Quantum AlgoLab Summer School 2024: Near-Term Quantum Algorithms

Simulating Quantum Many-Body Systems with Language Models

Stef Czischek (uOttawa)

June 12, 2024

I. Introduction

→ see slides

II. Qubit Systems and Measurements

Qubit Systems

We consider qubits as basic elements for quantum computation and quantum simulation. Analogous to a classical bit, a qubit can take two possible states. We describe the qubit states as vectors in the complex space $\mathbb{C}^2$, which is the single-qubit Hilbert space. The two basis states of a single qubit can be denoted as

$$|\uparrow\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \qquad |\downarrow\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix},$$

where we use the bra-ket notation, $\langle\uparrow| = |\uparrow\rangle^\dagger$.

The state vector of a qubit is given as a weighted superposition of the basis states,

$$|\psi\rangle = \alpha|\uparrow\rangle + \beta|\downarrow\rangle,$$

with amplitudes $\alpha, \beta \in \mathbb{C}$. In the following we assume that the state vectors are normalized,

$$\langle\psi|\psi\rangle = 1 \iff |\alpha|^2 + |\beta|^2 = 1.$$

In this case we can interpret the squared amplitudes as probabilities to find the qubit in each basis state,

$$P(\uparrow) = |\alpha|^2, \qquad P(\downarrow) = |\beta|^2.$$

If we consider a system of multiple qubits, its state vector is given by the tensor product of the individual single-qubit state vectors,

$$|\psi\rangle = |\psi\rangle_1 \otimes |\psi\rangle_2 \otimes \ldots \otimes |\psi\rangle_N \in \mathbb{C}^{2^N}.$$

($|\psi\rangle_1$: single-qubit state vector; $\otimes$: tensor product)

The tensor product is defined as

$$\vec{v} \otimes \vec{w} = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} \otimes \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} v_1 w_1 \\ v_1 w_2 \\ v_2 w_1 \\ v_2 w_2 \end{pmatrix}.$$

The basis states of the multiple qubit system are then given by the tensor products of all possible single-qubit basis state combinations,

$$|\vec{\sigma}\rangle \in \{|\downarrow\downarrow\downarrow\ldots\downarrow\rangle, |\downarrow\downarrow\downarrow\ldots\downarrow\uparrow\rangle, \ldots, |\uparrow\uparrow\uparrow\ldots\uparrow\rangle\}.$$

($|\downarrow\rangle = |\downarrow\rangle_1 \otimes |\downarrow\rangle_2 \otimes \ldots \otimes |\downarrow\rangle_N$)

We can thus write the system state vector as weighted sum over all basis states,

$$|\psi\rangle = \sum_{\{\vec{\sigma}\}} \psi(\vec{\sigma}) |\vec{\sigma}\rangle,$$

(sum over all basis states)

with wave function amplitudes $\psi(\vec{\sigma}) = \langle \vec{\sigma} | \psi \rangle$.

Projective Measurements

Information about a given quantum state can be obtained via measuring observables $\hat{O}$. These observables are operators that act on states in the Hilbert space and thus can be expressed as Hermitian

matrices.

Since the quantum state is a probabilistic superposition of basis states, also the measurement outcomes follow a probabilistic distribution. The possible measurement outcomes are given by the eigenvalues λ_i of the observable $\hat{O}$. The probability to obtain a specific measurement outcome, $P(\lambda_i)$, is given by

$$P(\lambda_i) = \langle \psi | \hat{\pi}_i | \psi \rangle,$$

according to the Born rule. Here $\hat{\pi}_i = |\lambda_i\rangle\langle\lambda_i|$ is a projector on the eigenstate $|\lambda_i\rangle$ corresponding to λ_i, $\hat{O}|\lambda_i\rangle = \lambda_i|\lambda_i\rangle$.

In the cases we consider in this session, $P(\lambda_i)$ corresponds to the squared wavefunction,

$$P(\lambda_i) = \langle \psi | \lambda_i \rangle \langle \lambda_i | \psi \rangle = |\psi(\lambda_i)|^2.$$

Once we obtain a measurement outcome λ_i, the qubit system is projected onto the corresponding eigenstate $|\lambda_i\rangle$. This is known as the collapse of the wavefunction, as the system state is fully determined. By re-preparing the target quantum state and performing multiple measurements, we obtain the operator expectation value

$$\langle \hat{O} \rangle = \sum_i \lambda_i P(\lambda_i) = \langle \psi | \hat{O} | \psi \rangle.$$

By performing measurements in an informationally complete set of observables, full information about the quantum state can be obtained. One such set can be generated with the single-qubit Pauli operators,

$$\hat{\sigma}_i^x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad \hat{\sigma}_i^y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \quad \hat{\sigma}_i^z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.$$

By measuring Pauli strings $\hat{\sigma}_1^{\alpha_1} \otimes \hat{\sigma}_2^{\alpha_2} \otimes \ldots \otimes \hat{\sigma}_N^{\alpha_N}$ with all possible combinations $\alpha_i \in \{x, y, z\}$, the wavefunction amplitudes of a quantum state can be fully determined. However, this requires measurements in 3^N bases and exponentially many measurements are required in each basis to estimate the probabilistic measurement outcomes with sufficient accuracy. Furthermore, to store the quantum state, we need to store 2^N complex wavefunction amplitudes. This all makes a complete determination of an experimentally prepared quantum state infeasible with classical computers. Instead of storing 2^N wavefunction amplitudes, we aim to find a function $\Psi(\vec{\sigma})$ that gives us the amplitude of a given basis state $\vec{\sigma}$ as output and is characterized by a polynomial amount of parameters. We will now see how we can tackle this task with language models.

III. Language Models

Language models are probabilistic models of a natural language and are used for speech recognition, machine translation, text generation and interpretation, handwriting recognition, and much more. Nowadays, language models based on artificial neural networks show remarkable performances for these tasks. The main ingredient of language models is the possibility to deal with sequential data, where they generalize to arbitrary sequence lengths. Language models further need to be flexible about the sequential order, since the word order can vary. As an example, the following two sentences have the same meaning, but the word order is different:

"On February 2nd, I swam in the lake."

"I swam in the lake on February 2nd."

To deal with these properties, language models consider each sequence element individually and introduce a dependence on previous elements as a memory. This technique is called autoregression and assumes that each sequence element depends on previous elements in the sequence. While this is true for speech recognition or text generation, a different technique

needs to be chosen for text translation. For our applications in quantum state representation, we will need autoregressive language models. Let's introduce two such models in detail.

III. 1 Recurrent Neural Networks

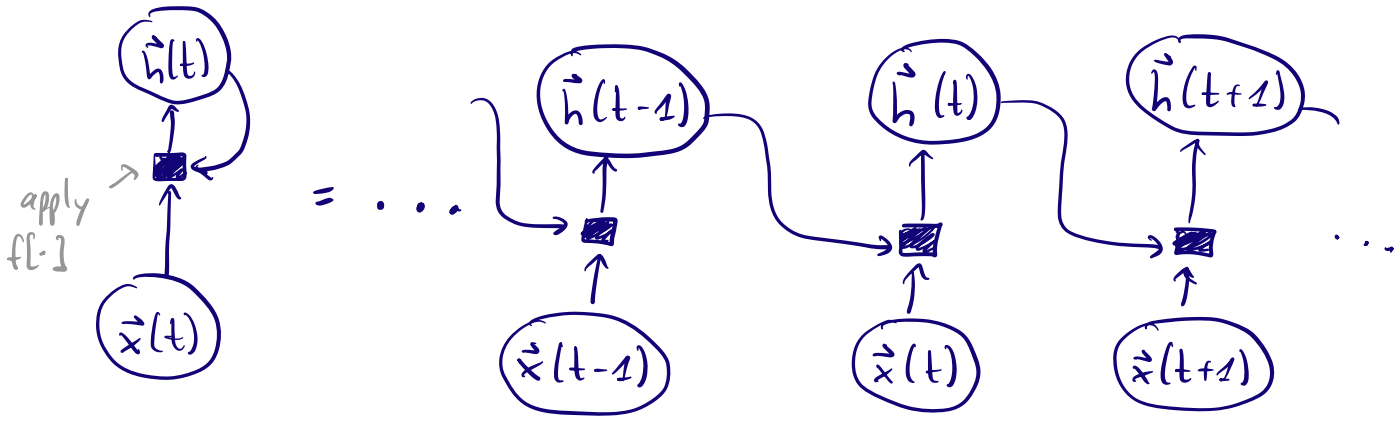
A recurrent neural network (RNN) receives each element $\vec{x}(t)$ of the sequence as individual input and processes it to generate an internal state $\vec{h}(t)$. Here, t denotes the position in the sequence. The sequence of states $\vec{h}(t)$ is calculated via repeated application of a non-linear function $f[\cdot]$,

$$\vec{h}(t) = f[\vec{h}(t-1), \vec{x}(t); \mathcal{W}].$$

This function depends on the sequence element $\vec{x}(t)$, the previous state $\vec{h}(t-1)$, and a set of variational parameters $\mathcal{W}$ which define the non-linear transformation. These parameters are tuned when the RNN is trained to achieve a desired behaviour. The same function $f[\cdot]$ with the same parameters $\mathcal{W}$ is applied recurrently to all sequence elements. This weight sharing allows the model to be applied to arbitrary sequence lengths. We can thus write

$$\vec{h}(t) = f[f[\vec{h}(t-2), \vec{x}(t-1); \mathcal{W}], \vec{x}(t); \mathcal{W}] = \dots,$$

which we can visualize as



This recurrent setup comes with two major benefits:

1. Regardless of the sequence length, the trained model always has the same input size, as it is specified in terms of transition from one to another state, rather than a variable-length history of states.
2. It is possible to use the same transition function with the same parameters at every sequence step.

The memory about previous sequence elements is fully encoded in the fixed-size state $\vec{h}(t)$, so it must be lossy when applied to sequences of arbitrary length. The model needs to learn to extract the relevant information from the sequence. The size of $\vec{h}(t)$ can be chosen freely depending on the task. A larger size, often called "more hidden parameters", increases the amount of encodable information and leads to more variational parameters and thus a higher network expressivity. It, however, also increases the computational cost for running and

training the RNN.

In order to apply the RNN as a language model, we define a network output $\vec{o}(t)$ at each sequence step, which we obtain from the hidden state $\vec{h}(t)$. We thus map an input sequence to an output sequence of the same length,

$$\vec{h}(t) = f[\vec{h}(t-1), \vec{x}(t); W],$$

$$\vec{o}(t) = g[\vec{h}(t); V],$$

where we introduce another non-linear function $g[\cdot]$ depending on variational parameters V. By choosing

$$g[\vec{h}(t); V] = \text{softmax}[V\,\vec{h}(t)],$$

$$\text{softmax}(x_i) = \frac{\exp[x_i]}{\sum_j \exp[x_j]},$$

the output corresponds to a probability distribution over all possible outcomes at each step in the sequence as it is non-negative and normalized. Due to the recurrent setup, this corresponds to the probability of each possible output conditioned on all previous inputs,

$$\vec{o}(t) = P[\vec{y}(t) \mid \vec{x}(t), \vec{x}(t-1), \dots, \vec{x}(0)].$$

A single output value $\hat{\vec{y}}(t)$ can then be sampled from this conditional distribution. In autoregressive models, where each sequence

element depends on the previous elements, the generated output $\hat{\vec{y}}(t)$ is used as input in the next iteration, $\vec{x}(t+1) = \hat{\vec{y}}(t)$. We thus get

$$\vec{o}(t) = P\left[\vec{y}(t) \,|\, \hat{\vec{y}}(t-1), \hat{\vec{y}}(t-2), \ldots, \hat{\vec{y}}(0)\right],$$

from which we can calculate the probability for an entire sequence of length τ as

$$P\left[\hat{\vec{y}}(0), \hat{\vec{y}}(1), \ldots, \hat{\vec{y}}(\tau-1)\right] = \prod_{t=1}^{\tau-1} P\left[\vec{y}(t) \,|\, \hat{\vec{y}}(t-1), \hat{\vec{y}}(t-2), \ldots, \hat{\vec{y}}(0)\right]$$
$$= \prod_{t=1}^{\tau-1} P\left[\vec{y}(t) \,|\, \hat{\vec{y}}_{<t}\right].$$

We call this the probability distribution encoded in the RNN.

In summary, an RNN can be trained to efficiently encode a probability distribution over a sequence. By running the trained RNN, we can further generate samples from this distribution with low computational cost. The performance of the RNN highly depends on the choice of the non-linear transformation $f[\cdot]$. While the vanilla RNN uses

$$f\left[\vec{h}, \vec{x}; W\right] = \tanh\left[W_h \vec{h} + W_x \vec{x}\right],$$

more stable and commonly applied architectures use a gated transformation that generates self-loops. For quantum state representation, gated recurrent units (GRUs) have proven very powerful. Those generate the state $\vec{h}(t)$ via the following transformations:

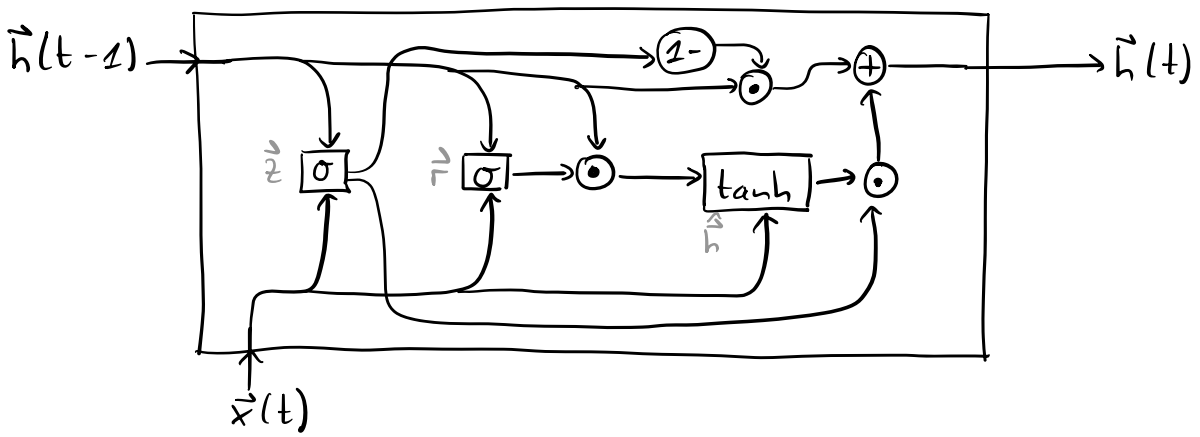
$$\vec{z}(t) = \sigma\left[W_z \vec{x}(t) + U_z \vec{h}(t-1)\right],$$

$$\vec{r}(t) = \sigma\left[W_r \vec{x}(t) + U_r \vec{h}(t-1)\right],$$

$$\hat{\vec{h}}(t) = \tanh\left[W_h \vec{x}(t) + U_h (\vec{r}(t) \odot \vec{h}(t-1))\right],$$

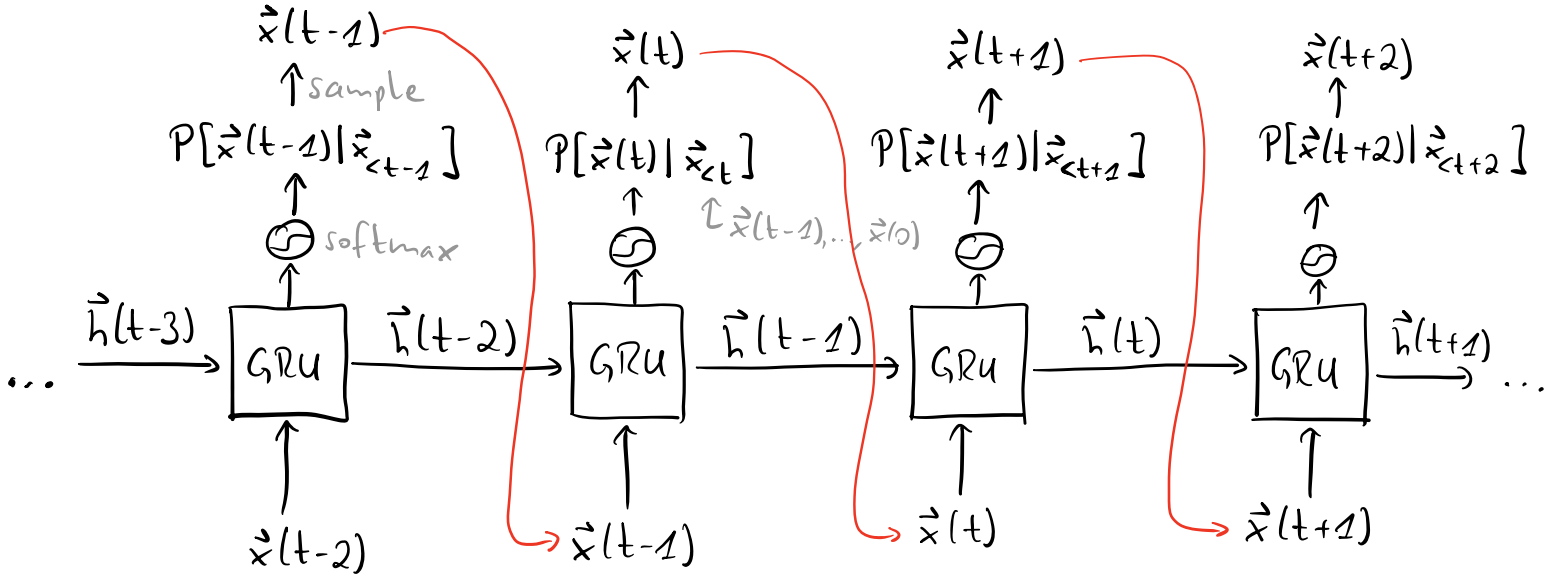
$$\vec{h}(t) = (1 - \vec{z}(t)) \odot \vec{h}(t-1) + \vec{z}(t) \odot \hat{\vec{h}}(t),$$

with the sigmoid function $\sigma[x] = \frac{1}{1+\exp[-x]}$ and the element-wise product $\odot$. We can visualize it as



With this, we can visualize our final RNN for encoding a probability distribution:

$$P[\vec{x}(0), \ldots, \vec{x}(\tau-1)] = \prod_{t=0}^{\tau-1} P[\vec{x}(t) \mid \vec{x}_{<t}]$$



III.2 Transformer Models

Transformer models are language models that have recently outperformed RNNs and led to novel technologies. They can be applied to sequential data similarly to RNNs, but do not have a recurrent behaviour. Instead, transformer models are based on a self-attention mechanism that provides access to all elements in the sequence and enables a dynamical highlighting of salient information. Initially, the transformer model was introduced consisting of an encoder and a decoder part, but we will only consider the encoder in the following.

The transformer model always takes the full sequence as input. Each input element is first embedded in a vector with a tunable embedding dimension. This is done via a linear transformation using trainable parameters that are shared over the entire sequence. As the sequence is sent in all at once, no information about the order of the elements is captured by default. Therefore, an extra layer is added to encode the position of each element in the sequence, which is done by adding a unique value to each element. This processed input information is then passed into the transformer cell, where the attention mechanism is used to

share the information from the individual inputs.

The self-attention mechanism projects each embedded sequence element $\vec{x}(t)$ to a query vector $\vec{q}(t)$, a key vector $\vec{k}(t)$, and a value vector $\vec{v}(t)$,

$$\vec{q}(t) = \sum_l W^q_{t,l} x_l(t), \quad \vec{k}(t) = \sum_l W^k_{t,l} x_l(t), \quad \vec{v}(t) = \sum_l W^v_{t,l} x_l(t),$$

with trainable weight matrices $W^{q/k/v}$. Query, key, and value vectors of all input elements can be summarized in the corresponding matrices,

$$Q = \begin{bmatrix} \vec{q}_0 \\ \vdots \\ \vec{q}_{\tau-1} \end{bmatrix}, \quad K = \begin{bmatrix} \vec{k}_0 \\ \vdots \\ \vec{k}_{\tau-1} \end{bmatrix}, \quad V = \begin{bmatrix} \vec{v}_0 \\ \vdots \\ \vec{v}_{\tau-1} \end{bmatrix},$$

with sequence length τ.

The attention mechanism then maps the queries and key-value pairs to an output for each sequence element, allowing for highlighting of connections to sequence elements with important information. For each sequence element $\vec{x}(t)$, the dot product of the query vector $\vec{q}(t)$ with the key vector $\vec{k}(\tilde{t})$ for all $\tilde{t} \in \{0, \ldots, \tau-1\}$ is evaluated. In an autoregressive model, a masking term

$$m_{t,t'} = \begin{cases} 0 & \text{if } t \leq t', \\ -\infty & \text{otherwise,} \end{cases}$$

is added to ensure that the self-attention only considers previous elements in the sequence and does not look at later elements that still need to be determined. Applying a softmax activation function to all

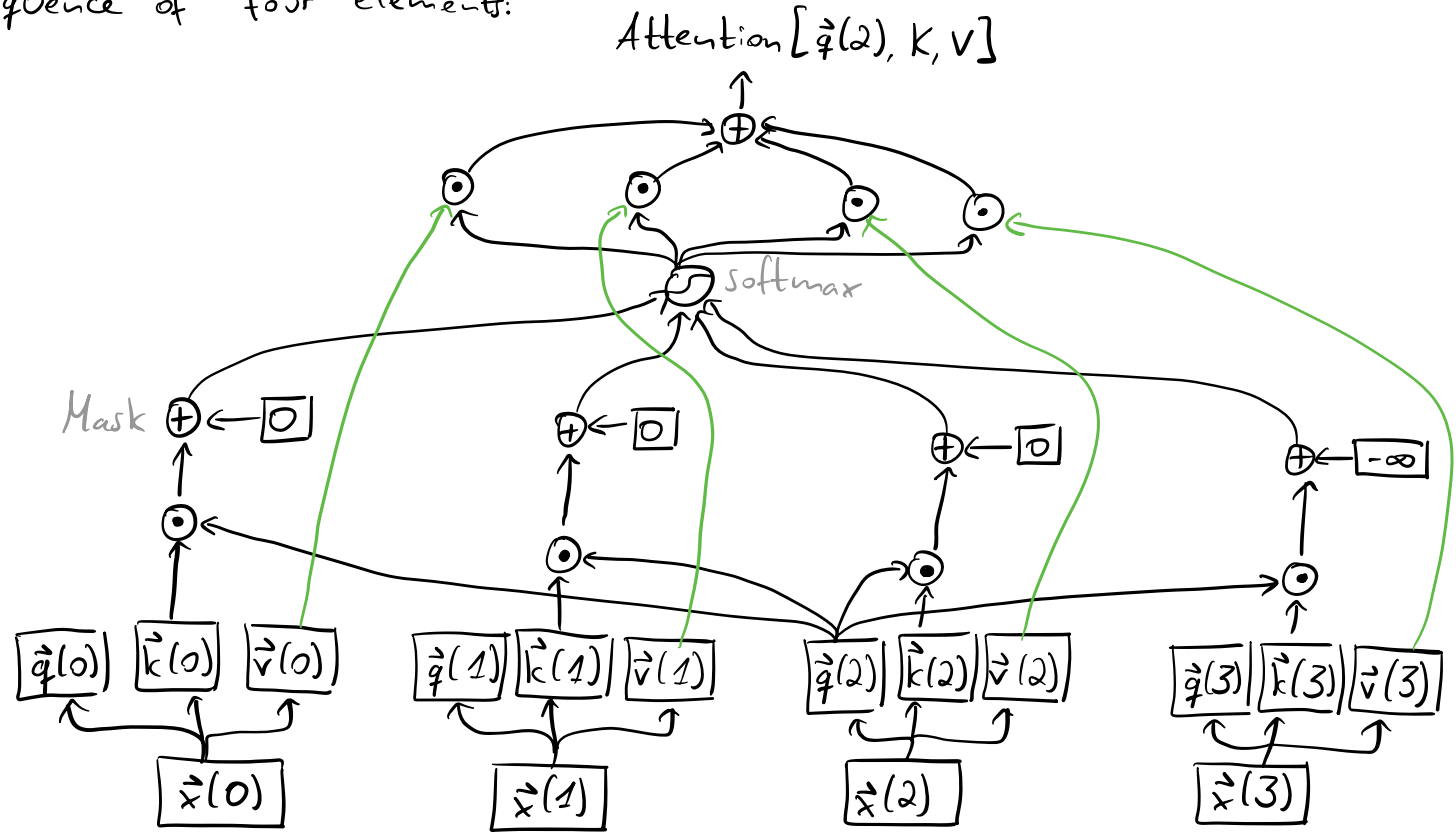
signals after adding the mask ensures that the contributions of all later sequence elements with $m_{t,t'} = -\infty$ are driven to zero. Then the dot product of each signal with the corresponding value vector $\vec{v}(t)$ is evaluated and all signals are summed to generate the output of the attention mechanism.

The complete formalism can be summarized as

$$\text{Attention}(Q, K, V) = \text{softmax}\left(QK^T + M\right)V,$$

with mask matrix M. To improve the performance, multiple attention heads can be used. This multi-headed attention projects each query, key, and value vector to multiple vectors using individually trainable projection matrices.

We can visualize the attention mechanism applied to the third element in a sequence of four elements:



Effectively, the attention mechanism provides connections between all input elements which are trainable to highlight connections between strongly correlated sequence elements. A mask is further applied to obtain an autoregressive behaviour, where each element only depends on previous elements in the sequence and future elements have not yet been determined and are thus masked out.

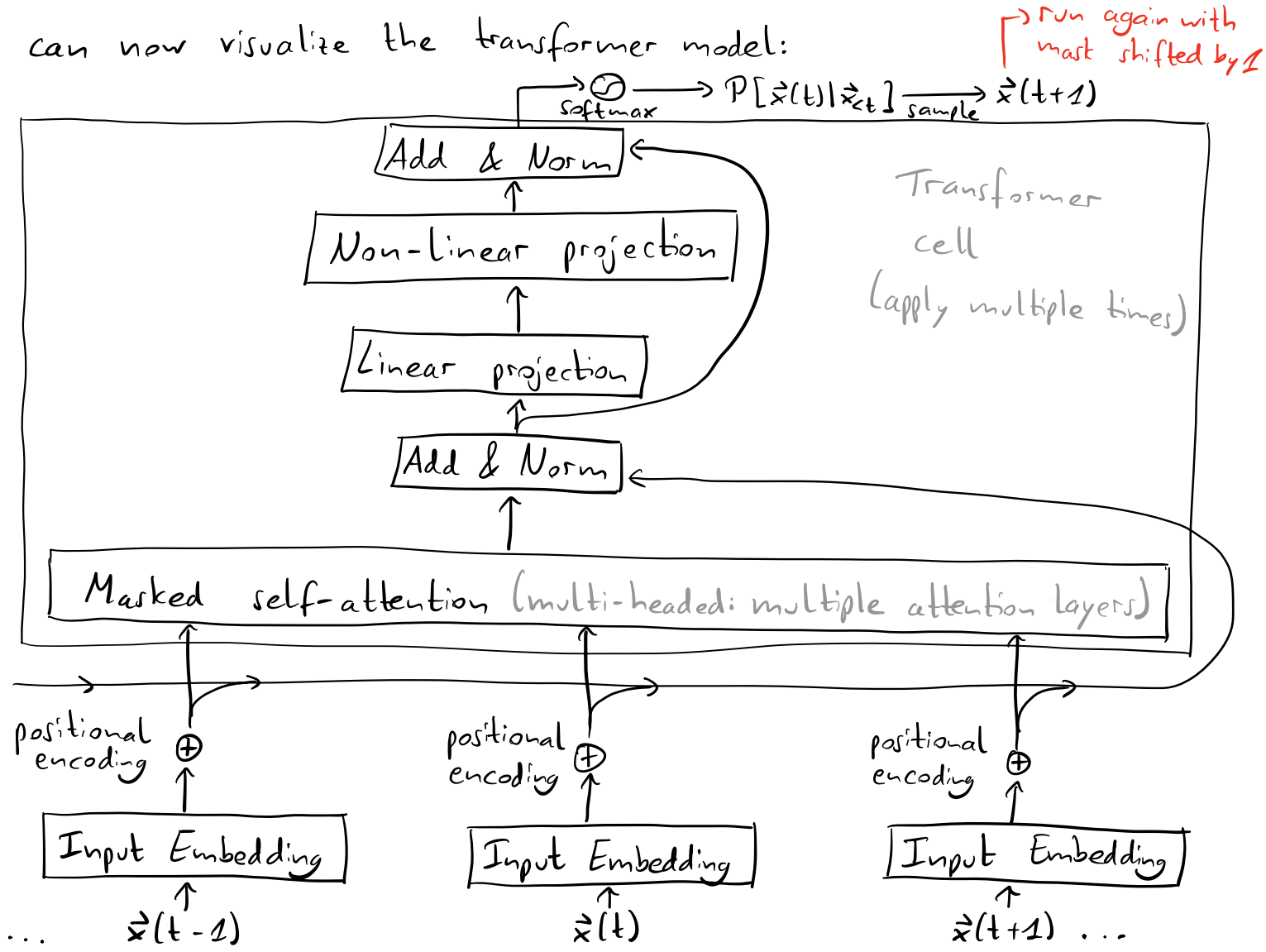
The attention mechanism provides an output vector, which is then added to the embedded input state and normalized. The result is progressed through a linear and a non-linear projection, which both provide trainable parameters. The output is again added to the output from the self-attention cell and the sum is normalized. The entire transformer cell, including the self-attention mechanism and the two projections, can be applied multiple times independently to improve the network expressivity.

To encode probability distributions as in the RNN, a softmax activation function is applied to the output of the transformer cell. Similar to the RNN, this leads to a probability distribution conditioned on all previous sequence elements, from which the next sequence element can

be sampled.

This allows us to run the transformer model in the same way as the RNN to encode a probability distribution underlying sequential data, from which we can efficiently draw samples by running the network. The difference between the two models lies in the recurrent setup versus the self-attention mechanism and thus influences the performance of the encoded memory. The transformer is more powerful in capturing long-range correlations.

We can now visualize the transformer model:



IV. Quantum State Representation with Language Models

Stoquastic Hamiltonians

The next step is to combine the two introduced technologies and find an efficient representation of quantum states using language models. Here, "efficient" refers to a computational cost that scales at most polynomially with the quantum system size. Within this session we will focus on finding ground state representations for a given Hamiltonian. The representation of excited states goes beyond the scope of this lecture.

We make another simplification by only considering stoquastic Hamiltonians. These are defined as operators which have only non-positive entries in the off-diagonal elements,

diagonal elements ↙ ↘ off-diagonal elements

$$\hat{H} = \begin{bmatrix} d_{11} & -|o_{12}| & -|o_{13}| & \dots & -|o_{12^N}| \\ -|o_{21}| & d_{22} & -|o_{23}| & \dots & -|o_{22^N}| \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -|o_{2^N 1}| & -|o_{2^N 2}| & -|o_{2^N 3}| & \dots & d_{2^N 2^N} \end{bmatrix}.$$

It can be proven mathematically that ground states of such Hamiltonians have only real-valued amplitudes. These quantum states are thus fully described by the squared wavefunction amplitudes $|\Psi(\vec{\sigma})|^2$, which correspond to the probabilities of finding the qubit system in each of the basis states

$\vec{\sigma}$. These probabilities can be determined from measurements of the Pauli string in the computational basis in which $\hat{H}$ is stoquastic. Here we choose the computational basis in the z-direction and thus consider

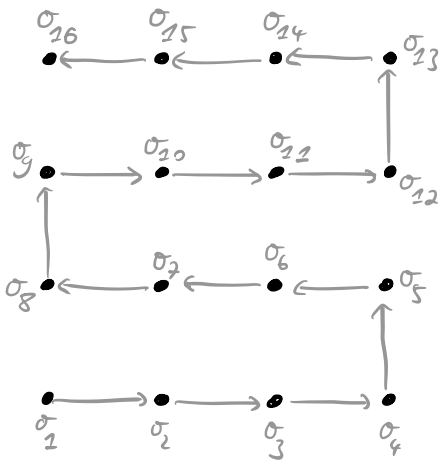
$$\langle \hat{\sigma}^z \rangle = \langle \sigma_1^z \otimes \sigma_2^z \otimes \dots \otimes \sigma_N^z \rangle,$$

since $\hat{\sigma}_i^z \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, $\hat{\sigma}_i^z \begin{bmatrix} 0 \\ 1 \end{bmatrix} = -\begin{bmatrix} 0 \\ 1 \end{bmatrix}$.

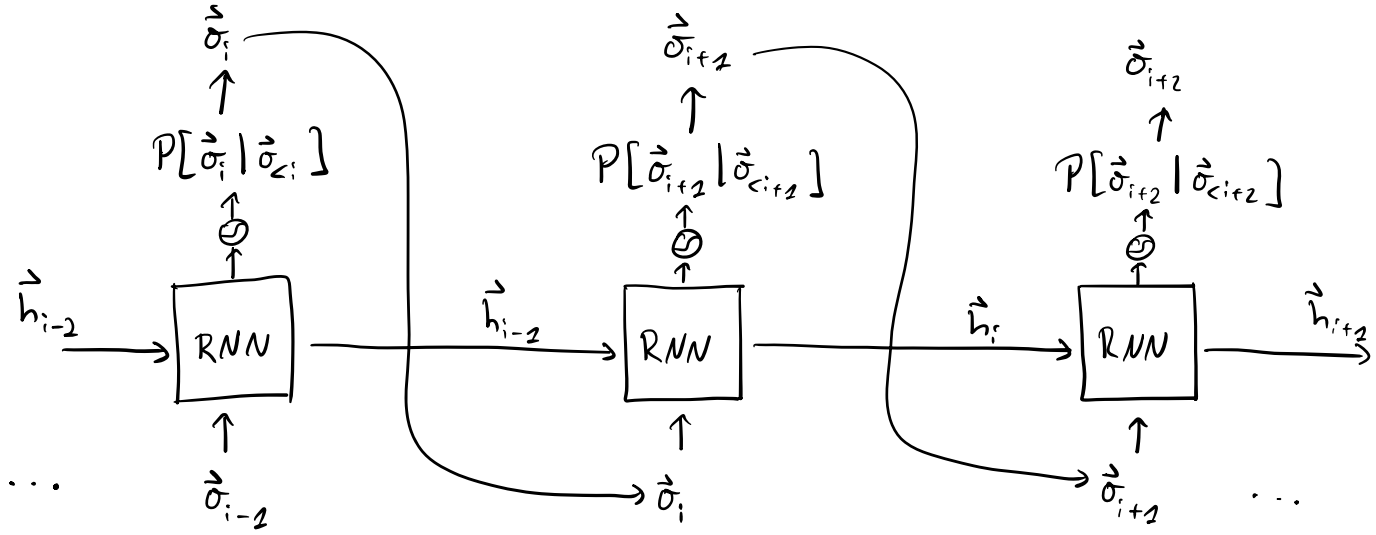
Wavefunction representation with language models

Since language models can approximate any probability distribution, we can use them to reconstruct quantum states and generate measurement data to evaluate operator expectation values. This is straight forward for ground states of stoquastic Hamiltonians.

To use an RNN as a wavefunction ansatz for representing quantum states, we consider the quantum system as a sequence of qubits. In one dimension we define this sequence from left to right, while we snake through a two-dimensional lattice.



We then sample the state of one qubit at a time and use it as the input in the next RNN iteration. The hidden state propagation captures correlations in the qubit system by carrying information about previously sampled qubit configurations.

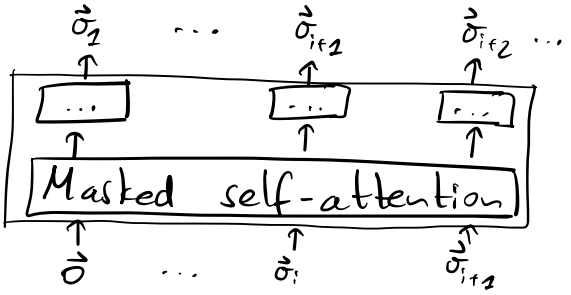


We then interpret the probability distribution encoded in the RNN as the squared wavefunction amplitude of the represented quantum state,

$$P[\vec{\sigma}; \mathcal{W}] = |\langle \vec{\sigma} | \psi_{\mathcal{W}} \rangle|^2 = |\psi_{\mathcal{W}}(\vec{\sigma})|^2.$$

This ansatz can model the complete information of ground states in stoquastic Hamiltonians.

Transformer models can be used similarly as a wavefunction ansatz by using a masked self-attention layer to ensure autoregressivity.



Evaluating operators

When we interpret the probability distribution encoded in a language model as squared wavefunction amplitude, generated samples correspond to outcomes of projective measurements in the computational basis, $\langle \hat{\sigma}^z \rangle$ in our case. They can thus be used to estimate expectation values of observables $\hat{O}$,

$$\langle \hat{O} \rangle = \langle \Psi_W | \hat{O} | \Psi_W \rangle$$

$$= \sum_{\{\vec{\sigma}, \vec{\sigma}'\}} \Psi_W^*(\vec{\sigma}) \Psi_W(\vec{\sigma}') \langle \vec{\sigma} | \hat{O} | \vec{\sigma}' \rangle .$$

For diagonal observables, $\hat{O}_{diag}$, the expectation value can be evaluated directly from the samples since

$$\langle \vec{\sigma} | \hat{O}_{diag} | \vec{\sigma}' \rangle = \hat{O}_{diag}(\vec{\sigma}) \delta_{\vec{\sigma},\vec{\sigma}'} \quad \left(= \hat{O}_{diag} | \vec{\sigma} \rangle \right)$$

$$\Rightarrow \langle \hat{O}_{diag} \rangle = \sum_{\{\vec{\sigma}\}} |\Psi_W(\vec{\sigma})|^2 \hat{O}_{diag}(\vec{\sigma}) \approx \frac{1}{N_S} \sum_{q=1}^{N_S} \hat{O}_{diag}(\vec{\sigma}_q).$$

Here, the index q indicates each of the N_S samples $\vec{\sigma}_q$ drawn from $|\Psi_W(\vec{\sigma})|^2 = P[\vec{\sigma}; W]$.

For general, non-diagonal observables $\hat{O}$, we introduce the local observable

$$\hat{O}_{loc}(\vec{\sigma}; W) = \frac{\langle \vec{\sigma} | \hat{O} | \Psi_W \rangle}{\langle \vec{\sigma} | \Psi_W \rangle} = \sum_{\{\vec{\sigma}'\}} \langle \vec{\sigma} | \hat{O} | \vec{\sigma}' \rangle \frac{\Psi_W(\vec{\sigma}')}{\Psi_W(\vec{\sigma})} .$$

With this, we obtain

$$\langle \hat{O} \rangle = \sum_{\{\vec{\sigma}, \vec{\sigma}'\}} \Psi_W^*(\vec{\sigma}) \Psi_W(\vec{\sigma}') \langle \vec{\sigma} | \hat{O} | \vec{\sigma}' \rangle$$

$$= \sum_{\{\vec{\sigma}\}} |\Psi_W(\vec{\sigma})|^2 \underbrace{\sum_{\{\vec{\sigma}'\}} \langle \vec{\sigma} | \hat{O} | \vec{\sigma}' \rangle \frac{\Psi_W(\vec{\sigma}')}{\Psi_W(\vec{\sigma})}}_{= \hat{O}_{loc}(\vec{\sigma}; W)} \approx \frac{1}{N_S} \sum_{q=1}^{N_S} \hat{O}_{loc}(\vec{\sigma}_q; W).$$

We thus evaluate the local observable on N_s samples and average the outcome. This can be done using the output of the language model since $\psi_N(\vec{\sigma}) = \sqrt{P[\vec{\sigma}; W]}$ is directly provided. However, in general the sum in the local observable still runs over exponentially many terms. In the cases we consider, it can be evaluated efficiently since we only look at Pauli operators acting on single qubits individually. For these cases, only one contribution to the sum in the local observable is non-zero.

For example, for $\langle \hat{\sigma}_i^x \rangle$ we get $\langle \vec{\sigma} | \hat{\sigma}_i^x | \vec{\sigma}' \rangle = \delta_{\vec{\sigma}, \vec{\sigma}_i}$ where $\vec{\sigma}_i$ denotes the basis state $\vec{\sigma}$ where qubit i is flipped.

IV. 1 Hamiltonian-driven training

Now that we know how to represent quantum states with language models and how to evaluate observables using generated samples, there is one thing we have not talked about yet: how to train the language models to represent the ground state of a given Hamiltonian. There exist two ways how we can train the models, which can be applied similarly in the RNN and the transformer model. The first training method we consider is Hamiltonian-driven training, which corresponds to a variational ground state search via energy minimization. For this, at each training

step, we evaluate the energy expectation value $\langle \hat{H} \rangle$ of the considered Hamiltonian using N_s samples $\vec{\sigma}_q$ generated from the language model,

$$\langle \hat{H} \rangle = \sum_{\{\vec{\sigma}\}} |\psi_W(\vec{\sigma})|^2 H_{loc}(\vec{\sigma}) \approx \frac{1}{N_s} \sum_{q=1}^{N_s} H_{loc}(\vec{\sigma}_q),$$

with local energy

$$H_{loc}(\vec{\sigma}) = \frac{\langle \vec{\sigma} | \hat{H} | \psi_W \rangle}{\langle \vec{\sigma} | \psi_W \rangle} = \sum_{\{\vec{\sigma}'\}} \langle \vec{\sigma} | \hat{H} | \vec{\sigma}' \rangle \frac{\psi_W(\vec{\sigma}')}{\psi_W(\vec{\sigma})}.$$

Since the ground state corresponds to the minimum energy state, we want to train the language model such that $\langle \hat{H} \rangle$ is minimized. We do this by updating the variational parameters in the language model to follow the negative gradient of the energy expectation value,

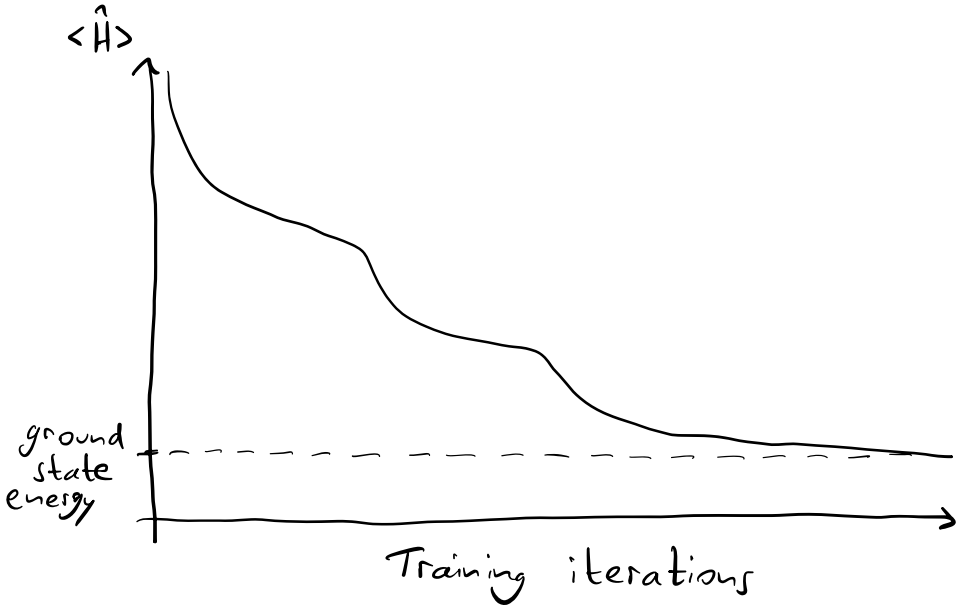
$$W_j \leftarrow W_j - \lambda \frac{\partial \langle \hat{H} \rangle}{\partial W_j},$$

$$\frac{\partial \langle \hat{H} \rangle}{\partial W_j} = \sum_{\{\vec{\sigma}\}} |\psi_W(\vec{\sigma})|^2 \frac{1}{\psi_W^*(\vec{\sigma})} \frac{\partial \psi_W^*(\vec{\sigma})}{\partial W_j} H_{loc}(\vec{\sigma}) + \sum_{\{\vec{\sigma}\}} |\psi_W(\vec{\sigma})|^2 \frac{1}{\psi_W(\vec{\sigma})} \frac{\partial \psi_W(\vec{\sigma})}{\partial W_j} H_{loc}(\vec{\sigma})$$

$$\approx \frac{2}{N_s} \mathrm{Re} \left\{ \sum_{q=1}^{N_s} \frac{\partial \psi_W(\vec{\sigma}_q)}{\partial W_j} \frac{1}{\psi_W(\vec{\sigma}_q)} H_{loc}(\vec{\sigma}_q) \right\}.$$

Here, W_j denotes a single variational parameter and λ is the learning rate (update step size). In each training iteration, all parameters are updated once. The gradient can be evaluated numerically using automatic differentiation algorithms. Once the parameters have been updated, we run the network again to generate a new set of samples from the updated encoded

probability distribution and evaluate again the energy expectation value. After a sufficient amount of training iterations, this method converges to a representation of the ground state of the given Hamiltonian.



IV.2 Data-driven training

The second training approach is data-driven training, where we train the language model on a given set of measurement data generated from the target quantum state. The trained network will approximately represent the state underlying the provided training data.

Assuming we have a projective measurement dataset $\mathcal{D}$ whose data points $\vec{\sigma} \in \mathcal{D}$ follow a distribution $P_{\mathcal{D}}(\vec{\sigma})$, we want the language model to approximate this distribution,

$$P[\vec{\sigma}; \mathcal{W}] \approx P_{\mathcal{D}}(\vec{\sigma}).$$

We use the so-called Kullback-Leibler divergence as a quasi-distance measure between the two distributions,

$$D_{KL}(P[W] \| P_{\mathcal{D}}) = \sum_{\{\vec{\sigma}\}} P_{\mathcal{D}}(\vec{\sigma}) \log\left[\frac{P_{\mathcal{D}}(\vec{\sigma})}{P[\vec{\sigma};W]}\right]$$
$$\approx -S_{\mathcal{D}} - \frac{1}{|\mathcal{D}|}\sum_{\vec{\sigma}\in\mathcal{D}} \log(P[\vec{\sigma};W]).$$

In the last line we approximate $P_{\mathcal{D}}$ with the sum over the data points in $\mathcal{D}$ and introduce the entropy

$$S_{\mathcal{D}} = -\sum_{\{\vec{\sigma}\}} P_{\mathcal{D}}(\vec{\sigma}) \log\left[P_{\mathcal{D}}(\vec{\sigma})\right],$$

which is a constant term independent of the trainable parameters W. We then train the language model to minimize the Kullback-Leibler divergence, where we again follow the negative gradient to update the network parameters,

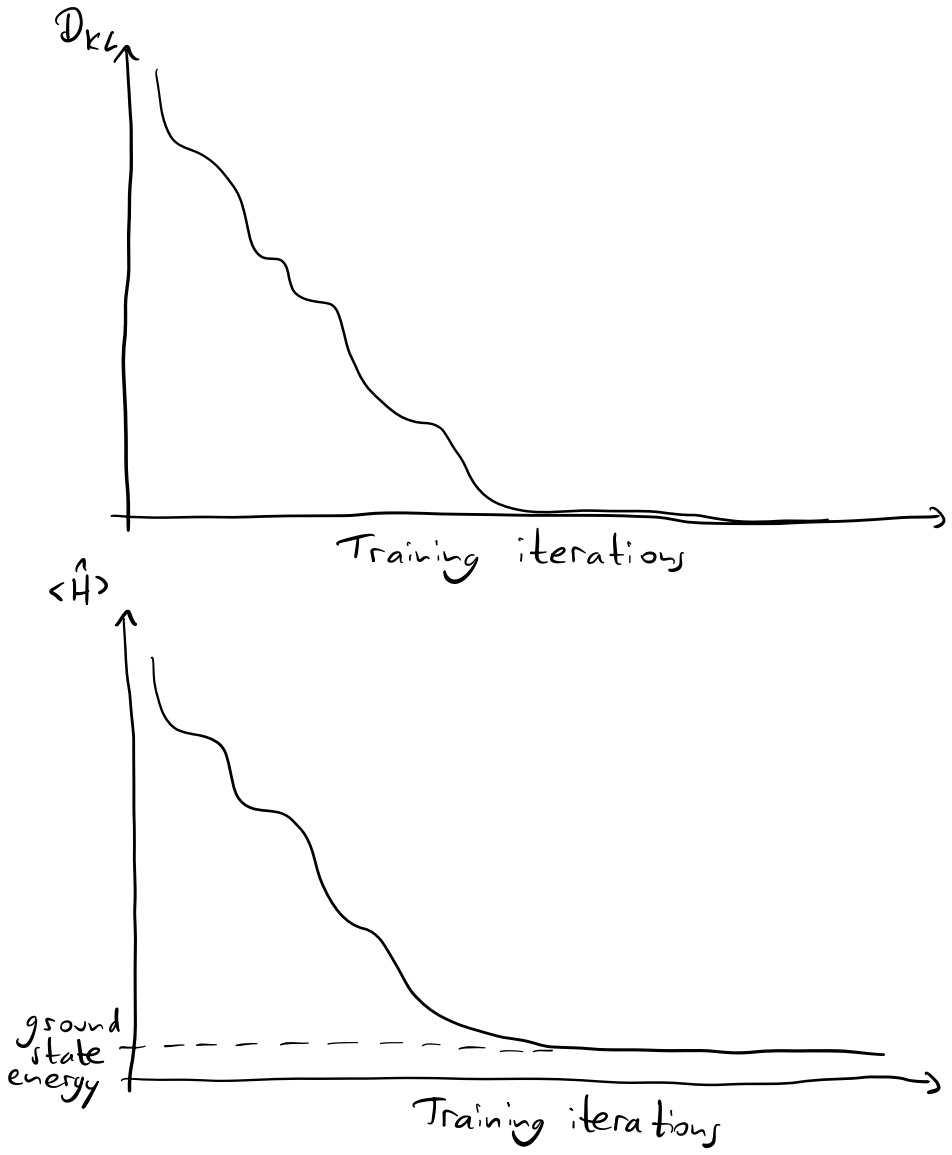
$$W_j \leftarrow W_j - \lambda \frac{\partial D_{KL}(P[W]\|P_{\mathcal{D}})}{\partial W_j},$$

$$\frac{\partial D_{KL}(P[W]\|P_{\mathcal{D}})}{\partial W_j} \approx -\frac{1}{|\mathcal{D}|}\sum_{\vec{\sigma}\in\mathcal{D}} \frac{1}{P[\vec{\sigma};W]} \frac{\partial P[\vec{\sigma};W]}{\partial W_j}.$$

We can evaluate $P[\vec{\sigma};W]$ by running the language model with the fixed input $\vec{\sigma}$. We further obtain the gradient numerically via automatic differentiation algorithms.

By using a training data set generated from the ground state of a

desired Hamiltonian, which can e.g. be produced with an experimental setup, we can train the language model to reconstruct this target ground state.



V. Summary & Outlook

→ see slides

VI. Tutorial

The tutorial was gratefully set up by Faith Oyedemi (uOttawa).

The Github link to access the tutorial will be shared.