true_class_probs = output_probs.gather(1,labels.view(-1,1))
21 print(true_class_probs)
22

```

```
23 # Calculate the focal loss
24 loss = torch.mean(-torch.log(true_class_probs))
25 focal_loss = torch.mean(-alpha*torch.pow((1-true_class_probs),gamma)*torch.
    log(true_class_probs))
26
27 print(loss)
28 print(focal_loss)
```

Listing 1.3: Focal loss calculating

1.4.7 Triplet loss

d

1.4.8 Tversky loss

1.5 Metrics

While loss function plays an essential role in deeplearning optimization, it also provides the sense of how good the model is learning through batches. Nevertheless, loss function is not always be able to indicate the quantitative perspective of the model's performance, especially in such case requires the model to focus in a more prioritized class than others. For example, in pathological cancer staging classification, the prize of precisely predicting stage 0 must be much lesser than predicting stage 2 since the number of samples that have positive in stage 2 is as not comparable as healthy samples. In this section, we would like to categorized the metrics types according to the output format of the task.

1.5.1 Metrics for classification

Accuracy

Recall

Precision

F1 score

Sensitivity

Specificity

Positive predicted value (PPV)/ Negative predicted value (NPV)

Received Operating Characteristic (ROC)

Precision-Recall curve

Confidence Interval (CI)

1.6 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 hardware (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.

In Im2col, we will trade off some memory for faster computation by transforming the 3 - channels image to a matrix form in which the convolution operation can be performed more efficiently. We list out 3 stages to implement the efficient way to do vectorized convolution:

1. Im2Col for input image and reshape the kernel: transform image and kernel to column form
2. Do the matrix multiplication between Im2Col output and reshaped kernel
3. Reshape the output of matrix multiplication to the form of convolution output.

In particular, for step 1, the intensity values of images are within the sliding window will be flattened to a column, then the matrix will consist of many transformed columns like this, covering all the sliding position of the kernel scans through the images, check out figure 2.2. Moreover, in the case the image is colored (presumably 3 channels RGB), the output is the concatenated matrix by Im2col output with respect to 3 channels. Obviously, in order to make matrix multiplication feasible, the kernel need to be reshaped to row vector form as well (refer figure 2.4,2.5).

Then the 2nd step is to do matrix multiplication between Im2Col output and reshaped kernels. Thus the result of 3-channels vectors multiply with 3-channels reshaped kernels rows is a single numeric value (figure 2.6). Once the we found the matrix output of step 2, we need to convert this matrix into the form of convolution operation's result. For example, regardless padding option and set stride as 1, the image size $7 \times 7 \times 3$ convolves with $3 \times 3 \times N$ kernels will produce $5 \times 5 \times N$ matrix (note: the number N is usually referred to the number of filter/the depth of feature map respectively for kernel/feature map case). So the output of matrix multiplication for vectorized convolution has the size $5 \times (5 * N)$ will be reshaped back to form $5 \times 5 \times N$ (figure 2.6)

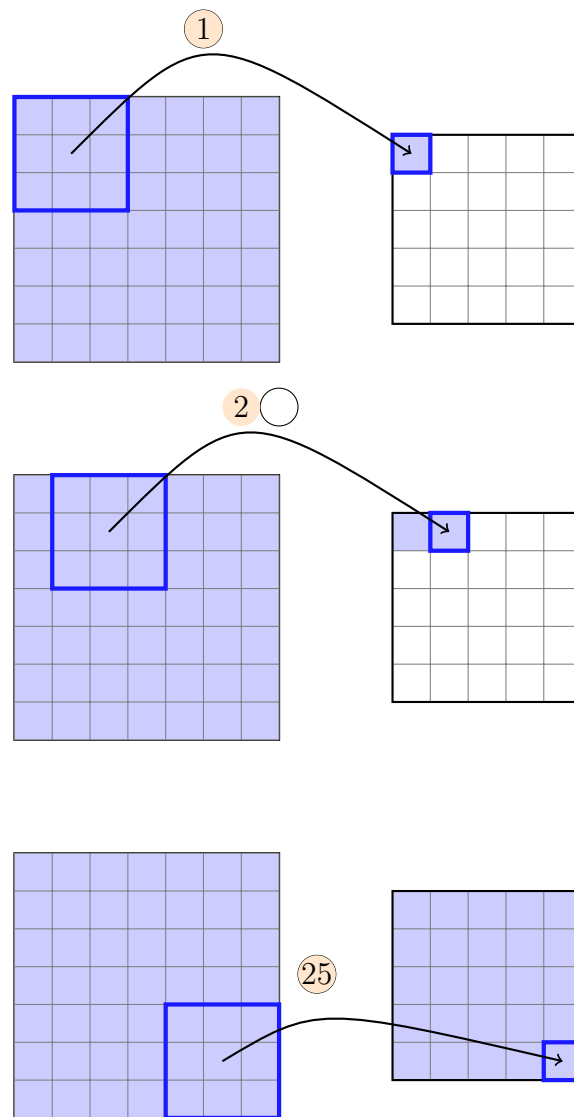


Figure 2.1: Convolution operation between 7×7 input and 3×3 kernel filter with $stride = 1$. If we use a loop then each iteration we slide over a 3×3 subgrid of input, we have to do 9 multiplications and 8 addition. The number operations we have to do is $9 \times 8 \times 25 = 1800$ operations for 1 convolution. Remind that with this looping implementation, we can only do $9 \times 8 = 72$ operations sequentially.

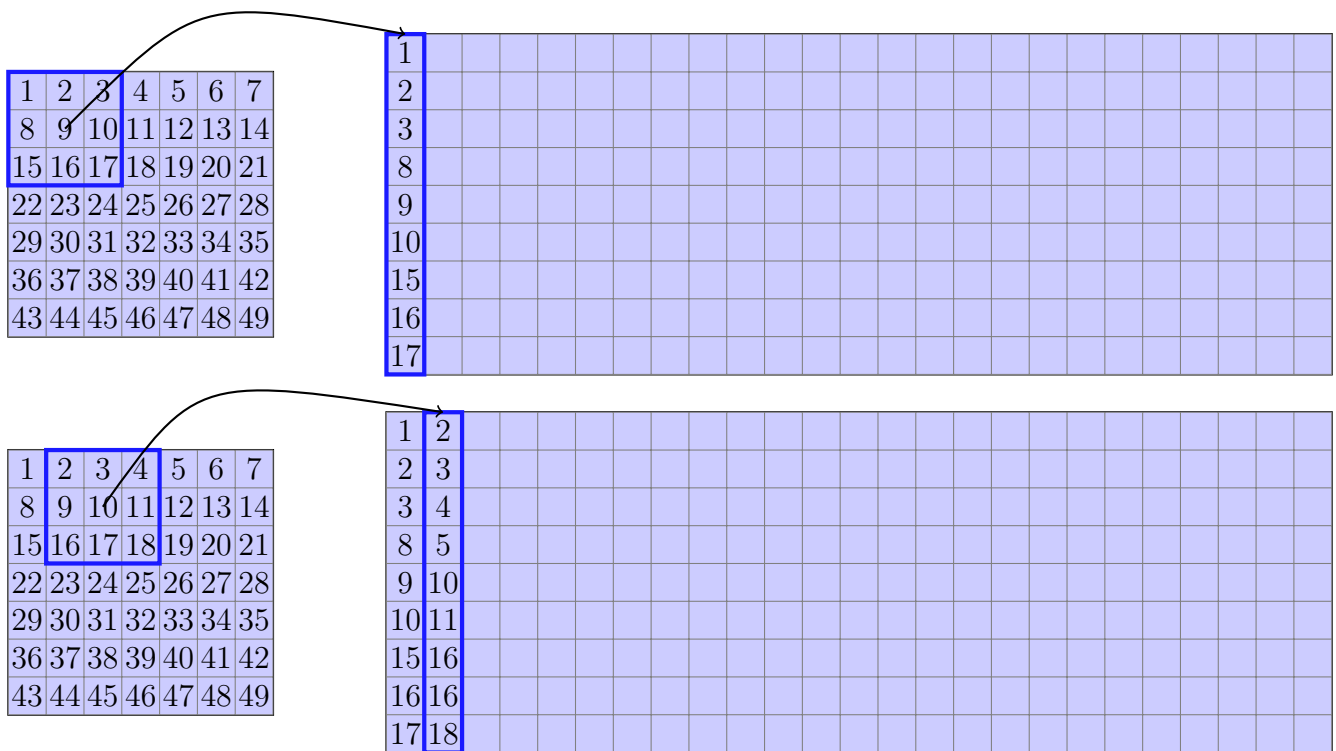


Figure 2.2: Step1: Image to Column (Im2Col) operation is to sort all values in convolutional kernel in a column

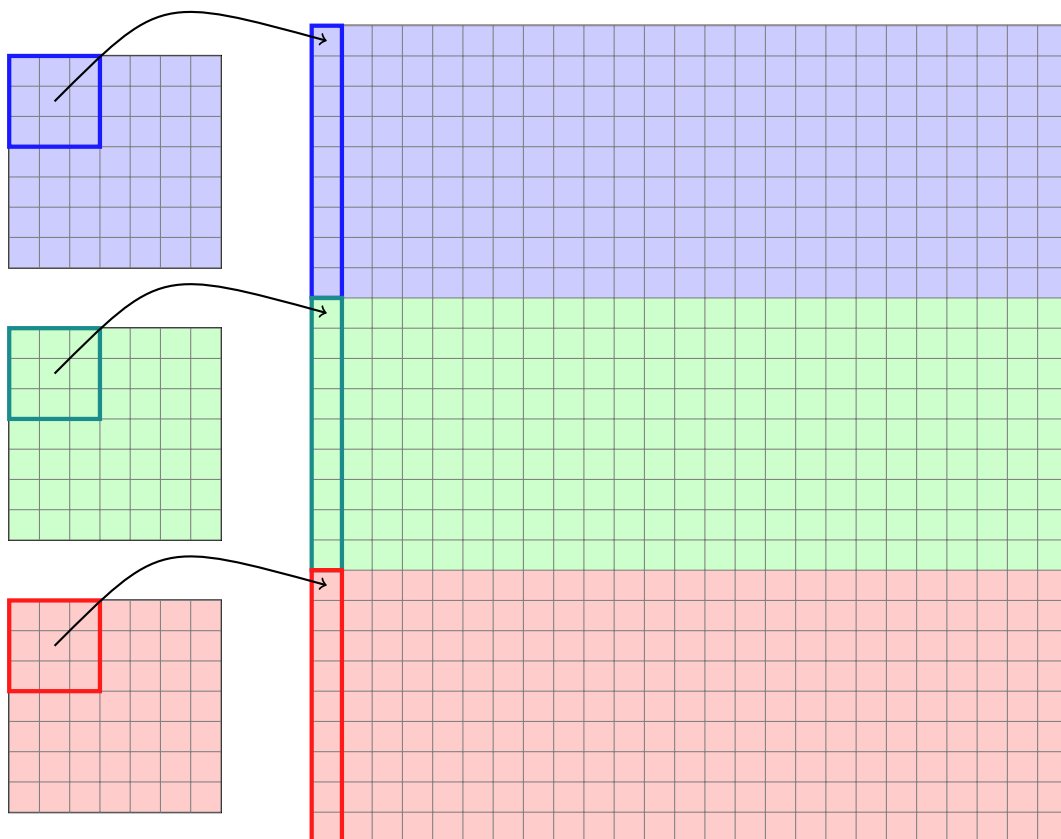


Figure 2.3: The output Im2col in 3-channels image

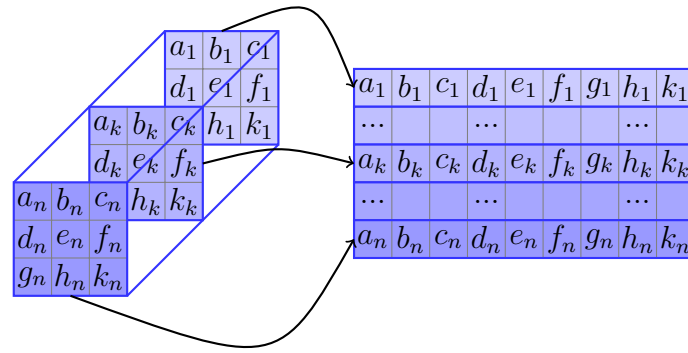


Figure 2.4: Reshape the kernel to row vector form. Given there are N kernels, then the reshaped kernels set will have the concatenated row vectors

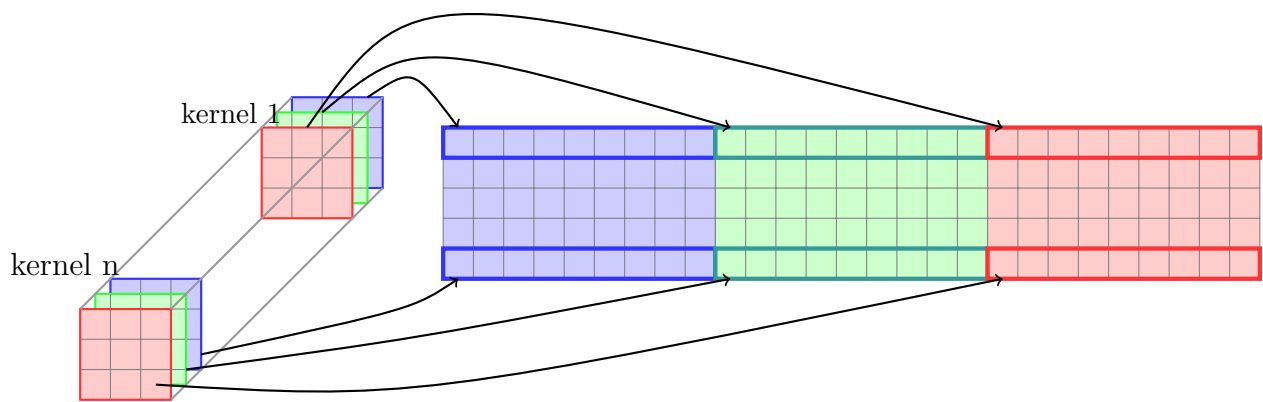


Figure 2.5: Reshape kernel in 3-channels

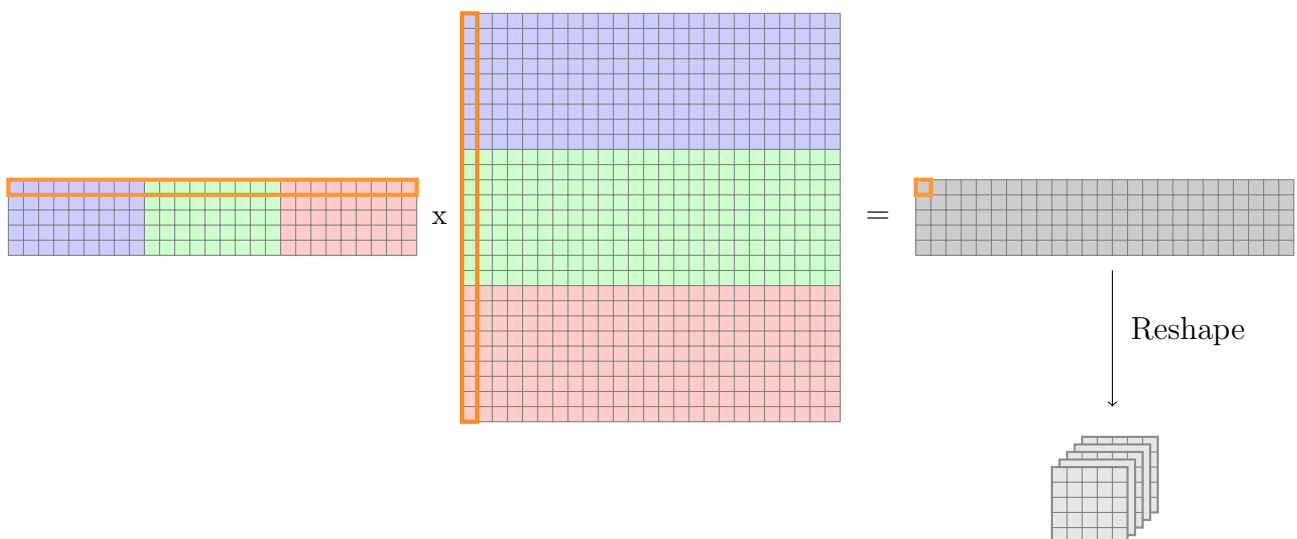


Figure 2.6: Matrix multiplication operation between reshaped kernels and Im2Col output. The output of this operation will be reshaped to the expected form as the same as after convolution.

Chapter 3

Configure deep learning model

3.1 Introduce common datatype in deep learning

In programming, variables are stored under a certain capacity that requires consistent precision in order to avoid loss in actions. For example, for positive integer computation, we would like to use at least *uint8* type to store the output with maximum value 255 but we can also use *int16*, *int32*, *int64* to store larger maximum values. However, when it comes to real values (in mathematical manner), rounding loss is unavoidable since irrational number doesn't have its end so we must accept a threshold that allow the computational output doesn't mismatch with expected result at a certain degree. In machine learning, most of dataset type will be casted under floating point with either following options: *float32*, *float16*, *bfloat16*.

Type *float32* also known as *single precision* or *full precision* can represent a wide range of real values with 1 sign bit, 8 exponent bits and 23 mantissa bits. Since many hardwares nowadays consider *float32* as a standard to measure processing power, it has been used widely in not only deep learning but also in computer graphics as well. Two disadvantages that need to be mentioned for *float32* are (1) high memory footprint and (2) slow due to more operations.

Type *float16* also known as *half precision* is a clipping version of *float32* since it can represent well floating point numbers within range $[-65504, 65504]$ decimal precision estimated around 3 digits after floating point. Compared to *float32*, this type only has 5 exponent bits and 11 mantissa bits, making it less expressive for significant values. Normally, *float16* is preferred in edge devices when launching deep learning due to its efficiency and memory-saving.

Type *bfloat16* is a special idea that was developed by Google Brain team in an effort to accelerate matrix multiplication operations on cloud TPUs platform [2]. In fact, *bfloat16* compromises 2 factors (1) the total number of bits used of *float16* and (2) exponent bits of *float32*, e.g 8 over 16 bits are obtained for representing exponent values. According to their article [2], neural network is more sensitive with exponent than mantissa so it's better to prioritize exponent bits when training deep learning.

	sign bit	exponent bits	mantissa bits	decimal precision
<i>float32</i>	1	8	23	≈ 3
<i>float16</i>	1	5	11	≈ 7
<i>bfloat16</i>	1	8	7	≈ 7

Table 3.1: Summary of common types used in deep learning training & inference

```

1 import numpy as np
2
3 a = np.array(2.666675, dtype=np.float32)
4 print(a)
5 #-> 2.666675
6
7 b = a.astype(np.float16)
8 print(b)
9 #-> 2.666

```

Listing 3.1: Code illustrates float16 and float32 type

3.2 Weight quantization

When a deep learning model is saved, its size will depend on the number of parameters and the data types to store those values. Regarding state-of-art image models and LLMs nowadays, they possess at least hundred millions to billions of params in order to perform well in specific tasks. Therefore, in the case we want to downgrade the size of the model just enough to be launched on local devices instead of calling APIs provided by any datacenter service, it is preferable to modify the data types that used to store params values through a phase called "quantization". According to [1], there are 2 quantization families:

- **Post-training quantization:** convert the numerical results of training process to a lower precision type. For example, the original LLama v3.2 11B model costs about 2 Gb when using full precision but after post-training quantization with half precision, the size of the model can be reduced by half which enough to run on edge device like Jetson Nano.
- **Quantization-aware training:** convert the weights and bias during pre-training or fine-tuning. For example: during forward pass the computation can adopt float16 type to reduce the precision but in backward pass and weight update, the outputs are stored under float32 to preserve the precision.

3.2.1 Post-training quantization

Nowadays, many companies and organizations decided to public not only the weights but also the model source codes for reseachers, users if one may interests with launching/finetuning the state-of-art models on their local devices. However, these individuals are less likely to possess thousands of A100 GPUs running parallely in their house, or at least to generate text for local GPT with 1000 tokens per minute so except your goal is for research or business, it is inefficient to run these models locally. As a matter of fact, when the absolute perfomance in terms of speed and accuracy of the model is not the priority, we can try to make the model run at a lower precision setting with approximate quality as original one by quantizing the weight datatype. From *float32* precision, we can convert the model to *float16* without severely damage the pretrained weights but it isn't really shrink the size significantly. For a higher degree of quantization, we consider **8 bit quantization** and **4 bit quantization** with 2 techniques: absolute maximum and zero-point quantization.

Absolute maximum 8 bit quantizer

$$X_{quant} = round\left(\frac{127}{max[X]} * X\right) \quad (3.1)$$

where 127 is the maximum value of *int8* type. The reverse equation from *int8* to *float16*, *float32* is:

$$X_{dequant} = \frac{\max[X]}{127} * X_{quant} \quad (3.2)$$

Explain: we calculate the scale of maximum value of *int8* type over the maximum entries of absolute weight, then multiplies this scale to all the weight entries.

```

1 import torch
2
3 weight = torch.tensor([0.1234, 0.6990, -0.7890, 0.0012], dtype=torch.float32)
4 print(weight)
5 #->tensor([0.1234, 0.6990, 0.7890, 0.0012])
6
7 scale = 127/torch.max(torch.abs(weight))
8 quantized_weight = (weight*scale).round().to(torch.int8)
9 print(quantized_weight)
10 #->tensor([ 20, 113, -127, 0], dtype=torch.int8)
11
12 dequantized_weight = quantized_weight/scale
13 print(dequantized_weight)
14 #->tensor([ 0.1243, 0.7020, -0.7890, 0.0000])>

```

Listing 3.2: Code for absolute maximum quantization

Zero-point quantization

First we need to calculate the scale due to the assumption that the input value range distribute asymetrically. Then we find the zero point that offset the input to $[-128, 127]$ range of *int8*.

$$scale = \frac{255}{\max(X) - \min(X)} \quad (3.3)$$

$$zeropoint = -\text{round}(scale * \min(X)) - 128 \quad (3.4)$$

The quantized outputs will be computed by multiplying input with the scale and add the zero point.

$$X_{quant} = \text{round}(scale * X + zeropoint) \quad (3.5)$$

$$X_{dequant} = \frac{X_{quant} - zeropoint}{scale} \quad (3.6)$$

```

1 import torch
2
3 weight = torch.tensor([0.1234, 0.6990, -0.7890, 0.0012], dtype=torch.float32)
4 print(weight)
5 #->tensor([ 0.1234, 0.6990, -0.7890, 0.0012])
6
7 scale = 255/(torch.max(weight) - torch.min(weight))
8 zeropoint = -(scale*torch.min(weight)).round() - 128
9
10 quantized_weight = (scale*weight + zeropoint).round().to(torch.int8)
11 print(quantized_weight)
12 #->tensor([ 28, 127, -128, 7], dtype=torch.int8)
13
14 dequantized_weight = (quantized_weight - zeropoint)/ scale
15 print(dequantized_weight)
16 #->tensor([ 0.1225, 0.7002, -0.7878, 0.0000])

```

Listing 3.3: Code for zero point quantization

3.2.2 Quantization-aware training