

Advanced NLP

Summer 2023

Anoop Sarkar

LayerNorm

<https://arxiv.org/abs/1607.06450>

also see: <https://arxiv.org/abs/1911.07013>

$$\mathbf{x} = (x_1, x_2, \dots, x_H)$$

$$\left\| \frac{\partial \text{LN}(\mathbf{x})}{\partial \mathbf{x}} \right\| = O\left(\frac{\sqrt{d}}{\|\mathbf{x}\|} \right)$$

<https://proceedings.mlr.press/v119/huang20f.html>

$$\mu = \frac{1}{H} \sum_{i=1}^H x_i \quad \sigma^2 = \frac{1}{H} \sum_{i=1}^H (x_i - \mu)^2$$

$$N(\mathbf{x}) = \frac{\mathbf{x} - \mu}{\sigma + \epsilon} \quad \epsilon \text{ avoids div by zero}$$

$$\mathbf{h} = \mathbf{g} \cdot N(\mathbf{x}) + \mathbf{b}$$

\mathbf{g} and \mathbf{b} are hyperparameters with dimension H

In PyTorch

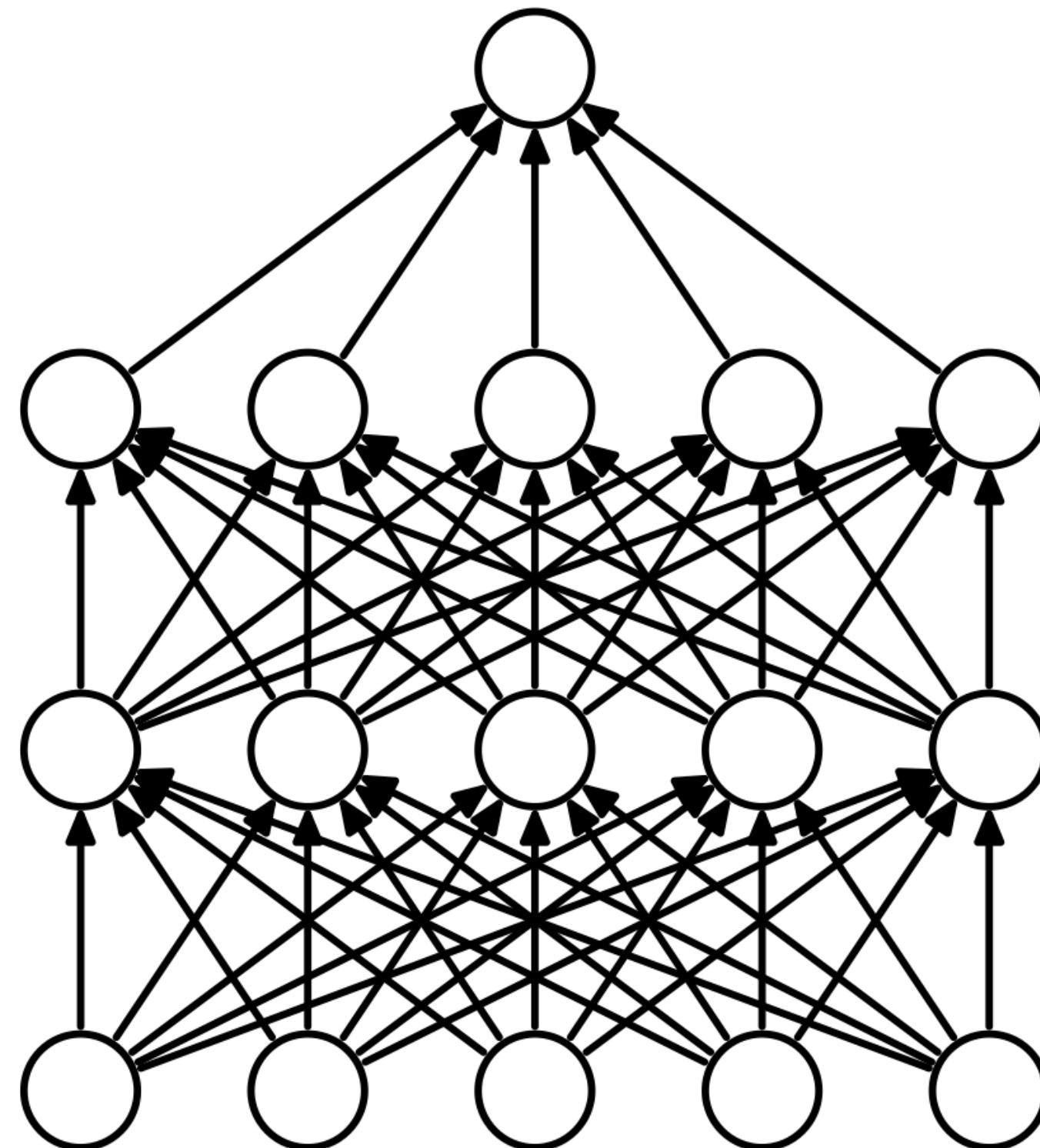
```
>>> # NLP Example
>>> batch, sentence_length, embedding_dim = 20, 5, 10
>>> embedding = torch.randn(batch, sentence_length, embedding_dim)
>>> layer_norm = nn.LayerNorm(embedding_dim)
>>> # Activate module
>>> layer_norm(embedding)
```

Dropout

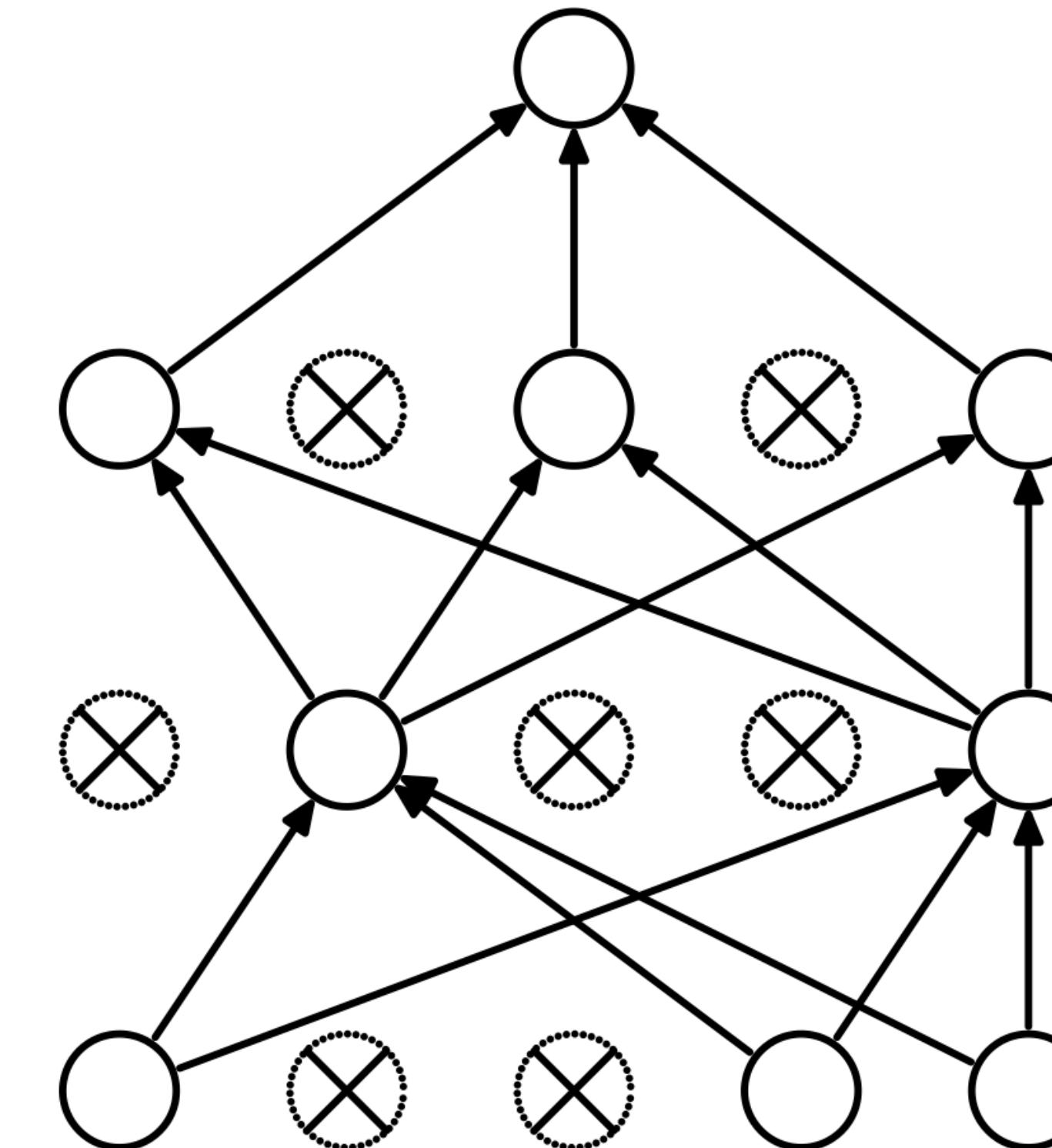
<https://jmlr.org/papers/v15/srivastava14a.html>

<https://arxiv.org/abs/1207.0580>

aka how to train 2^n neural networks when it has n units



(a) Standard Neural Net



(b) After applying dropout.

Before dropout

$$\begin{aligned} z_i^{(l+1)} &= \mathbf{w}_i^{(l+1)} \mathbf{y}^l + b_i^{(l+1)}, \\ y_i^{(l+1)} &= f(z_i^{(l+1)}), \end{aligned}$$

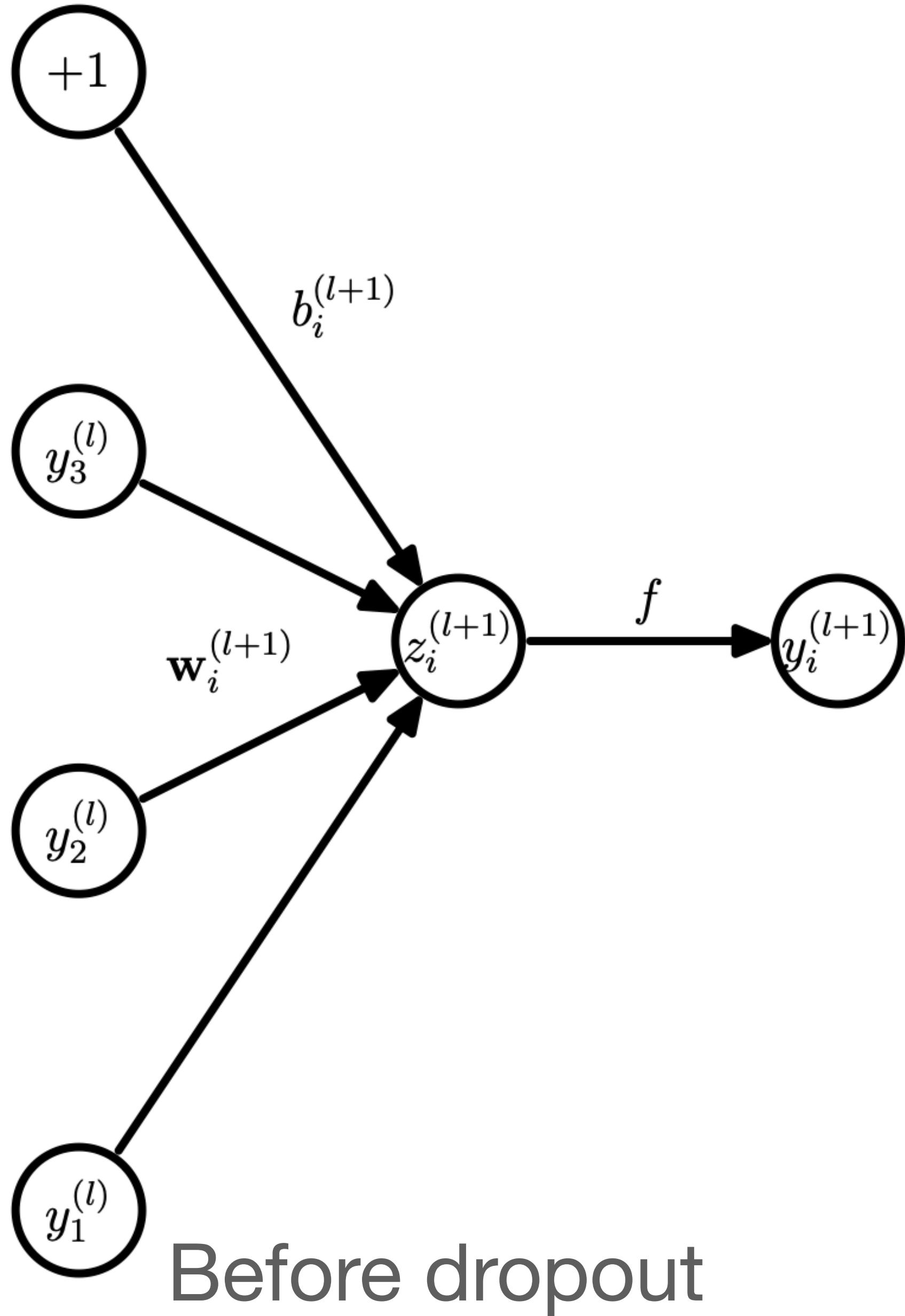
After dropout

$$\begin{aligned} r_j^{(l)} &\sim \text{Bernoulli}(p), \\ \tilde{\mathbf{y}}^{(l)} &= \mathbf{r}^{(l)} * \mathbf{y}^{(l)}, \\ z_i^{(l+1)} &= \mathbf{w}_i^{(l+1)} \tilde{\mathbf{y}}^l + b_i^{(l+1)}, \\ y_i^{(l+1)} &= f(z_i^{(l+1)}). \end{aligned}$$

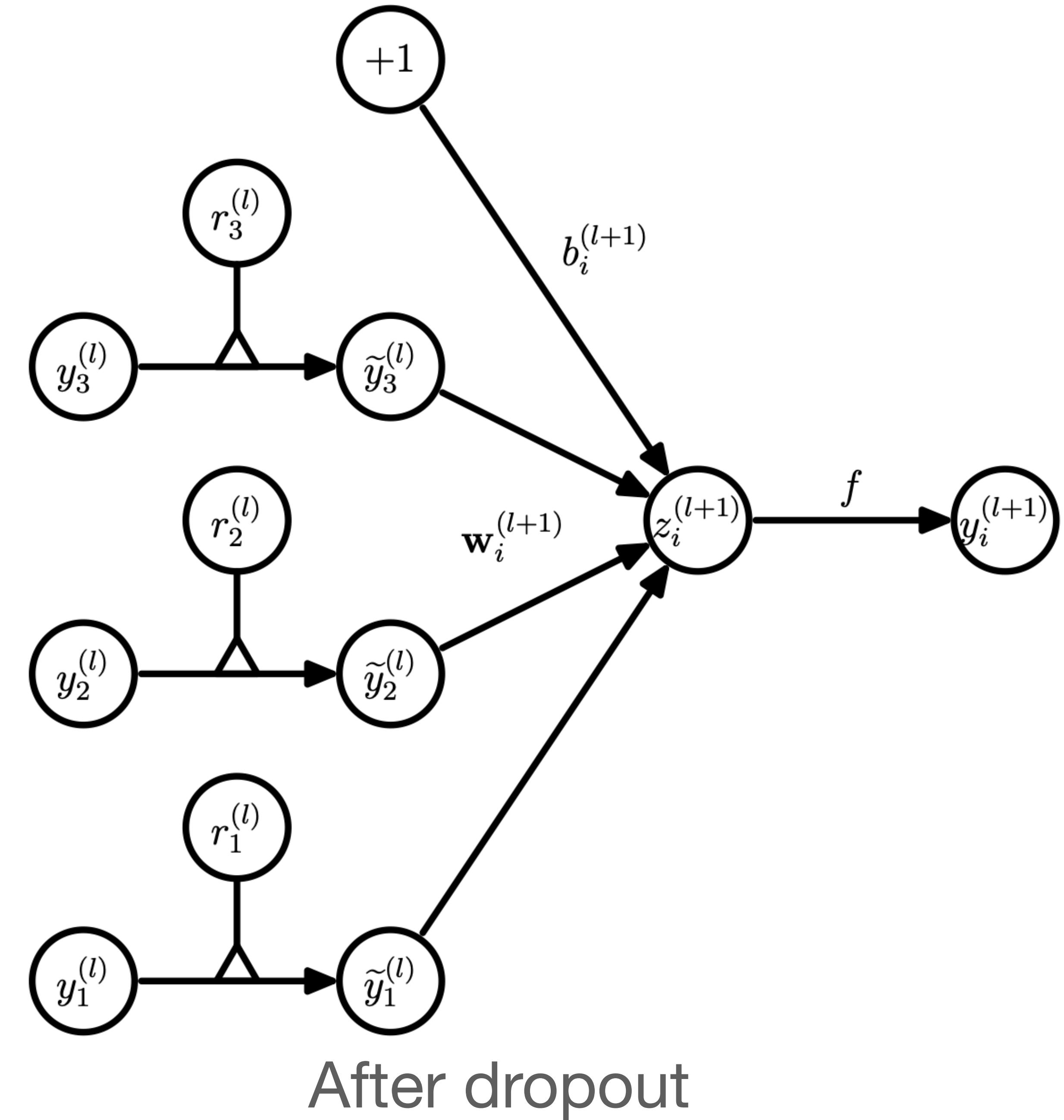
In PyTorch

```
>>> m = nn.Dropout(p=0.2)
>>> input = torch.randn(20, 16)
>>> output = m(input)
```

default: 0.5



Before dropout



After dropout

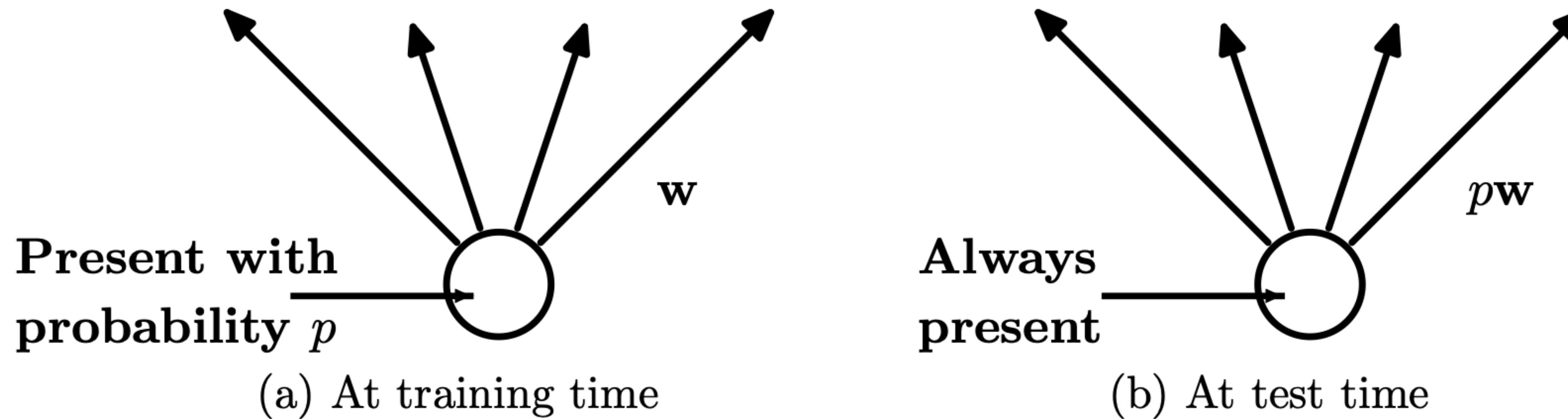


Figure 2: **Left:** A unit at training time that is present with probability p and is connected to units in the next layer with weights w . **Right:** At test time, the unit is always present and the weights are multiplied by p . The output at test time is same as the expected output at training time.

In Pytorch the outputs are scaled by a factor of $\frac{1}{1-p}$ during training so at inference/test/evaluation time the dropout function simply computes the identity function

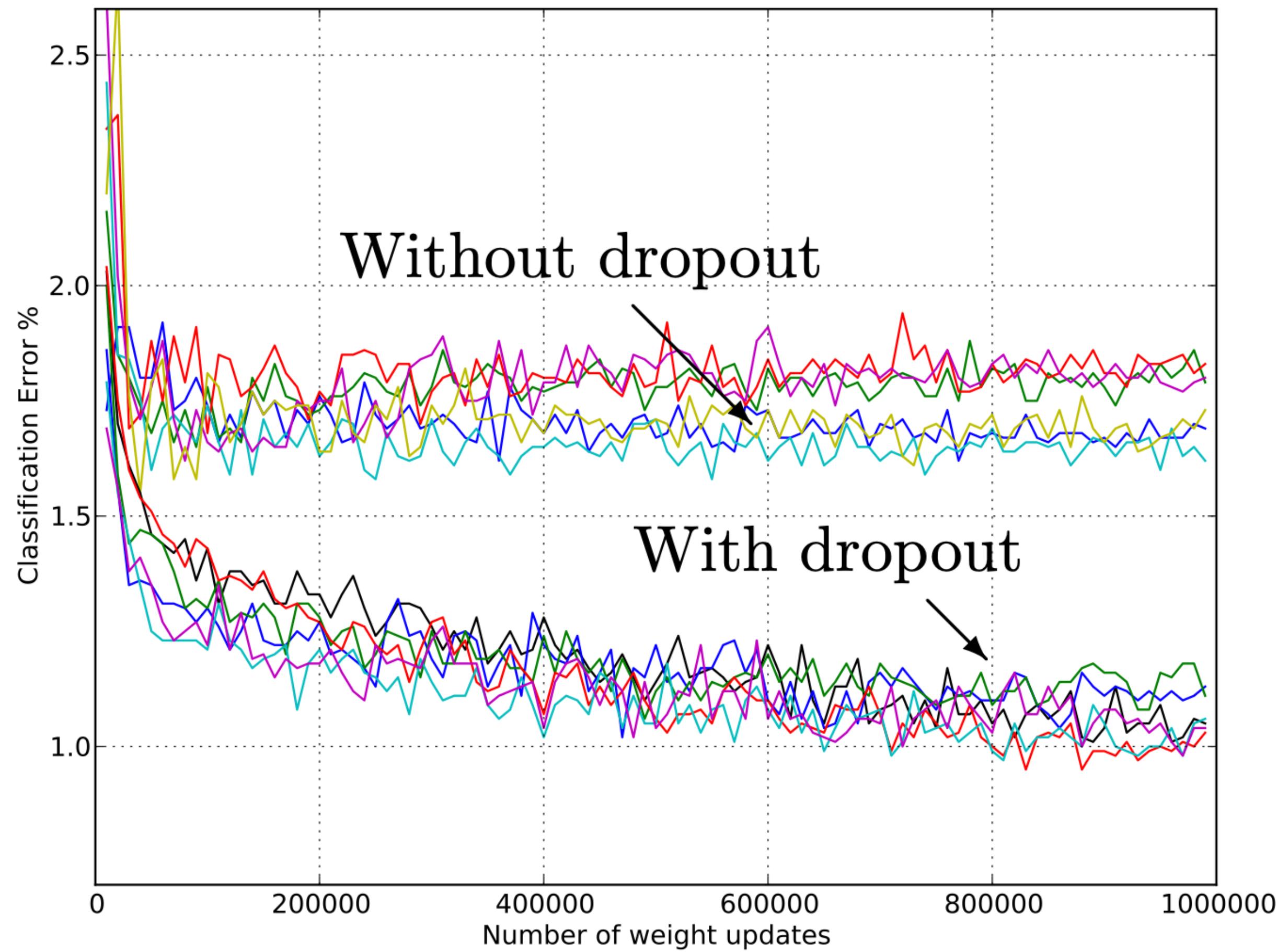
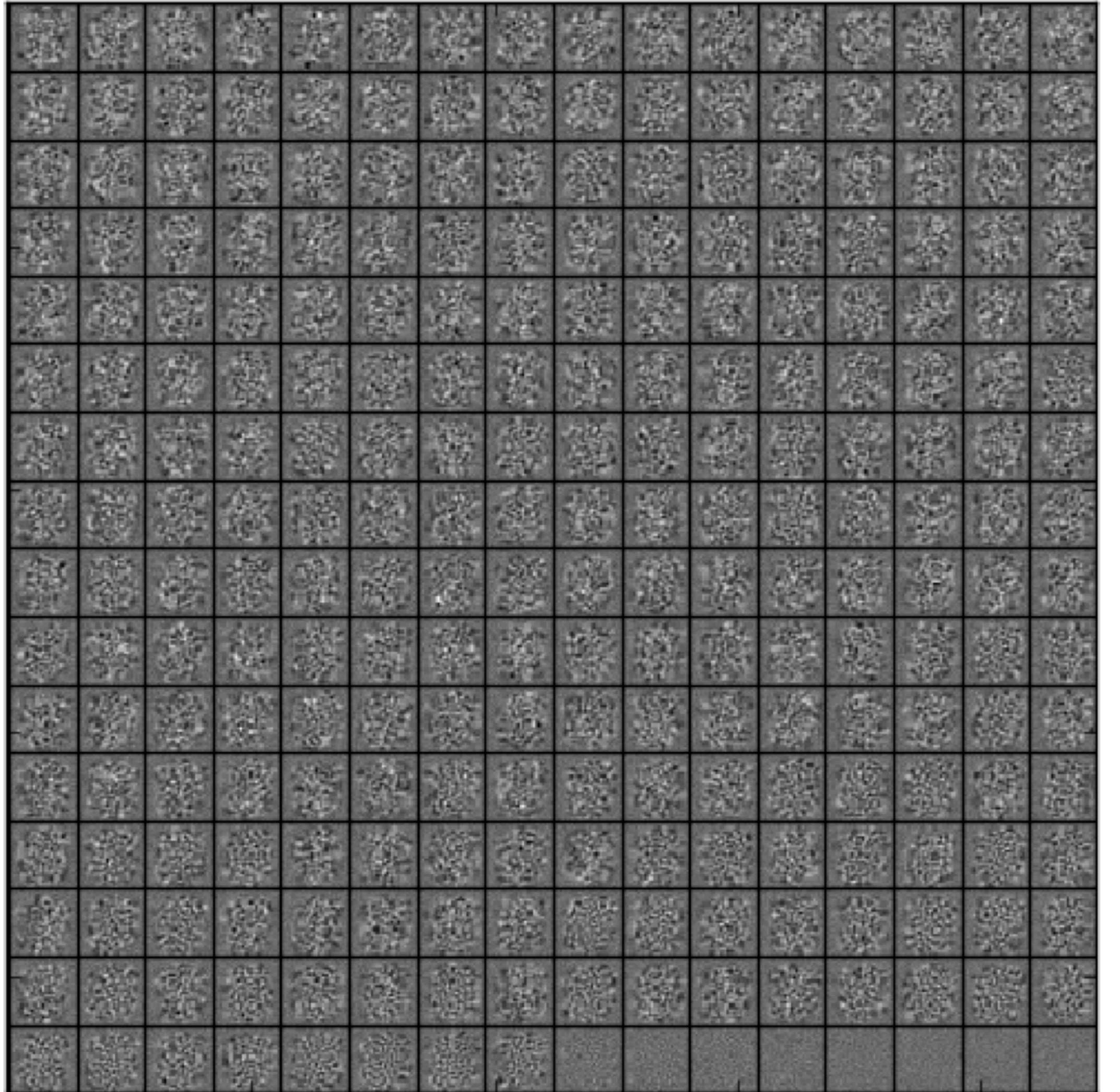
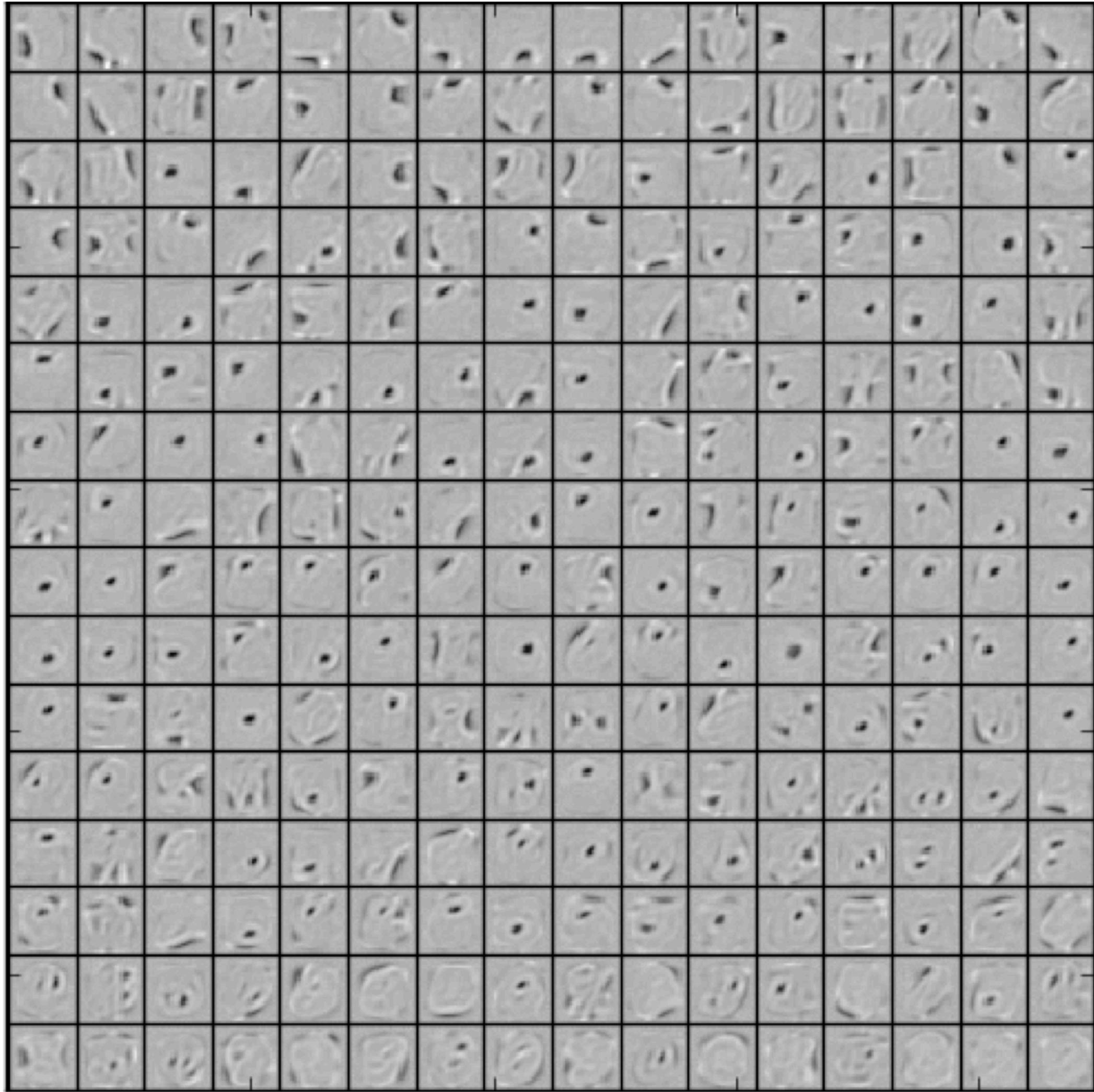


Figure 4: Test error for different architectures with and without dropout. The networks have 2 to 4 hidden layers each with 1024 to 2048 units.

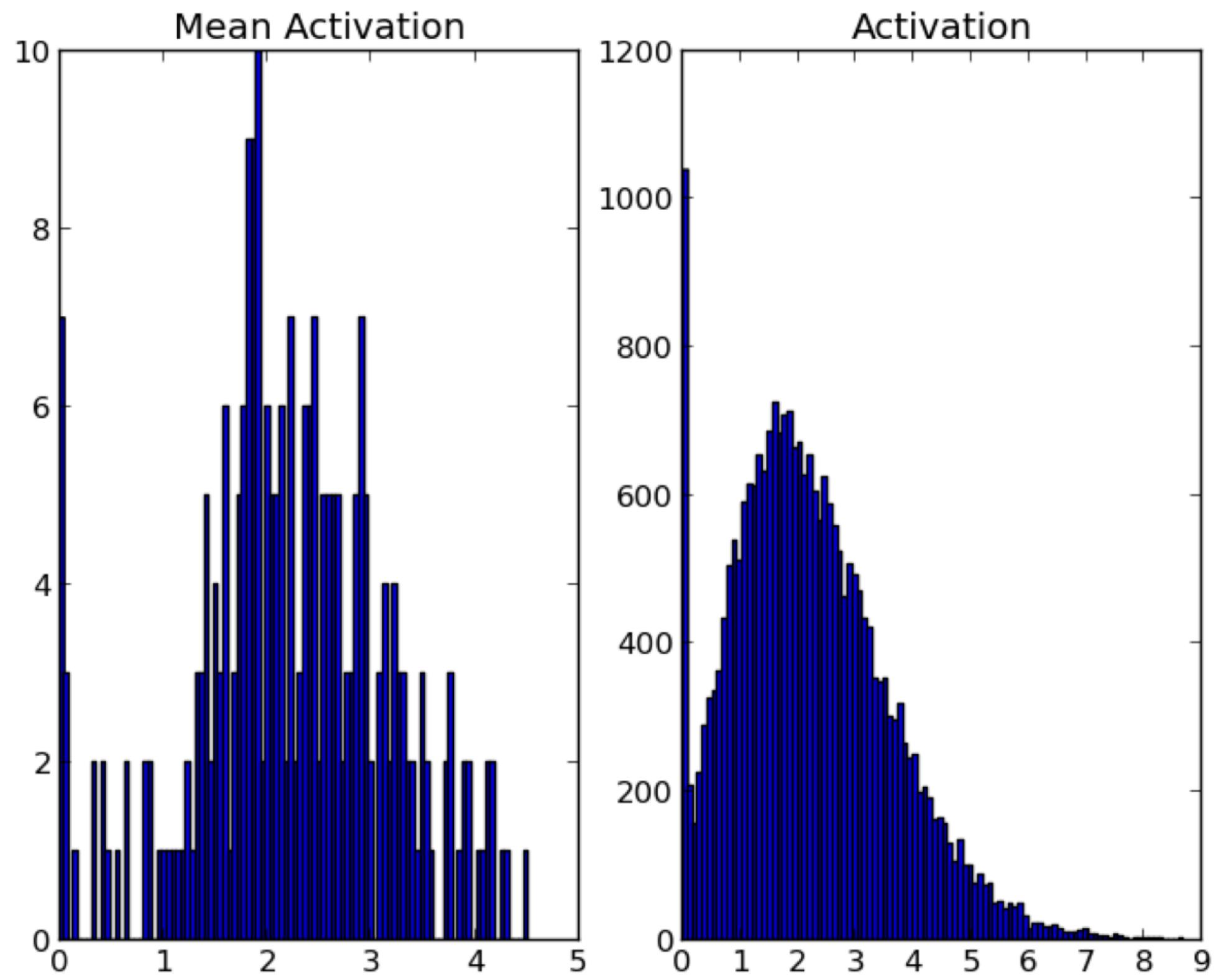


(a) Without dropout

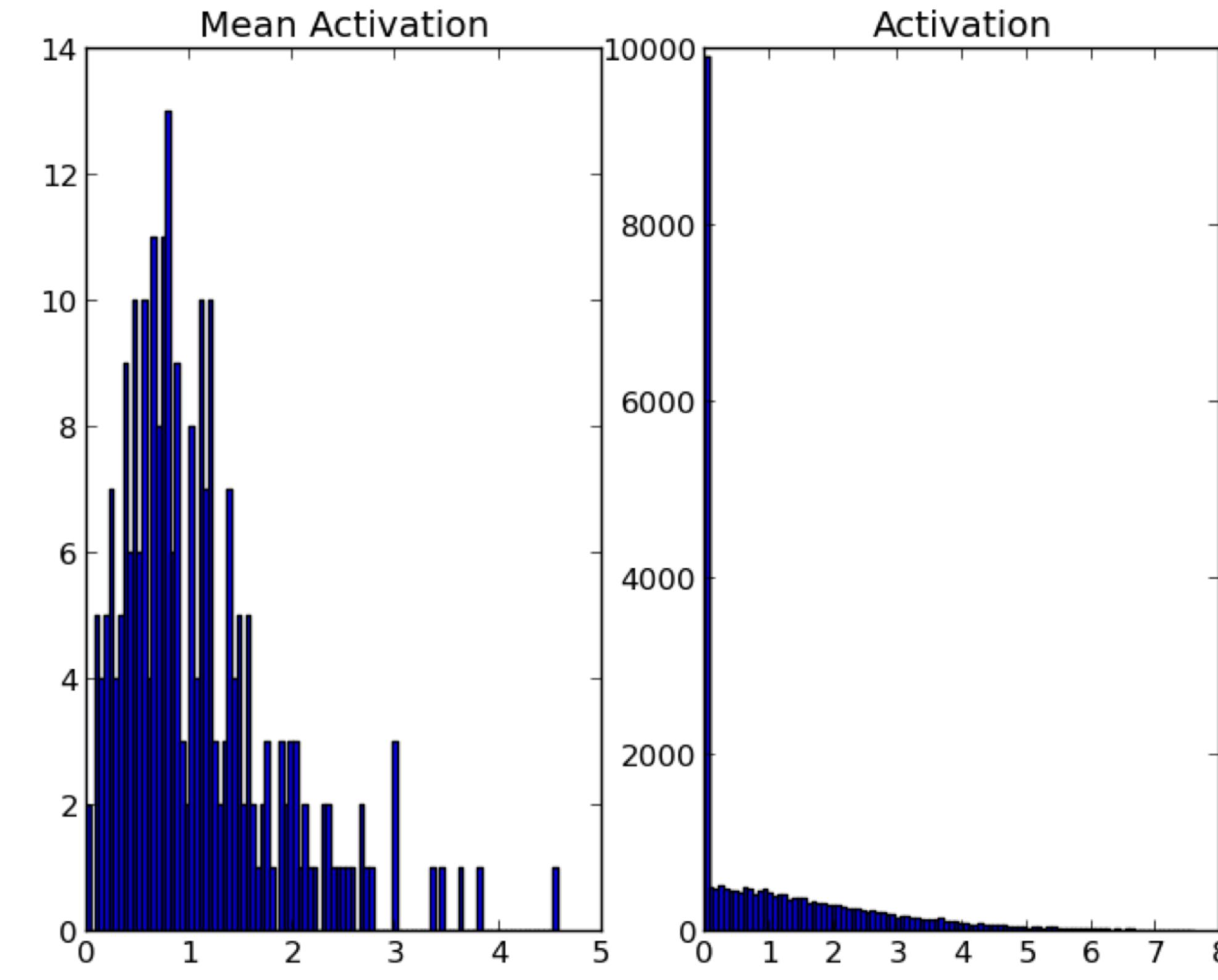


(b) Dropout with $p = 0.5$.

Figure 7: Features learned on MNIST with one hidden layer autoencoders having 256 rectified linear units.



(a) Without dropout



(b) Dropout with $p = 0.5$.

Figure 8: Effect of dropout on sparsity. ReLUs were used for both models. **Left:** The histogram of mean activations shows that most units have a mean activation of about 2.0. The histogram of activations shows a huge mode away from zero. Clearly, a large fraction of units have high activation. **Right:** The histogram of mean activations shows that most units have a smaller mean activation of about 0.7. The histogram of activations shows a sharp peak at zero. Very few units have high activation.

Attention Is All You Need

Ashish Vaswani*
Google Brain
avaswani@google.com

Noam Shazeer*
Google Brain
noam@google.com

Niki Parmar*
Google Research
nikip@google.com

Jakob Uszkoreit*
Google Research
usz@google.com

Llion Jones*
Google Research
llion@google.com

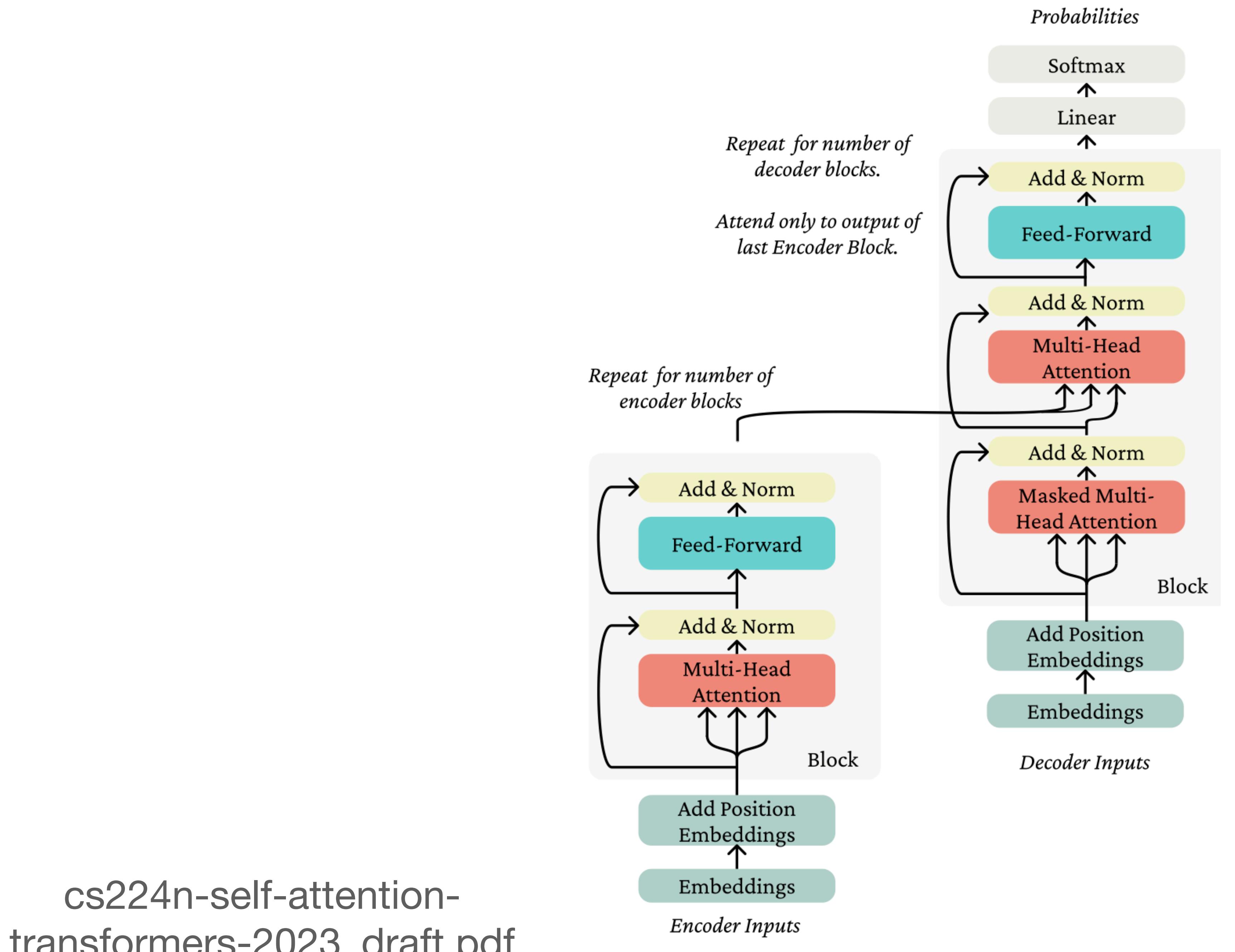
Aidan N. Gomez* †
University of Toronto
aidan@cs.toronto.edu

Łukasz Kaiser*
Google Brain
lukaszkaiser@google.com

Illia Polosukhin* ‡
illia.polosukhin@gmail.com

<https://arxiv.org/abs/1409.0473>

NIPS (2017)



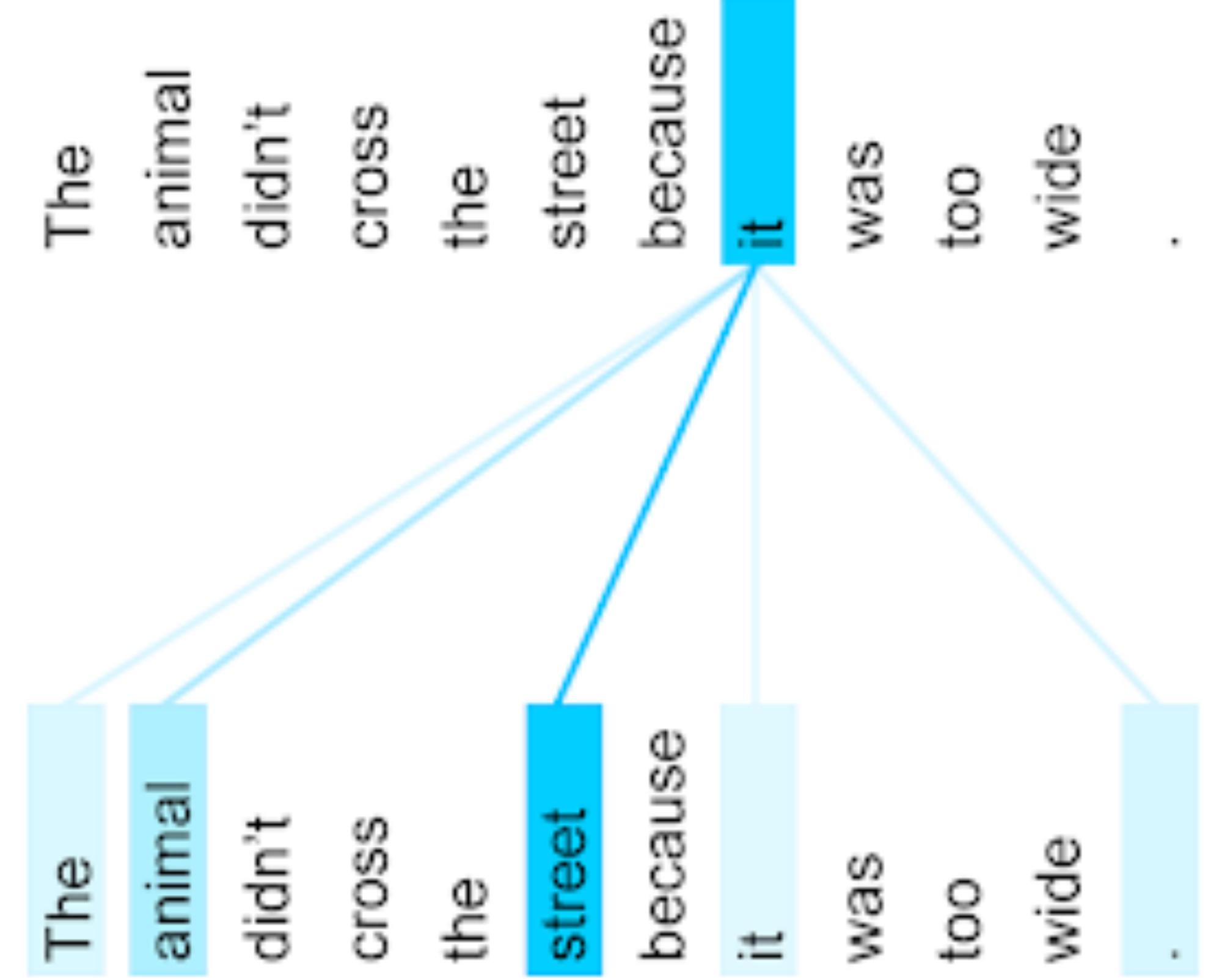
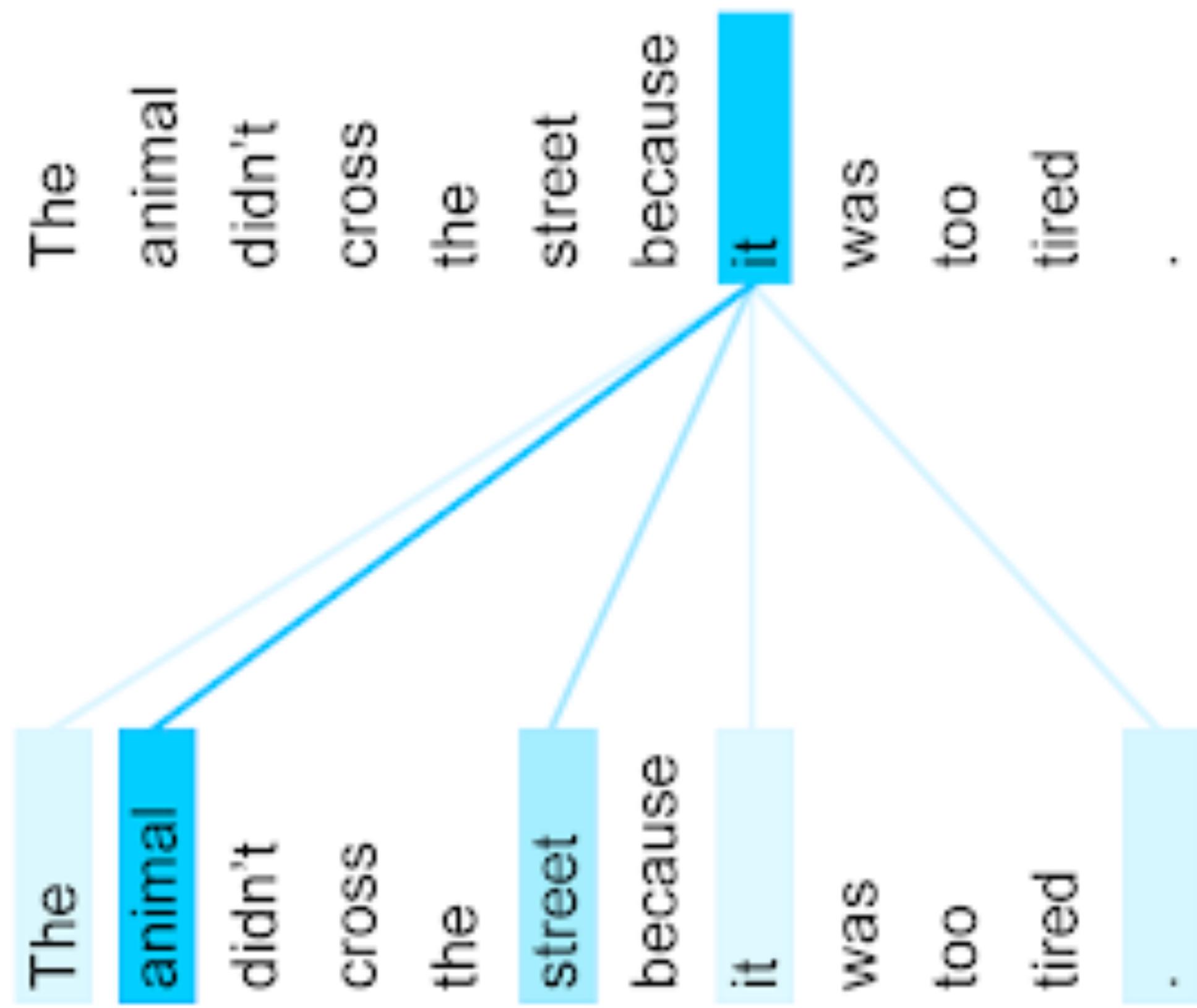
cs224n-self-attention-transformers-2023_draft.pdf

<https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html>

*The animal didn't cross the street because it was too tired.
L'animal n'a pas traversé la rue parce qu'il était trop fatigué.*

*The animal didn't cross the street because it was too wide.
L'animal n'a pas traversé la rue parce qu'elle était trop large.*

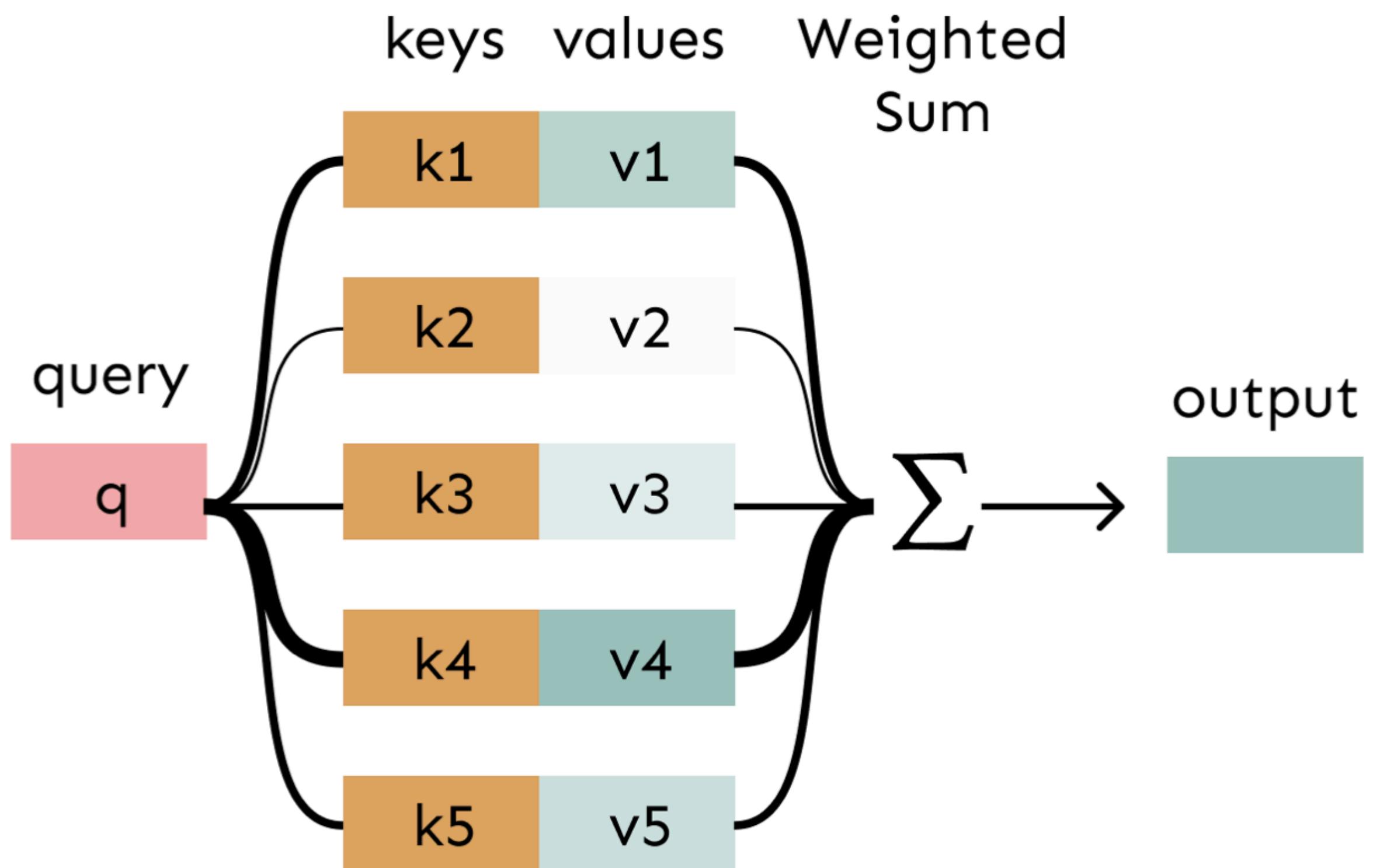
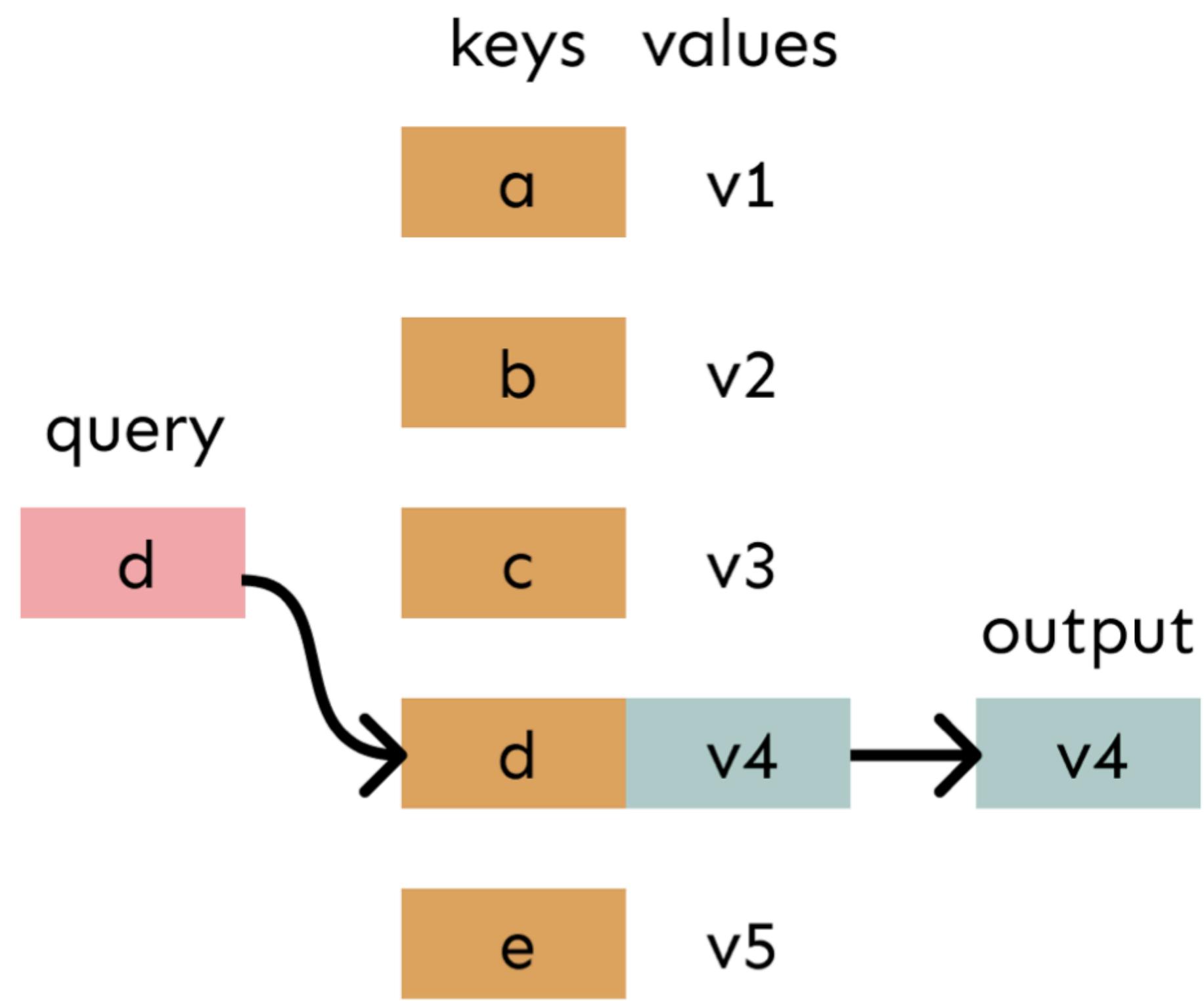
the translation for “it” depends on the gender of the noun it refers to - and in French “animal” and “street” have different genders



The encoder self-attention distribution for the word “it” from the 5th to the 6th layer of a Transformer trained on English to French translation (one of eight attention heads).

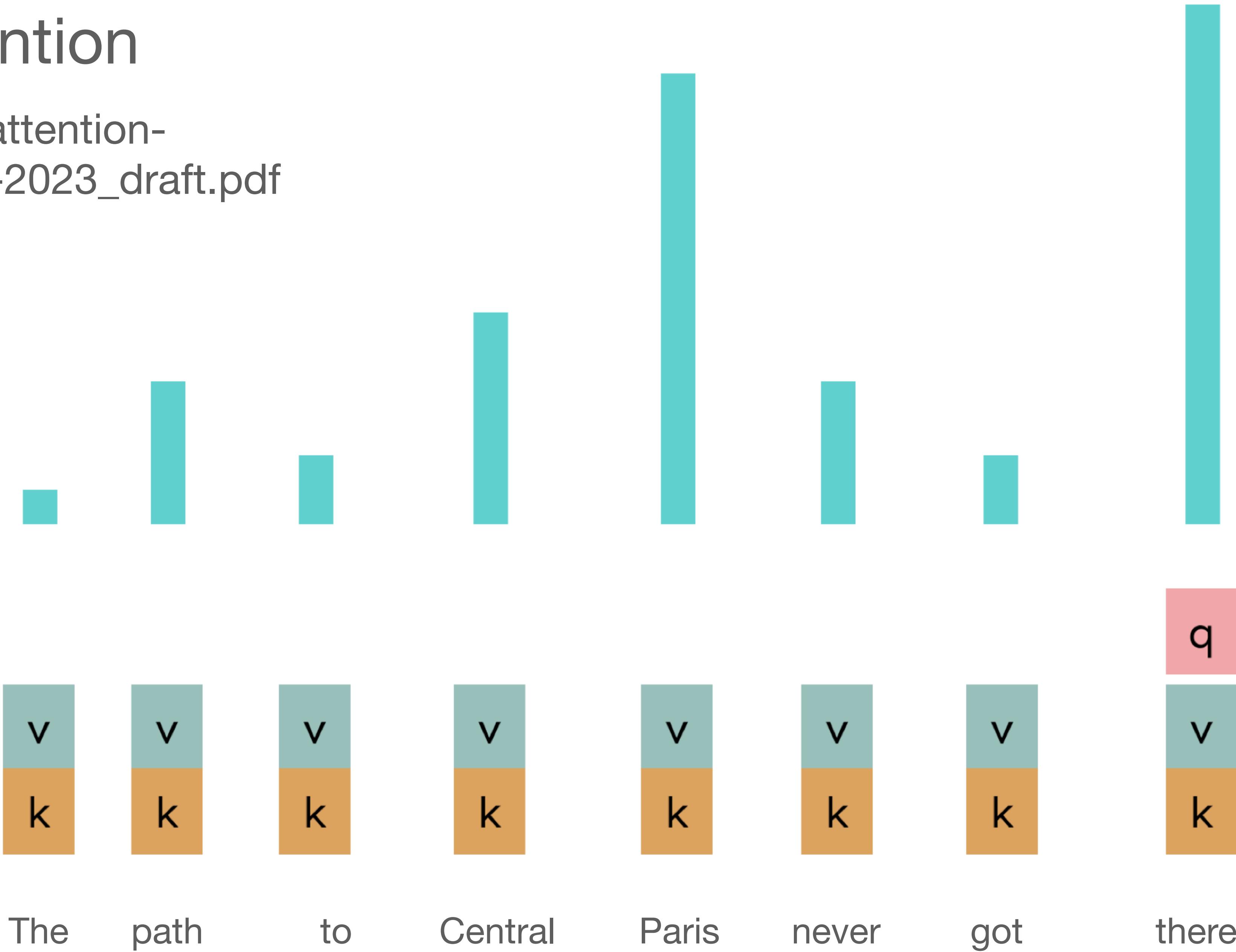
Self Attention

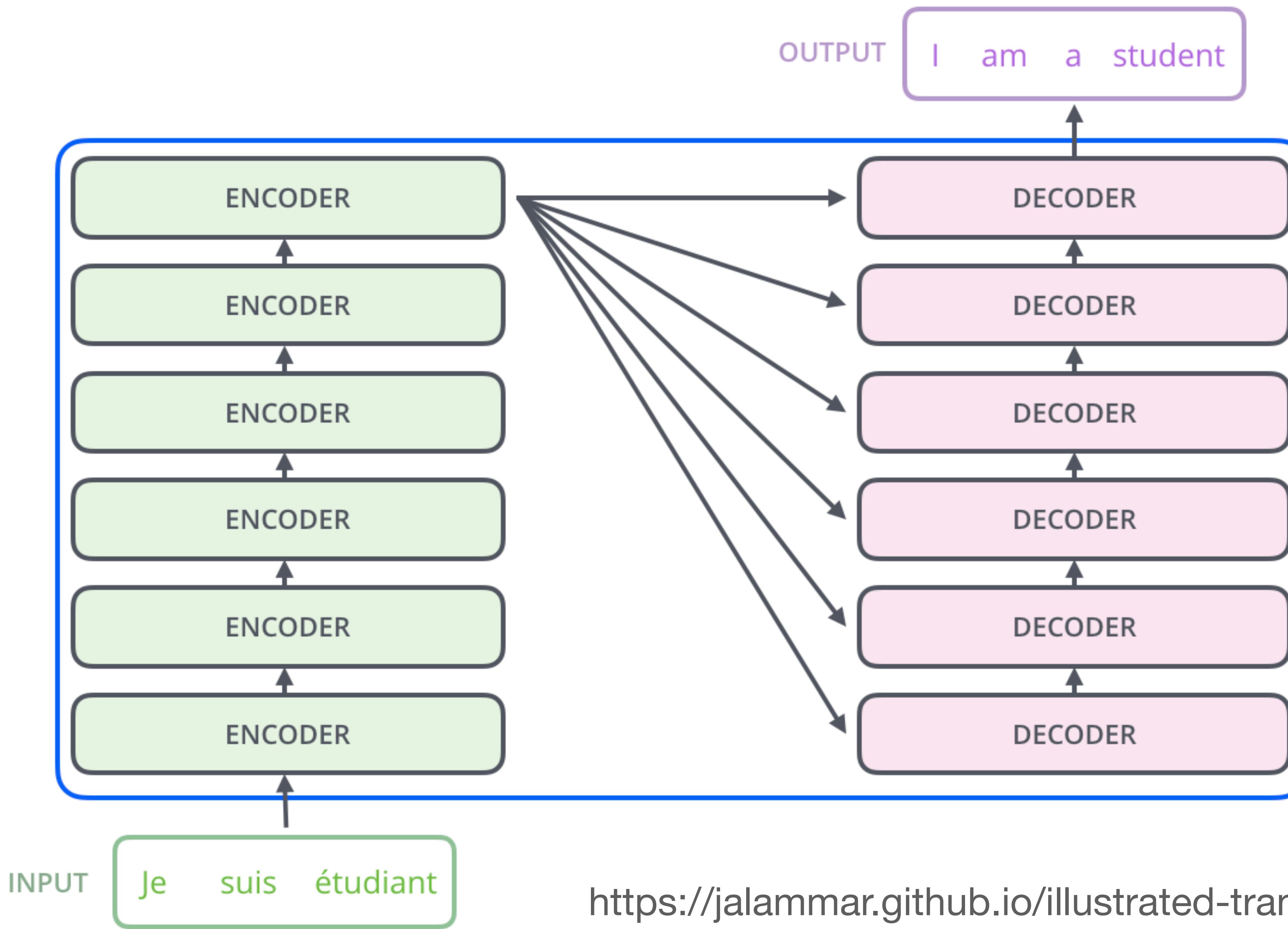
- Take a query vector (based on one token)
- Do a "**soft lookup**" in a key-value store; **pick up** the key **most like** the query and return the value vector
- "**pick up**" = return the average value based on a probability distribution
- "**most like**" = higher probability for a key means it is **more like** the query
- "**more like**" = dot product e.g.
- In *self attention* we use the same tokens for queries, keys and values



Self Attention

cs224n-self-attention-transformers-2023_draft.pdf





<https://jalammar.github.io/illustrated-transformer/>

Self-attention

keys, queries and values from the same sequence

- Let $\mathbf{w} = (w_1, \dots, w_n)$ be a sequence of tokens, like "Cuba is the capital of"
 - For each w_i let $\mathbf{x}_i = E\mathbf{w}_i$ where $E \in \mathbb{R}^{d \times |V|}$ is an embedding matrix. V is the vocabulary.
 - Let Q, K, V be matrices in $\mathbb{R}^{d \times d}$
 - $\mathbf{q}_i = Q\mathbf{x}_i$
 - $\mathbf{k}_i = K\mathbf{x}_i$
 - $\mathbf{v}_i = V\mathbf{x}_i$
- Output for each word is a weighted sum of values:
- $$\mathbf{o}_i = \sum_j \text{softmax}_j(\mathbf{q}^T \mathbf{k}_j) \cdot \mathbf{v}_i$$

Self Attention: Three Problems

Problem	Solution
Encoder and decoder has no inherent notion of ordering. It's just a bag of words.	Add position representations to each token
Just a weighted average of a vector. No non-linearities.	Apply feedforward network to each self attention output
Decoder should not look into the future while training the predictor.	Mask out the future by setting attention weights to zero.

Self-attention

Fixing the sequence order problem

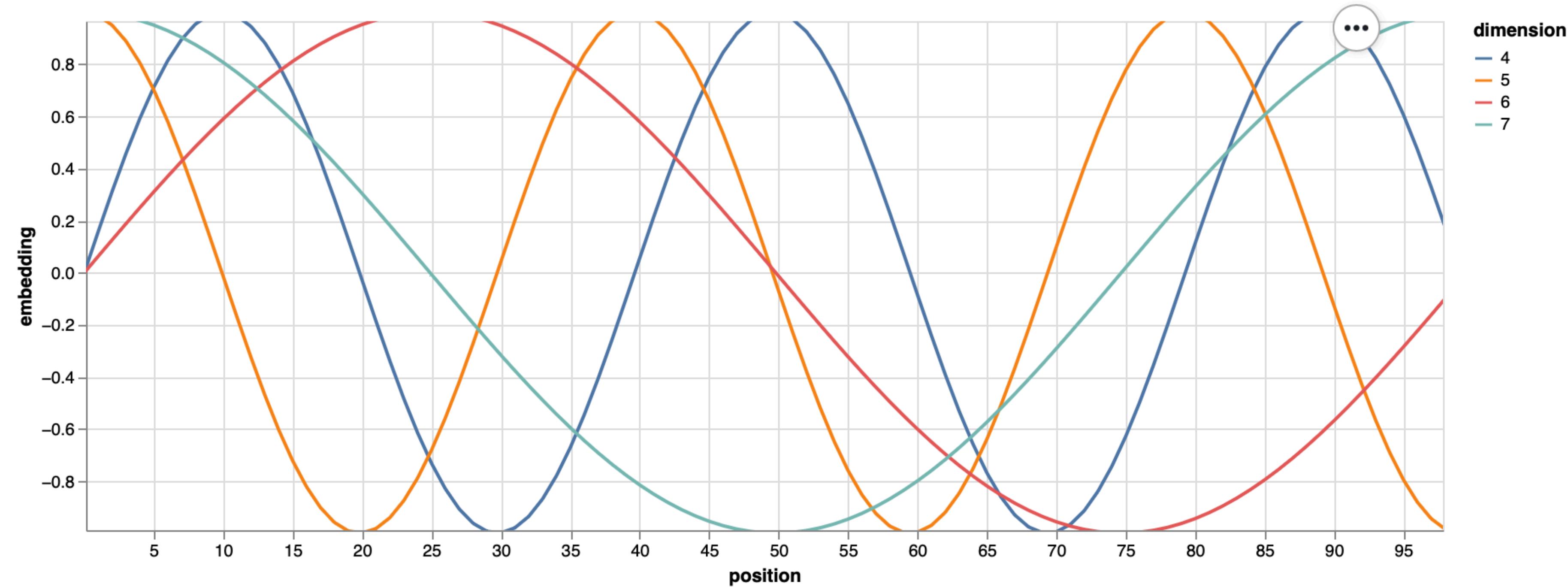
- We need to encode the order of the tokens in a sentence in the keys, values and queries
- We want a position embedding (similar to a word embedding)
- Let $\mathbf{p}_i \in \mathbb{R}^d$ for $i \in 1, \dots, n$ be the position embeddings
- If \mathbf{x}_i is the embedding for the word w_i then the combined word plus position embedding is $\tilde{\mathbf{x}}_i = \mathbf{x}_i + \mathbf{p}_i$
- Either concatenate \mathbf{x}_i and \mathbf{p}_i or just add them. Adding is more common.

Position embeddings without learning

cs224n-self-attention-transformers-2023_draft.pdf

Use a periodic function like sine and cosine with different periods to get an embedding vector without any parameter updates.

$$\mathbf{p}_i = \begin{pmatrix} \sin(i/10000^{2*1/d}) \\ \cos(i/10000^{2*1/d}) \\ \vdots \\ \sin(i/10000^{2*\frac{d}{2}/d}) \\ \cos(i/10000^{2*\frac{d}{2}/d}) \end{pmatrix}$$



Pros:

- * Periodicity means absolute position is not important
- * Can extrapolate to longer sequences as periods restart

Cons:

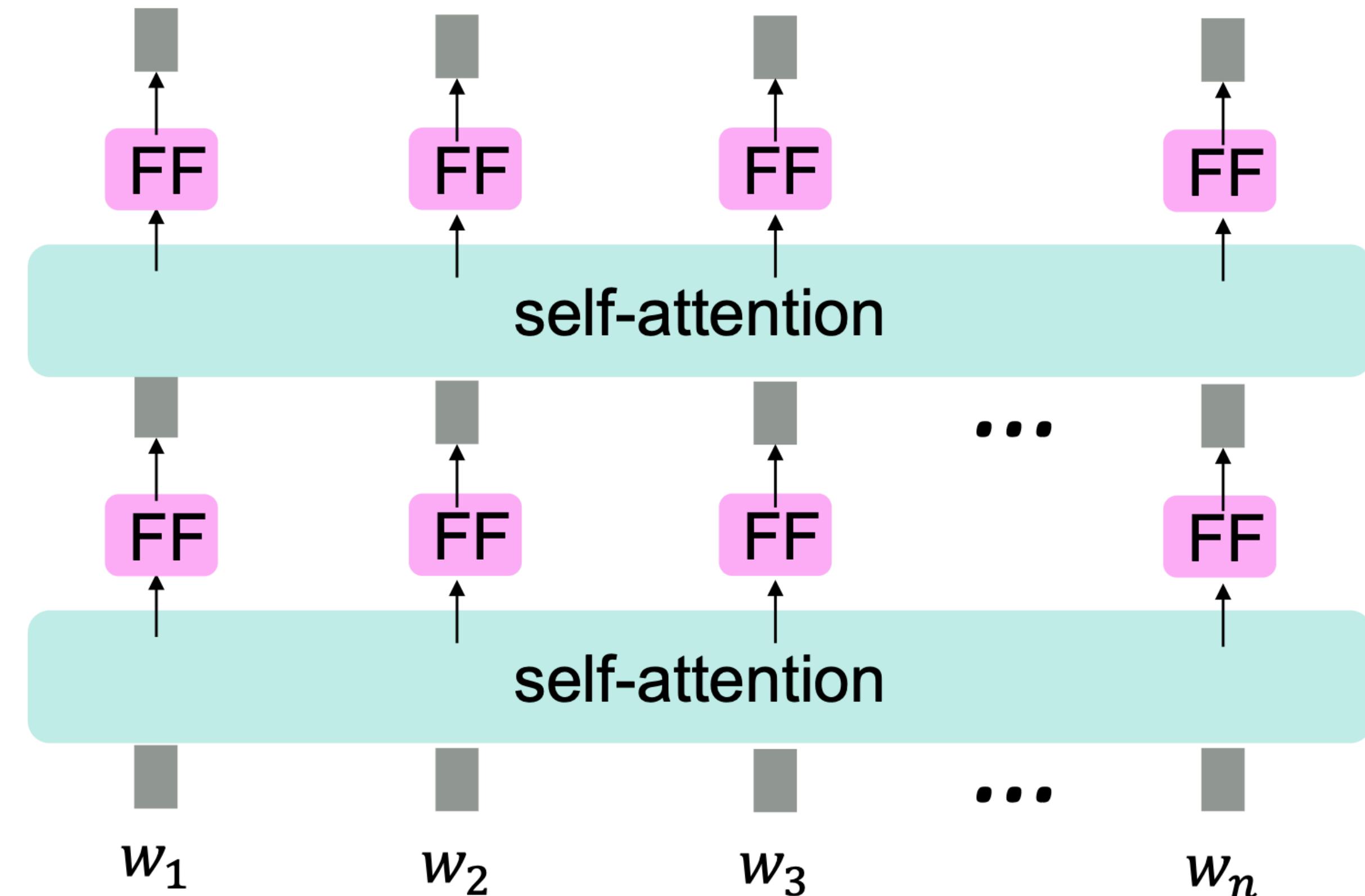
- * Not learnable
- * Extrapolation does not work that well for some applications

<http://nlp.seas.harvard.edu/annotated-transformer/#positional-encoding>

Self Attention Encoder using a Feed-forward Network

$$\begin{aligned} m_i &= \text{MLP}(\text{output}_i) \\ &= W_2 * \text{ReLU}(W_1 \text{ output}_i + b_1) + b_2 \end{aligned}$$

Intuition: the feed-forward (FF) network processes the attention vector and makes it usable by the next layer



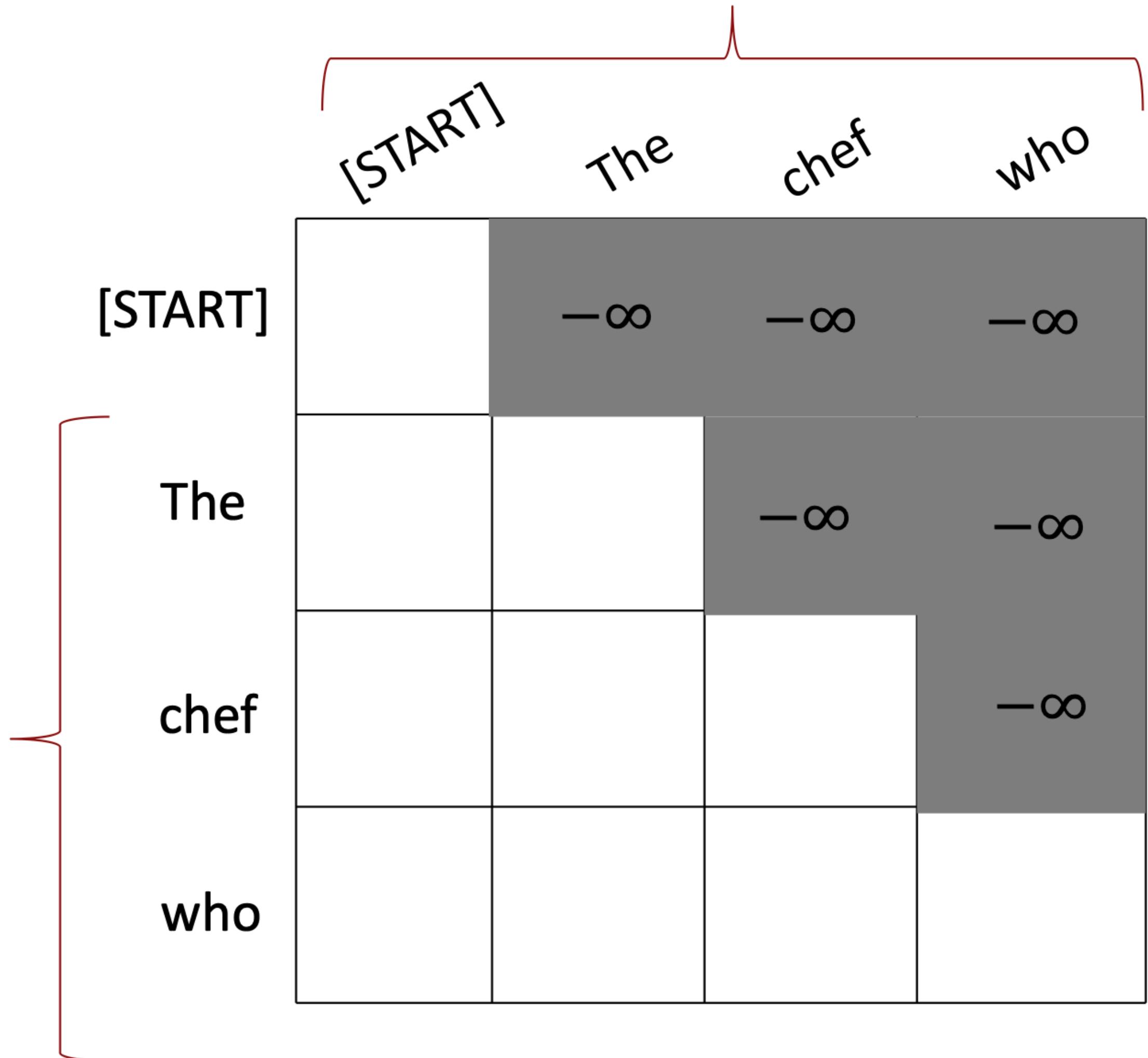
Decoders should **not** see into the future

- * During training we mask the attention vector by setting attention scores to $-\infty$
- * During inference, we decode from left to right and use the output from previous time-step as input to the next

$$e_{ij} = \begin{cases} q_i^T k_j, & j \leq i \\ -\infty, & j > i \end{cases}$$

For encoding
these words

We can only look at the non-greyed out words in the attention vector



Self-attention building block

- * **Self attention**
 - * need this!
- * **Position embeddings**
 - * since self-attention is unordered
- * **Nonlinearities**
 - * For the output of attention block
 - * Simple feed-forward network that is easy to train
- * **Masking**
 - * To parallelize operations while not looking at the future (during training)
 - * Enforces training to behave like inference

