# Sequence models

Author: Juvid Aryaman
Last compiled: December 21, 2022

This document contains my personal notes on sequence models.

## 1. Embeddings

Embeddings are tensors. You interact with that tensor by indexing into it. It is often used to store encodings of collections of words. For example:

```
>>> nn.Embedding(vocab_sz, n_hidden)
```

creates a set of `vocab_sz` tensors, each of size `n_hidden`.

A common thing to do is to something like:

```
>>> embedding = nn.Embedding(10, 3)
>>> input = torch.LongTensor([[1,2,4,5],[4,3,2,9]])
>>> embedding(input)

tensor([[[-0.0251, -1.6902,  0.7172],
         [-0.6431,  0.0748,  0.6969],
         [ 1.4970,  1.3448, -0.9685],
         [-0.3677, -2.7265, -0.1685]],

        [[ 1.4970,  1.3448, -0.9685],
         [ 0.4362, -0.4004,  0.9400],
         [-0.6431,  0.0748,  0.6969],
         [ 0.9124, -2.3616,  1.1151]]])
```

so you can see that the input is `[sentence1, sentence2]`, where sentence 1 consists of words `[1,2,4,5]`. As an output, we get the corresponding 3-vectors for each word. So the output is:

```
[[[embedding_word_1,    # length 3 vector
   embedding_word_2,
   embedding_word_4,
   embedding_word_5],

  [embedding_word_4,
   embedding_word_3,
   embedding_word_2,
   embedding_word_9]
]]
```

## 2. Linear layer

Applies a linear transformation to the incoming data: $y = xA^T + b$

```
>>> m = nn.Linear(20, 30)
>>> input = torch.randn(128, 20)
>>> output = m(input)
>>> print(output.size())
torch.Size([128, 30])
```

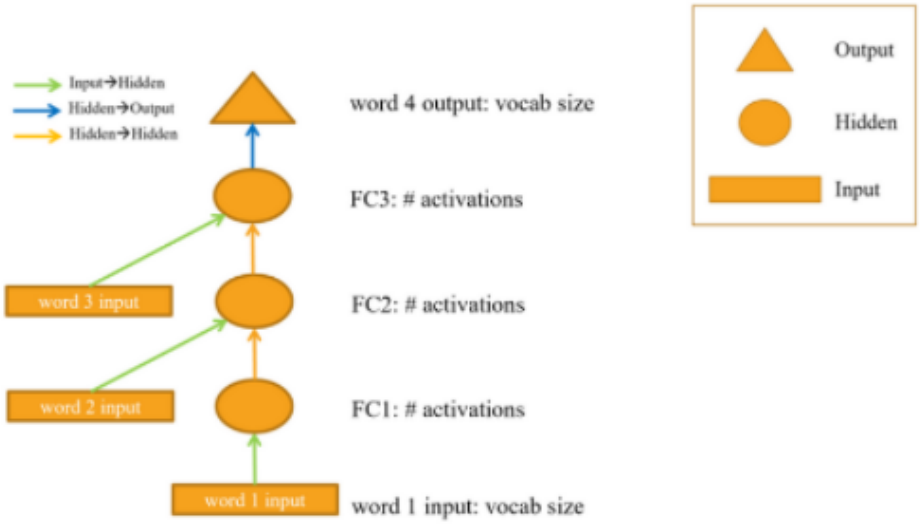


**Figure 1.** Graphical representation of RNN

## 3. Recurrent neural network

Torch, by default, applies a multi-layer Elman RNN. This is defined as applying the following function to each element of the input sequence

$$h_t = \sigma_h(W_h x_t + U_h h_{t-1} + b_h) \tag{3.1}$$
$$y_t = \sigma_y(W_y h_t + b_y) \tag{3.2}$$

where $x_t$ is an input vector, $h_t$ is a hidden layer vector, $y_t$ is an output vector, $W, U, b$ are parameter matrices and vector, $\sigma_h, \sigma_y$ are activation functions. Note that we don't actually retain the hidden state between lines – we throw it away after every complete training example (a line). We will typically initialize the hidden state to be $h_{t=0} = 0$. Within a particular training instance, on a particular line, we may have different maximum values of $t = T$.

For example, in word classification, where we construct a character-level RNN, in each training loop we will

1. Get an input and target tensor
2. Create a zeroed initial hidden state
3. Read each letter in and:
    - Keep hidden state for next letter
    - Feed the previous hidden state $h_{t-1}$ in with the current input $x_t$
4. Compare output at the end of the RNN loop to the target
5. Back-propagate

Then return the output and loss.

### 3.1. Gated recurrent unit

A GRU is a type of RNN. They are similar to LSTMs but have fewer parameters and can be easier to train. The key innovation is that they allow the network to control the amount of information which flows between consecutive time steps, and allows the network to forget.

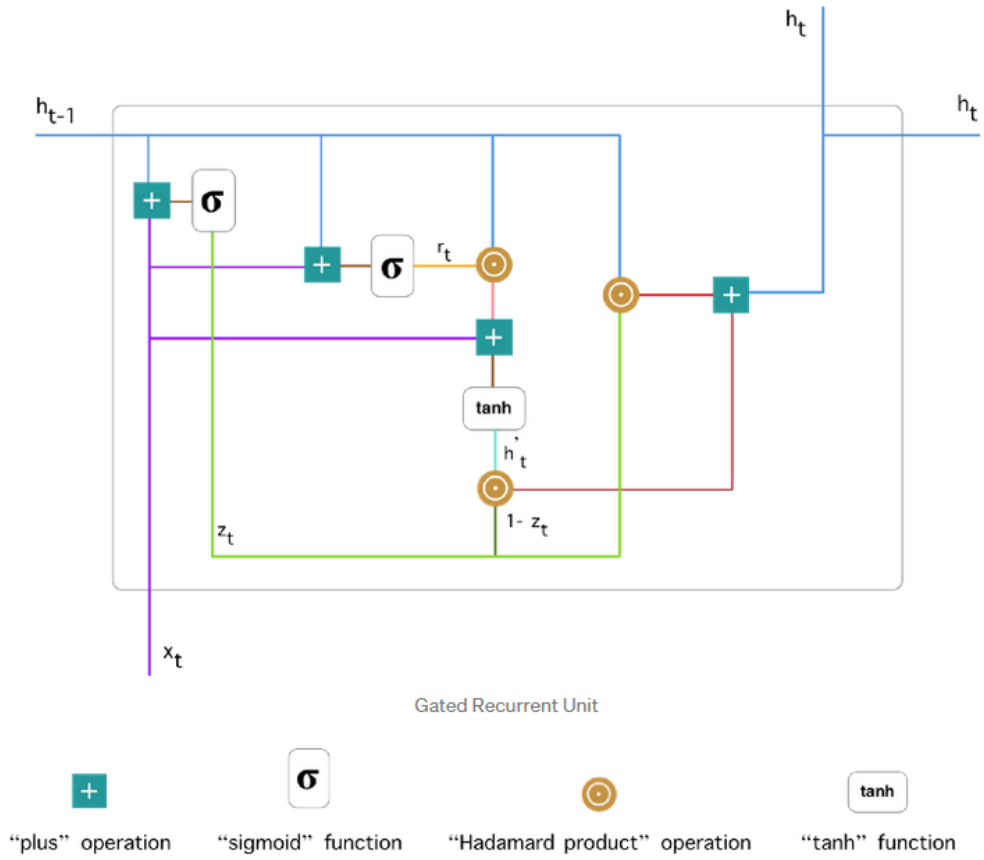


**Figure 2.** Graphical representation of GRU. I don't actually find these super-helpful.

For each element in the input sequence, each layer computes the following function:

$$z_t = \sigma(W_{iz}x_t + b_{iz} + W_{hz}h_{(t-1)} + b_{hz}) \tag{3.3}$$
$$r_t = \sigma(W_{ir}x_t + b_{ir} + W_{hr}h_{(t-1)} + b_{hr}) \tag{3.4}$$
$$n_t = \tanh(W_{in}x_t + b_{in} + r_t \odot (W_{hn}h_{(t-1)} + b_{hn})) \tag{3.5}$$
$$h_t = (1 - z_t) \odot n_t + z_t \odot h_{(t-1)} \tag{3.6}$$

where $x_t$ is the input at time $t$, $h_t$ is the hidden state at time $t$. $r_t$, $z_t$, and $n_t$ are the reset, update, and new gates respectively. $\sigma$ is the sigmoid function, and $\odot$ is the Hadamard product.

1. Eq.(3.3) is called the **update gate**. The update gate combines the input with the previous hidden state. It determines how much of the previous step's hidden state $h_{t-1}$ is passed onto the new hidden state $h_t$ in Eq.(3.6).
2. Eq.(3.4) is called the **reset gate**. The formula is the same as Eq.(3.3). It will be used to decide how much of the past information to **forget** in Eq.(3.5).
3. Eq.(3.5) is a **candidate** hidden state. It combines the current input with some weighting of the previous hidden state. The reset gate $r_t$ has an element-wise product with $h_{t-1}$, allowing the network to forget $h_{t-1}$ as $r_t \to 0$.
4. Eq.(3.6) mixes the previous hidden state $h_{t-1}$ with the candidate hidden state $n_t$ through a convex combination weighted by $z_t$.

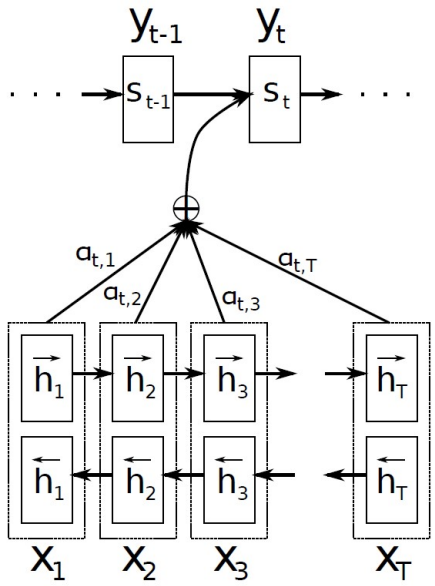


Figure 1: The graphical illustration of the proposed model trying to generate the $t$-th target word $y_t$ given a source sentence $(x_1, x_2, \ldots, x_T)$.

**Figure 3.** Bahdanau attention (Bahdanau *et al.*, 2014), depicting Eq.(4.2) and Eq.(4.3) graphically.

## 4. Attention

### 4.1. Bahdanau attention

Bahdanau *et al.* (2014) were the first to describe an attention model in the context of an encoder-decoder recurrent model. The model is broadly as follows. An encoder reads an input sequence of vectors $x = (x_1, ..., x_{T_x})$, where each element corresponds to e.g. a word in a sentence and the vector is an embedding vector with some fixed embedding dimension. They use an RNN to generate a hidden state $h_t \in \mathbb{R}^n$ for each time $t$

$$h_t = f(x_t, h_{t-1}) \tag{4.1}$$

and a context vector which, in general, is written as $c = q(\{h_1, ..., h_{T_x}\})$ where both $f$ and $q$ are some non-linear functions. We concatenate the forwards and backwards hidden states of a bidirectional GRU in `seq.enc_dec_attn` to form $c$, see Fig. 3.

For the decoder, we have a sequence of target vectors $y = (y_1, ..., y_{T_y})$ and an RNN hidden state $s_i$ where

$$s_i = f(s_{i-1}, y_{i-1}, c_i). \tag{4.2}$$

Notice that, unlike the encoder RNN, the decoder RNN is conditioned on a distinct context vector $c_i$ for each target word $y_i$.

The context vector $c_i$ depends upon the full sequence of encoder hidden states $(h_1, ..., h_{T_x})$, where each of these 'annotations' contains information about the whole input sequence but with a strong focus

on the parts surrounding the $i$-th word. The context vector is computed using an **attention mechanism**

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j \tag{4.3}$$

where $\alpha_{ij}$ is an **attention probability**. In this equation, $h_j$ is commonly referred to as a "**value**" $(V)$, as it is the quantity being reweighted by an attention mechanism $(\alpha)$. The attention probability is defined by

$$\alpha_{ij} = \text{softmax}\,(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})} \tag{4.4}$$

where $e_{ij}$ is an **attention energy** defined by an **alignment model**

$$e_{ij} = a(s_{i-1}, h_j) = v_a^T \tanh(W_a s_{i-1} + U_a h_j) \tag{4.5}$$

where $W_a \in \mathbb{R}^{n \times n}$, $U_a \in \mathbb{R}^{n \times 2n}$ and $v_a \in \mathbb{R}^n$. $W_a s_{i-1}$ is commonly referred to as a "**query**" $(Q)$, as $s_{i-1}$ is the information being used to look up which of the encoder states to combine with. $U_a h_j$ is referred to as the "**key**" $(K)$ – which is the quantity being reweighted.

The alignment models scores how well the inputs around position $j$ and the output at position $j$ match. The context vector for output position $j$ is therefore a reweighting of the input sequence annotations, according to the 'probability' that a particular input is 'relevant' for the current output position $j$. See `seq.enc_dec_attn` for an implementation.

### 4.2. Scaled dot product attention

Scaled dot product attention is a slightly simpler approach to attention than Bahdanau attention. We simply pack together a set of $n$ queries, keys, and values (each of dimension $d_k$) into matrices $Q$, $K$, and $V$ respectively. We then compute the function

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V \tag{4.6}$$

where $Q, K \in \mathbb{R}^{n \times d_k}$ $V \in \mathbb{R}^{n \times d_v}$. The factor of $\sqrt{d_k}$ attempts to reduce the variance of the dot product, and therefore keep the softmax function in regions of non-negligible gradient. (Note that, in general, $d_k \neq d_v \neq d_{\text{emb}}$, where $d_{\text{emb}}$ = embedding dimension, as there may be an intervening linear projection between the embeddings and the query/key/value vectors, as is the case of self-attention.)

**Intuition for queries, keys, and values** The notion of queries, keys, and values comes from information retrieval systems. To better understand these terms, an analogy might be useful. Imagine you're at the supermarket buying a set of ingredients on your shopping list. Then,

- Query: Ingredient on your shopping list
- Key: Label on an item on the shelf
- Value: The actual item on the shelf

As you go through the supermarket, you check the similarity between the item on your list (query) and the label of the item (key). If there is a match, you take the item (value). This analogy breaks down in the sense that attention is a more "smoothed-out" concept: for a given ingredient on your shopping list, the query matches every item in the supermarket (key) to some extent.

### 4.3. Self-attention

Let's go into more conceptual detail on Section 4.2, and specifically discuss self-attention. The attention mechanism behind recurrent models is involves computing the relevance of each encoder state to the decoder hidden state, at a particular decoder time-step. In contrast, the main idea behind self-attention is to use the whole sequence to re-weight the embeddings, in order to encode some contextual information. E.g. the word "apple" would be updated to be more "company-like" and less "fruit-like" if the words "keynote" or "phone" are close to it.

Given a sequence of token embeddings $x_1, ..., x_n$, self-attention produces a sequence of new embeddings $x'_1, ..., x'_n$ where each $x'_i$ is a linear combination of all $x_j$:

$$x'_i = \sum_{j=1}^{n} w_{ji} x_j \tag{4.7}$$

$$\sum_j w_{ji} = 1 \tag{4.8}$$

The coefficients $w_{ji}$ are called **attention weights**, and the resulting embeddings are called *contextualized embeddings* and predate the invention of the transformer.

In practice, when we perform self-attention, we apply independent weight matrices $W_{Q,K,V}$ to the input embeddings $x_1, ..., x_n$ to generate the queries, keys, and values, and then apply Eq.(4.6). The result is a so-called "attention head". Generating multiple independent weight matrices $W_{Q,K,V,i}$ for $i = 1, ..., h$ results in multi-headed attention – see Section 5.2. In other words, when computing attention weights, we do not directly ask which token embedding is most similar to which other token embedding through a naive dot-product, but rather we first make a linear projection of the embeddings into another space for the queries/keys/values and then compute our dot-product similarity metric there.

## 5. Transformers

Transformers (Vaswani *et al.*, 2017) use an encoder/decoder architecture, where we'll denote the embedding dimension as $d_{\text{emb}}$ (they call it $d_{\text{model}}$ in the paper, but I prefer $d_{\text{emb}}$). The following subsections contain detail on each of the sublayers. See Fig. 4 for how the model is wired up, and `seq.transformer` for an implementation.

### 5.1. Positional encoding

After generating the embeddings, we **add** a different sinusoidal function to each dimension of the embedding, where the sinusoid is continuous along the sequence length of the input (namely either the source or target). Doing this gives the input positional information, which is necessary as there are no recurrent or convolutional components to the architecture. For dimension $i$, at position $pos$, we add the following functions to the embeddings

$$PE_{pos,2i} = \sin\left(pos/10000^{2i/d_{\text{emb}}}\right) \tag{5.1}$$

$$PE_{pos,2i+1} = \cos\left(pos/10000^{2i/d_{\text{emb}}}\right). \tag{5.2}$$

In other words, for even dimensions we add a sine wave and for odd dimensions we add a cosine wave, and each dimension goes in a geometric progression of frequency. The intuition is that for any fixed offset $k$, then $PE_{pos+k}, i \propto PE_{pos}, i$ which is easy to learn and should help the model attend to relative positions. [**TODO: Don't quite follow the intuition here.**]

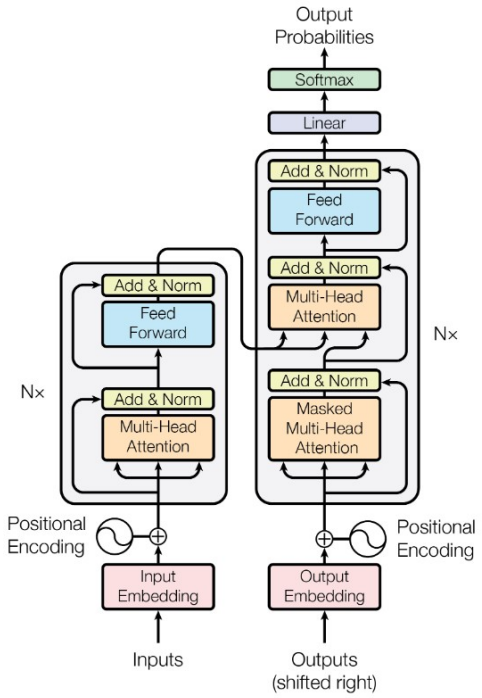


**Figure 4.** The infamous transformer diagram. Left is the encoder, right is the decoder. (Left) Tokenized input sentences are placed in an embedding space, and then a sinusoidal function (called a positional encoding, see Section 5.1) is added onto the embeddings, so the model has access to positional information. The embeddings are then fed through multi-head self-attention (Section 5.2), a normalization step via a residual connection (Section 5.3), and finally a fully-connected feed-forward network with another normalization/residual connection. This is all repeated $N_x$ times. (Right) Has all the same ingredients as the encoder, just wired differently. Again, the tokenized output sentences are placed in an embedding space, and positionally encoded. Embeddings are then fed through multi-head self-attention, with normalization/residual connection. The output of the encoder is fed into multi-head attention as the keys ($K$) and values ($V$), and the output of the decoder's self-attention the query ($Q$), with normalization/residual connection. Lastly, we apply a fully-connected feed-forward network, with normalization/residual connection. This is all repeated $N_x$ times. A linear projection and softmax are finally used to generate probabilities over the output vocabulary.

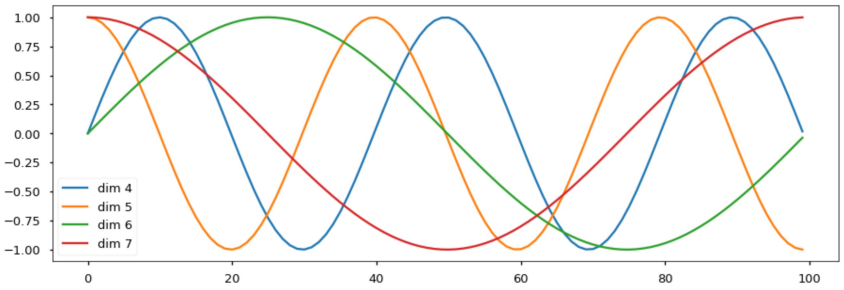


**Figure 5.** Positional encodings. The $x$-axis corresponds to position along the sequence ($pos$). The vertical axis is the value of $PE_{pos,i}$ for dimension $i$. Notice how each function has a different frequency, and the phase at $pos = 0$ alternates between odd and even dimensions.

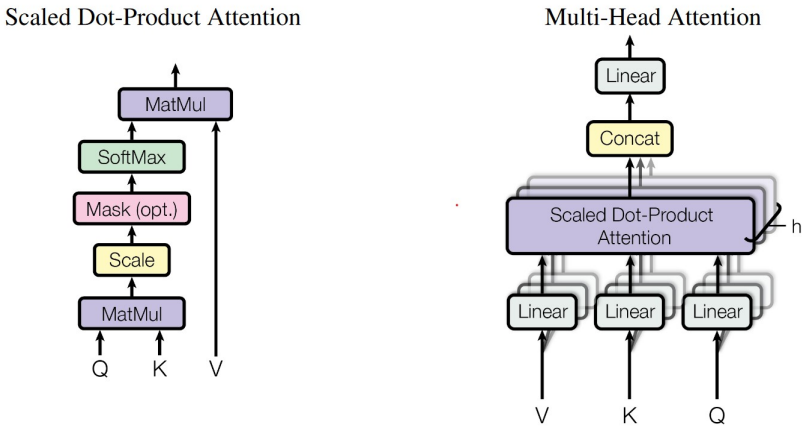


**Figure 6.** Attention mechanisms for transformers. (Left) Depiction of scaled dot-product attentions (see Section 4.2. Masking can be used to induce non-anticipation in the attention mechanism of the target, i.e. not using future tokens of the target to generate a prediction. Masking is also used for padding. (Right) Depiction of multi-head attention. Queries/keys/values are projected into smaller subspaces of dimension $d_k/d_k/d_v$ respectively. This is done $h$ times for all of the queries/keys/values. Scaled dot-product attention is applied to each of these attention heads. The attention heads are then concatenated together to recover the original dimension of the objects. Finally, a linear transformation is applied.

### 5.2. Multi-head attention

Transformers use scaled dot product attention (Section 4.2), but instead of performing a single attention function with $d_{\text{emb}}$-dimensional keys, values, and queries, they instead break the space up into $h$ subspaces via (learnt) linear projections on the queries, keys, and values. They then perform scaled dot-product attention on each of these $h$ subspaces, which they call "**attention heads**":

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V), \text{for } i = 1, ..., h \tag{5.3}$$

where $W_i^Q, W_i^K \in \mathbb{R}^{d_{\text{emb}} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{\text{emb}} \times d_v}$ and $d_v \cdot h = d_k \cdot h = d_{\text{emb}}$. They then concatenate each of the attention heads and apply a linear transformation

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h) \cdot W^O \tag{5.4}$$

where $W^O \in \mathbb{R}^{hd_v \times d_{\text{emb}}}$, to pop the result back into $\mathbb{R}^{d_{\text{emb}}}$ (see Fig. 6). Multi-head attention allows the model to jointly attend to information from different representation subspaces, at different positions.

### 5.3. Add & norm

Every point in Fig. 4 with the step "add & norm" implements a layer normalization (Ba *et al.*, 2016), dropout, and then a residual connection[1] (He *et al.*, 2016). This corresponds to

$$\text{AddAndNorm}(x) = x + \text{Dropout}(\text{Sublayer}(\text{LayerNorm}(x))) \tag{5.5}$$

$$\text{LayerNorm}(x) = a \cdot \frac{x - \bar{x}}{s_x + \epsilon} + b \tag{5.6}$$

where $\bar{x}$ is the sample mean of $x$ over the final dimension (for us, $d_{\text{emb}}$), $s_x$ is the sample standard deviation over the final dimension. $a$ and $b$ are learnable scalar parameters called the gain and bias

[1] This is actually not what is said in the paper, but what modern implementations do because it's apparently better, see here.

respectively. Sublayer$(x)$ refers to the layer immediately preceding AddAndNorm$(x)$ in Fig. 4 – either multi-head attention or a feed-forward layer. The layer is called a residual connection because it is of the form $f(x) = x + R$, so the network is trying to learn residuals rather than $f(x)$ outright (He *et al.*, 2016).

## 6. Hopfield networks

For an implementation see `notebooks/05-Hopfield.ipynb`.

### 6.1. Classical Hopfield networks

Classical Hopfield networks (Hopfield, 1982) are mathematical models of memory. They may be written down as spin systems, where the interaction strength between spins (or "neurons") are tuned such that some number of spin configurations which we wish to store (called "memories") are dynamical steady states of the system.

The state vector of the system is the set of spins $x_i \in \{-1, 1\}$ for $i = 1, ..., d$. The strength of connections between spins $i$ and $j$ is $W_{ij}$. The system evolves under the following dynamics. At some rate, spins flip according to the following rule:

$$x_i \to 1 \text{ if } \sum_{i \neq j} W_{ij} x_j > U_i \tag{6.1}$$

$$x_i \to 0 \text{ if } \sum_{i \neq j} W_{ij} x_j < U_i \tag{6.2}$$

until an equilibrium is reached. We usually set $U_i = 0 \; \forall \; i$, meaning that a spin-up has the same energy as a spin-down.

If we configure $W_{ij}$ in a particular way, we can encode any arbitrary state vector $x$ to be an equilibrium of the dynamical system. Indeed, we can even attempt to store a **collection** of $N$ state vectors $x^s$ for $s = 1, ..., N$ (memories) as equilibria of the spin system, by encoding the collection of state vectors in the interaction strengths of the spins. We use the encoding rule

$$W_{ij} = \sum_{s=1}^{N} x_i^s x_j^s$$

i.e. a sum of outer products of each of the state vectors we wish to encode.

We then take some arbitrary state vector $\xi$ and allow this vector to evolve under the dynamics of the system (where the dimensions of $\xi$ can be updated **synchronously** or **asynchronously**). A synchronous update rule corresponds to all spins/neurons of the system being updated at the same time using a centralised clock; such an update rule is

$$\xi^{t+1} = \text{sgn}(W\xi^t - U). \tag{6.3}$$

Synchronous update rules are often thought of as unrealistic biological models (and are also not great in machine learning because we often want to leverage large parallelism). An asynchronous update rule corresponds to applying the previous equation, but for only one component of $\xi$ at a time, until convergence is reached and $\xi^{t+1} = \xi^t$. This asynchronous update rule corresponds to Hamiltonian dynamics with energy:

$$E = -\frac{1}{2}\xi^T W \xi + \xi^T U.$$

The asynchronous update rule corresponds to simply applying Eq.(6.3) element-wise on $\xi$, in any order.

It turns out that the Classical Hopfield network can have poor recall because the equilibria are quite shallow in configuration space: so given an initial state $\xi$ which is "quite similar" to some state $x^s$, classical Hopfield networks will often return some other state $x^q$ for $q \neq s$.

### 6.2. Modern Hopfield networks

Modern Hopfield networks (Krotov and Hopfield, 2016) are a generalisation of classical Hopfield networks. In general, they use a non-linearity in order to achieve sharper equilibria, allowing better separation of each memory in configuration space. They use an energy function

$$E = -\sum_{i=1}^{N} F(x_i^T \xi) \tag{6.4}$$

where $F$ is some non-linear function, typically a polynomial like $F(z) = z^a$.

Demircigil *et al.* (2017) use an exponential function $F(z) = \exp(z)$. The corresponding update rule can be shown to be

$$\xi^{\text{new}}[l] = \text{sgn}\left[-E(\xi^{(l+)}) + E(\xi^{(l-)})\right] \xi[l] \tag{6.5}$$

where $\xi^{(l+)} = \xi$ and $\xi^{(l-)}[l] = -\xi[l],\ \xi^{(l-)}[i] = \xi[i]\ \forall\ i \neq l$ [**TODO: NB:** The product with $\xi[l]$ seems to disagree with the paper and this blog, yet it's what I had to implement to get it to work! I might be misunderstanding something...]

### 6.3. Continuous states and self-attention

For continuous-valued states, (Ramsauer *et al.*, 2020) propose a new energy function

$$E = -\text{logsumexp}(\beta, X^T\xi) + \frac{1}{2}\xi^T\xi + \beta^{-1}\log N + \frac{1}{2}M^2 \tag{6.6}$$

where $\beta$ is an inverse temperature, which sets how sharp the equilibria are in configuration space and $M$ is the largest norm of all stored patterns[2]. The corresponding (asynchronous) update rule is

$$\xi^{\text{new}} = X\text{softmax}(\beta X^T \xi). \tag{6.7}$$

It turns out that if we appropriately project the patterns and the result, then Eq.(6.7) is equivalent to dot-product self-attention. If we consider $S$ raw state patterns $R = (\xi_1, ..., \xi_S)^T$ and $N$ raw stored patterns $Y = (y_1, ..., y_N)^T$ and then use the projections

$$Q = RW_Q \tag{6.8}$$
$$K = YW_K \tag{6.9}$$
$$V = KW_V \tag{6.10}$$

then if we let $\beta = 1/\sqrt{d_k}$ then we obtain

$$Z = Q^{\text{new}}W_V = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)KW_V = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{6.11}$$

which is the form for dot-product self-attention (Eq.(4.6)). This equation for $Z$ defines a Hopfield layer, see Fig. 7 (for an implementation, see the authors' GitHub repository).

Compared to self-attention, Hopfield layers are useful in the following contexts:

1. Association of two sets of vectors
2. Searching a set of fixed patterns
3. Storing or learning static patterns

Some settings where this could be used include:

[2] In practice, I found that I needed to Z-score all input patterns for the retrieval to work.

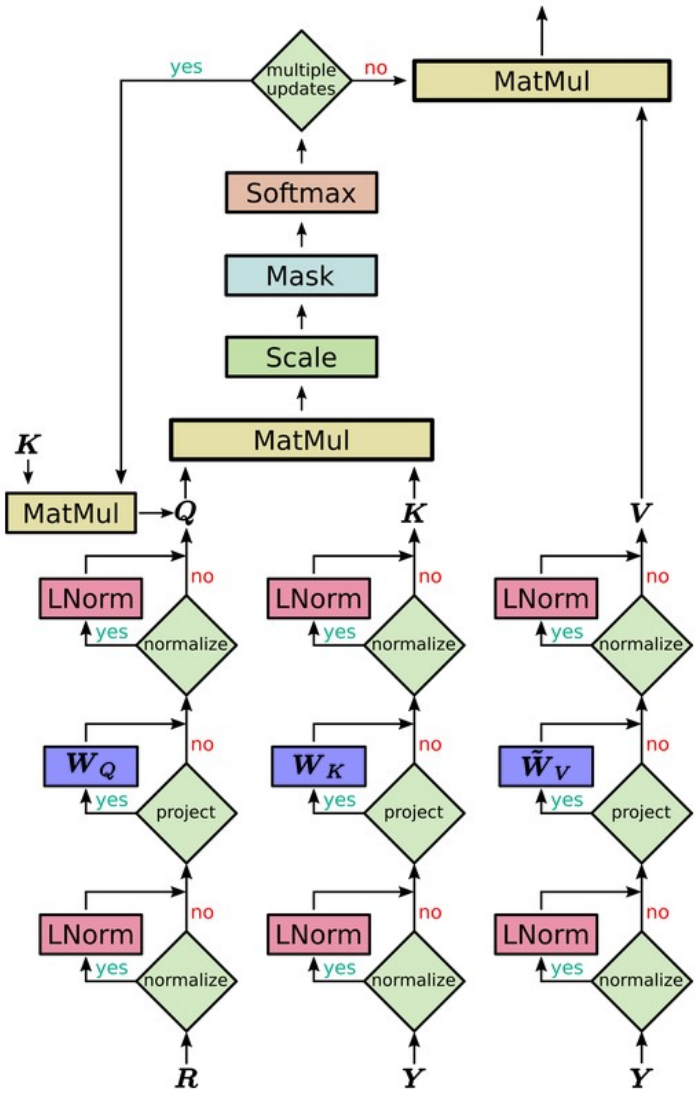


**Figure 7.** Hopfield layer architecture

- The training data is used as stored patterns, the new data as state pattern, and the training label to project the output of the Hopfield layer.
- When only one static pattern (query) exists, then a Hopfield layer is de facto pooling over the sequence. The static pattern is a prototype pattern, i.e. $Q$ is learned. **[TODO: Don't understand.]**
- Needle in a haystack problems, where there exist a large number of patterns, but only a tiny minority are relevant for the model output. The inputs $K$ are partially learned by a neural network. See (Widrich *et al.*, 2020) for an example.

## 7. Loss functions

### 7.1. Cross entropy

The cross-entropy $H(p, q)$ between two probability distributions $p$ and $q$ is over the same underlying set of events measures the number of bits needed to identify an event drawn from the set if a coding scheme used for the set is optimized for an estimated probability distribution $q$ rather than the true distribution $p$. It is defined as

$$H(p, q) = -\sum_{x \in \mathcal{X}} p(x) \log q(x) \tag{7.1}$$

for discrete probability distributions $p$ and $q$. The cross entropy can be written in terms of the entropy of the true distribution ($H(p)$) and the Kullback-Leibler divergence ($D_{\mathsf{KL}}(p||q)$)

$$H(p, q) = H(p) + D_{\mathsf{KL}}(p||q). \tag{7.2}$$

[**TODO**: which will almost certainly have some sort of energetics interpretation too]

Minimizing the cross-entropy is the same as maximizing the log-likelihood for a multinomial model (which is called a "bag of words" model in language modelling). We typically use $p$ as the empirical probability in the test set, and $q$ as the model predicted probability.

### 7.2. Label smoothing

Suppose that there exists a single true label $y$ over classification classes $k$. Using the same notation above for cross entropy, then for a particular training example $x$ we will have $p(k) = \delta_{k,y}$. We typically generate $q(k|x) = \text{softmax}(z_k)$ where $z_k$ is a logit. The fact that the target distribution is a Dirac delta function means that the only way to achieve the target distribution is to have $z_y \gg z_k$. This can cause: i) overfitting; ii) the model to become less adaptive, due to the gradient of $H(p, q)$ w.r.t. $z_k$ being bounded between -1 and 1. Szegedy *et al.* (2016) propose replacing the label distribution $q(k|x) = \delta_{k,y}$ with

$$q'(k|x) = (1 - \epsilon)\delta_{k,y} + \epsilon u(k) \tag{7.3}$$

where $u(k)$ is independent of the training example $x$. They typically choose this to be the model prior, namely the discrete uniform distribution. This amounts to replacing the target with a uniformly random category with probability $\epsilon$ during training.

## References


Ba, J. L., J. R. Kiros, and G. E. Hinton, 2016 Layer normalization. arXiv preprint arXiv:1607.06450 .

Bahdanau, D., K. Cho, and Y. Bengio, 2014 Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473 .

Demircigil, M., J. Heusel, M. Löwe, S. Upgang, and F. Vermet, 2017 On a model of associative memory with huge storage capacity. Journal of Statistical Physics **168**: 288–299.

He, K., X. Zhang, S. Ren, and J. Sun, 2016 Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778.

Hopfield, J. J., 1982 Neural networks and physical systems with emergent collective computational abilities. Proceedings of the national academy of sciences **79**: 2554–2558.

Krotov, D. and J. J. Hopfield, 2016 Dense associative memory for pattern recognition. Advances in neural information processing systems **29**: 1172–1180.

Ramsauer, H., B. Schäfl, J. Lehner, P. Seidl, M. Widrich, *et al.*, 2020 Hopfield networks is all you need. arXiv preprint arXiv:2008.02217 .

Szegedy, C., V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, 2016 Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2818–2826.

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, *et al.*, 2017 Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008.

Widrich, M., B. Schäfl, H. Ramsauer, M. Pavlović, L. Gruber, *et al.*, 2020 Modern hopfield networks and attention for immune repertoire classification. arXiv preprint arXiv:2007.13505 .