Intro to Neural Network
Xingyu Ren 2025 0314 (updating)
https://github.com/Phy-Ren

This is a general intro for physicist to Neural Networks (NNs). This note is expected to gather some chosen topics for physicist to study NNs and Machine Learning (ML). The contents are collected from many resources, so the references may not be complete. The most important ones are listed as possible.

1. Neural Quantum State

1.1. Variational wave function

Let us first start with a specific application of NNs on lattice models. A quantum many-body state $|\psi\rangle$ can be represented by its wave function

$$|\psi\rangle = \sum\nolimits_s \psi(s)|s\rangle \tag{1}$$

on a complete orthonormal basis $\left\{|s\rangle\right\}$. The essential problem of simulating a quantum many-body system is that the exponentially increasing Hilbert space dimension (2^N for qubit systems) with the system size N. Given the Hamiltonian $\hat{H}$ of the system, one of the central concern is to find the ground state $|GS\rangle$ of this system

$$\hat{H}|GS\rangle = E_{min}|GS\rangle, \tag{2}$$

where E_{min} is the minimal eigenvalue of the Hamiltonian.

The idea of variational ansatz is to assume that the wave function $\psi(s)$ has some specific structure such that it can be parametrized by some parameters θ

$$(\theta, s) \mapsto \psi_\theta(s) = \langle s|\psi_\theta\rangle. \tag{3}$$

Usually, if the number of independent parameters θ is less than the Hilbert space dimension 2^N so that this wave function can not represent the most general states in the Hilbert space. Here is the point: Most low-lying physical states we care about have very spacial structure such that they only occupy exponentially small parts in the whole Hilbert space. If the subspace span by a variational state covers the low-lying physical states, then this variational ansatz can represent the physics very well in a much smaller subspace instead of the whole Hilbert space.

1.2. Variational Monte Carlo

The quantum expectation value of an operator $\hat{A}$ on a non-normalized pure state $|\psi\rangle$ can be written as a classical expectation value $\mathbb{E}[\tilde{A}]$ over the Born distribution
$\rho(s) \propto |\psi(s)|^2$

$$\langle \hat{A} \rangle = \frac{\langle \psi | \hat{A} | \psi \rangle}{\langle \psi | \psi \rangle} = \sum_s \frac{|\psi(s)|^2}{\langle \psi | \psi \rangle} \tilde{A}(s) = \sum_s p(s) \tilde{A}(s) = \mathbb{E}[\tilde{A}], \tag{4}$$

where $\tilde{A}$ is the local estimator

$$\tilde{A}(s) = \frac{\langle s | \hat{A} | \psi \rangle}{\langle s | \psi \rangle} = \sum_{s'} \frac{\psi(s')}{\psi(s)} \langle s | \hat{A} | s' \rangle. \tag{5}$$

Note that even though the sum in Eq. (5) runs over the whole Hilbert space basis, it can still be efficiently computed if the operator $\hat{A}$ is sparse enough. However, the sum in Eq. (4) is too hard to calculate such that the classical expectation value $\mathbb{E}[\tilde{A}]$ must be estimated by averaging over a sequence $\{s_i\}_{i=1}^{N_s}$ of configurations distributed according to the Born distribution $\rho(s) \propto |\psi(s)|^2$

$$\mathbb{E}[\tilde{A}] \approx \frac{1}{N_s} \sum_{i=1}^{N_s} \tilde{A}(s_i). \tag{6}$$

Given the derivatives of the log-amplitudes

$$O_i(s) = \frac{\partial \ln \psi_\theta(s)}{\partial \theta_i}, \tag{7}$$

the force vector can be defined as the covariance

$$\tilde{f}_i = \mathrm{Cov}[O_i, \tilde{A}] = \mathbb{E}\Big[O_i^* (\tilde{A} - \mathbb{E}[\tilde{A}])\Big]. \tag{8}$$

If the parameters $\theta_i \in \mathbb{R}$ are real, the gradients of the expectation value are

$$\frac{\partial \langle \hat{A} \rangle}{\partial \theta_i} = 2 \Re[\tilde{f}_i]. \tag{9}$$

If the parameters $\theta_i \in \mathbb{C}$ and the mapping $\theta_i \mapsto \psi_\theta(s)$ is complex differentiable (holomorphic), the gradients are

$$\frac{\partial \langle \hat{A} \rangle}{\partial \theta_i^*} = \tilde{f}_i. \tag{10}$$

In the case of a non-holomorphic mapping, the real part $\Re[\theta_i]$ and the imaginary part $\Im[\theta_i]$ can be treated independently as two real parameters.

Above is the essence of Variational Monte Carlo (VMC): chose an varational wave function ansatz, use the Monte Carlo sampling to approximately calculate the expectation value of operators, then minimize the energy expectation value to get the ground state.

1.3. PEPS

For strongly correlated systems, one of the most important wave function ansatz is Tensor Network (TN). Especially for 2D quantum systems, the Projected Entangled-Pair State (PEPS) is one of the most important ansatz. PEPS is a natural generalization of the Matrix Product State (MPS). The advantage of these methods is that their entanglement entropies scale with the boundary of the system, which is the characteristic of most gaped ground states of local Hamiltonian. The MPS ansatz can be written as

$$\psi(s_1, s_2, \cdots, s_n) = \mathrm{Tr}\Big[A_1^{s_1} A_2^{s_2} \cdots A_n^{s_n}\Big], \tag{11}$$

where $A_i^{s_i}$ is a rank-3 tensor (in the bulk) with the physical leg labeled by s_i and the two inner legs form a matrix. The PEPS ansatz replaces the rank-3 tensor with a rank-5 tensor and forms a 2-dimensional tensor networks.

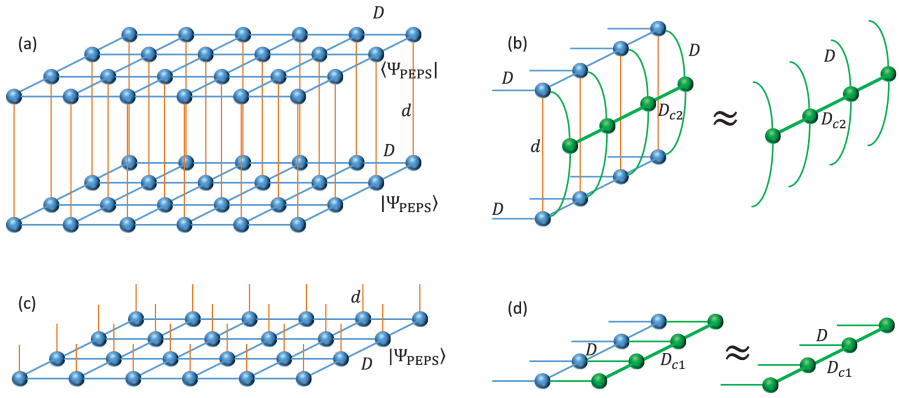


Figure 1: PEPS+VMC

(a) Exact contraction of a PEPS.
(b) Boundary-MPO method to reduce the computational complexity of (a).
(c) VMC can reduce from contracting (a) to a single layer PEPS with fixed physical legs.
(d) Boundary-MPS method to reduce the computational complexity of (c).

In principle, we can directly contract a tensor network state to get the physical quantities. However, the size of the PEPS is usually too large to contract rigorously. Therefore, Ref. [1] introduced the VMC method and boundary-MPS method to reduce the computational complexity. Figure 1 shows how this works:

1. Introducing VMC can simplify the contraction process from (a) to (c).
2. Introducing the boundary-MPS method in (d), truncating the bond dimension of the green bold bond at a fixed value D_{c1} (usually $D_{c1} \approx 2D$).

1.4. NN

In the seminal paper [6], the authors introduced Restricted Boltzmann Machine (RBM, a simplest type of NN) as the wave function ansatz to calculate the ground state of quantum lattice models: 1D Ising model, 1D and 2D Heisenberg model. Since the RBM is a very simple NN architecture and the parameter number is not so large, so this paper uses a purely variational method, compute the operator expectation value exactly without

Monte Carlo.

Now in the era of Large Language Models (LLMs), the number of model parameters becomes extremely large, so the VMC method is necessary in the training process to reduce computation complexity. Various NN architectures including Transformer are also introduced by many papers as variational wave function ansatz. Also many training techniques are introduced from the ML community.

Now here is a package NetKet 3 written by JAX/Flax implementing this NN+VMC method.

2. NN Architectures

Above we motivated the NN studies by an application of Neural Quantum State. Now we start to introduce basics of NNs and ML. There are so many NN architectures that we can only mention several most important ones.

2.1. RBM

The Restricted Boltzmann Machine is simplified from Boltzmann Machine, so let's start with the latter. The Boltzmann machine has a set $\{s_i\} = \{v_i\} \amalg \{h_j\}$ of binary variables $s_i \in \{0, 1\}$, which can be divided into two groups: the visible variables $\{v_i\}$ and the hidden variables $\{h_j\}$. Just as a classical Ising model, the Boltzmann machine has an energy function

$$E_\lambda(\{s_i\}) = -\sum_{i<j} w_{ij} s_i s_j - \sum_i \theta_i s_i, \tag{12}$$

where $\{\lambda\} \equiv \{w_{ij}, \theta_i\}$ is the set of model parameters. Figure 2 shows the structure of Boltzmann machine: the circles are binary variables $\{s_i\}$, the white circles are visible variables $\{v_i\}$, the blue circles are the hidden variable $\{h_j\}$, and the energy function E introduces interactions between any two variables, making it a fully connected graph.

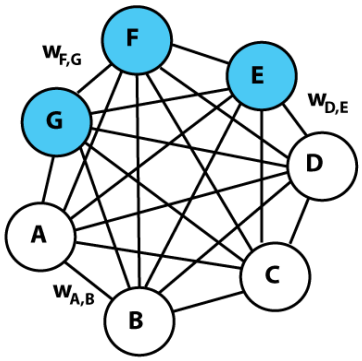


Figure 2: Boltzmann Machine: a fully connected graph

Then the joint probability distribution is

$$p_\lambda(\{s_i\}) = \frac{e^{-E(\{s_i\})}}{\mathcal{Z}}, \tag{13}$$

$$\mathcal{Z} = \mathrm{Tr}_{s_i} e^{-E(\{s_i\})} = \sum\nolimits_{s_i} e^{-E(\{s_i\})}. \tag{14}$$

We then trace out the hidden variables $\{h_j\}$ to get the variational distribution of the visible variables

$$p_\lambda(\{v_i\}) = \mathrm{Tr}_{h_j} p_\lambda(\{v_i\}, \{h_j\}) \equiv \frac{e^{-H_\lambda(\{v_i\})}}{\mathcal{Z}}, \tag{15}$$

where $H_\lambda(\{v_i\})$ is the variational Hamiltonian if viewed as an effective statistical model in physics. In general, the problem we consider can have nothing to do with physics. The general setup is that given a probability distribution of binary variables $P(\{v_i\})$, we want to approximate this ground truth distribution $P(\{v_i\})$ by the variational distribution $p_\lambda(\{v_i\})$, which done by minimize the Kullback-Leibler divergence

$$D_{KL}[P(\{v_i\})||p_\lambda(\{v_i\})] = \sum_{\{v_i\}} P(\{v_i\}) \log \frac{P(\{v_i\})}{p_\lambda(\{v_i\})}. \tag{16}$$

The RBM is a simpler version of the above Boltzmann machine. It is called restricted because the interactions are only between one visible variable and one hidden variable, so there is no interaction between visible variables themselves or between hidden variables themselves. Then the energy function would be slightly different from Eq. (12) as below

$$E(\{v_i\}, \{h_j\}) = \sum_j b_j h_j + \sum_{ij} v_i w_{ij} h_j + \sum_i c_i v_i, \tag{17}$$

where the variational parameters are $\lambda = \{w_{ij}, b_j, c_i\}$. Though the joint probability distribution actually has the same form as Eq. (13), we still rewrite it to emphasize the difference of the energy functions

$$p_\lambda(\{v_i\}, \{h_j\}) = \frac{e^{-E(\{v_i\},\{h_j\})}}{\mathcal{Z}}, \tag{18}$$

$$\mathcal{Z} = \mathrm{Tr}_{v_i,h_j} e^{-E(\{v_i\},\{h_j\})} = \sum\nolimits_{v_i,h_j} e^{-E(\{v_i\},\{h_j\})}. \tag{19}$$

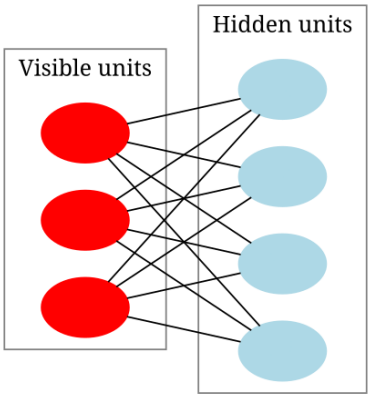


Figure 3: Restricted Boltzmann Machine: interactions only between one visible variable and one hidden variable.

2.2. CNN

The Convolutional NN (CNN) is motivated from the way the brain achieves vision processing in animals. Work by Hubel and Wiesel in the 1950s and 1960s showed that cat visual cortices contain neurons that individually respond to small regions of the visual field. Provided the eyes are not moving, the region of visual space within which visual stimuli affect the firing of a single neuron is known as its receptive field. Neighboring cells have similar and overlapping receptive fields. Receptive field size and location varies systematically across the cortex to form a complete map of visual space. Let's see below how this matches the architecture of CNN. A CNN architecture is formed by a stack of distinct layers that transform the input volume into an output volume through a differentiable function.

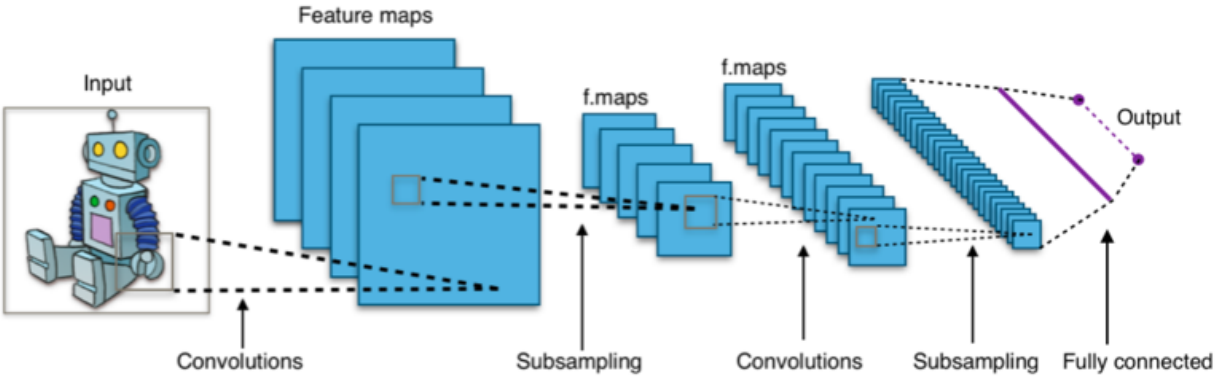


Figure 4: Typical CNN architecture

2.2.1. Convolutional layer

Usually, we are dealing with pictures using CNN. Assuming the input is a 2D picture, then it is passed to a convolutional layer, which is the core building block of a CNN. The

whole process is shown in Figure 5. The layer's parameters consist of a set of learnable filters (or kernels, in Figure 5 it is $\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$), which have a small receptive field, but extend through the full depth of the input volume. During the forward pass, each filter is convolved across the width and height of the input volume, computing the dot product between the filter entries and the input, producing a 2-dimensional activation map of that filter. It is important to note that the computation of the filter with the input is not the matrix multiplication producing a matrix, it is more like the vector multiplication producing a scalar.

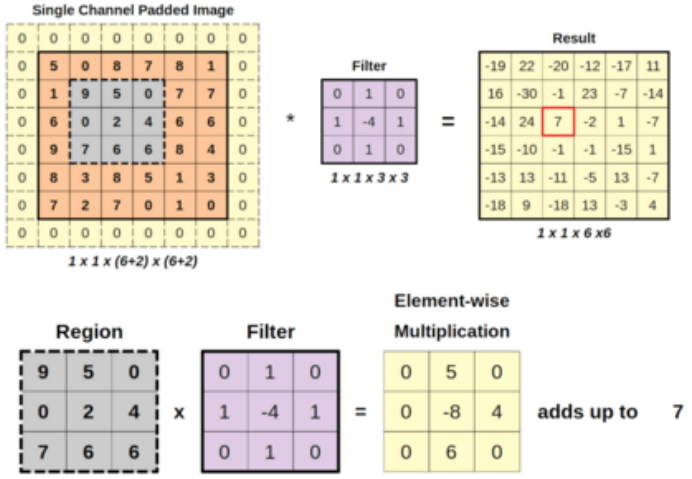


Figure 5: Convolutional kernel

Another thing to note is the locality of the CNN. The images have a natural spacial organization, which may not be true for some other kind of data. As the filter is passing through the input picture, we can see the locality of the spacial organization is necessary for the computation. CNNs not only work for 2D images, but also for some high dimensional data with good locality. The convolutional layer exploits spatially local correlation by enforcing a sparse local connectivity pattern between neurons of adjacent layers: each neuron is connected to only a small region of the input volume. The extent of this connectivity is a hyperparameter called the receptive field of the neuron.

Three hyperparameters control the size of the output volume of the convolutional layer: the depth, stride, and padding size:

1. Depth of the output volume: It controls the number of neurons in a layer that connect to the same region of the input volume. In other words, it is the number of distinct filters (also the number of output layers) you have.
2. Stride: It controls how depth columns around the width and height are allocated. If the stride is 1, then we move the filters one pixel at a time. This leads to heavily overlapping receptive fields between the columns, and to large output volumes. For any integer S, a stride S means that the filter is translated S units at a time per

output.

3. Padding: it is convenient to pad the input with zeros (or other values, such as the average of the region) on the border of the input volume. The size of this padding provides control of the output volume's spatial size.

2.2.2. Pooling layer

Another important concept of CNNs is pooling, which is used as a form of non-linear down-sampling. There are several non-linear functions to implement pooling, where max pooling and average pooling are the most common.

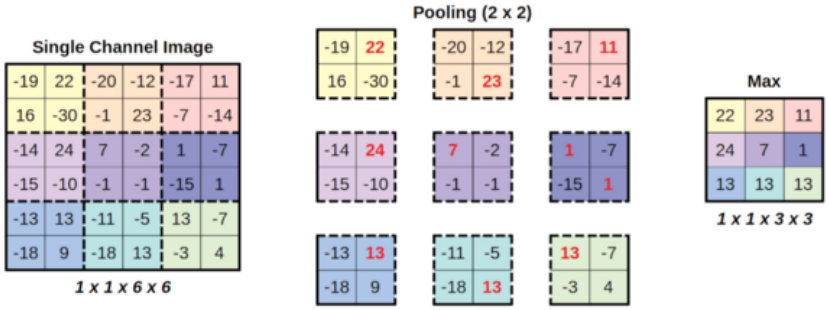


Figure 6: An example of 2×2 maxpooling of stride 2.

Intuitively, the exact location of a feature is less important than its rough location relative to other features. This is the idea behind the use of pooling in CNNs. The pooling layer serves to progressively reduce the spatial size of the representation, to reduce the number of parameters, memory footprint and amount of computation in the network, and hence to also control overfitting. This is known as down-sampling. It is common to periodically insert a pooling layer between successive convolutional layers (each one typically followed by an activation function, such as a ReLU layer) in a CNN architecture.

2.2.3. Activation function

The activation function of a node in a NN is a function that calculates the output of the node based on its individual inputs and their weights. Nontrivial problems can be solved using only a few nodes if the activation function is nonlinear. Modern activation functions include:

1. Logistic (sigmoid) function used in the 2012 speech recognition model developed by Hinton et al;
2. ReLU used in the 2012 AlexNet computer vision model and in the 2015 ResNet model;
3. GELU, the smooth version of the ReLU, which was used in the 2018 BERT model.

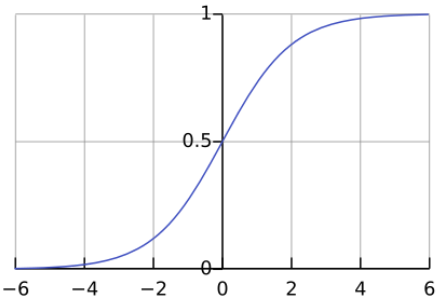


Figure 7: Logistic (sigmoid) activation function

Aside from their empirical performance, activation functions also have different mathematical properties:

1. Nonlinear: When the activation function is non-linear, then a two-layer neural network can be proven to be a universal function approximator. This is known as the Universal Approximation Theorem. The identity activation function does not satisfy this property. When multiple layers use the identity activation function, the entire network is equivalent to a single-layer model.
2. Range: When the range of the activation function is finite, gradient-based training methods tend to be more stable, because pattern presentations significantly affect only limited weights. When the range is infinite, training is generally more efficient because pattern presentations significantly affect most of the weights. In the latter case, smaller learning rates are typically necessary.
3. Continuously differentiable: This property is desirable (ReLU is not continuously differentiable and has some issues with gradient-based optimization, but it is still possible) for enabling gradient-based optimization methods.

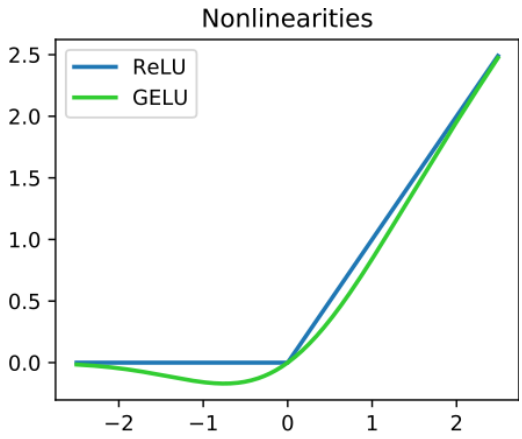


Figure 8: ReLU (blue) and GELU (green) activation functions

The Rectified Linear Unite (ReLU) is one of the most popular activation function used in NNs. As plotted in Figure 8, it just effectively removes negative values from an activation map by setting them to zero

$$\text{ReLU}(x) = \max(0, x). \tag{20}$$

ReLU is often preferred to other functions because it trains the neural network several times faster without a significant penalty to generalization accuracy. For the CNN, the ReLU is applied to every single neuron after every convolution layer and every dense layer.

2.2.4. Fully connected layer

After several convolutional and max pooling layers, the final classification is done via fully connected layers (dense layers). Fully connected layers in a neural networks are those layers where all the inputs from one layer are connected to every activation unit of the next layer. In most popular machine learning models, the last few layers are full connected layers which compiles the data extracted by previous layers to form the final output. It is the second most time consuming layer second to convolution layer. While convolutional layers are good at detecting features in input data, dense layers are essential for integrating these features into predictions.

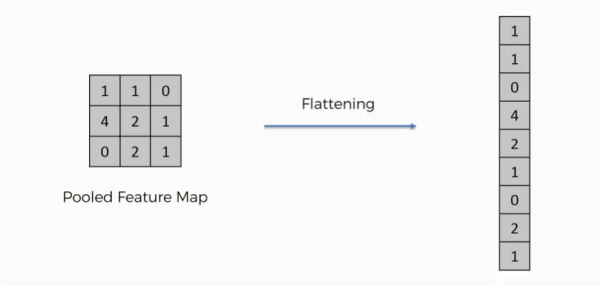


Figure 9: Flattening before the dense layers in CNNs

Before the fully connected NN, we usually get the outputs from the pooling layer, which should flattened before put them into the dense layer as Figure 9. As shown in Figure 10, the structure of the fully connected NN is very similar as the RBM introduced before. The differences are:

1. The input variables are not restricted to binary values, but can take any real values for the fully connected NN.
2. There is an activation function applied to every neuron in the hidden layers and output layer in the fully connected NN.

Each neuron in the hidden layer or output layer can be written as

$$a_j = f(z_j) = f\left(\sum\nolimits_i w_{ij} x_i + b_j\right), \tag{21}$$

where a_j is the output value of the j-th neuron in the latter hidden layer, f is the activation function, x_i is the value of the i-th neuron in the previous input layer, and $\{w_{ij}, b_j\}$ are the wights of the parameters.

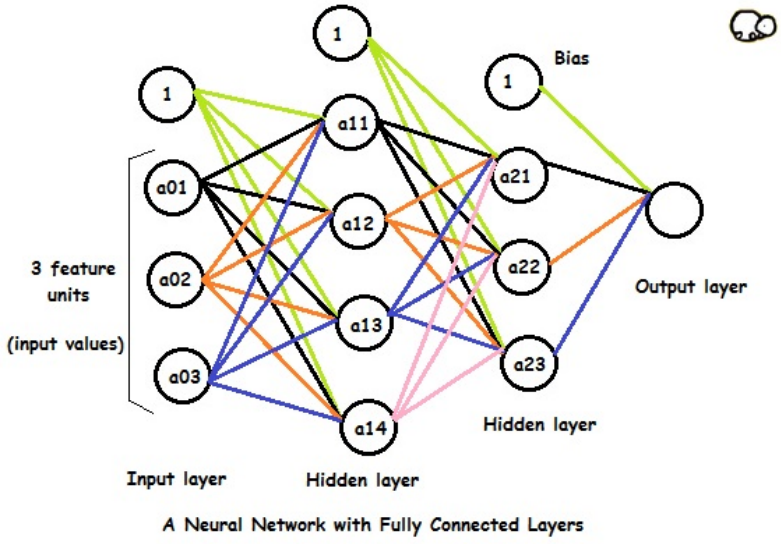


Figure 10: Typical structure of a fully connected NN.

2.2.5. Loss function

The "loss layer", or "loss function", specifies how training penalizes the deviation between the predicted output of the network, and the true data labels (during supervised learning). Various loss functions can be used, depending on the specific task. The training process is basically minimize the loss function to the global minimum (ideally) using automatic differentiable programming.

The Softmax loss function is used for predicting a single class of K mutually exclusive classes. Formally, the standard (unit) softmax function $\operatorname{softmax}: \mathbb{R}^K \to (0,1)^K$ takes a vector $\mathbf{x} = (x_1, \cdots, x_K) \in \mathbb{R}^K$ and computes each component i of vector $\operatorname{softmax}(\mathbf{x}) \in (0,1)^K$ with

$$\operatorname{softmax}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}. \tag{22}$$

The term "softmax" derives from the amplifying effects of the exponential on any maxima in the input vector. For example, the standard softmax of $(1, 2, 8)$ is approximately $(0.001, 0.002, 0.997)$, which amounts to assigning almost all of the total unit weight in the result to the position of the vector's maximal element.

Sigmoid cross-entropy loss is used for predicting K independent probability values in $[0, 1]$. Euclidean loss is used for regressing to real-valued labels $(-\infty, +\infty)$.

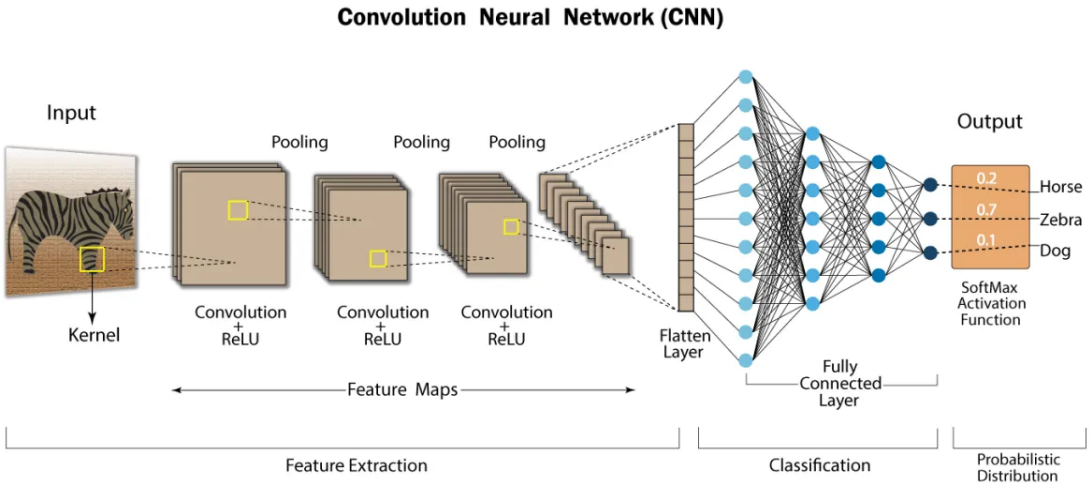


Figure 11: Typical CNN architecture

2.3. Transformer

As the foundation of Large Language Models (LLMs), the transformer is now the most important deep learning architecture that was developed by researchers at Google and is based on the multi-head attention mechanism, which was proposed in the 2017 paper "Attention Is All You Need". Figure 12 is the encoder-decoder transformer proposed in Google's original paper. Starting in 2018, the OpenAI GPT series of decoder-only Transformers became state of the art in natural language generation. In 2022, a chatbot based on GPT-3, ChatGPT, became unexpectedly popular, triggering a boom around large language models. All transformers have the same primary components:

1. Tokenizers, which convert text into tokens.
2. Embedding layer, which converts tokens and positions of the tokens into vector representations.
3. Transformer layers, which carry out repeated transformations on the vector representations, extracting more and more linguistic information. These consist of alternating attention and feedforward layers. There are two major types of transformer layers: encoder layers and decoder layers, with further variants.
4. Un-embedding layer, which converts the final vector representations back to a probability distribution over the tokens.

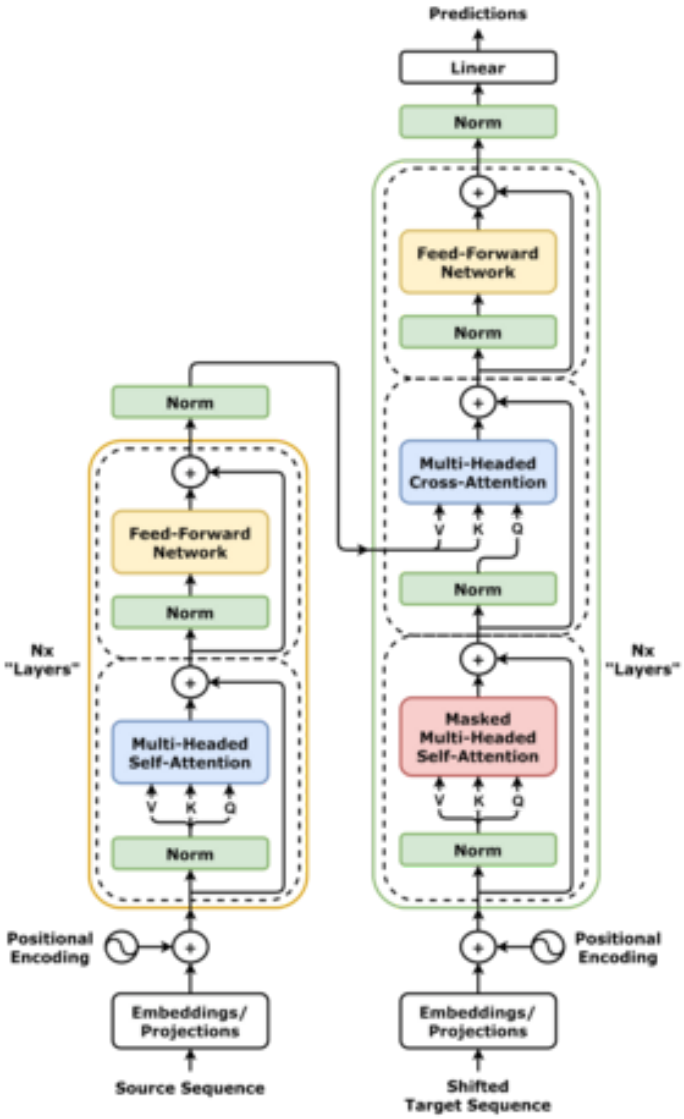


Figure 12: Encoder-Decoder Transformer architecture

2.3.1. Tokenization

As the Transformer architecture natively processes numerical data, not text, there must be a translation between text and tokens. A token is an integer that represents a character, or a short segment of characters. On the input side, the input text is parsed into a token sequence. Similarly, on the output side, the output tokens are parsed back to text. The module doing the conversion between texts and token sequences is a tokenizer.

The set of all tokens is the vocabulary of the tokenizer, and its size is the vocabulary size n_{voca}. When faced with tokens outside the vocabulary, typically a special token is used, written as "[UNK]" for "unknown".

2.3.2. Embedding

Each token is converted into an embedding vector via a lookup table. Equivalently stated, it multiplies a one-hot representation of the token by an embedding matrix M. For example, if the input token is 3, then the one-hot representation is $[0, 0, 0, 1, 0, 0, \cdots]$, and its embedding vector is

$$\text{Embed}(3) = [0, 0, 0, 1, 0, 0, \cdots] \cdot M, \tag{23}$$

where $\cdot$ means the matrix/vector multiplication. The number of dimensions in an embedding vector is called hidden size or embedding size d_{emb}. The token embedding vectors are added to their respective positional encoding vectors (see below), producing the sequence of input vectors.

2.3.3. Positional encoding

A positional encoding is a fixed-size vector representation of the relative positions of tokens within a sequence: it provides the transformer model with information about where the words are in the input sequence. The positional encoding is defined as a function $\text{PE} : \mathbb{R} \to \mathbb{R}^{d_{\text{emb}}}$, where d_{emb} is usually the embedding dimension defined above. Denote the position of a token as $t_{\text{pos}} \in \mathbb{N}$ and the i-the component of the vector $\text{PE}(t_{\text{pos}})$ is

$$\text{PE}(t_{\text{pos}})_i = \begin{cases} \sin \dfrac{t_{\text{pos}}}{N^{i/d_{\text{emb}}}}, & \text{even } i \\ \cos \dfrac{t_{\text{pos}}}{N^{(i-1)/d_{\text{emb}}}}, & \text{odd } i \end{cases}, \tag{24}$$

where N is a free parameter that should be significantly larger than d_{emb} (the original paper uses $N = 10000$). Figure 13 shows the pattern for this positional encoding function. The main reason for using this positional encoding function is that using it, shifts of Δt_{pos} are linear transformations:

$$\text{PE}(t_{\text{pos}} + \Delta t_{\text{pos}}) = \text{diag}[\text{PE}(\Delta \text{t}_{\text{pos}})] \cdot \text{PE}(t_{\text{pos}}), \tag{25}$$

where $\text{diag}[\text{PE}(\Delta \text{t}_{\text{pos}})]$ means putting the elements of the vector $\text{PE}(\Delta t_{\text{pos}})$ on the diagonal positions to form an diagonal matrix. This nice property of relative positions can be manipulated to give

$$\text{PE}(t_{\text{pos}}) = \left\{ \sum_j c_j \times \text{diag}[\text{PE}(\Delta t_{\text{pos}})] \right\}^{-1} \left\{ \sum_j c_j \times \text{PE}(\Delta t_{\text{pos},j}) \right\} \tag{26}$$

for any constants c_j. This allows the transformer to take any encoded position and find a linear sum of the encoded locations of its neighbors. This sum of encoded positions, when fed into the attention mechanism, would create attention weights on its neighbors, much like what happens in a CNN language model. In the author's words, "we hypothesized it would allow the model to easily learn to attend by relative position."

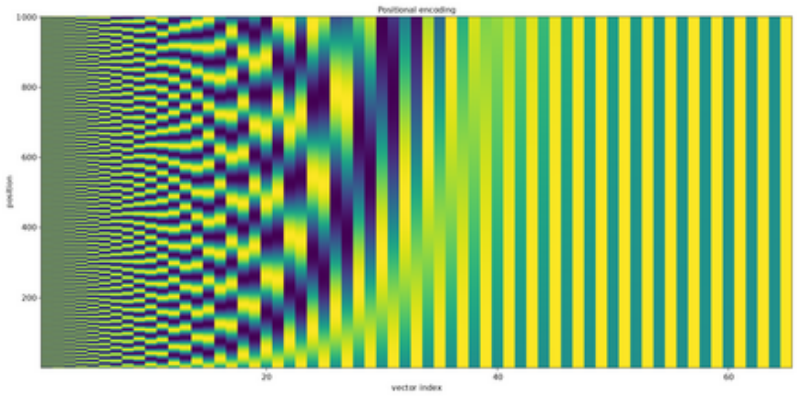


Figure 13: Sinusoidal positional encoding with parameters $N = 10000, d_{\text{emb}} = 100$

2.3.4. Encoder

The encoder consists of encoding layers that process all the input tokens together one layer after another, while the decoder consists of decoding layers that iteratively process the encoder's output and the decoder's output tokens so far. The purpose of each encoder layer is to create contextualized representations of the tokens, where each representation corresponds to a token that "mixes" information from other input tokens via self-attention mechanism.

Attention head

The attention mechanism used in the Transformer architecture are scaled dot-product attention units, which is the most important structure. For each unit, the transformer model learns three weight matrices: the query weights W^Q with dimension $d^Q_{\text{emb}} \times d_{\text{query}}$, the key weights W^K with dimension $d^K_{\text{emb}} \times d_{\text{key}}$, and the value weights W^V with dimension $d^V_{\text{emb}} \times d_{\text{value}}$. The module takes three sequences, a query sequence, a key sequence, and a value sequence. Here is the point: the attention mechanism describes a weighted average of (sequence) elements with the weights dynamically computed based on an input query and elements' keys. Denote the sequence length of the input token embedding vectors (with dimension d_{emb}) as ℓ_{seq}. Put all these vectors x_i ($i = 0, \cdots, \ell_{\text{seq}} - 1$) in the sequence together, we get an input matrix X with dimension $\ell_{\text{seq}} \times d_{\text{emb}}$. For a general scaled dot-product attention mechanism, the input sequences can be different for query, key and value such that we have different X^Q, X^K, X^V with dimensions $\ell^Q_{\text{seq}} \times d^Q_{\text{emb}}, \ell^K_{\text{seq}} \times d^K_{\text{emb}}, \ell^V_{\text{seq}} \times d^V_{\text{emb}}$. The corresponding query, key and value matrices are

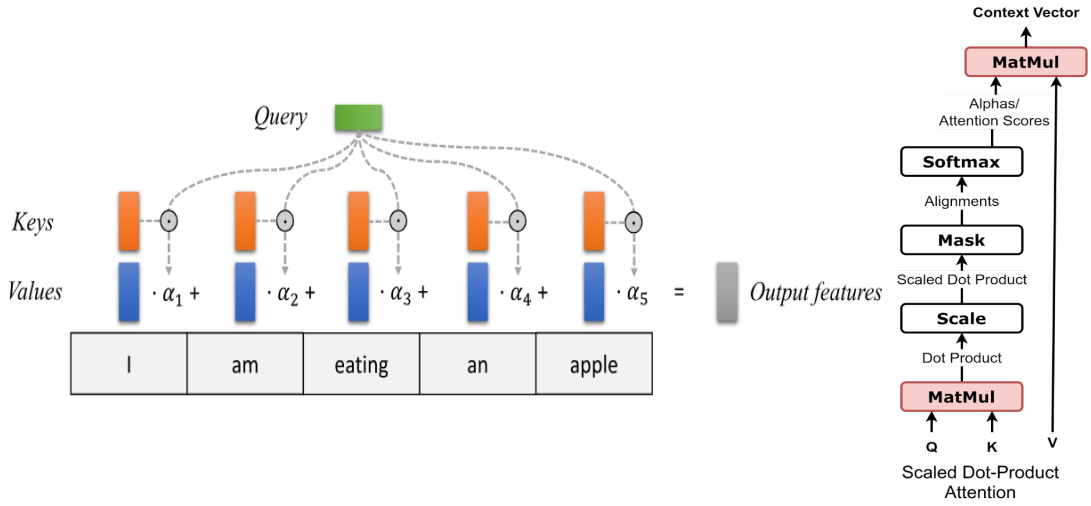
$$Q = X^Q \cdot W^Q, \tag{27}$$
$$K = X^K \cdot W^K, \tag{28}$$
$$V = X^V \cdot W^V \tag{29}$$

with dimensions $\ell^Q_{\text{seq}} \times d_{\text{query}}, \ell^K_{\text{seq}} \times d_{\text{key}}, \ell^V_{\text{seq}} \times d_{\text{value}}$. Then as shown in Drawing 1,

the attention result is a sequence of context vectors

$$\text{Atten}(Q, K, V) = \text{softmax}\left(\frac{Q \cdot K^T}{\sqrt{d_{\text{key}}}}\right) \cdot V, \tag{30}$$

where the softmax is applied over each of the rows of the matrix (over the computation results of every vector q_i for $i = 0, \cdots, \ell^Q_{\text{seq}} - 1$), and we can see the requirement that $d_{\text{query}} = d_{\text{key}}$ and $\ell^K_{\text{seq}} = \ell^V_{\text{seq}}$ for the matrix multiplication to be well defined. The dimension of the output vectors in attention mechanism is usually called head dimension $d_{\text{head}} = d_{\text{value}}$.



Drawing 1: Scaled dot-product attention

We call the matrix $Q \cdot K^T$ attention weights, which are calculated using the query and key vectors: the attention weight a_{ij} from token i to token j is the dot product between q_i and k_j. The attention weights are divided by the square root of the dimension of the key vectors, $\sqrt{d_{\text{key}}}$, which stabilizes gradients during training, and passed through a softmax which normalizes the weights. The fact that W^Q and W^K are different matrices allows attention to be non-symmetric: if token i attends to token j ($q_i \cdot k_j^T$ is large), this does not necessarily mean that token j will attend to token i ($q_j \cdot k_i^T$ could be small).

For attention mechanism in the encoder which is just self-attention, the input matrix X with dimension $\ell_{\text{seq}} \times d_{\text{emb}}$ are the same for query, key and value. Usually we also want these query, key matrices to be square, then $d_{\text{query}} = d_{\text{key}} = d_{\text{emb}}$. If the attention head is used in a cross-attention fashion, then usually $X^Q \neq X^K = X^V$. It is theoretically possible for all three to be different, but that is rarely the case in practice.

Multiheaded attention

One set of $\left(W^Q, W^K, W^V\right)$ matrices is called an attention head, and each layer in a transformer model has multiple attention heads. While each attention head attends to the tokens that are relevant to each token, multiple attention heads allow the model to do this

for different definitions of "relevance". The computations for each attention head can be performed in parallel, which allows for fast processing on GPUs. The outputs for the attention layer are concatenated to pass into the feed-forward neural network layers.

$$\text{MultiAtten}\{Q, K, V\} = \text{Concat}_{i=0}^{n_{\text{head}}}[\text{Atten}(Q_i, K_i, V_i)] \cdot W^O. \tag{31}$$

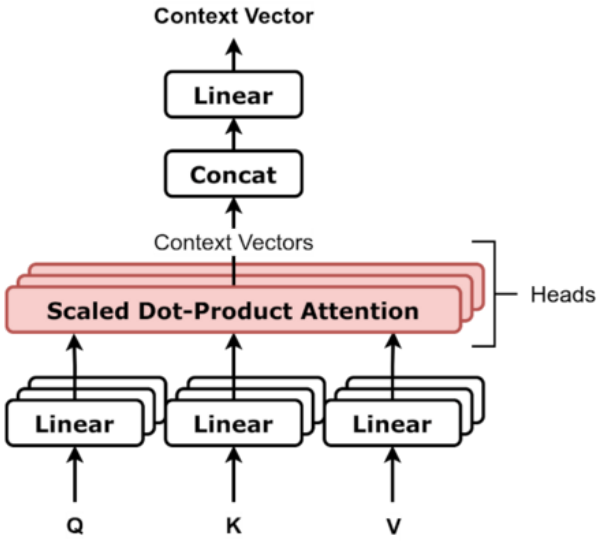


Figure 14: Multiheaded attention

It is theoretically possible for each attention head to have a different head dimension d_{head}, but that is rarely the case in practice. As an example, in the smallest GPT-2 model, there are only self-attention mechanisms. It has the following dimensions: $d_{\text{emb}} = 768, n_{\text{head}} = 12, d_{\text{head}} = 64$. We observe that $d_{\text{emb}} = n_{\text{head}} \times d_{\text{head}}$, so its output projection matrix $W^O \in \mathbb{R}^{(n_{\text{head}} \times d_{\text{head}}) \times d_{\text{emb}}}$ is a square matrix.

2.3.5. Decoder

Each decoder layer contains two attention sublayers: (1) cross-attention for incorporating the output of encoder (contextualized input token representations), and (2) self-attention for "mixing" information among the input tokens to the decoder (i.e. the tokens generated so far during inference time).

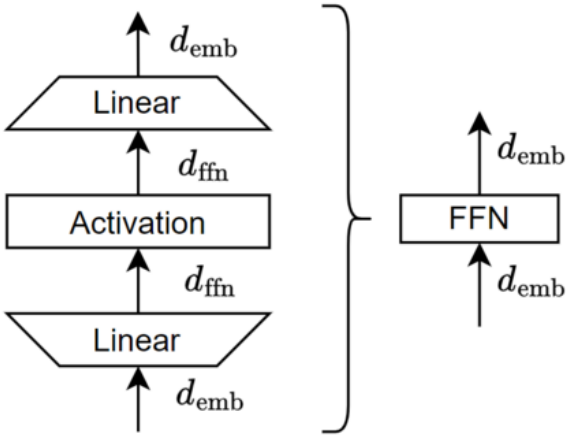
2.3.6. Feedforward network

The feedforward network (FFN) modules in a Transformer are 2-layered multilayer perceptrons:

$$\text{FFN}(x) = f\big(x \cdot W^{(1)} + b^{(1)}\big) \cdot W^{(2)} + b^{(2)}, \tag{32}$$

which is basically the same as Eq. (21) but with two linear layers. The number of neurons in the middle layer is called intermediate size (GPT), filter size (BERT), or feedforward

size (BERT). It is typically larger than the embedding size. For example, in both GPT-2 series and BERT series, the intermediate size of a model is 4 times its embedding size: $d_{\text{ffn}} = 4d_{\text{emb}}$.



2.3.7. Un-embedding

An un-embedding layer is almost the reverse of an embedding layer. Whereas an embedding layer converts a token into a vector, an un-embedding layer converts a vector into a probability distribution over tokens. The un-embedding layer is a linear-softmax layer:

$$\text{UnEmbed}(x) = \text{softmax}(x \cdot W + b), \tag{33}$$

where the matrix W has the shape $d_{\text{emb}} \times n_{\text{voca}}$, sometimes required to be the transpose of the embedding matrix M (this is called weight tying).

3. Coding

[1] W.-Y. Liu, S.-J. Dong, Y.-J. Han, G.-C. Guo, and L. He, *Gradient Optimization of Finite Projected Entangled Pair States*, Phys. Rev. B **95**, 195154 (2017).

[2] R. Rende, L. L. Viteritti, L. Bardone, F. Becca, and S. Goldt, *A Simple Linear Algebra Identity to Optimize Large-Scale Neural Network Quantum States* (2023).

[3] L. L. Viteritti, R. Rende, A. Parola, S. Goldt, and F. Becca, *Transformer Wave Function for the Shastry-Sutherland Model: Emergence of a Spin-Liquid Phase* (2024).

[4] R. Rende, S. Goldt, F. Becca, and L. L. Viteritti, *Fine-Tuning Neural Network Quantum States* (2024).

[5] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, *Attention Is All You Need* (2023).

[6] G. Carleo and M. Troyer, *Solving the Quantum Many-Body Problem with Artificial Neural Networks*, Science **355**, 602 (2017).

[7] F. Vicentini, D. Hofmann, A. Szabó, D. Wu, C. Roth, C. Giuliani, G. Pescia, J. Nys, V. Vargas-Calderón, N. Astrakhantsev, and G. Carleo, *NetKet 3: Machine Learning Toolbox for Many-Body Quantum Systems*, SciPost Phys. Codebases 7 (2022).

[8] F. Vicentini, D. Hofmann, A. Szabó, D. Wu, C. Roth, C. Giuliani, G. Pescia, J. Nys, V. Vargas-Calderón, N. Astrakhantsev, and G. Carleo, *Codebase Release 3.4 for NetKet*, SciPost Phys. Codebases 7 (2022).

[9] *Tutorial 6 (JAX): Transformers and Multi-Head Attention — UvA DL Notebooks v1.2 Documentation — Uvadlc-Notebooks.Readthedocs.Io* (https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/JAX/tutorial6/Transformers_and_MHAttention.html, n.d.).

[10] *Tutorial 15: Vision Transformers — UvA DL Notebooks v1.2 Documentation — Uvadlc-Notebooks.Readthedocs.Io* (https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial15/Vision_Transformer.html, n.d.).

[11] *Ground-State: J1-J2 Model — NetKet — Netket.Readthedocs.Io* (https://netket.readthedocs.io/en/latest/tutorials/gs-j1j2.html, n.d.).

[12] *NetKet - The Machine Learning Toolbox for Quantum Physics — Netket.Org* (https://www.netket.org/, n.d.).