
BASICS OF CONVOLUTIONAL NEURAL NETWORKS

Junyu Chen
jchen245@jhmi.edu
Johns Hopkins University

May 24, 2021

Contents

1	Multilayer perceptron network	3
1.1	Bias	3
1.2	Activation function	3
1.3	Backpropagation in MLP	3
1.4	Problems with MLP on Image Tasks	5
2	Convolutional Neural Networks	5
2.1	Backpropagation in ConvNet	6
2.2	Batch Normalization	7
2.3	The Effective Receptive Fields of ConvNets	8
3	Advancement in Network Design	8
3.1	Recurrent ConvNet	8
3.2	Transformers	9
3.2.1	Sinusoidal Positional Embedding	10
3.2.2	Relative Positioning	10
3.2.3	Vision Transformer	10

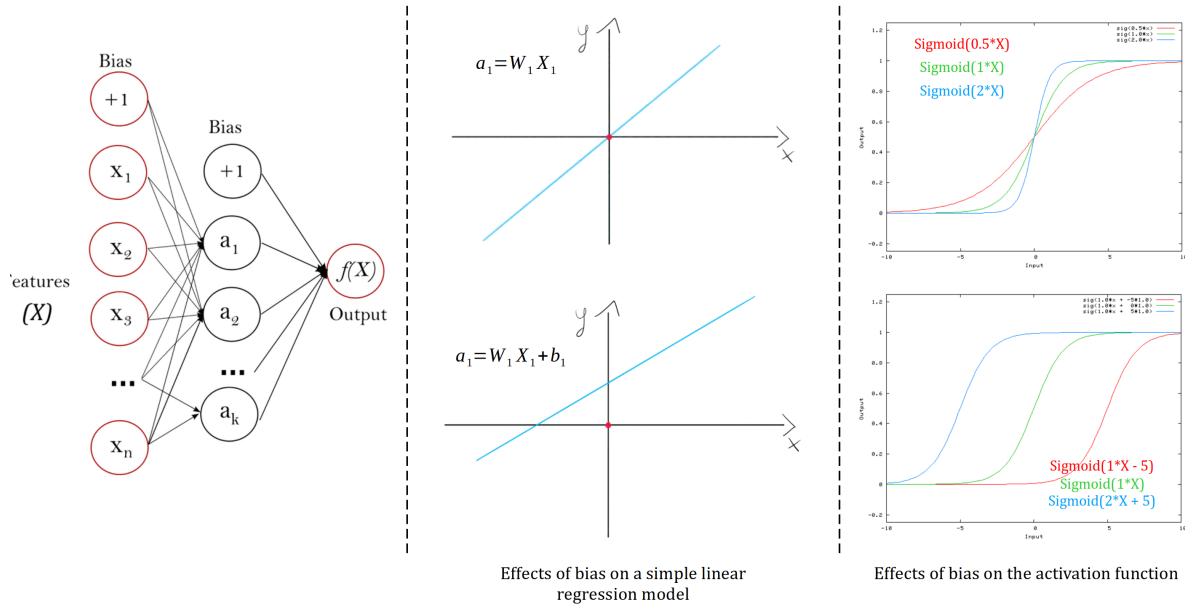


Figure 1: Left: One hidden layer MLP (image was obtained from [1]). Middle: Effects of bias on a linear regression model (image was obtained from [2]). Right: Effects of bias on the Sigmoid activation (image was obtained from [3]).

1 Multilayer perceptron network

Multilayer perceptron network (MLP) models the human brain, whereby the neurons in MLP are stimulated by connected nodes and are only activated when a certain threshold value is reached. The output of a neuron can be written as:

$$a_i = act\left(\sum_j w_{ij}x_j + b_i\right), \quad (1)$$

where w_{ij} represents the weight at index i and j , x_j is the j^{th} input neuron (neuron in the previous layer), b_i is the i^{th} bias, and act denotes the activation function.

1.1 Bias

Bias is an additional parameter in the network which is used to adjust the activation function by adding a constant (i.e. the given bias) to the input. It is a constant that helps the model to fit better for the given data. As shown in the middle and the right panels of Fig. 1, adding a bias shifts the linear regression model and the activation function to provide better model fitting to the data.

1.2 Activation function

Without activation function (i.e., act in Eqn. 1), the output a_i can take values from $-\infty$ to ∞ , which does not precisely model how brain neurons work. An activation function is used to decide whether the neuron should fire or not. As we shall see in section 1.3, without activation function, every neuron will only be performing a linear transformation on the inputs using the weights and biases, which will not be useful to learn complex patterns from real world data. Therefore, nonlinearities introduced by activation functions are essential in a neural network. Different choices of activation functions are shown in Fig. 2.

1.3 Backpropagation in MLP

Backpropagation is an algorithm used to train neural networks. It is used along with an optimization routine such as gradient descent. Backpropagation updates the weights in the network to provide a better fitting to given data. Let us consider a single layer MLP model:

$$p = act\left(\sum_j w_jx_j + b\right). \quad (2)$$










Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a. Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
Tanh		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$

Figure 2: Different activation functions (image was obtained from [4]).

Let p (i.e., $f(x)$ in the left panel of Fig. 1) be the output of the network, and t be the target (or ground truth), the loss function (mean squared error) can be defined as:

$$\mathcal{L} = \frac{1}{2} \|p - t\|^2. \tag{3}$$

In backpropagation, a weight (i.e., w_{ij}) is updated by computing the partial derivative of the loss function \mathcal{L} with respect to (w.r.t) the weight using the chain rule:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial w_j} &= \frac{\partial \mathcal{L}}{\partial p} \frac{\partial p}{\partial w_j} \dots \dots \text{chain rule} \\ &= \frac{\partial [\frac{1}{2} \|p - t\|^2]}{\partial p} \frac{\partial p}{\partial w_j} = (p - t) \frac{\partial p}{\partial w_j} \\ &= (p - t) \frac{\partial \text{act}(\sum_j w_j x_j + b)}{\partial w_j} \\ &= (p - t) \frac{\partial \text{act}(\sum_j w_j x_j + b)}{\partial (\sum_j w_j x_j + b)} \frac{\partial (\sum_j w_j x_j + b)}{\partial w_j} \dots \dots \text{chain rule} \\ &= \boxed{(p - t) \frac{\partial \text{act}(\sum_j w_j x_j + b)}{\partial (\sum_j w_j x_j + b)} x_j}, \end{aligned} \tag{4}$$

where $\frac{\partial \text{act}(\sum_j w_j x_j + b)}{\partial (\sum_j w_j x_j + b)}$ is simply the derivative of the activation function (activation functions shown in Fig. 2). Similarly, the partial derivative of the loss function \mathcal{L} w.r.t b can be computed as:

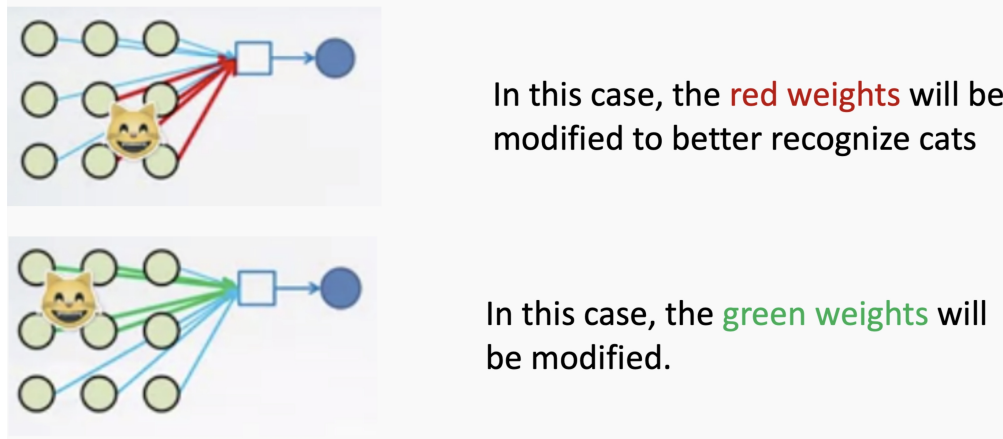


Figure 3: An image and its shift version activates different neurons (image was obtained from [5]).

$$\begin{aligned}
 \frac{\partial \mathcal{L}}{\partial b} &= \frac{\partial \mathcal{L}}{\partial p} \frac{\partial p}{\partial b} \dots \dots \dots \text{chain rule} \\
 &= (p - t) \frac{\partial \text{act}(\sum_j w_j x_j + b)}{\partial b} \\
 &= (p - t) \frac{\partial \text{act}(\sum_j w_j x_j + b)}{\partial(\sum_j w_j x_j + b)} \frac{\partial(\sum_j w_j x_j + b)}{\partial b} \dots \dots \dots \text{chain rule} \\
 &= \boxed{(p - t) \frac{\partial \text{act}(\sum_j w_j x_j + b)}{\partial(\sum_j w_j x_j + b)}}.
 \end{aligned} \tag{5}$$

From Eqn. 4 and 5 we can see that without the activation function, the derivative $\frac{\partial \text{act}(\sum_j w_j x_j + b)}{\partial(\sum_j w_j x_j + b)}$ goes away, and the overall partial derivative becomes a linear scaling. The derivatives of different activation functions are shown in Fig. 2. Finally, the classical gradient descent is used to update weights:

$$\mathbf{w}^{n+1} = \mathbf{w}^n - a_w \frac{\partial \mathcal{L}}{\partial \mathbf{w}^n}, \tag{6}$$

where a_w is the learning rate that is specified by an optimization method.

1.4 Problems with MLP on Image Tasks

MLPs are found to have limited performance on image tasks, such as image classification and segmentation. The drawbacks are two folds:

- MLPs use one perceptron (i.e., neuron) for each input (i.e., for each pixel or voxel in an image). The amount of weights rapidly becomes unmanageable for large network architectures and large images.
- Another drawback is that MLPs are not translation invariant. They tends to react differently to an input and its shifted version (as illustrated in Fig. 3)

Therefore, MLPs are not optimal for image tasks. We introduce another variant of neural networks in the following section.

2 Convolutional Neural Networks

Convolutional neural networks (ConvNets) solves the problems mentioned in section 1.4. Instead of using one neuron for each pixel, ConvNets learns a set of small convolutional kernels (typically 3×3 or 5×5), which dramatically decreases the number of weights. Besides, due to the nature of convolutional filters, ConvNets inherently learns the

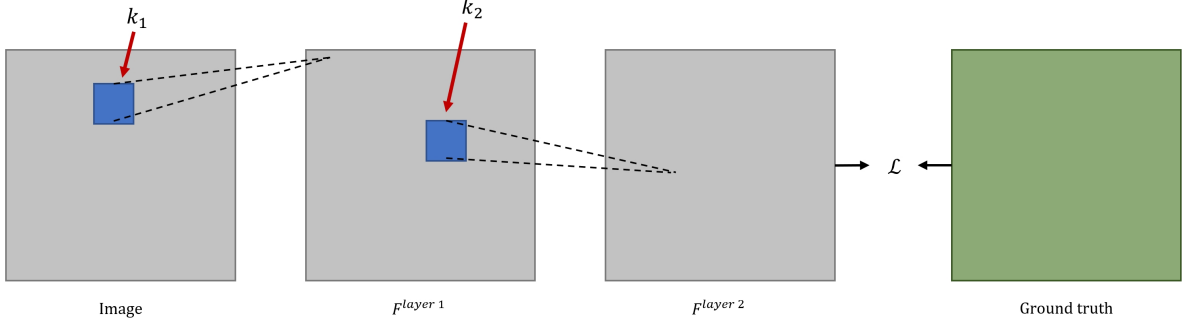


Figure 4: A simple two-layer ConvNet.

correlations between a pixel and its neighboring pixels. The filters (or kernels) in ConvNet could be related to anything. For example, in an image of cat, a filter could be associated with seeing cat's ears, and this filter will provide an indication of how strongly the cat's ears seem to appear in a given image, and in what locations in the image they appear. The output from a convolutional operation is defined as:

$$F_n = \text{act}\left(\sum_c k_{nc} * I_c + b_n\right), \quad (7)$$

where F_i denotes the n^{th} output feature map corresponds to the n^{th} kernel of c^{th} channel (i.e., k_{nc}), b_n is the bias, and I_c denotes the c^{th} input feature map. Similar to MLPs, the weights in a ConvNet is updated by backpropagation algorithm.

2.1 Backpropagation in ConvNet

Let us consider a simple ConvNet, as shown in Fig. 4, and assume the kernels k_1 and k_2 are 3×3 kernels. We, again, choose the loss function to be MSE due to its simplicity:

$$\mathcal{L} = \frac{1}{N} \|\mathbf{p} - \mathbf{t}\|^2, \quad (8)$$

where \mathbf{p} represents the output image (i.e., $\mathbf{p} = F^{\text{layer } 2}$), \mathbf{t} denotes the ground truth, and N is the total number of pixels. We first calculate the partial derivative of \mathcal{L} w.r.t the kernel k_2 :

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial k_2(u)} &= \frac{\partial \mathcal{L}}{\partial \mathbf{p}} \frac{\partial \mathbf{p}}{\partial k_2(u)} \dots \dots \text{chain rule} \\ &= \sum_n \frac{2}{N} (p(n) - t(n)) \frac{\partial p(n)}{\partial k_2(u)} \\ &= \sum_n \frac{2}{N} (p(n) - t(n)) \frac{\partial}{\partial k_2(u)} \text{act} \left(\sum_{u=-4}^4 F(n-u)k_2(u) + b^2(n) \right) \\ &= \sum_n \frac{2}{N} (p(n) - t(n)) \frac{\partial \text{act} \left(\sum_{u=-4}^4 F(n-u)k_2(u) + b^2(n) \right)}{\partial \sum_{u=-4}^4 F(n-u)k_2(u) + b^2(n)} F(n-u), \end{aligned} \quad (9)$$

where, again, $\frac{\partial \text{act}(\sum_{u=-4}^4 F(n-u)k_2(u) + b^2(n))}{\partial \sum_{u=-4}^4 F(n-u)k_2(u) + b^2(n)} \equiv \text{act}'$ is the derivative of the activation function. Next, we calculate the partial derivative of \mathcal{L} w.r.t the kernel k_1 :

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial k_1(u)} &= \frac{\partial \mathcal{L}}{\partial F} \frac{\partial F}{\partial k_1(u)} \dots \dots \text{chain rule} \\ &= \frac{\partial \mathcal{L}}{\partial \mathbf{p}} \frac{\partial \mathbf{p}}{\partial F} \frac{\partial F}{\partial k_1(u)} \dots \dots \text{chain rule}, \\ &= \sum_n \frac{\partial \mathcal{L}}{\partial p(n)} \frac{\partial p(n)}{\partial F(n)} \frac{\partial F(n)}{\partial k_1(u)} \end{aligned} \quad (10)$$

where $\frac{\partial p(n)}{\partial F(n)}$ can be computed as:

$$\begin{aligned}\frac{\partial p(n)}{\partial F(n)} &= \frac{\partial}{\partial F(n)} \text{act} \left(\sum_{u=-4}^4 F(n-u)k_2(u) + b^2(n) \right) \\ &= \text{act}' \frac{\partial}{\partial F(n)} \left(\sum_{u=-4}^4 F(n-u)k_2(u) + b^2(n) \right) \\ &= \text{act}' \cdot k_2(u).\end{aligned}\tag{11}$$

Then, the partial derivative of \mathcal{L} w.r.t the kernel k_1 becomes:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial k_1(u)} &= \frac{\partial \mathcal{L}}{\partial F} \frac{\partial F}{\partial k_1(u)} \dots \dots \text{chain rule} \\ &= \frac{\partial \mathcal{L}}{\partial \mathbf{p}} \frac{\partial \mathbf{p}}{\partial F} \frac{\partial F}{\partial k_1(u)} \dots \dots \text{chain rule,} \\ &= \sum_n \frac{\partial \mathcal{L}}{\partial p(n)} \frac{\partial p(n)}{\partial F(n)} \frac{\partial F(n)}{\partial k_1(u)} \\ &= \sum_n \frac{\partial \mathcal{L}}{\partial p(n)} \text{act}' \cdot k_2(u) \frac{\partial}{\partial k_1(u)} \text{act} \left(\sum_{u=-4}^4 I(n-u)k_1(u) + b^1(n) \right) \\ &= \boxed{\sum_n \frac{\partial \mathcal{L}}{\partial p(n)} \text{act}' \cdot k_2(u) \frac{\partial \text{act} \left(\sum_{u=-4}^4 I(n-u)k_1(u) + b^1(n) \right)}{\partial \sum_{u=-4}^4 I(n-u)k_1(u) + b^1(n)} \cdot I(n-u)}.\end{aligned}\tag{12}$$

Similar equations can be obtained for the biases \mathbf{b}^1 and \mathbf{b}^2 .

2.2 Batch Normalization

Normalization is a data pre-processing method used to bring data to a common scale. In medical imaging, normalization is usually done by scaling the the intensity of an image into a range of [0, 1], or it is done by normalizing the images into their z-scores. The reason of doing normalization is to ensure that the trained models can generalize well on the testing dataset.

In batch normalization, the term "batch" refers to the amount of data used to train a network in a single iteration (or a single forward/backward pass). Therefore, batch normalization is performed on a batch of images, instead of each individual input. Without batch normalization, input feature maps to each layer will not be normalized (except the first layer, assuming data normalization is performed in a pre-processing step). Batch normalization is found to speedup training (i.e., converging faster). It is performed by first standardize the batch of data to its z-score, then scale and shift the data by some trainable parameters:

1. Compute mean of the batch:

$$\mu = \frac{1}{N} \sum_n X_n,\tag{13}$$

where N is the total number of images (or samples), and X_n is the n^{th} image (or feature map).

2. Compute standard deviation of the batch:

$$\sigma = \left[\frac{1}{N} \sum_n (X_n - \mu)^2 \right]^{1/2}.\tag{14}$$

3. Normalize to z-score:

$$\bar{X}_n = \frac{X_n - \mu}{\sigma + \eta},\tag{15}$$

where η is a small value to prevent dividing by zero.

4. Scale and shift:

$$\hat{X}_n^{BN} = \gamma \bar{X}_n + \beta,\tag{16}$$

where γ and β are, respectively, scaling and shifting factors.

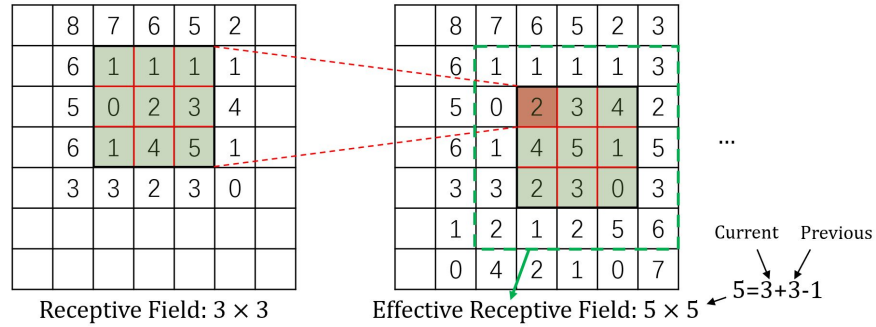


Figure 5: Illustration of the receptive field of convolution layers.

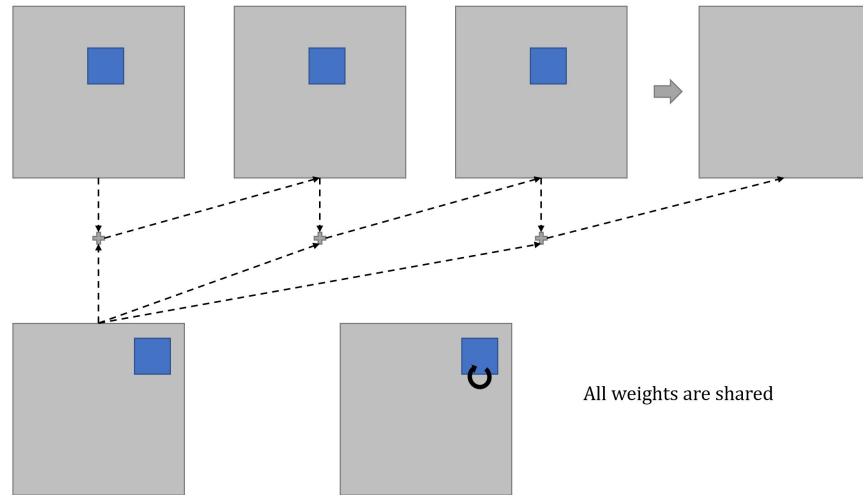


Figure 6: A recurrent convolution layer.

2.3 The Effective Receptive Fields of ConvNets

The receptive field (RF) can be referred to as the region in the input image that a particular CNN's feature is looking at (i.e. be affected by), or the region in the image that a kernel (i.e., filter) of a convolution layer can see. The effective receptive field (ERF) of the network grows with its depth. Let us look at an example shown in Fig. 5, where the RF of the first convolution operation is 3×3 (this equals to its filter size). Each pixel in the second image contains the information from a 3×3 region in the previous image. Therefore, the effective receptive field of the second convolution operation is:

$$\text{Current ERF} = \text{Size of Current RF} + \text{Size of Previous RF} - 1. \quad (17)$$

We can see that the ERF increases with the number of convolution layers used.

3 Advancement in Network Design

3.1 Recurrent ConvNet

One disadvantage of a conventional ConvNet is that the ERFs are small for the first several layers, that is, the ConvNet will not be able to "see" the whole image until the very last of layers. To overcome this problem, Liang et al. [6] proposed a recurrent ConvNet (RCNN), where the kernel of a convolution layer can "see" a larger region without increasing the kernel size. Therefore, RCNN can help a ConvNet to understand the context in an image better. The main component in a RCNN is the recurrent convolution layer (RCL), and it is shown in Fig. 6. An RCL can be expanded into t regular convolution operations, while the kernel weights in the t layers are share (i.e., identical weights). Except the first layer, the input to each layer is the sum of the output from the first layer with that of the previous layer (as illustrated in Fig. 6).

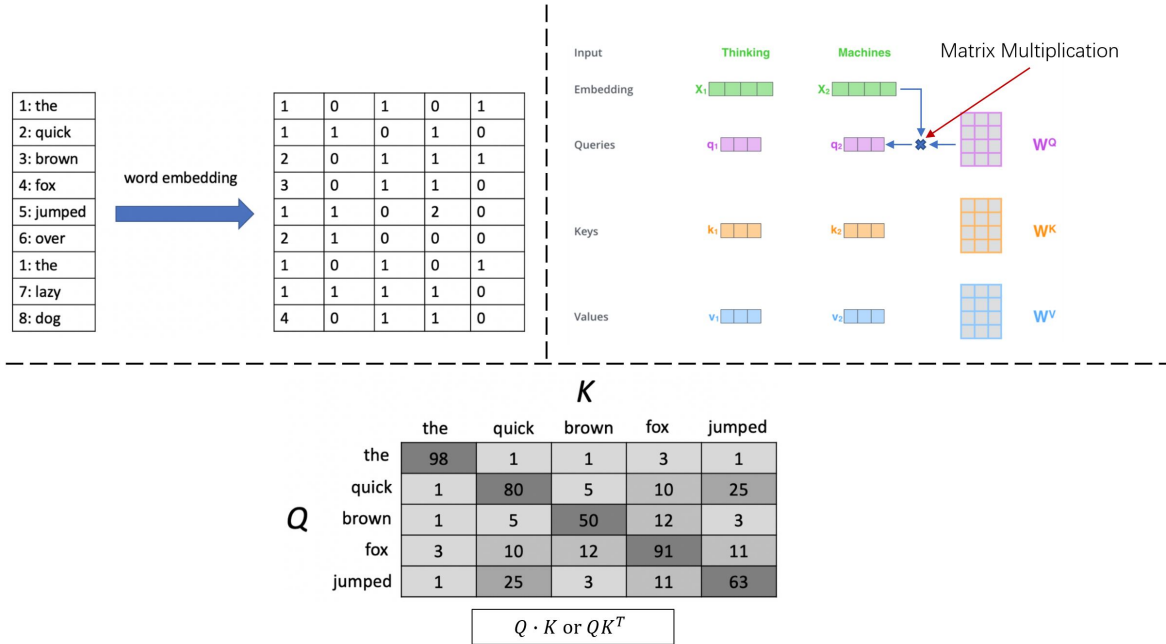


Figure 7: Graphical illustration of vector embedding, forming Q , K , and V vectors/matrices, and score matrix computing (images in the top left and bottom panels were obtained from [8], and the image in the top right panel was obtained from [9])

3.2 Transformers

Transformer was first introduced by Vaswani et al. in [7] for natural language processing (NLP). The most important component in the Transformer is the self-attention mechanism. Next, we take look at an example in NLP, and we briefly describe implementation of the self-attention in 4 steps:

1. The first step is to embed input words into vectors with the uniform length. This embedding is learned by the neural network, which outputs a vector representation for each word.
2. Add positional embeddings to the vectors. Positional embeddings are essential, and they allow the network to understand the relative position of each word vector. Otherwise, the positional information would be lost. We discuss positional embedding in detail in the next section (section 3.2.1).
3. Three vectors from each input vector (i.e., the embedding of each word) are created. These vectors consist of a *Query* vector, a *Key* vector, and a *Value* vector. The vectors are created by multiplying (i.e., matrix multiplication) the embedding vector by three matrices learned by the network during training. Notice that the Q , K , and V vectors are usually much smaller in dimension than the embedding vectors. For example, if the embedding vector has a dimension of 512, then the size of the Q , K , and V vectors usually is 64. To summarize, we create a *Query*, a *Key*, and a *Value* projection of each input vector in this stage.
4. Calculate scores by doing dot-product between a Q vector and a K vector of each of input embeddings. We are essentially computing a matrix of similarity scores between the *Query* matrix and the *Key* matrix (as shown in Fig. 7).
5. Finally, these scores are normalized by dividing the score matrix by the square root of the number of dimensions (e.g., 64). Then, a softmax activation function is applied to the scaled scores to convert them into "probabilities" (values range in $[0, 1]$). These "probabilities" are referred to as the *attention* weights, which are then multiplied by the *Value* matrix. The self-attention mechanism is defined mathematically as:

$$Attention(Q, K, V) = softmax\left(\frac{Q \cdot K}{\sqrt{n}}\right) V. \quad (18)$$

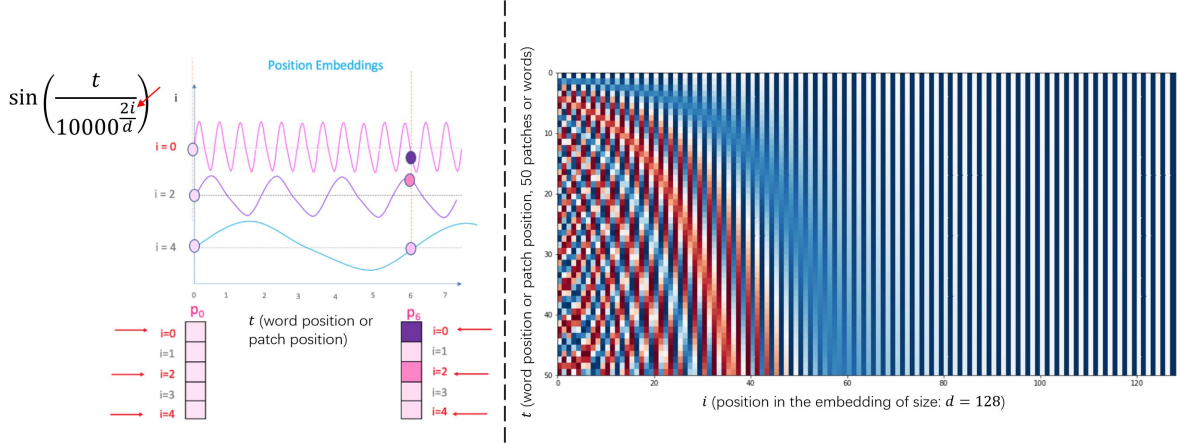


Figure 8: Graphical illustration of the sinusoidal positional embedding. (Image in the left panel was obtained from [10], and the image in the right panel was obtained from [11])

3.2.1 Sinusoidal Positional Embedding

Because the whole sequence of data (sentences in NLP or images in computer vision) is split into pieces (i.e., patches), positions or orders of the patches are essential. To embed positional information into a Transformer, a straightforward solution is adding a piece of information to each embedding about its position in the entire data (i.e., the whole image or sentence). A naive way to do this is to assign a number to each embedding. However, the problem is that if the length of the sequence is large, then some embeddings will be added by huge positional embedding values, which is very unfavorable for network training as the values may be way larger than the information in the patch embeddings.

The sinusoidal positional embedding (SPE) is a simple yet clever solution which satisfies our requirements. The resulting embedding is not a single number, but it is a d -dimensional vector (the same dimension as the patch embedding). Image in the left panel of Fig. 8 illustrates the idea of SPE. SPE is defined as:

$$\mathbf{p}_t(k) = \begin{cases} \sin\left(\frac{t}{10000^{2i/d}}\right), & \text{if } k = 2i \\ \cos\left(\frac{t}{10000^{2i/d}}\right), & \text{if } k = 2i + 1 \end{cases} \quad (19)$$

Here, t refers to the position of the patch in an image or the "word" in the sequence (e.g., \mathbf{p}_0 refers to the position embedding of the first patch or word), d represents the dimensions of the embedding, and i denotes the index in the t^{th} positional embedding. In the actual implementation, the embedding size d is often pre-defined or fixed. Image in the right panel of Fig. 8 presents an example of SPE using $d = 128$ and $t_{\max} = 50$ (i.e., 50 words/image patches).

3.2.2 Relative Positioning

An important feature that SPE provides is the relative positioning between an image patch and another. The SPE at $t + \phi$ can be represented as the linear transformation of the SPE at t :

$$\begin{bmatrix} \sin(w_i \cdot (t + \phi)) \\ \cos(w_i \cdot (t + \phi)) \end{bmatrix} = \begin{bmatrix} \cos(w_i \cdot \phi) & \sin(w_i \cdot \phi) \\ -\sin(w_i \cdot \phi) & \cos(w_i \cdot \phi) \end{bmatrix} \begin{bmatrix} \sin(w_i \cdot t) \\ \cos(w_i \cdot t) \end{bmatrix}. \quad (20)$$

This property makes it easy for the model to learn to attend by relative positions.

3.2.3 Vision Transformer

Although Transformer was originally proposed for NLP related tasks, recently Dosovitskiy et al. [12] have successfully demonstrated Transformer's ability to achieve superior performance in computer vision tasks. The authors proposed the Vision Transformer (ViT) that is essentially the same as a conventional Transformer architecture used in NLP. Next, we briefly describe how the ViT works:

1. Split an image into patches. Similar to splitting sentences into individual words in NLP, ViT splits an image into small patches and flatten the images to form vectors.
2. Produce linear embeddings of the flattened patches (same as step 1 in the section 3.2).

3. Add positional embeddings to retain positional information (see sectopm 3.2.1).
4. Feed the sequence as an input to a conventional transformer encoder.
5. Train the model for image classification tasks.

References

- [1] Neural network models (supervised). https://scikit-learn.org/stable/modules/neural_networks_supervised.html. Accessed: 2021-05-18.
- [2] Łukasz Gebel. Why we need bias in neural networks. <https://towardsdatascience.com/why-we-need-bias-in-neural-networks-db8f7e07cb98>. Accessed: 2021-05-18.
- [3] What is the role of the bias in neural networks? [closed]. <https://stackoverflow.com/questions/2480650/what-is-the-role-of-the-bias-in-neural-networks>. Accessed: 2021-05-18.
- [4] SAGAR SHARMA. Activation functions in neural networks. <https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6>. Accessed: 2021-05-18.
- [5] Matthew Stewart. Simple introduction to convolutional neural networks. <https://towardsdatascience.com/simple-introduction-to-convolutional-neural-networks-cdf8d3077bac>. Accessed: 2021-05-19.
- [6] Ming Liang and Xiaolin Hu. Recurrent convolutional neural network for object recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3367–3375, 2015.
- [7] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.
- [8] Amol Mavuduru. What are transformers and how can you use them? <https://towardsdatascience.com/what-are-transformers-and-how-can-you-use-them-f7ccd546071a>. Accessed: 2021-05-19.
- [9] Giuliano Giacaglia. How transformers work. <https://towardsdatascience.com/transformers-141e32e69591>. Accessed: 2021-05-19.
- [10] What is the positional encoding in the transformer model? <https://datascience.stackexchange.com/questions/51065/what-is-the-positional-encoding-in-the-transformer-model>. Accessed: 2021-05-19.
- [11] Amirhossein Kazemnejad. Transformer architecture: The positional encoding. https://kazemnejad.com/blog/transformer_architecture_positional_encoding/. Accessed: 2021-05-19.
- [12] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale, 2020.