Batch Normalization and Layer Normalization

Ruixin Guo

Department of Computer Science
Kent State University

September 29, 2023

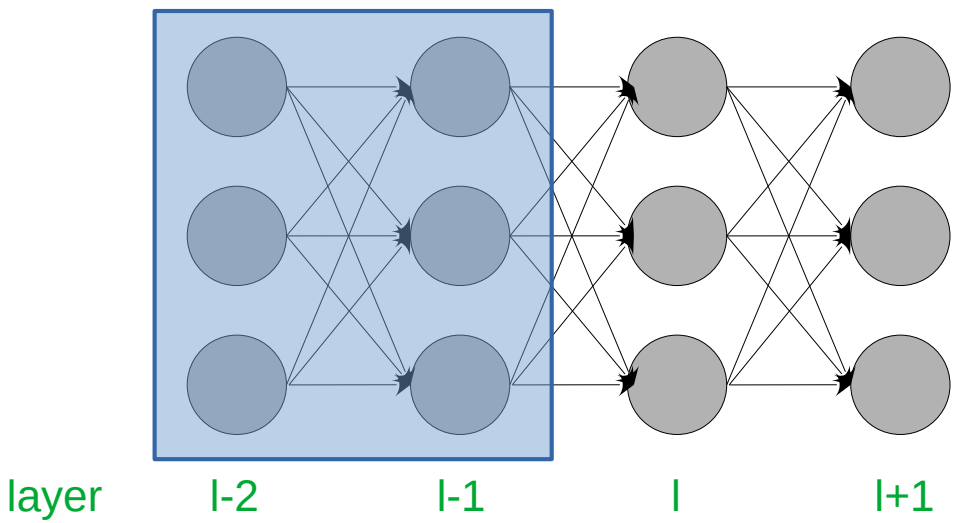
Contents

Contents

Covariate Shift

Batch Normalization (BN) is proposed by Ioffe and Szegedy in 2015 [1]. It aims to solve the Internal Covariate Shift (ICS) problem. BN can significantly accelerate the convergence.

Let's consider the input of the l-th layer in a neural network, which is the output of the $l-1$-th layer. Covariate Shift means in each iteration, the distribution of input of the l-th layer will change, caused by the changing of parameters of layers 1 to $l-1$.



[1] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating deep network training by reducing internal covariate shift. International Conference on Machine Learning, pages 448–456. pmlr, 2015.

Internal Covariate Shift

The Internal Covariate Shift means the Covariate Shift happened in a neuron node. Let u be the input vector, z be the output vector, W be the weight matrix, b be the bias vector, $g(x)$ be the activation function. A neuron performs

$$z = g(Wu + b)$$

Suppose g is a sigmoid function, i.e., $g(x) = \frac{1}{1+e^{-x}}$. Then $g'(x) = g(x)(1 - g(x))$. We call x an activation of g.

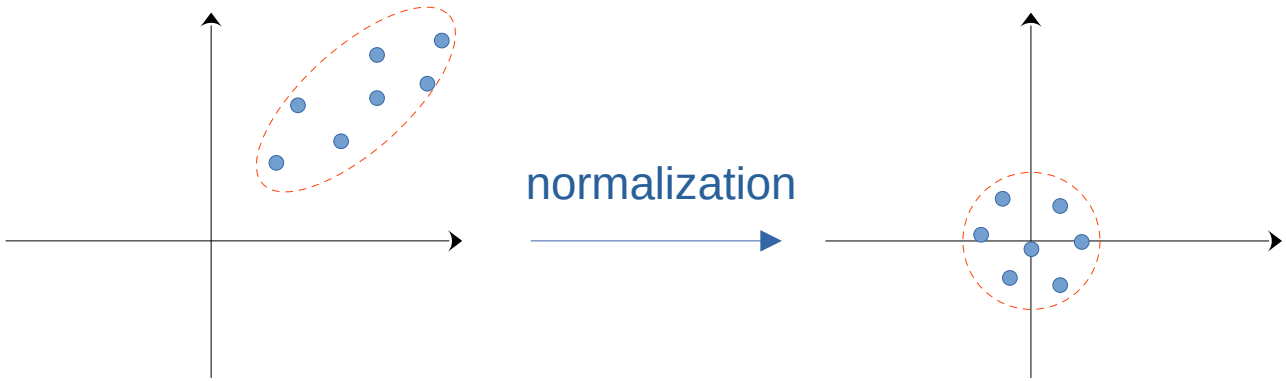
Since $x = Wu + b$, the changes in W and b will make x shift. Moreover, when $|x|$ increases, $g(x)$ will either be too large or too small, and both cases will make $g'(x)$ close to 0. This will slow down the convergence of training.

One way to solve this problem is to normalize x. x will be close to 0 after normalization, which will not make $g'(x)$ trivial.

Normalization

Yann LeCun et al [2] showed that normalizing input can accelerate the convergence of training the neural network. The normalization is to make all the input sample vectors having 0 mean and all dimensions of these vectors have the same variance.

Since each neuron has an activation function, and each activation function has an input, we can apply normalization to all these inputs. The layers inside a neuron now becomes: Linear Transformation $\rightarrow$ Normalization $\rightarrow$ Activation Function.



Let $x^{(k)}$ be an input of the activation function of the k-th layer. The normalization of $x^{(k)}$ can be implemented by

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Here $\mathrm{E}[x^{(k)}]$ and $\mathrm{Var}[x^{(k)}]$ are the expectation and variance of all samples.

[2] Yann A LeCun, Leon Bottou, Genevieve B Orr, and Klaus Robert Muller. Efficient backprop. Tricks of the Trade, page 9, 1998.

Batch Normalization

However, calculating $\mathrm{E}[x^{(k)}]$ and $\mathrm{Var}[x^{(k)}]$ are impractical, because they are different in each iteration, and we need to recalculate them over and over again. Instead of calculating the mean and variance of all samples, we can calculate the mean and variance of the samples in a mini-batch instead.

Algorithm (Batch Normalization): Given a mini-batch of feature vectors $\mathcal{B} = \{x_1, x_2, ..., x_m\}$ where $x_i \in \mathbb{R}^d$. The output is a mini-batch of normalized feature vectors $\mathcal{B}_N = \{h_1, h_2, ..., h_m\}$ where each h_i is obtained as follows:

$$\mu_{\mathcal{B}} = \frac{1}{m} \sum_{i=1}^{m} x_i$$

$$\sigma_{\mathcal{B}}^2 = \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2$$

$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$$

$$h_i = \gamma \cdot \hat{x}_i + \beta$$

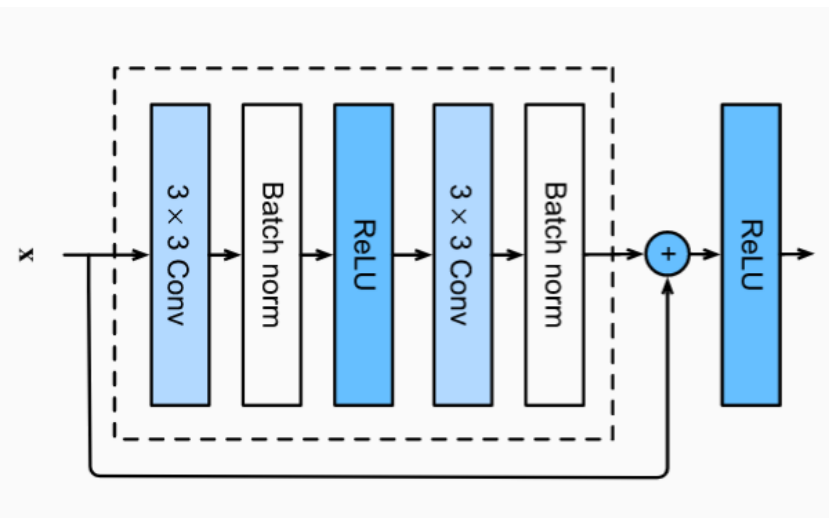
where $\gamma \in \mathbb{R}^d, \beta \in \mathbb{R}^d$ are learned parameters and are shared by all samples in the mini-batch. $\mathcal{B}_N$ will have β mean and γ^2 variance. The square, division, square root and dot product are all element-wise.

Batch Normalization

The reason that Ioffe and Szegedy feeds h_i instead of $\hat{x}_i$ to the activation is: the normalization would change the activation pattern. For example, it would put all the samples in the linear region of the sigmoid function and makes the sigmoid function lose non-linearity.

The learned parameters β and γ are used to make sure the transformation from x_i to h_i can represent identity transform. When $\beta = \mu_{\mathcal{B}}$ and $\gamma = \sigma_{\mathcal{B}}$, the transformation will be identical.

Finally, let's see an example about how Batch Normalization is applied in neural network. Below is the structure of a residual block in ResNet. We can find that Batch Normalization is applied before every ReLU activation.



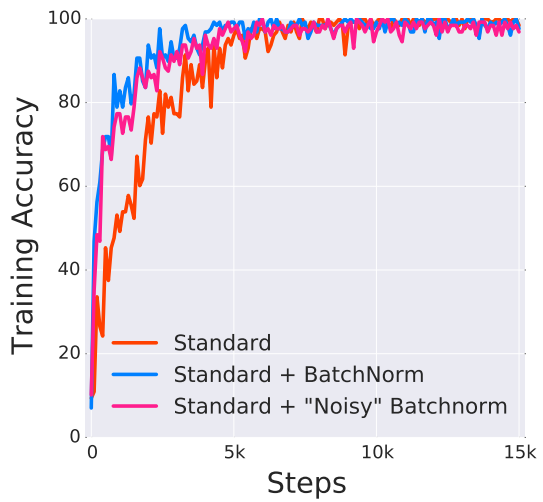
Contents

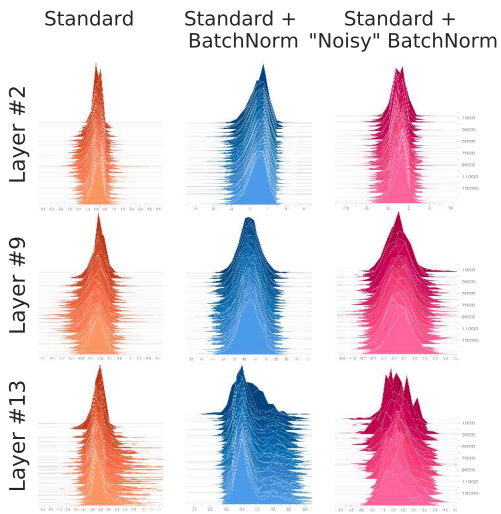
Does Batch Normalization Benefit from Internal Covariate Shift?

The exact reason for Batch Normalization's effectiveness is still not well understood. One popular belief is that Batch Normalization changes the input distribution during training to reduce the Internal Covariate Shift.

Santurkar et al [3] demonstrated that the effectiveness of Batch Normalization stems from not reducing the Internal Covariate Shift but making the optimization landscape significantly smoother.

In the first experiment, Santurkar et al trained the network with random noise injected after BatchNorm layers. The results showed that the "noisy" BarchNorm network has less stable distributions, but still perform better in training.





[3] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does Batch Normalization help Optimization? Advances in Neural Information Processing Systems, 31, 2018.

Batch Normalization makes Landscape Smoother

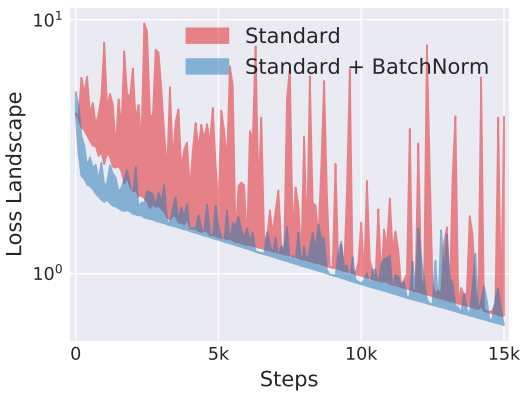
Suppose the Empirical Risk Function $R^{\text{emp}}(\theta)$ of a neural network is L-Lipschitz and β-smooth (see Appendix). Adding Batch Normalization will make both $R^{\text{emp}}(\theta)$ and β smaller, thus make the landscape of f smoother and easier to be trained.

The second experiment analyze optimization landscape of VGG network. Blue is with Batch Normalization and Red is without Batch Normalization. The VGG network updates the parameter θ by SGD: $\theta_t = \theta_{t-1} - \alpha_t \nabla_\theta R^{\text{emp}}(x_i, y_i; \theta)$. At each step t, taking the point θ_t and $\theta_{t,\eta} = \theta_t - \eta \nabla_\theta R^{\text{emp}}(x_i, y_i; \theta)$ where η is varied in $[0, 0.4]$. Here $\theta_{t,\eta}$s are the points on the trajectory that θ_t moves in the direction of its gradient[4].

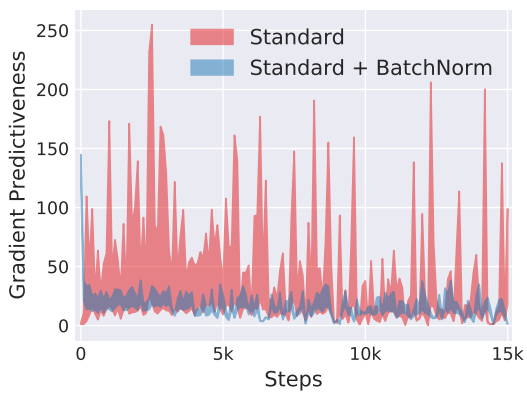
[4] If taking random direction instead of the gradient of θ_t here, the experiment results will be similar, as the authors claimed.

Batch Normalization makes Landscape Smoother

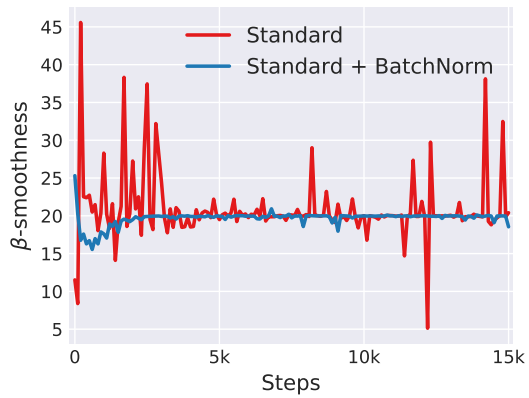
The results are shown as follows. Fig (a) and (b) show the variation (shaded region) of the Empirical Risk $|R^{\text{emp}}(\theta_t) - R^{\text{emp}}(\theta_{t,\eta})|$ and L_2 norm of the gradients $\|\nabla R^{\text{emp}}(\theta_t) - \nabla R^{\text{emp}}(\theta_{t,\eta})\|_2$ with η. Fig (c) shows the effective β-smoothness where $\beta = \max_\eta \|\nabla R^{\text{emp}}(\theta_t) - \nabla R^{\text{emp}}(\theta_{t,\eta})\|_2 / \|\theta_t - \theta_{t,\eta}\|_2$.



(a) loss landscape



(b) gradient predictiveness



(c) "effective" β-smoothness

The results show that the Batch-Normalized VGG has smaller variation of Empirical Risk, gradient differences and β. This means the optimization landscape is smoother in both L-Lipschitz and β-Smooth.

Contents

Word2Vec

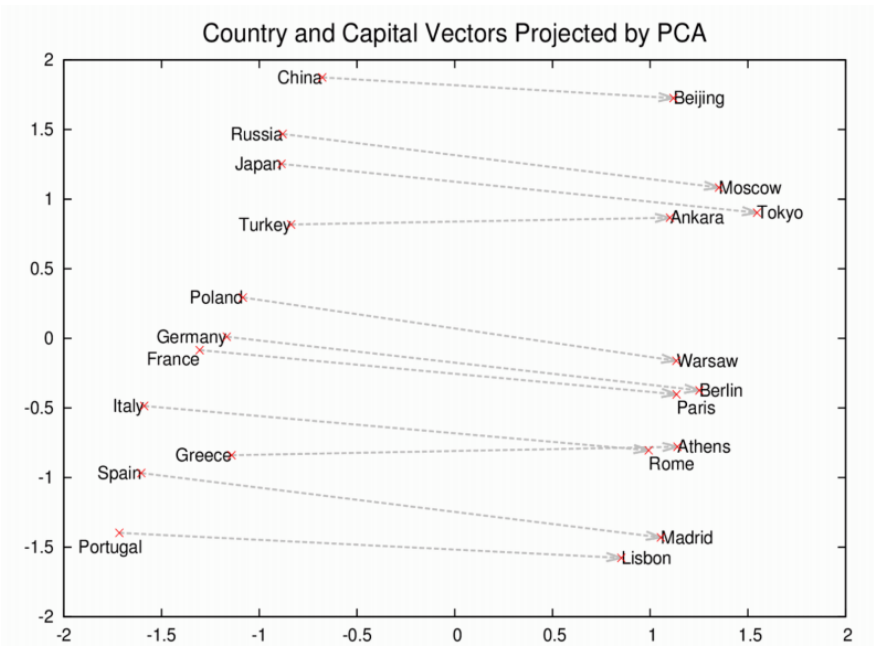
Word2Vec[5] is a method to generate representations for words. It maps each word to an embedding vector.

The embeddings are created based on the similarity of words. For any two words w1 and w2, their similarity is defined as

$$\cos(vec(\text{w1}), vec(\text{w2})) = \frac{\langle vec(\text{w1}), vec(\text{w2}) \rangle}{\|vec(\text{w1})\| \|vec(\text{w2})\|}$$

For example, the word "king" is closer to "queen" than "zebra", so $\cos(vec(\text{king}), vec(\text{queen}))$ is closer to 1 while $\cos(vec(\text{king}), vec(\text{zebra}))$ is closer to 0.

Researchers found that the similarity of word representations has simple syntactic regularities. For example, using simple algebraic operation, the vector $vec(\text{"Beijing"}) - vec(\text{"China"}) + vec(\text{"Russia"})$ will have large cosine similarity with $vec(\text{"Moscow"})$.



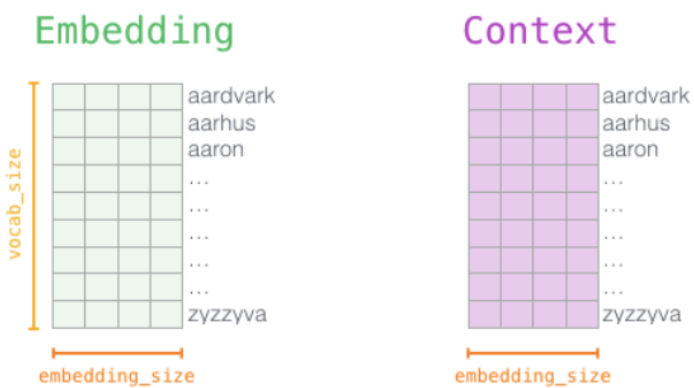
[5] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781, 2013.

Word2Vec Training Process

The word embeddings are obtained in a machine learning way[6]:

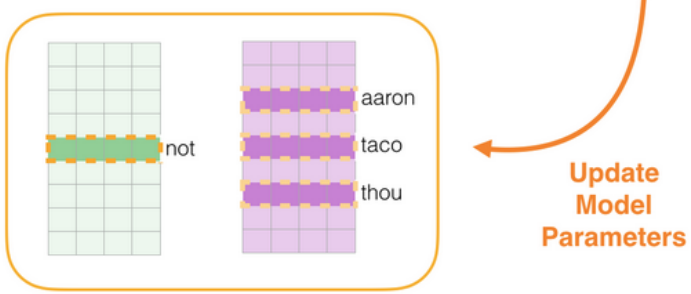
1. Create two matrices: the Embedding Matrix $E \in \mathbb{R}^{m \times n}$ and the Context Matrix $C \in \mathbb{R}^{m \times n}$, where m is the number of words and n is the size of the feature vector. Initialize both E and C with random values.

2. Each time pick a word in E, and one positive samples and few negative samples from C, calculate the loss, and update both E and C.



Initialize E and C

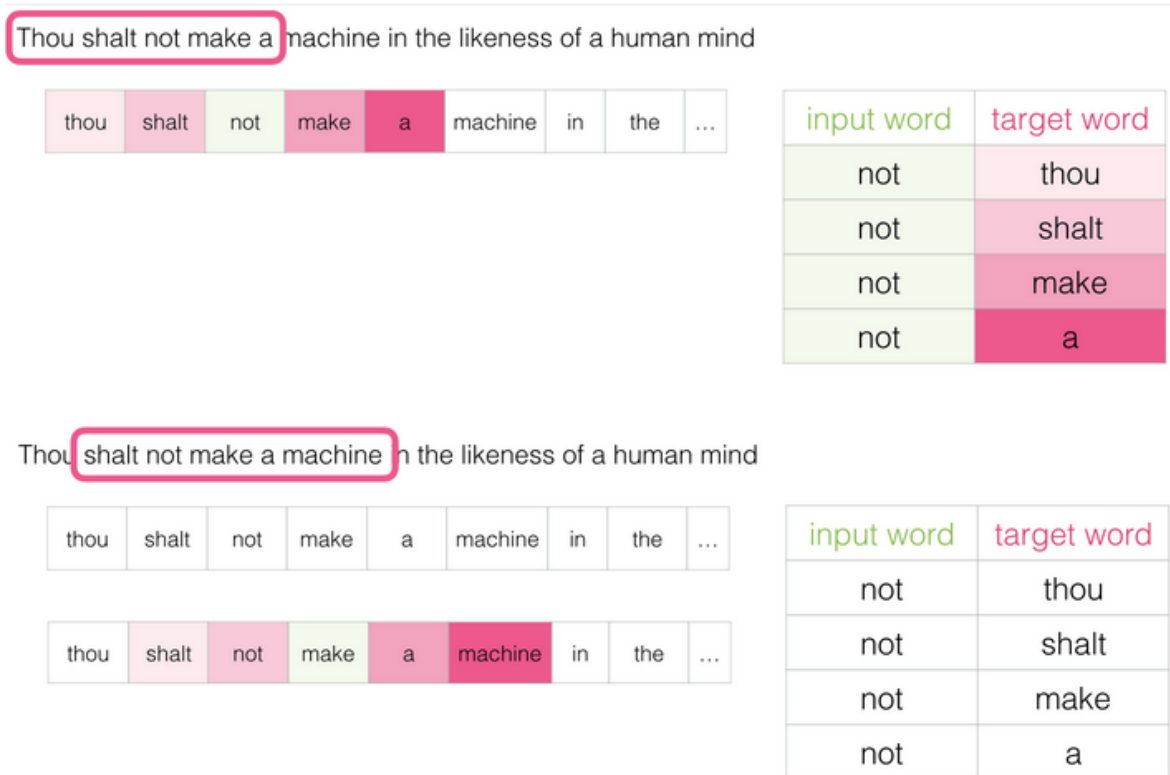
input word	output word	target	input • output	sigmoid()	Error
not	thou	1	0.2	0.55	0.45
not	aaron	0	-1.11	0.25	-0.25
not	taco	0	0.74	0.68	-0.68



Update E and C

[6] https://jalammar.github.io/illustrated-word2vec/

The way to find positive samples is using a sliding window to scan through the text. The word at the center of the sliding window is the word chosen in E and its neighbors are positive samples in C. Other words not in the sliding window are the negative samples.



3. When the training is finished, discard C, and use E as the embeddings for words.

Recurrent Neural Network (RNN)

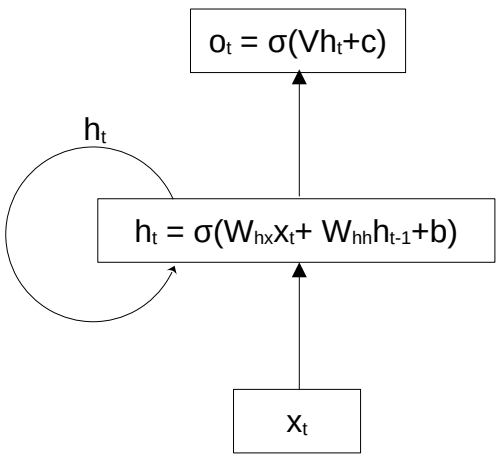
An RNN is a neural network that the output from the previous step is fed as input to the current step.

Let x_t and h_t be the input and output of a neuron at step t. Let W_{xh} be the weight matrix, b be the bias vector and σ be the activation function. A general neuron will calculate

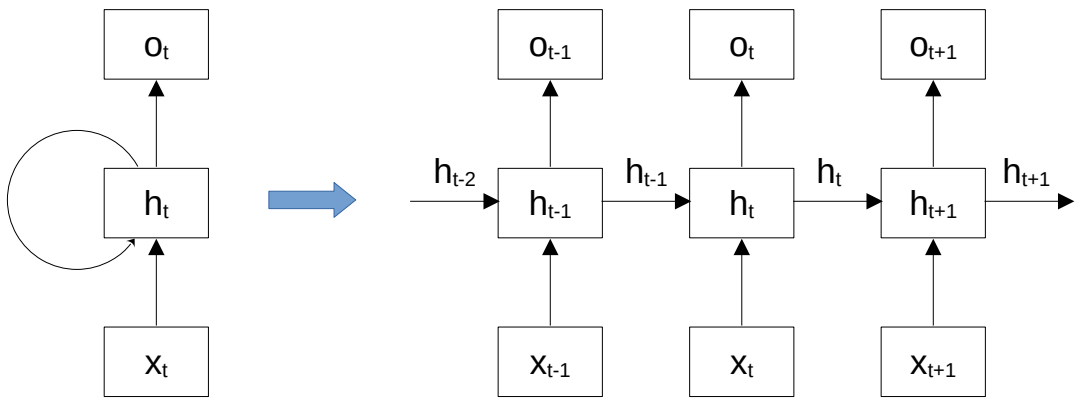
$$h_t = \sigma(W_{xh}x_t + b)$$

Let W_{hh} be the weight matrix of the hidden state. An recurrent neuron will calculate

$$h_t = \sigma(W_{xh}x_t + W_{hh}h_{t-1} + b)$$



A single RNN with one neuron in the hidden layer



Unfolding the RNN

The Mini-batch in RNN

In most cases, the input of RNN is a mini-batch of samples. Each sample is a sentence, each sentence contains multiple words. The mini-batch size means the number of sentences.

The words in a sentence are processed sequentially. The i-th words in all sentences are processed in parallel ($i = 1, 2, 3, ...$). The parallelization speeds up the training of RNN.

Processed in parallel

Processed sequentially

samples

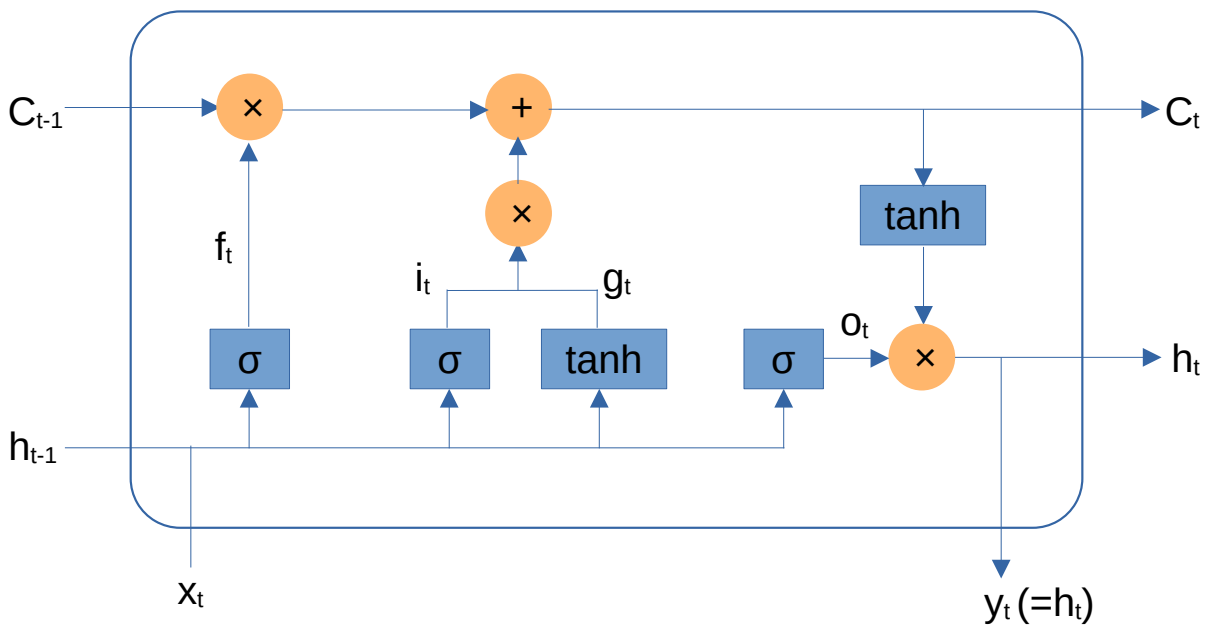
1	Alice	follows	the	White	Rabbit
2	She	found	a	beautiful	garden
3	Caterpillar	seated	on	a	mushroom
4	The	Queen	orders	his	beheading
5	Play	croquet	with	a	Flamingo

Suppose the number of words in a mini-batch is fixed. The longer the sentences are, the less sentences will be contained in the mini-batch.

Long Short Term Memory

The Long Short Term Memory (LSTM) is a variant of RNN. Comparing with typical RNN, LSTM has two recurrent branches: the hidden state h and the cell state C.

The cell state maintain the memory of all past knowledge. In each step t, the cell state is changed by forgetting some knowledge through the forget gate $\times$, and obtain some new knowledge through the input gate $+$.



Long Short Term Memory

In each step t, the input x_t and hidden state h_{t-1} is concatenated and sent do different gates for different purposes:

- Forget Gate f: Through the sigmoid activator it generates an output vector with each element between 0 and 1. 0 means forget everything and 1 means remember everything. $f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$.
- Input Gate i: Determines what kind of information that should be remembered. $i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$.
- Input Modulation Gate g: Normalize i_t such that the normalized i_t will have zero mean. $g_t = \tanh(W_g \cdot [h_{t-1}, x_t] + b_g)$.
- Output Gate o: $o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$.

The gate $\times$ means element-wise multiplication and the gate $+$ means the element-wise addition. First, the cell state is updated by

$$C_t = C_{t-1} \times f_t + i_t \times g_t$$

Then we update the hidden state with C_t and o_t,

$$h_t = o_t \times \tanh(C_t)$$

Finally, we take $y_t = h_t$ as the output.

Attention Model

Attention Model[7] is a neural network model. Recent years Attention Model has replaced RNN to be the foundation of Large Language Models. However, the principle of Attention Model, like many other deep learning methods, is still not clear yet.

Here is the main idea about Attention Model[8]. Consider we have a sentence:

Tom is a cat and he wants to catch Jerry.

Here we know that "Tom", "cat", "he" represents the same object in this sentence. So their word embeddings should have high similarity with each other. The original word embeddings "Tom", "cat", "he" may not have high similarity with each other, so we need to regenerate the word embeddings for this sentence.

Self-Attention is a way to regenerate the word embeddings. Suppose we have a sentence of m words, each word corresponds to an embedding $e_i \in \mathbb{R}^d$. So the embeddings that represent this sentence is $\{e_1, e_2, ..., e_m\}$.

[7] Ashish Vaswani, Noam Shazeer, Niki Parmar et al. Attention is all you need. Advances in neural information processing systems, 30, 2017.

[8] https://towardsdatascience.com/all-you-need-to-know-about-attention-and-transformers-in-depth-understanding-part-1-552f0b41d021

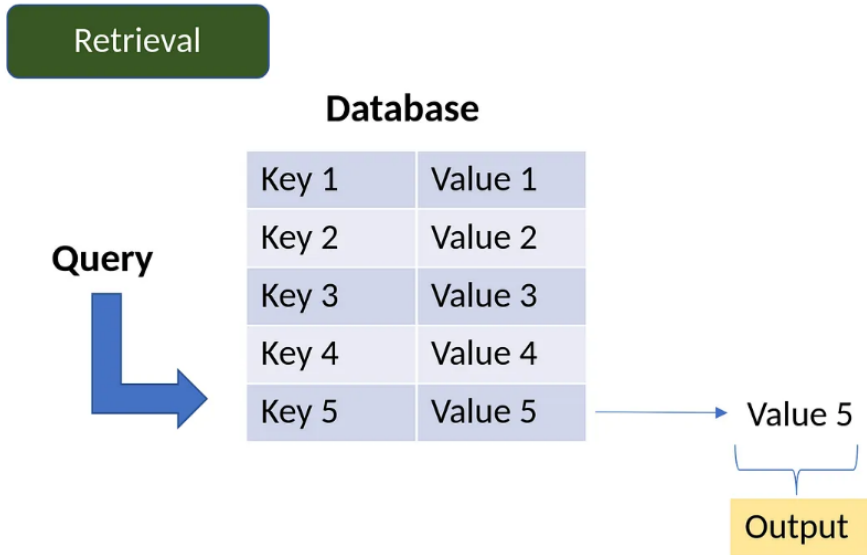
Consider that we want to regenerate the embedding of e_1. First, we generate a set of weights by making inner products of e_1 with all embeddings: $e_1^T e_1, e_1^T e_2, ..., e_1^T e_m$. Then, we normalize these weights such that $\alpha_{1i} = \text{Norm}(e_1^T e_i)$ and $\sum_{i=1}^m \alpha_{1i} = 1$. Finally, we calculate the weighted embedding $y_1 = \sum_{i=1}^m \alpha_{1i} e_i$. So y_1 is the regenerated embedding of e_1.

y_1 will contain the relationship information of e_1 with all other words. If e_1 has larger similarity with e_j, then $e_1^T e_j$ will be large, and α_{1j} will be large, thus gives a large weight for e_j.

However, this embedding regeneration method has no trainable parameters. Here we use three trainable matrices $M_q, M_k, M_v \in \mathbb{R}^{d \times d}$ such that for each e_i, we calculate $q_i = M_q e_i, k_i = M_k e_i, v_i = M_v e_i$. We know that $q_i, k_i, v_i \in \mathbb{R}^d$. So the generation of y_i becomes

$$y_i = \sum_{j=1}^{m} \text{Norm}(q_i^T k_j) v_j \tag{1}$$

Here we call q_i the query, k_js the keys and v_js the values. This process is like a retrieval in database.



We can write Eq (1) in matrix form. Let $Q = [q_1, ..., q_m]^T \in \mathbb{R}^{m \times d}$, $K = [k_1, ..., k_m]^T \in \mathbb{R}^{m \times d}$, $V = [v_1, ..., v_m]^T \in \mathbb{R}^{m \times d}$, $Y = [y_1, ..., y_m]^T \in \mathbb{R}^{m \times d}$. Use the row-wise softmax as the normalization function, then we have the Attention function

$$Y = \text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V \tag{2}$$

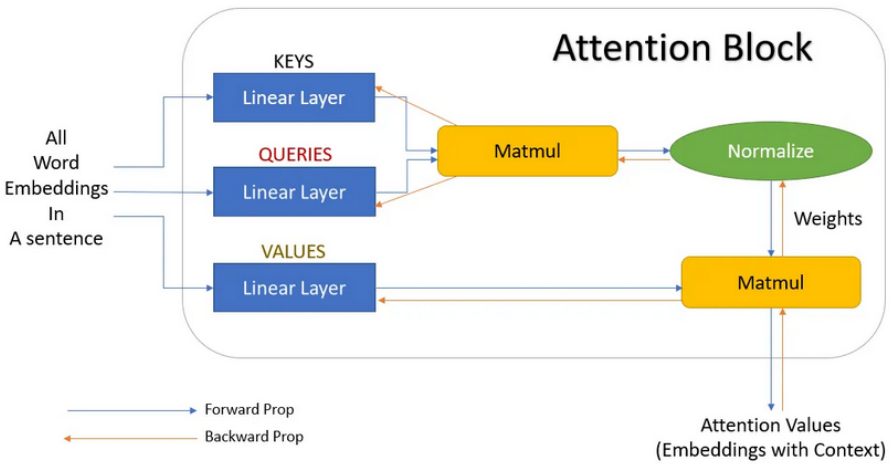
Note that the element of row i column j in QK^T is $q_i^T k_j$. The value of $q_i^T k_j$ is likely to grow with d. The scaler $\frac{1}{\sqrt{d}}$ is used above to prevent $q_i^T k_j$ from being too large and goes to the small gradient region of softmax.

Multi-head Attention

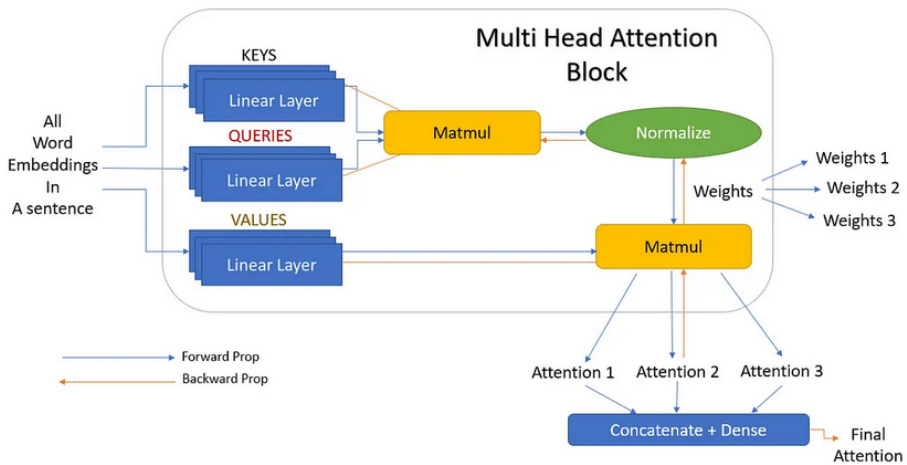
Eq (2) is called an Attention Block. The authors found it beneficial to replicate the Attention Block multiple times.

- Each Attention Block takes the same input $e_1, ..., e_m$ but with different linear transformation matrix M_q, M_k, M_v.
- The outputs of these Attention Blocks are concatenated and pass though a linear to get a final output, which is like a weighted sum of all the outputs where the weights are learned through training.

This structure is called Multi-Head Attention Block.



Attention Block

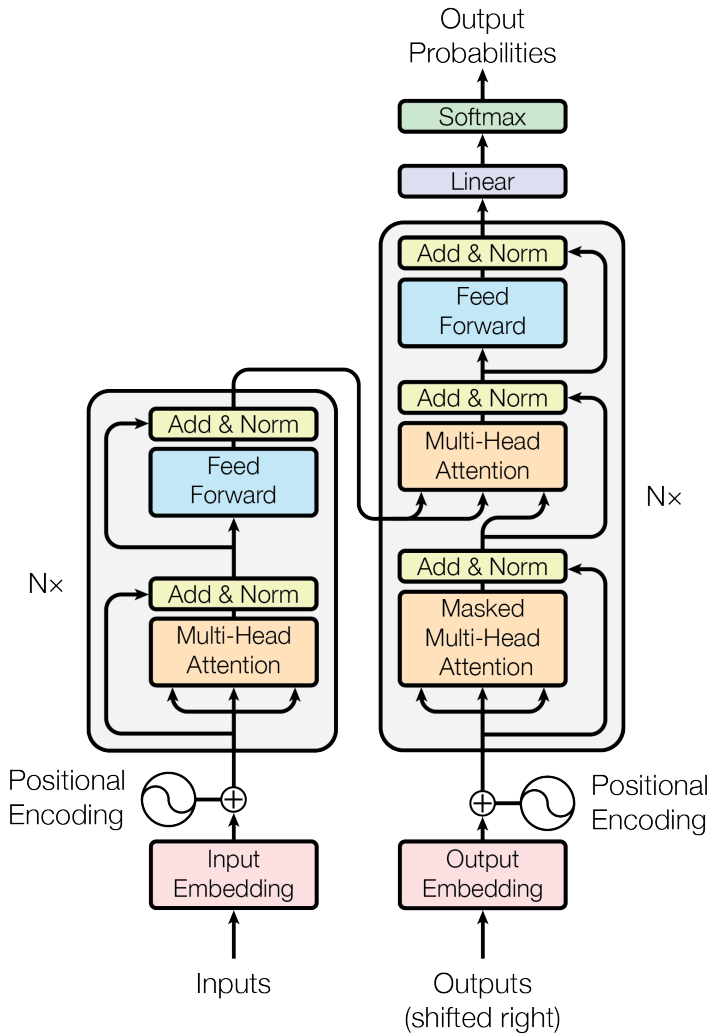


Multi-Head Attention Block

Transformer

Transformer is the basic structure of Large Language Models like BERT, GPT and Llama. It includes encoder and decoder part. Each part is composed of a stack of N layers. Each layer contains Multi-Head Attention Block. The Norm in each layer is Layer Normalization.

The Transformer works in this way [8]. Suppose we want to translate the French *Je suis étudiant* to English *I am a student*. First, we feed *Je suis étudiant* as the Inputs of the encoder. Then, the decoder will output *I* in the Output Probabilities part. Feed *I* to the Outputs (shifted right) part, then Output Probabilities will output *am*. Feed *am* to the Outputs (shifted right) part, then Output Probabilities will output *a*... Continue this step until the decoder finishes the entire sentence.



The structure of Transformer, the left part is encoder and the right part is decoder.

[8] https://jalammar.github.io/illustrated-transformer/

Why Attention Model

There are few reasons that Attention Model is superior to RNN:

- Parallelizable: Unlike RNN that the words in a sentence must be processed sequentially, the Attention Model can process the words in a sentence in parallel. Consider Eq (2), we can parallelize it using m threads, each thread calculates Eq (1).
- Long-Range Dependency: Consider we have a long sentence and two dependent words in the sentence has a long distance. RNN can hardly catch such dependency because there are many forward-backward operations between two words, causing signal fade. Transformer can catch such dependency because the entire sentence only forward once.
- Less computation: Below is a table comparing the computation cost between RNN and Attention Model per layer. Attention Model will outperform RNN when $m < d$, which is often the case in Language Models.

Layer Type	Complexity Per layer	Sequential Operations	Maximum Path Length
Attention	$O(m^2 d)$	$O(1)$	$O(1)$
RNN	$O(md^2)$	$O(m)$	$O(m)$

m is the number of words in a sentence. d is the length of embedding vector of each word.

Contents

The Problems of Batch Normalization

Batch Normalization has two problems:
(1) The normalization depends on the mini-batch size. When the mini-batch is small or equals to 1, Batch Normalization will not be effective.
(2) Cannot be easily applied to RNN. Consider the unfolding RNN, we need to apply Batch Normalization on each step, and each step needs a pair of learned parameters γ and β. However, the number of steps required depends on the length of the input, which is unpredictable.

To fix these issues, Ba et al proposed Layer Normalization[9].

[9] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. arXiv preprint arXiv:1607.06450, 2016.

Layer Normalization

Algorithm (Layer Normalization): Suppose a Layer has N neurons. The input mini-batch of the activation function of the ith neuron is $a^i = [a_1^i, a_2^i, ..., a_m^i]$ where $a_j^i \in \mathbb{R}^d$. The Layer Normalization takes the input $a = [a^1, a^2, ..., a^N]$ and gives the output $o = [o^1, o^2, ..., o^N]$ where each o^i is obtained by:

$$\mu = \frac{1}{N}\sum_{i=1}^{N} a^i$$

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(a^i - \mu)^2$$

$$\hat{a}^i = \frac{a^i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

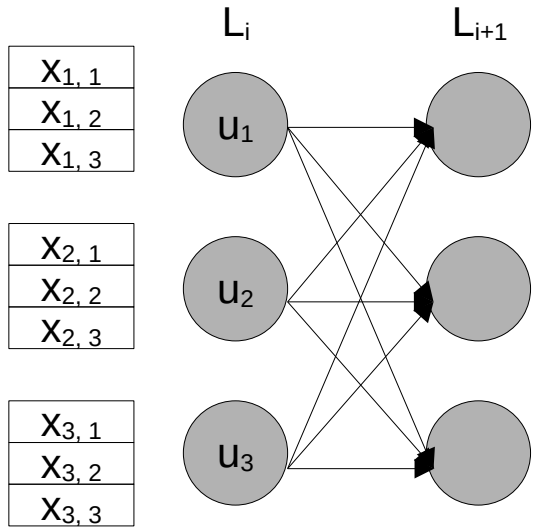
$$o^i = \text{diag}(\gamma)\hat{a}^i + \text{diag}(\beta)\mathbb{I}_{d\times m}$$

where $\gamma \in \mathbb{R}^d$ and $\beta \in \mathbb{R}^d$ are learned parameters shared by all samples in a mini-batch. Square, square root and division are element-wise.

For an RNN, $a^i = W_{xh}x_t + W_{hh}h_{t-1} + b$, where $x_t \in \mathbb{R}^d$ is the current input and $h_{t-1} \in \mathbb{R}^d$ is the hidden state. Let f be the activation function, the output of the ith neuron is $h_t = f(o^i) = f(\text{LayerNorm}(a^i))$.

Differences between Batch Normalization and Layer Normalization

Suppose we have a fully connected network as shown in the right figure. It shows two layers L_i and L_{i+1}. L_i layer has 3 neurons u_1, u_2, u_3. The minibatch of the neuron $u_j (j = 1, 2, 3)$ is $x_{j,1}, x_{j,2}, x_{j,3}$. The minibatch is the input of the activation function is each neuron.



Here is the difference in the computation of Batch Normalization and Layer Normalization:

Batch Normalization:

For each neuron $u_j (j = 1, 2, 3)$:

$$\mu = \frac{1}{3}\sum_{k=1}^{3} x_{j,k}, \quad \sigma^2 = \frac{1}{3}\sum_{k=1}^{3}(x_{j,k} - \mu)^2$$

For each sample $x_{j,k} (k = 1, 2, 3)$ in u_j:

$$\hat{x}_{j,k} = \frac{x_{j,k} - \mu}{\sigma}, \quad h_{j,k} = \gamma \cdot \hat{x}_{j,k} + \beta$$

Layer Normalization:

For the kth element $(k = 1, 2, 3)$ in the batch of each neuron u_j:

$$\mu = \frac{1}{3}\sum_{j=1}^{3} x_{j,k}, \quad \sigma^2 = \frac{1}{3}\sum_{j=1}^{3}(x_{j,k} - \mu)^2$$

$$\hat{x}_{j,k} = \frac{x_{j,k} - \mu}{\sigma}$$

For each sample $x_{j,k} (k = 1, 2, 3)$ in u_j:

$$h_{j,k} = \gamma \cdot \hat{x}_{j,k} + \beta$$

The Advantages of Layer Normalization

Layer Normalization does not depend on the batch size. This means it can be used when the batch size is small or even when the batch size is 1.

Here are some cases that Layer Normalization can be applied:

(1) In online learning, the batch size is 1 because the samples come in a sequence.

(2) When we want to process very long sequences but the capacity of memory is limited, to store the data, the mini-batch size will be small.

Appendix: Lipschitz Continuous and Lipschitz Smooth

Lipschitz Continuous: A differentiable function $f : \mathbb{R}^d \to \mathbb{R}$ is said to be Lipschitz Continuous if there exists a constant $L \geq 0$ such that for any $x, y \in \mathbb{R}^d$,

$$|f(y) - f(x)| \leq L\|y - x\|_2$$

We also say that f is L-Lipschitz.

Lemma: f is L-Lipschitz if only if $\sup \|\nabla f(x)\|_2 \leq L$.
Proof:
$\Longleftarrow$: Let $g(t) = x + t(y - x)$ where $t \in [0, 1]$ such that $g(0) = x$ and $g(1) = y$. Then

$$\begin{aligned}
|f(y) - f(x)| &= |f(g(1)) - f(g(0))| \\
&= |\int_0^1 \frac{df}{dg}\frac{dg}{dt}dt| \qquad \text{[Chain Rule]} \\
&= |\int_0^1 \nabla_g f(g(t))^T (y - x)dt| \leq \int_0^1 |\nabla_g f(g(t))^T (y - x)|dt \\
&\leq \int_0^1 \|\nabla_g f(g(t))\|_2 \|y - x\|_2 dt \qquad \text{[Cauchy]} \\
&= \|y - x\|_2 \int_0^1 \|\nabla_g f(g(t))\|_2 dt
\end{aligned}$$

Appendix: Lipschitz Continuous and Lipschitz Smooth

By the Mean value theorem, there exist a $\xi \in [0, 1]$ such that

$$\int_0^1 \|\nabla_g f(g(t))\|_2 dt = \|\nabla_g f(g(\xi))\|_2 \int_0^1 dt = \|\nabla_g f(g(\xi))\|_2$$

Since for any $g(\xi)$, $\|\nabla_g f(g(\xi))\|_2 \leq L$, we have

$$|f(y) - f(x)| \leq L\|y - x\|_2$$

$\Longrightarrow$: Let v be a unit vector and let $y = x + hv$ where $h \in \mathbb{R}$, then for any x, y

$$\lim_{y \to x} \frac{|f(y) - f(x)|}{\|y - x\|_2} = \lim_{h \to 0} \frac{|f(x + hv) - f(x)|}{\|hv\|_2} = \left| \lim_{h \to 0} \frac{f(x + hv) - f(x)}{h} \right| \leq L$$

Remember that

$$\lim_{h \to 0} \frac{f(x + hv) - f(x)}{h} = D_v f(x) = \nabla f(x)^T v$$

Here $D_v f(x)$ is the directional derivative at x in the direction of v. We know that for any v,

$$|D_v f(x)| = |\nabla f(x)^T v| \leq L$$

Appendix: Lipschitz Continuous and Lipschitz Smooth

Then we can take the v that maximizes $D_v f(x)$. The maximizer v is known to be in the same direction as $\nabla f(x)$, i.e. ,$v = \frac{\nabla f(x)}{\|\nabla f(x)\|_2}$. Thus, for any x

$$\max_v |D_v f(x)| = \left| \nabla f(x)^T \frac{\nabla f(x)}{\|\nabla f(x)\|_2} \right| = \|\nabla f(x)\|_2 \leq L$$

Lipschitz Smooth: A function $f : \mathbb{R}^d \to \mathbb{R}$ is said to be Lipschitz Smooth if there exists a constant $\beta \geq 0$ such that for any $x, y \in \mathbb{R}^d$,

$$\|\nabla f(y) - \nabla f(x)\|_2 \leq \beta \|y - x\|_2$$

We also say that f is β-Smooth.

A function is Lipschitz Smooth does not mean it is Lipschitz Continuous. For example, $f(x) = x^2$. A function is Lipschitz Continuous also does not mean it is Lipschitz Smooth. For example, $f(x) = \frac{1}{2}\arcsin x + \frac{1}{2}x\sqrt{1-x^2}$ where $f'(x) = \sqrt{1-x^2}$.