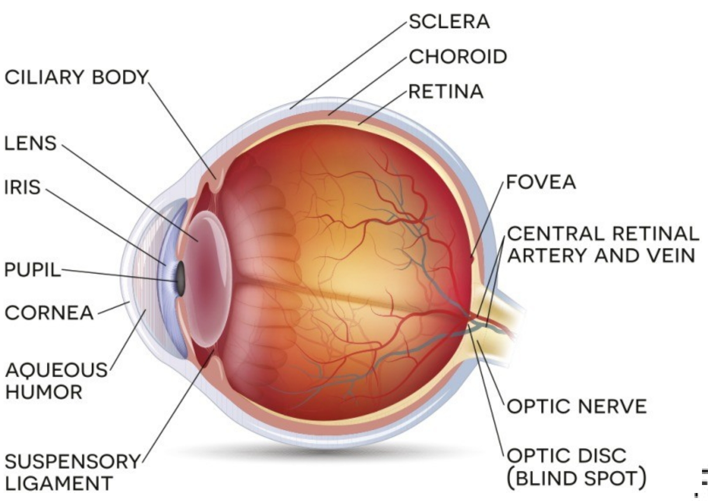
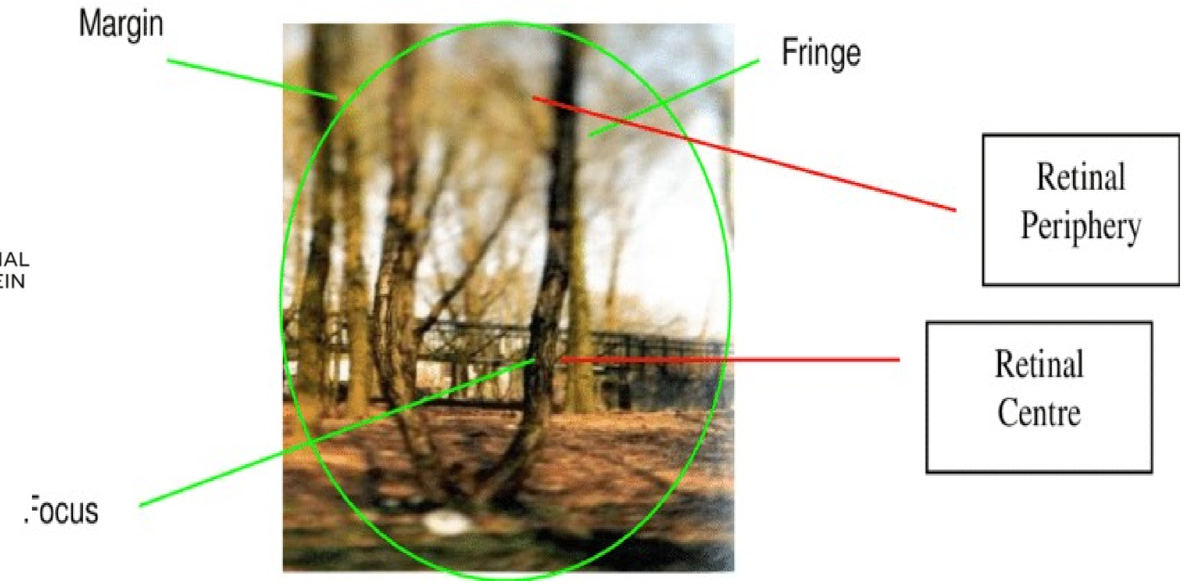
Computer Vision and Deep Learning

Lecture 10

Attention





humans exploit a sequence of partial glimpses and selectively focus on salient parts in order to capture visual structure better

https://www.researchgate.net/figure/Spotlight-model-of-visual-attention-The-attentional-spotlight-and-visual-sensitivity-is_fig5_266283321

CBAM – Convolutional Block Attention Module

Convolutional Block Attention Module (CBAM):

- a simple yet effective attention module for feed-forward convolutional neural networks

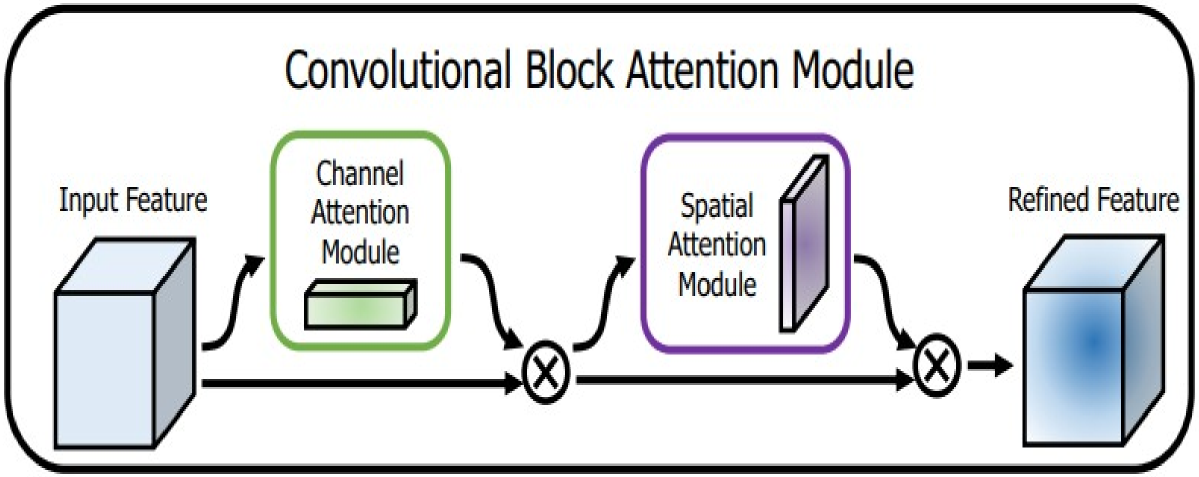


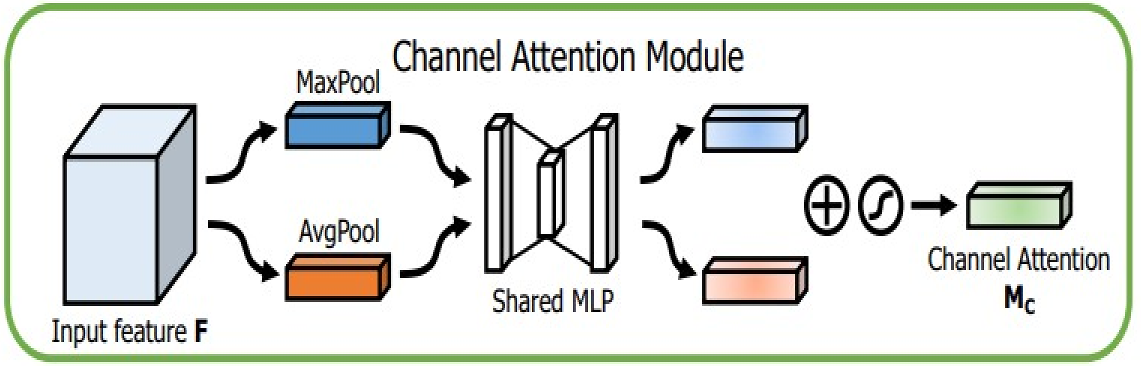
Fig. 1: **The overview of CBAM**. The module has two sequential sub-modules: *channel* and *spatial*. The intermediate feature map is adaptively refined through our module (CBAM) at every convolutional block of deep networks.

https://openaccess.thecvf.com/content_ECCV_2018/papers/Sanghyun_Woo_Convolutional_Block_Attention_ECCV_2018_paper.pdf

Channel Attention

- GAP(Global Average Pooling)
  - Aggregate spatial information
- GMP
  - Preserve richer context information
- Multilayer perceptron (MLP)
- Sigmoid activation
  - Gives the weights for each channel

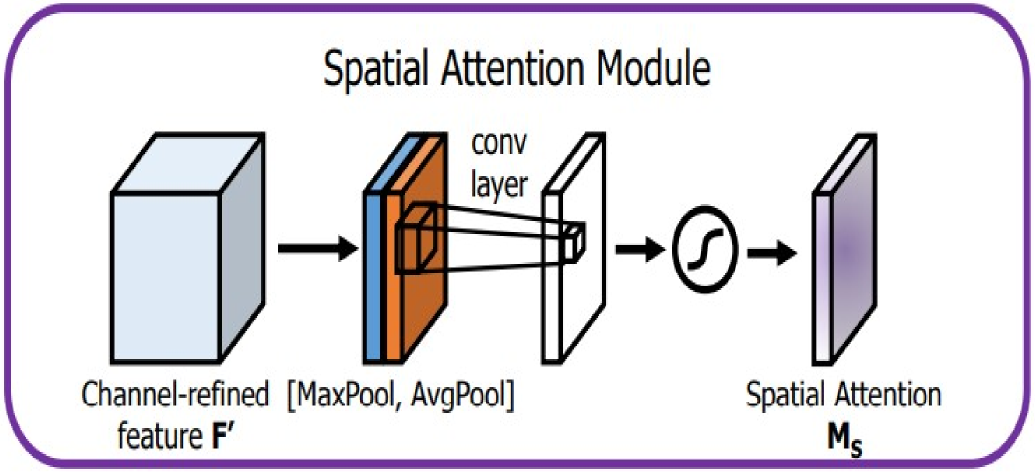
$$\begin{aligned}\mathbf{M_c}(\mathbf{F}) &= \sigma(MLP(AvgPool(\mathbf{F})) + MLP(MaxPool(\mathbf{F}))) \\ &= \sigma(\mathbf{W_1}(\mathbf{W_0}(\mathbf{F^c_{avg}})) + \mathbf{W_1}(\mathbf{W_0}(\mathbf{F^c_{max}}))),\end{aligned}$$



https://openaccess.thecvf.com/content_ECCV_2018/papers/Sanghyun_Woo_Convolutional_Block_Attention_ECCV_2018_paper.pdf

Spatial Attention

- Two pooling operations
- 1x1 Conv
- Sigmoid activation
  - Can be applied element-wise to all the positions in the input feature map



$$\begin{aligned}\mathbf{M_s}(\mathbf{F}) &= \sigma(f^{7\times7}([AvgPool(\mathbf{F}); MaxPool(\mathbf{F})])) \\ &= \sigma(f^{7\times7}([\mathbf{F}^{\mathbf{s}}_{\mathbf{avg}}; \mathbf{F}^{\mathbf{s}}_{\mathbf{max}}])),\end{aligned}$$

https://openaccess.thecvf.com/content_ECCV_2018/papers/Sanghyun_Woo_Convolutional_Block_Attention_ECCV_2018_paper.pdf

CBAM – Convolutional Block Attention Module

Architecture	Param.	GFLOPs	Top-1 Error (%)	Top-5 Error (%)
ResNet18 [5]	11.69M	1.814	29.60	10.55
ResNet18 [5] + SE [28]	11.78M	1.814	29.41	10.22
ResNet18 [5] + CBAM	11.78M	1.815	**29.27**	**10.09**
ResNet34 [5]	21.80M	3.664	26.69	8.60
ResNet34 [5] + SE [28]	21.96M	3.664	26.13	8.35
ResNet34 [5] + CBAM	21.96M	3.665	**25.99**	**8.24**
ResNet50 [5]	25.56M	3.858	24.56	7.50
ResNet50 [5] + SE [28]	28.09M	3.860	23.14	6.70
ResNet50 [5] + CBAM	28.09M	3.864	**22.66**	**6.31**
ResNet101 [5]	44.55M	7.570	23.38	6.88
ResNet101 [5] + SE [28]	49.33M	7.575	22.35	6.19
ResNet101 [5] + CBAM	49.33M	7.581	**21.51**	**5.69**
WideResNet18 [6] (widen=1.5)	25.88M	3.866	26.85	8.88
WideResNet18 [6] (widen=1.5) + SE [28]	26.07M	3.867	26.21	8.47
WideResNet18 [6] (widen=1.5) + CBAM	26.08M	3.868	**26.10**	**8.43**
WideResNet18 [6] (widen=2.0)	45.62M	6.696	25.63	8.20
WideResNet18 [6] (widen=2.0) + SE [28]	45.97M	6.696	24.93	7.65
WideResNet18 [6] (widen=2.0) + CBAM	45.97M	6.697	**24.84**	**7.63**
ResNeXt50 [7] (32x4d)	25.03M	3.768	22.85	6.48
ResNeXt50 [7] (32x4d) + SE [28]	27.56M	3.771	**21.91**	6.04
ResNeXt50 [7] (32x4d) + CBAM	27.56M	3.774	21.92	**5.91**
ResNeXt101 [7] (32x4d)	44.18M	7.508	21.54	5.75
ResNeXt101 [7] (32x4d) + SE [28]	48.96M	7.512	21.17	5.66
ResNeXt101 [7] (32x4d) + CBAM	48.96M	7.519	**21.07**	**5.59**

https://openaccess.thecvf.com/content_ECCV_2018/papers/Sanghyun_Woo_Convolutional_Block_Attention_ECCV_2018_paper.pdf

Recurrent neural networks

Other resources:

RNNs: http://karpathy.github.io/2015/05/21/rnn-effectiveness/ (**HIGHLY RECOMMENDED**)

https://www.youtube.com/watch?v=yCC09vCHzF8

https://colah.github.io/posts/2015-08-Understanding-LSTMs/

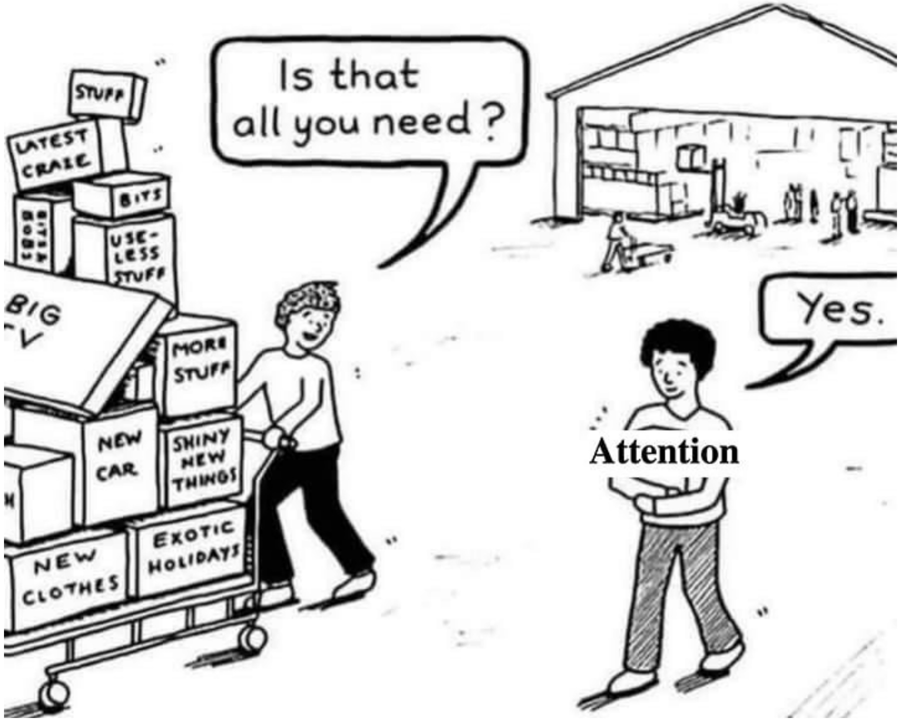
Self attention and transformers (Stanford lecture 2021): https://www.youtube.com/watch?v=ptuGllU5SQQ

Jay Alammar series on Transformers & BERT:

- https://jalammar.github.io/illustrated-transformer/
- https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/

- https://jalammar.github.io/illustrated-bert/

Attention is all you need



https://arxiv.org/abs/1706.03762

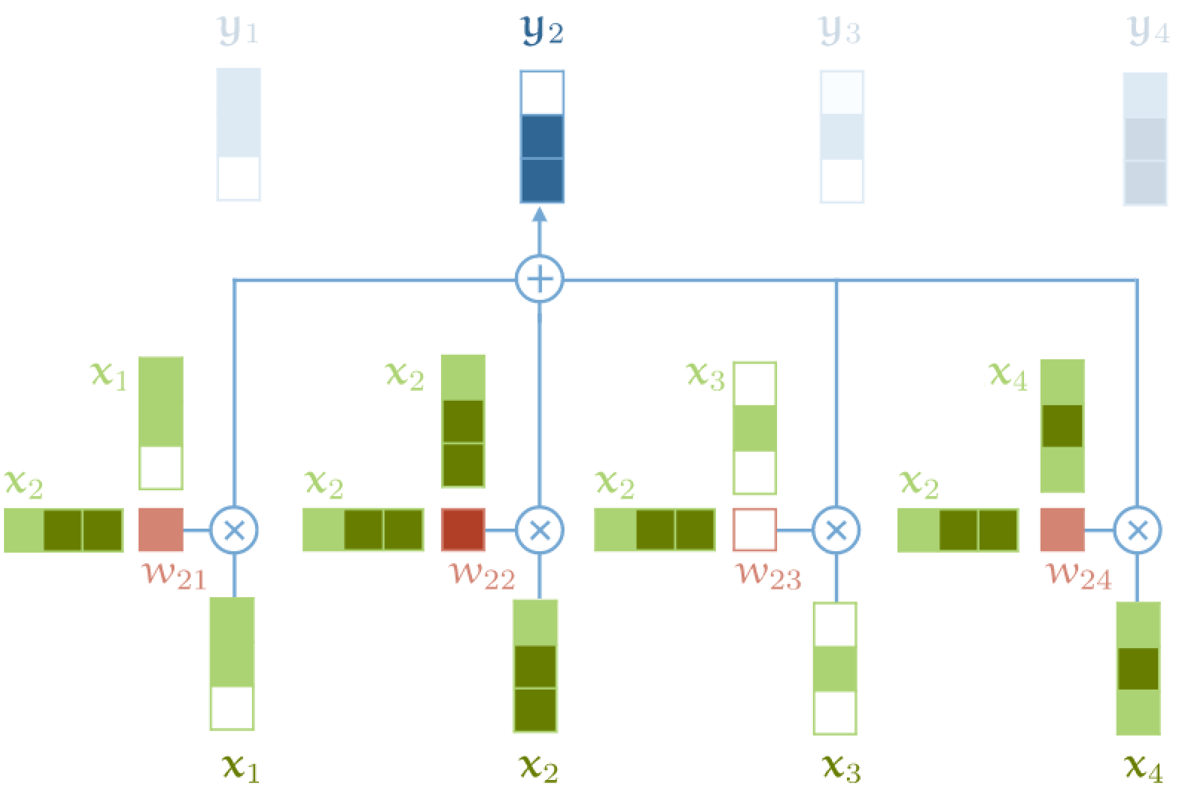
Self attention

Self-attention, sometimes called intra-attention is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence. Self-attention has been used successfully in a variety of tasks including reading comprehension, abstractive summarization, textual entailment and learning task-independent sentence representations [4, 27, 28, 22].

$$\mathbf{y}_i = \sum_j w_{ij} \mathbf{x}_j \, .$$

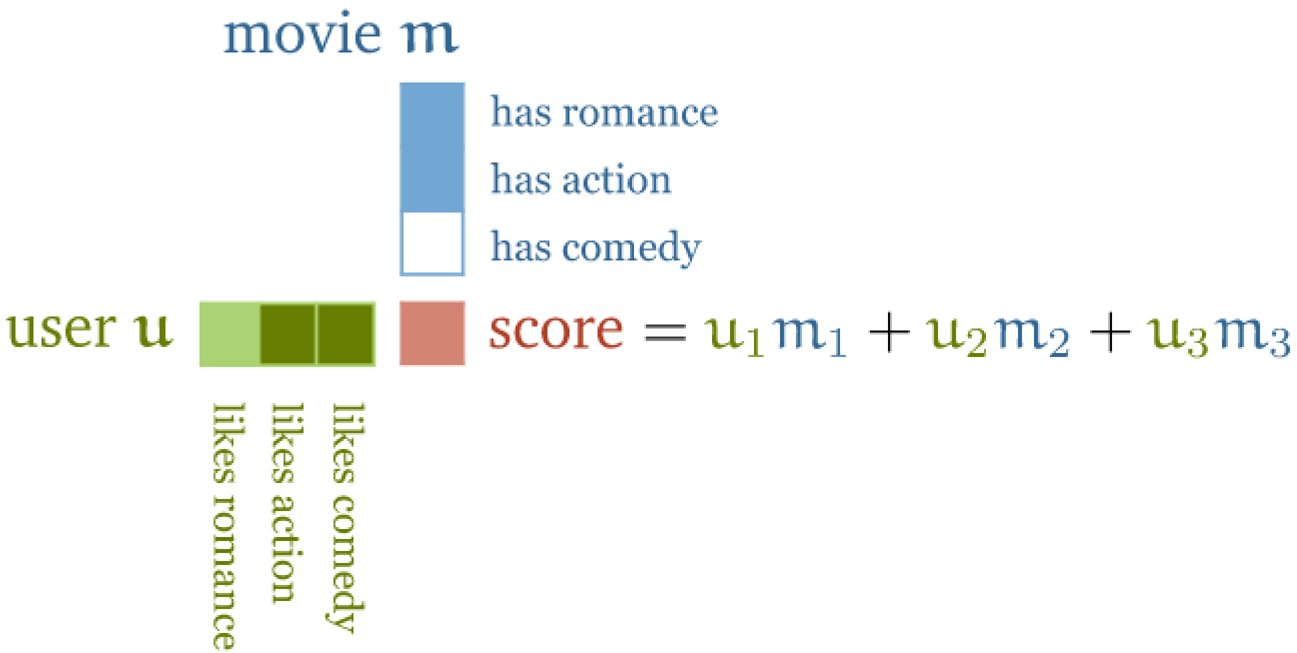
$$w'_{ij} = \mathbf{x}_i^{\mathsf{T}} \mathbf{x}_j \, .$$

$$w_{ij} = \frac{\exp w'_{ij}}{\sum_j \exp w'_{ij}} \, .$$



http://peterbloem.nl/blog/transformers

Self attention



http://peterbloem.nl/blog/transformers

Attention: query, keys and values

"An ***attention function*** can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors."

$$\mathbf{q}_i = \boldsymbol{W}_q \boldsymbol{x}_i \qquad \mathbf{k}_i = \boldsymbol{W}_k \boldsymbol{x}_i \qquad \boldsymbol{v}_i = \boldsymbol{W}_v \boldsymbol{x}_i$$

$$w'_{ij} = \mathbf{q}_i{}^\top \mathbf{k}_j$$

$$w_{ij} = \mathrm{softmax}(w'_{ij})$$

$$\mathbf{y}_i = \sum_j w_{ij} \boldsymbol{v}_j \,.$$

Attention: query, keys and values

$$\mathbf{q}_i = \mathbf{W}_q \mathbf{x}_i \qquad \mathbf{k}_i = \mathbf{W}_k \mathbf{x}_i \qquad \mathbf{v}_i = \mathbf{W}_v \mathbf{x}_i$$

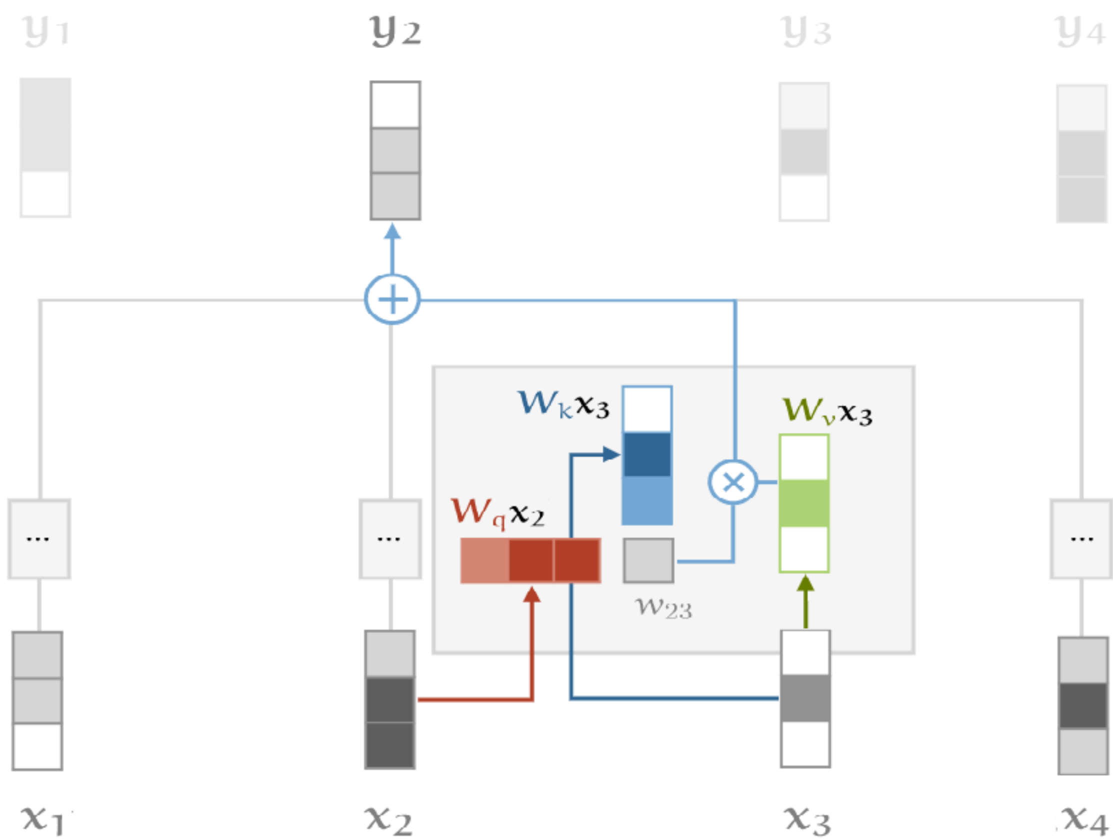


Illustration of the self-attention with key, query and value

Scaled Dot-Product Attention

We call our particular attention "Scaled Dot-Product Attention" (Figure 2). The input consists of queries and keys of dimension d_k, and values of dimension d_v. We compute the dot products of the query with all keys, divide each by $\sqrt{d_k}$, and apply a softmax function to obtain the weights on the values.

While for small values of d_k the two mechanisms perform similarly, additive attention outperforms dot product attention without scaling for larger values of d_k [3]. We suspect that for large values of d_k, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients [4]. To counteract this effect, we scale the dot products by $\frac{1}{\sqrt{d_k}}$.

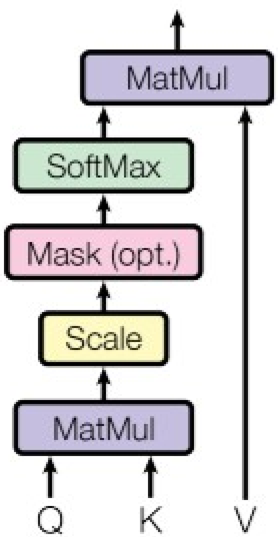
k(d_k in the paper) – dimension of the embedding

$$w'_{ij} = \frac{\mathbf{q}_i^{\mathrm{T}} \mathbf{k}_j}{\sqrt{k}}$$

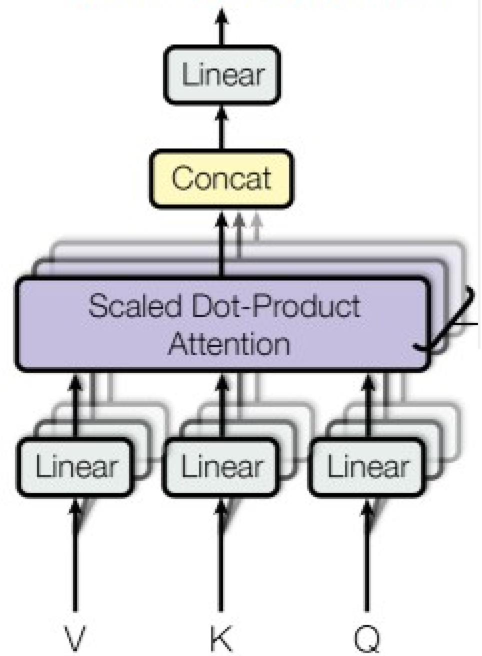
```python
def scaled_dot_product_attention(queries, keys, values, mask):
    # Calculate the dot product, QK_transpose
    product = tf.matmul(queries, keys, transpose_b=True)
    # Get the scale factor
    keys_dim = tf.cast(tf.shape(keys)[-1], tf.float32)
    # Apply the scale factor to the dot product
    scaled_product = product / tf.math.sqrt(keys_dim)
    # Apply masking when it is requiered
    if mask is not None:
        scaled_product += (mask * -1e9)
    # dot product with Values
    attention = tf.matmul(tf.nn.softmax(scaled_product, axis=-1), values)

    return attention
```

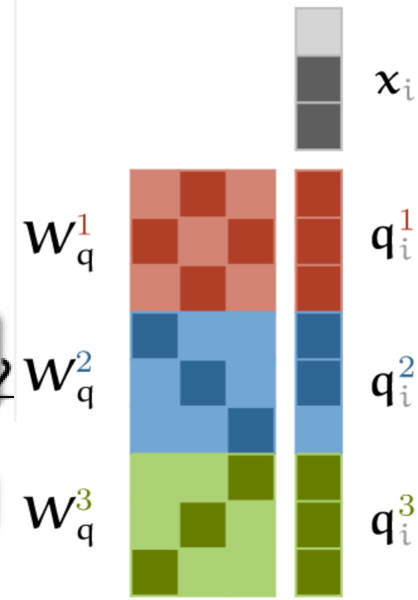
Multi-head attention

Scaled Dot-Product Attention



Multi-Head Attention





Combining three attention heads into one matrix multiplication (for the queries).

$$\mathrm{MultiHead}(Q, K, V) = \mathrm{Concat}(\mathrm{head_1}, ..., \mathrm{head_h})W^O$$
$$\text{where } \mathrm{head_i} = \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

```python
class MultiHeadAttention(layers.Layer):

    def __init__(self, n_heads):
        super(MultiHeadAttention, self).__init__()
        self.n_heads = n_heads

    def build(self, input_shape):
        self.d_model = input_shape[-1]
        assert self.d_model % self.n_heads == 0
        # Calculate the dimension of every head or projection
        self.d_head = self.d_model // self.n_heads
        # Set the weight matrices for Q, K and V
        self.query_lin = layers.Dense(units=self.d_model)
        self.key_lin = layers.Dense(units=self.d_model)
        self.value_lin = layers.Dense(units=self.d_model)
        # Set the weight matrix for the output of the multi-head attention W0
        self.final_lin = layers.Dense(units=self.d_model)
```

https://towardsdatascience.com/attention-is-all-you-need-discovering-the-transformer-paper-73e5ff5e0634

```python
def call(self, queries, keys, values, mask):
    # Get the batch size
    batch_size = tf.shape(queries)[0]
    # Set the Query, Key and Value matrices
    queries = self.query_lin(queries)
    keys = self.key_lin(keys)
    values = self.value_lin(values)
    # Split Q, K y V between the heads or projections
    queries = self.split_proj(queries, batch_size)
    keys = self.split_proj(keys, batch_size)
    values = self.split_proj(values, batch_size)
    # Apply the scaled dot product
    attention = scaled_dot_product_attention(queries, keys, values, mask)
    # Get the attention scores
    attention = tf.transpose(attention, perm=[0, 2, 1, 3])
    # Concat the h heads or projections
    concat_attention = tf.reshape(attention,
                                  shape=(batch_size, -1, self.d_model))
    # Apply W0 to get the output of the multi-head attention
    outputs = self.final_lin(concat_attention)

    return outputs
```

https://towardsdatascience.com/attention-is-all-you-need-discovering-the-transformer-paper-73e5ff5e0634

```
lef split_proj(self, inputs, batch_size): # inputs: (batch_size, seq_length, d_model)
    # Set the dimension of the projections
    shape = (batch_size,
             -1,
             self.n_heads,
             self.d_head)
    # Split the input vectors
    splited_inputs = tf.reshape(inputs, shape=shape) # (batch_size, seq_length, nb_proj, d_proj)
    return tf.transpose(splited_inputs, perm=[0, 2, 1, 3]) # (batch_size, nb_proj, seq_length, d_proj)
```

https://towardsdatascience.com/attention-is-all-you-need-discovering-the-transformer-paper-73e5ff5e0634

Transformer architecture

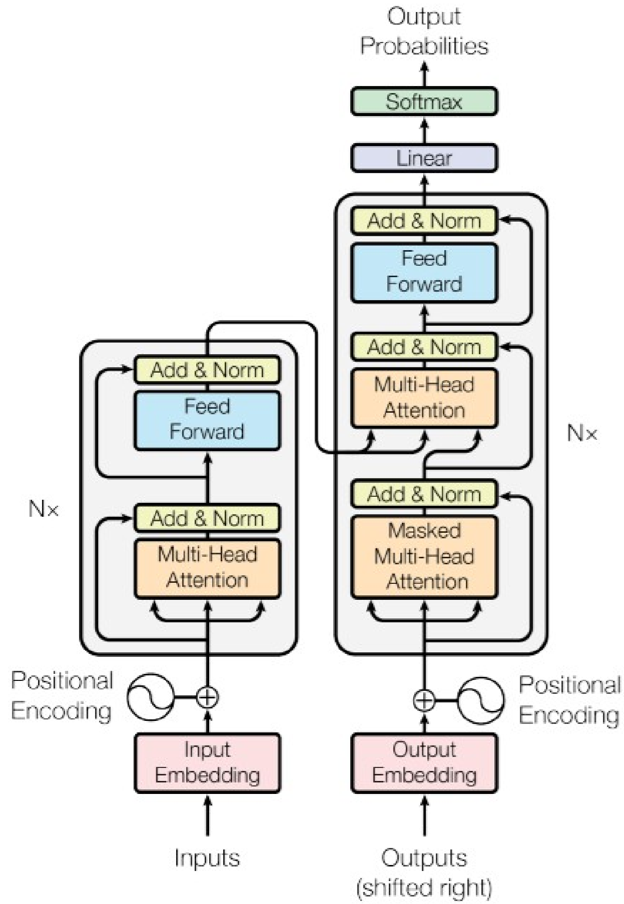


Figure 1: The Transformer - model architecture.

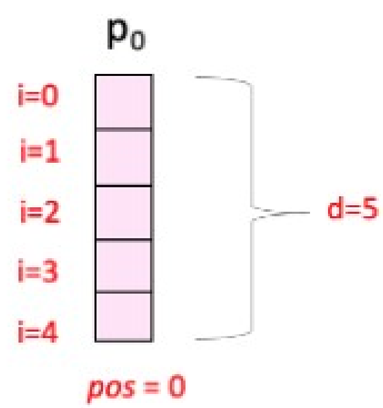
Most competitive neural sequence transduction models have an encoder-decoder structure [5, 2, 35]. Here, the encoder maps an input sequence of symbol representations $(x_1, ..., x_n)$ to a sequence of continuous representations $\mathbf{z} = (z_1, ..., z_n)$. Given $\mathbf{z}$, the decoder then generates an output sequence $(y_1, ..., y_m)$ of symbols one element at a time. At each step the model is auto-regressive [10], consuming the previously generated symbols as additional input when generating the next.

Positional Encoding

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

where pos is the position and i is the dimension. That is, each dimension of the positional encoding corresponds to a sinusoid. The wavelengths form a geometric progression from 2π to $10000 \cdot 2\pi$. We

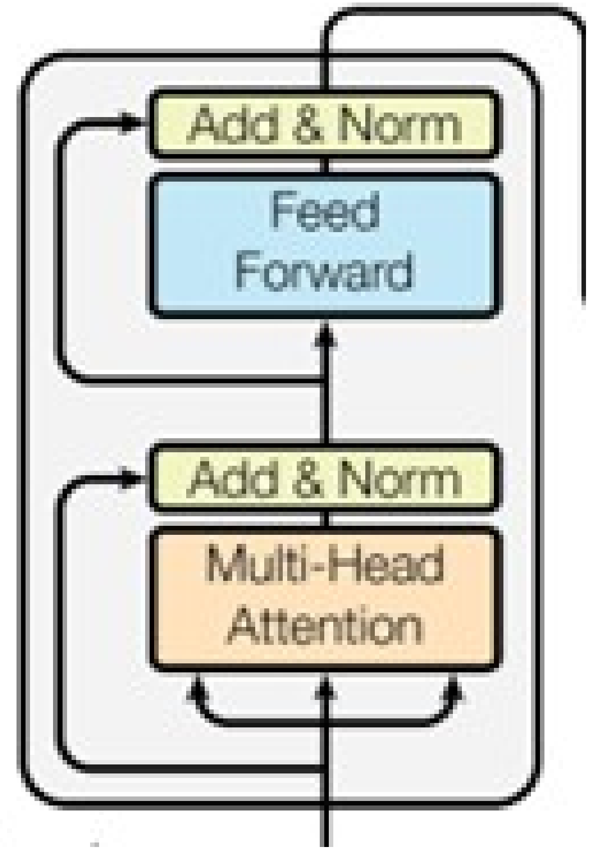
$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$



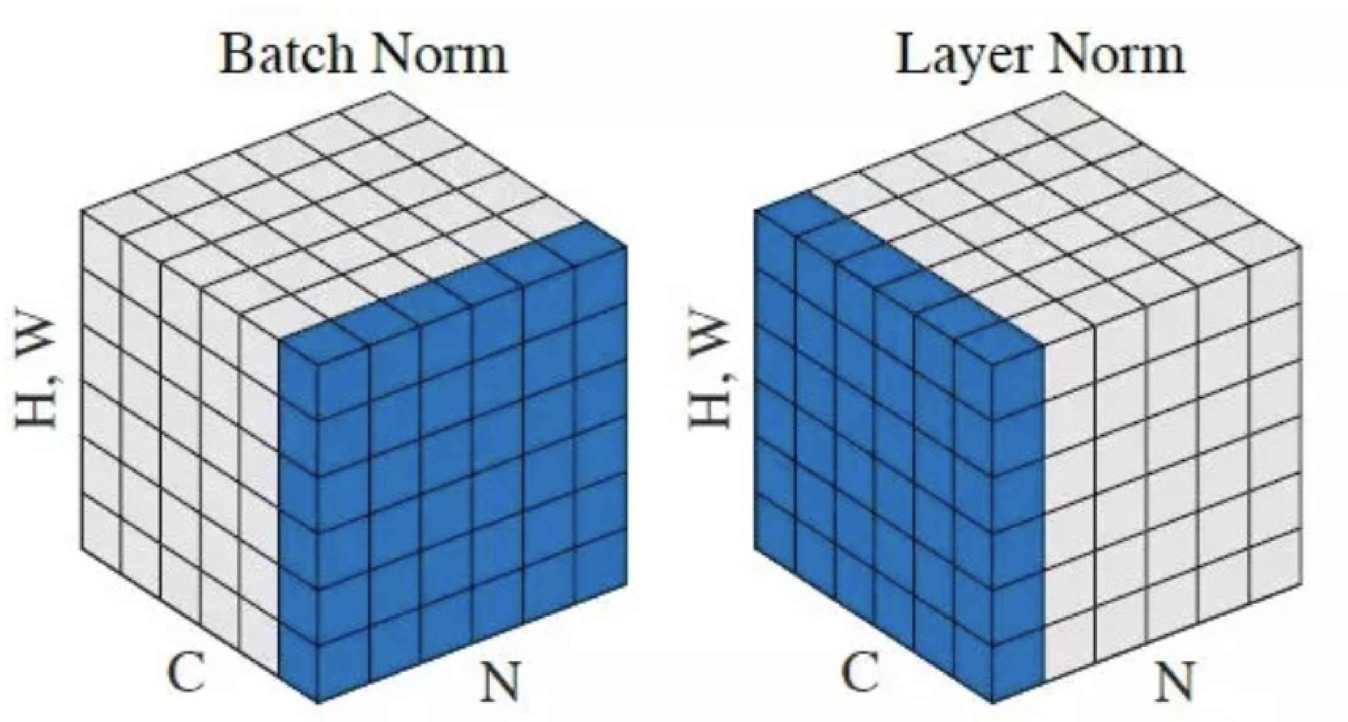
Transformer: encoder block

- Multi head attention layer
- Layer normalization
- Feed forward
  layer (applied
  independently to each
  vector)
- Layer normalization

+ Residual
connections (before the
normalizations)

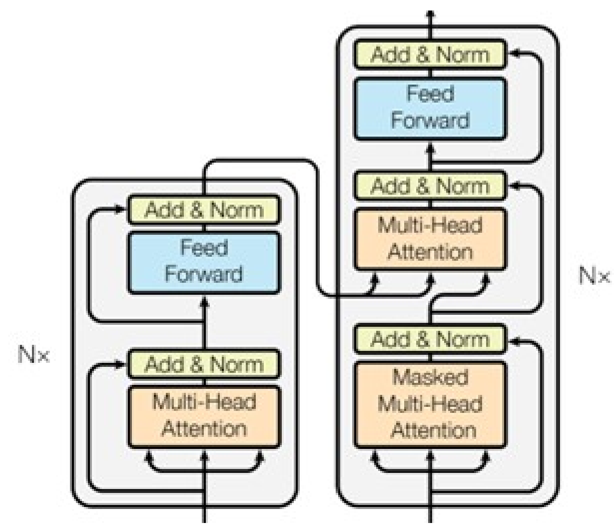


Batch norm vs Layer norm

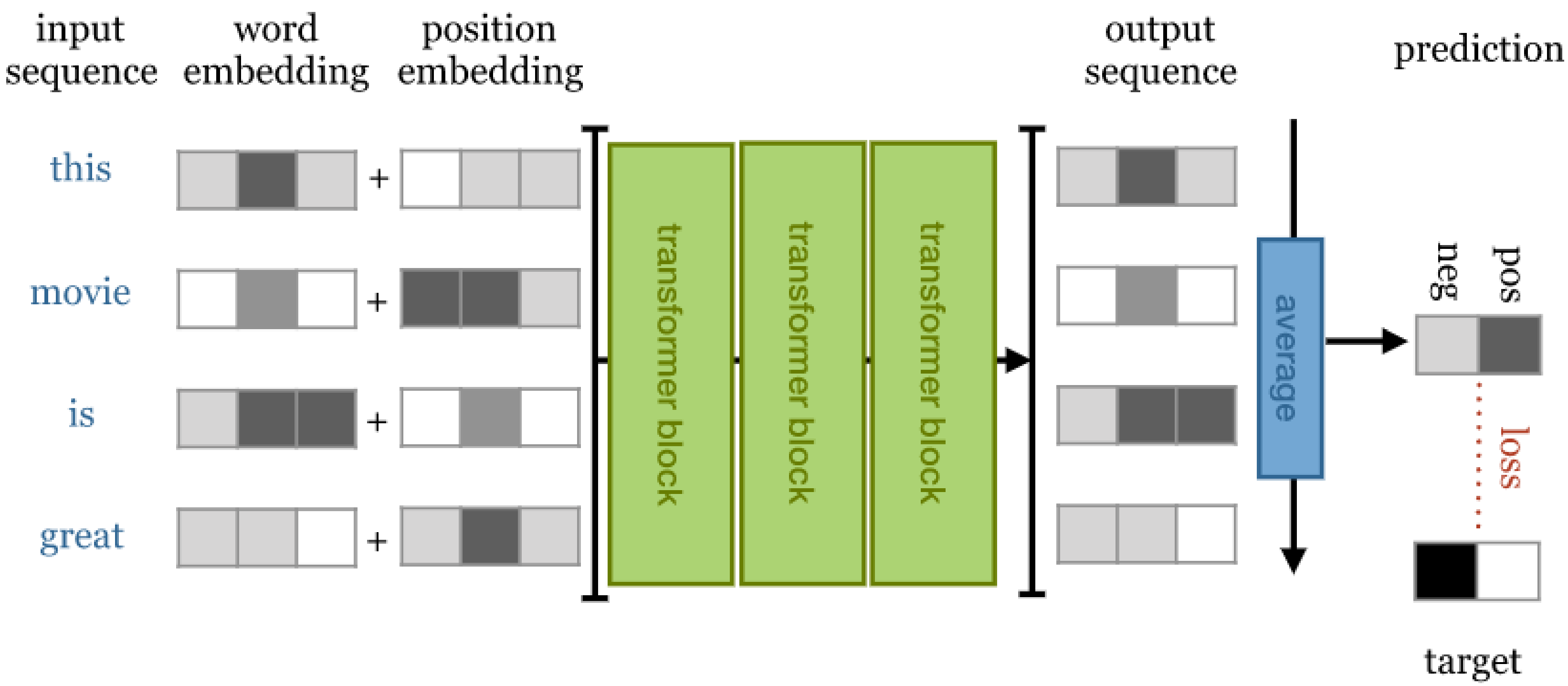


Transformer: decoder block

- Input: block on the same level in the encoder and the previous level in the decoder
- Masked multi head attention



Simple transformer for sentiment analysis



BERT

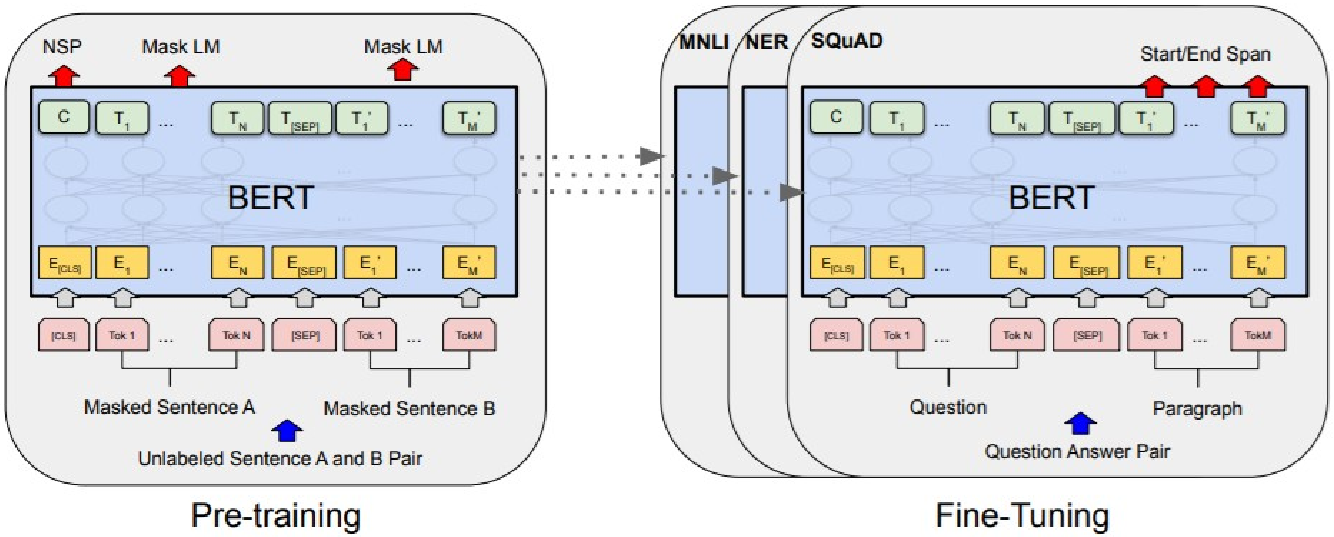


Figure 1: Overall pre-training and fine-tuning procedures for BERT. Apart from output layers, the same architectures are used in both pre-training and fine-tuning. The same pre-trained model parameters are used to initialize models for different down-stream tasks. During fine-tuning, all parameters are fine-tuned. [CLS] is a special symbol added in front of every input example, and [SEP] is a special separator token (e.g. separating questions/answers).

https://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/

BERT

- ***PRE-TRAINED*** on a large general-domain corpus consisting of 800M words from English books 2.5B words of text from English Wikipedia articles.
  - **Masking:** words are masked out, replaced with a random word or kept as they are. The model is then asked to predict, for **only** these words, what the original words were.

BERT

- ***PRETRAINING***
  - **Masking**
  - **Next Sentence Prediction:** Two sequences of about 256 words are sampled that either (a) follow each other directly in the corpus, or (b) are both taken from random places. The model must then predict whether a or b is the case.

BERT

- Input is prepended with a special <CLS> token. The output vector corresponding to this token is used as a sentence representation in sequence classification tasks like the next sentence classification.

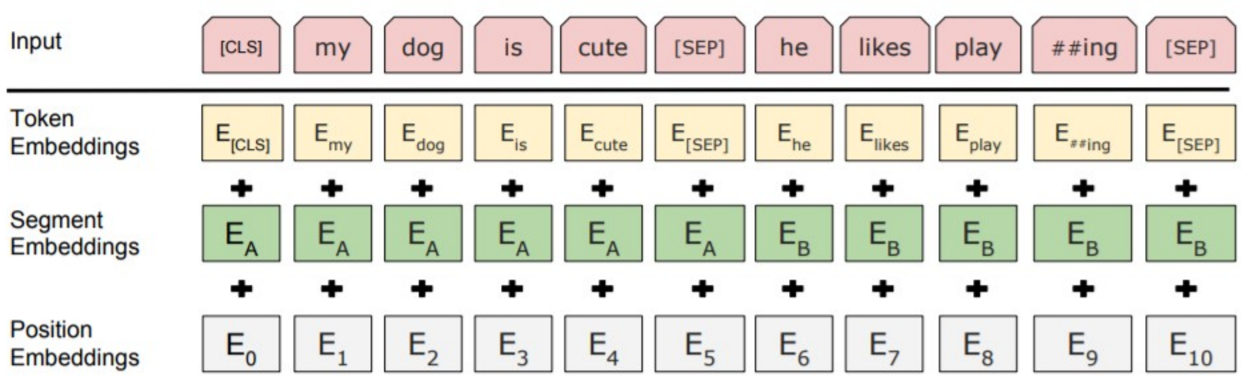


Figure 2: BERT input representation. The input embeddings are the sum of the token embeddings, the segmentation embeddings and the position embeddings.

BERT

- **FINE-TUNING**
  - a single task-specific layer is placed after the transformer blocks, which maps the general purpose representation to a task specific output.
- **Classification:** maps the first output token (corresponding to <CLS>) to softmax probabilities over the classes

An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale

Vision transformer

Official implementation in JAX:

https://github.com/google-research/vision_transformer#vision-transformer

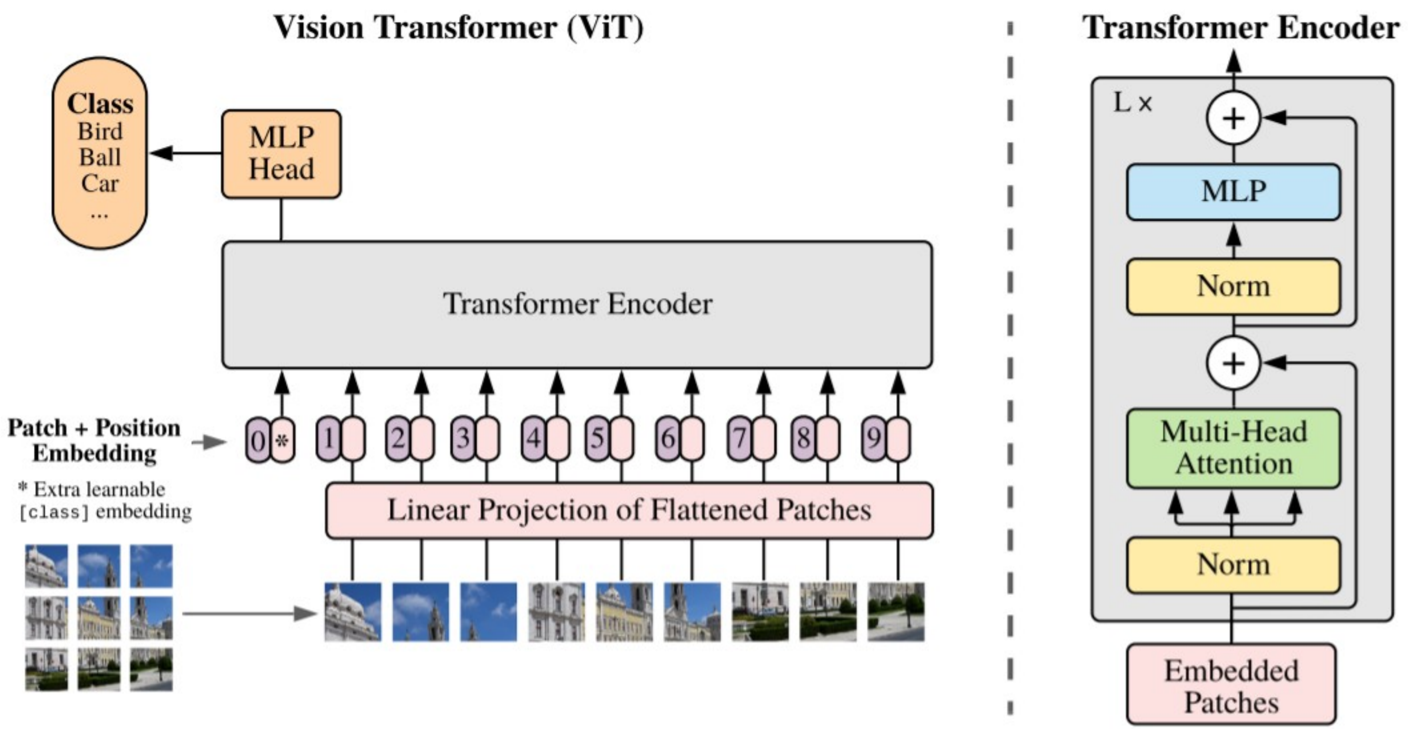
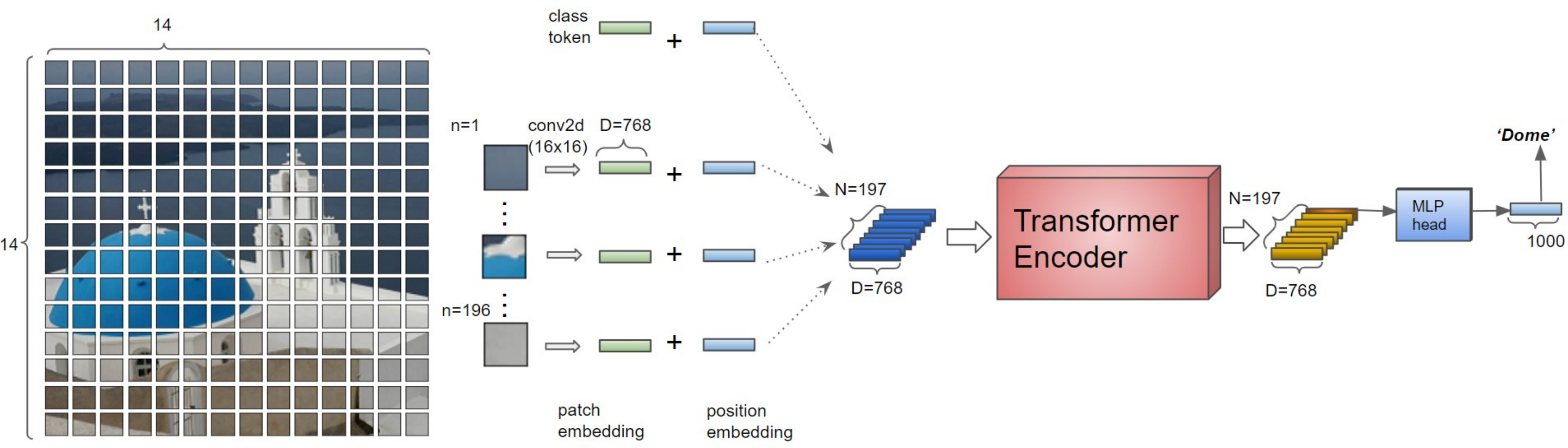


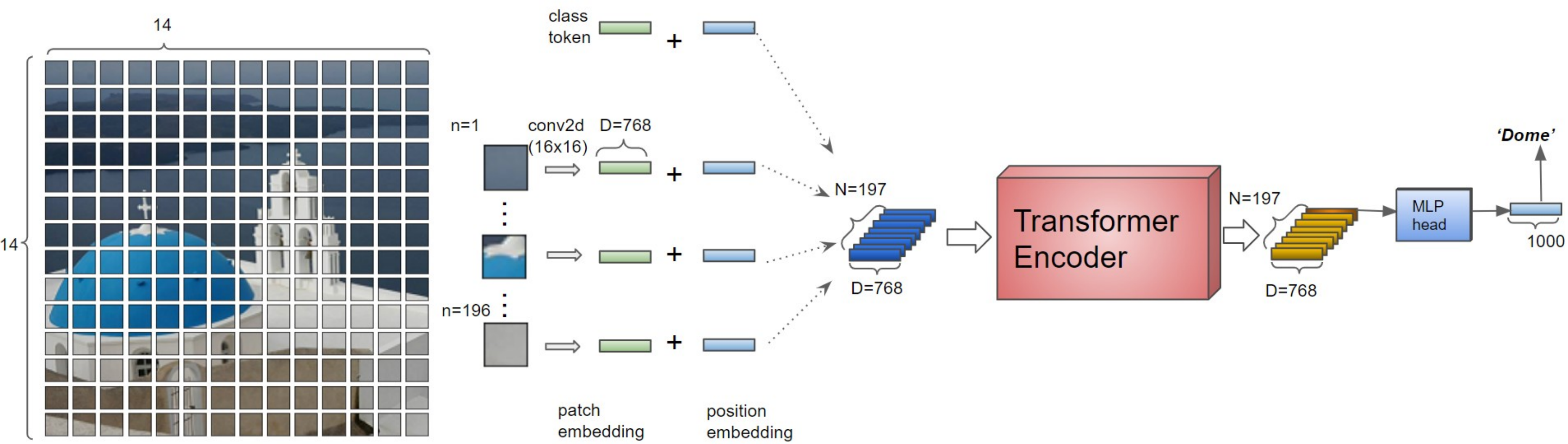
Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable "classification token" to the sequence. The illustration of the Transformer encoder was inspired by Vaswani et al. (2017).



1. Split the image into patches and encode each patch

```
x = nn.Conv(
    features=self.hidden_size,
    kernel_size=self.patches.size,
    strides=self.patches.size,
    padding='VALID',
    name='embedding')(
        x)
```

https://colab.research.google.com/github/hirotomusiker/schwert_colab_data_storage/blob/master/notebook/Vision_Transformer_Tutorial.ipynb#scrollTo=nI6rRunEO6bI

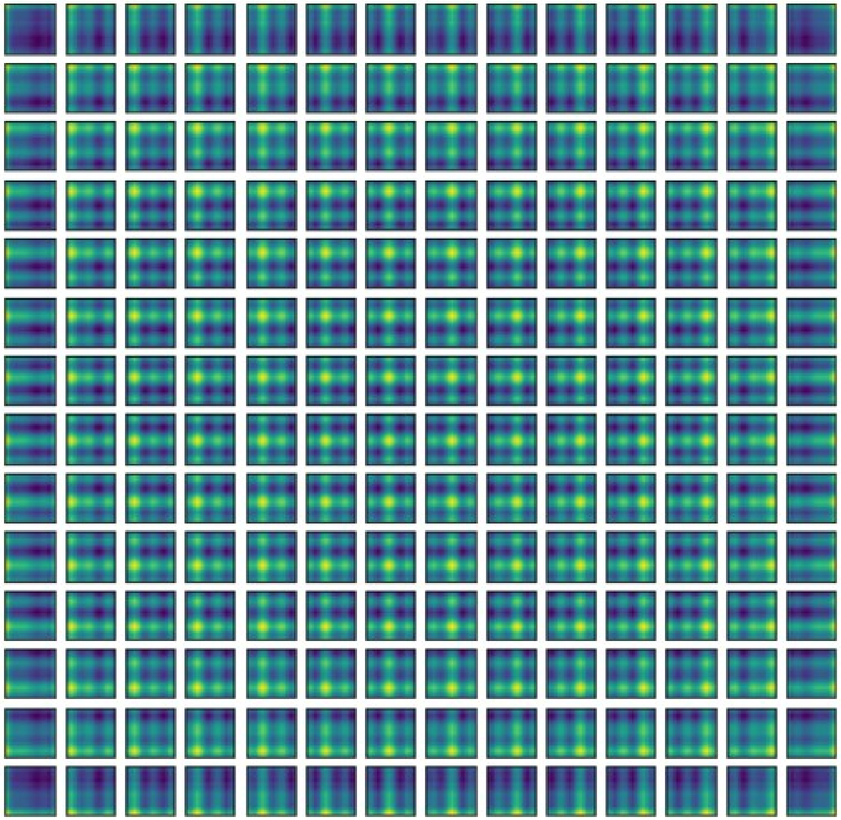


1. Split the image into patches and encode each patch
2. Add Position Embeddings

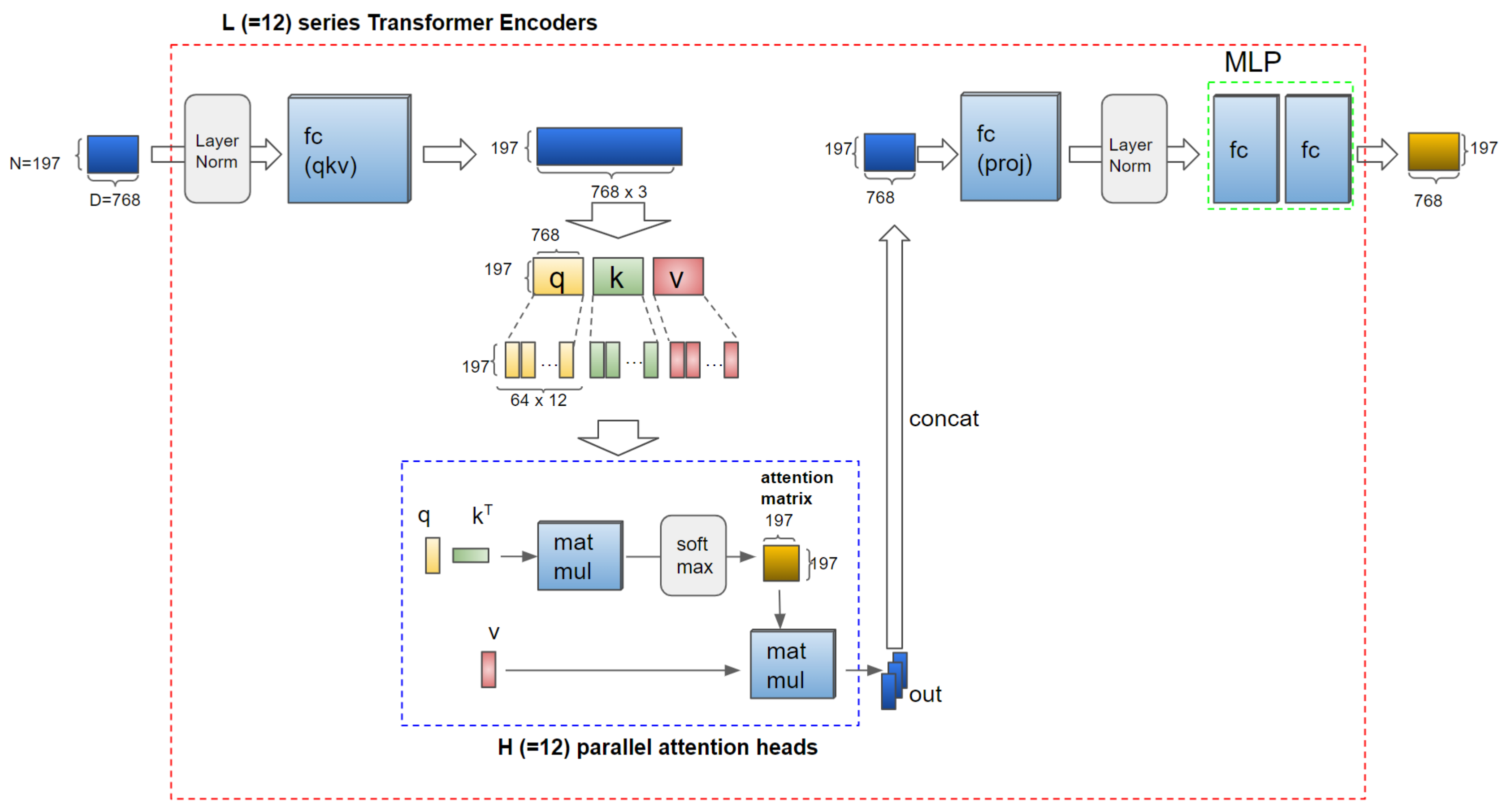
https://colab.research.google.com/github/hirotomusiker/schwert_colab_data_storage/blob/master/notebook/Vision_Transformer_Tutorial.ipynb#scrollTo=nI6rRunEO6bI

Position Embeddings Visualisation

```
# Visualize position embedding similarities.
# One cell shows cos similarity between an embedding and all the other embeddin
cos = torch.nn.CosineSimilarity(dim=1, eps=1e-6)
fig = plt.figure(figsize=(8, 8))
fig.suptitle("Visualization of position embedding similarities", fontsize=24)
for i in range(1, pos_embed.shape[1]):
    sim = F.cosine_similarity(pos_embed[0, i:i+1], pos_embed[0, 1:], dim=1)
    sim = sim.reshape((14, 14)).detach().cpu().numpy()
    ax = fig.add_subplot(14, 14, i)
    ax.axes.get_xaxis().set_visible(False)
    ax.axes.get_yaxis().set_visible(False)
    ax.imshow(sim)
```



1. Split the image into patches and encode each patch
2. Add Position Embeddings
3. Transformer Encoder



https://colab.research.google.com/github/hirotomusiker/schwert_colab_data_storage/blob/master/notebook/Vision_Transformer_Tutorial.ipynb#scrollTo=nI6rRunEO6bI

VISION TRANSFORMER

1. Split the image into patches and encode each patch
2. Add Position Embeddings
3. Transformer Encoder
4. 4. MLP (Classification) Head

https://colab.research.google.com/github/hirotomusiker/schwert_colab_data_storage/blob/master/notebook/Vision_Transformer_Tutorial.ipynb#scrollTo=nI6rRunEO6bI

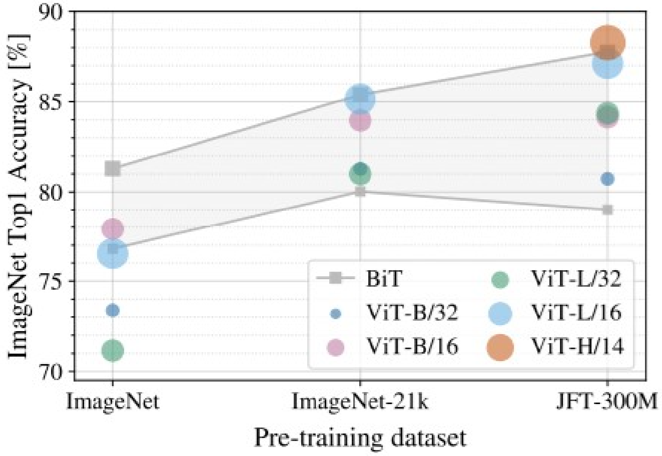


Figure 3: Transfer to ImageNet. While large ViT models perform worse than BiT ResNets (shaded area) when pre-trained on small datasets, they shine when pre-trained on larger datasets. Similarly, larger ViT variants overtake smaller ones as the dataset grows.

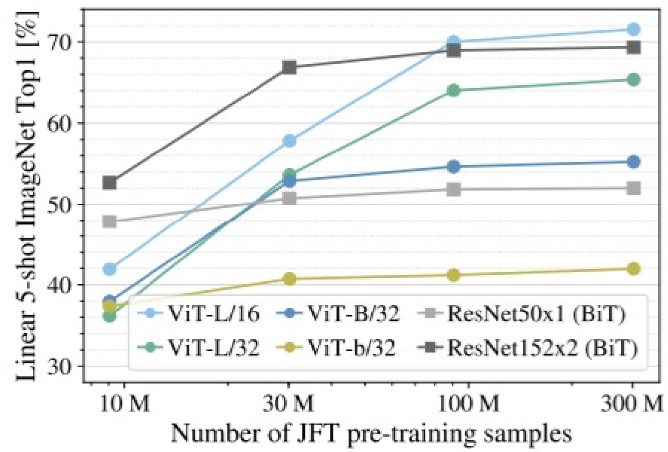
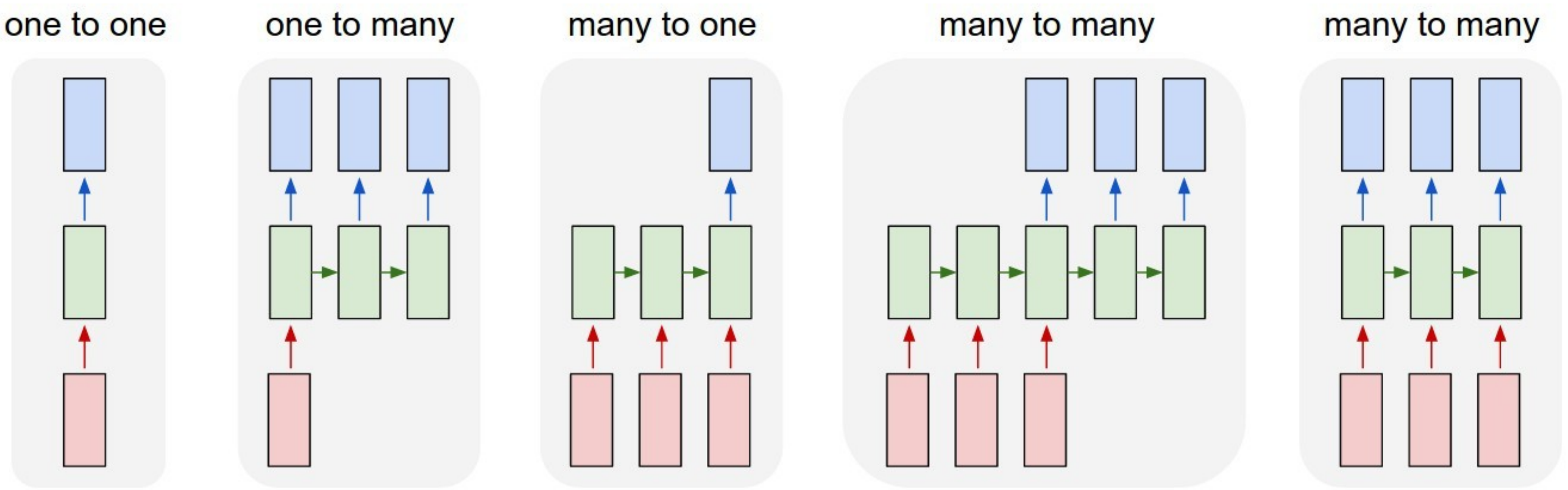


Figure 4: Linear few-shot evaluation on ImageNet versus pre-training size. ResNets perform better with smaller pre-training datasets but plateau sooner than ViT, which performs better with larger pre-training. ViT-b is ViT-B with all hidden dimensions halved.

Recurrent Neural Networks

YOU DON'T NEED TO LEARN THIS FOR THE EXAM!!!

Process sequences



Process sequences

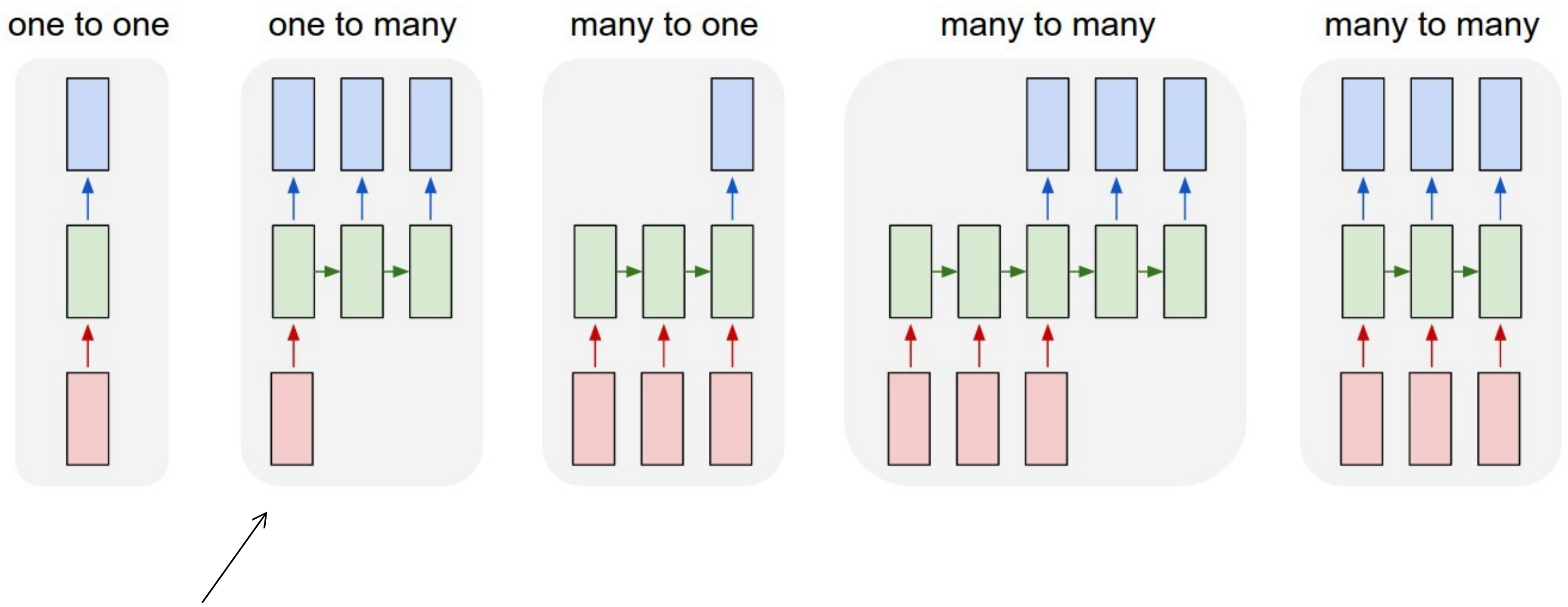
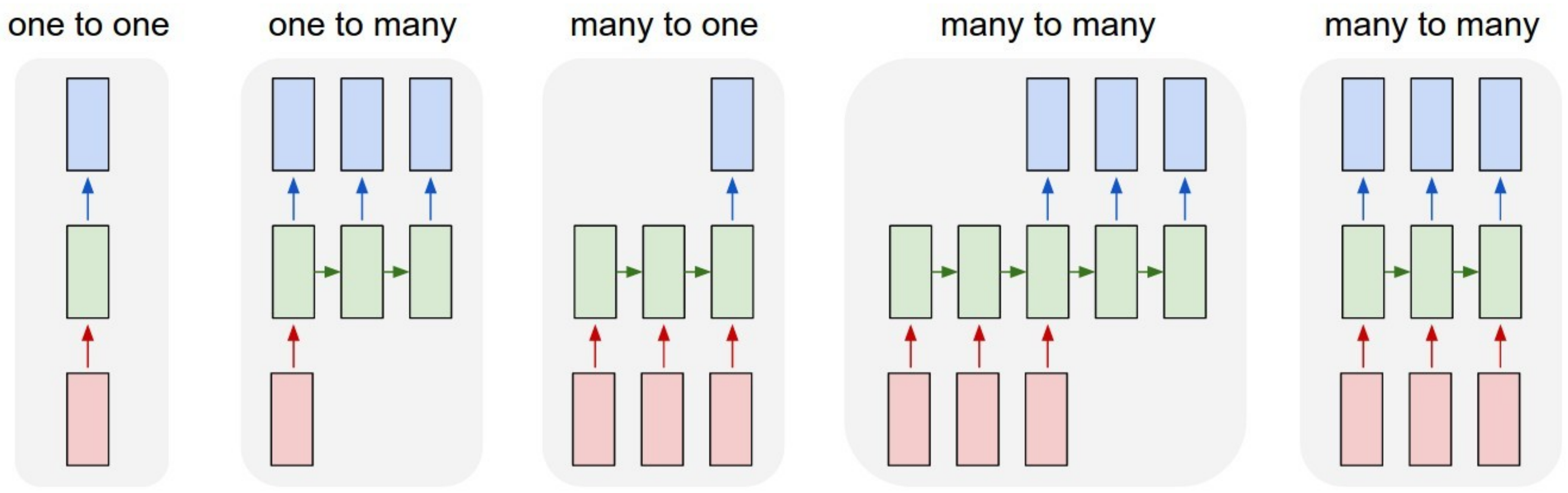


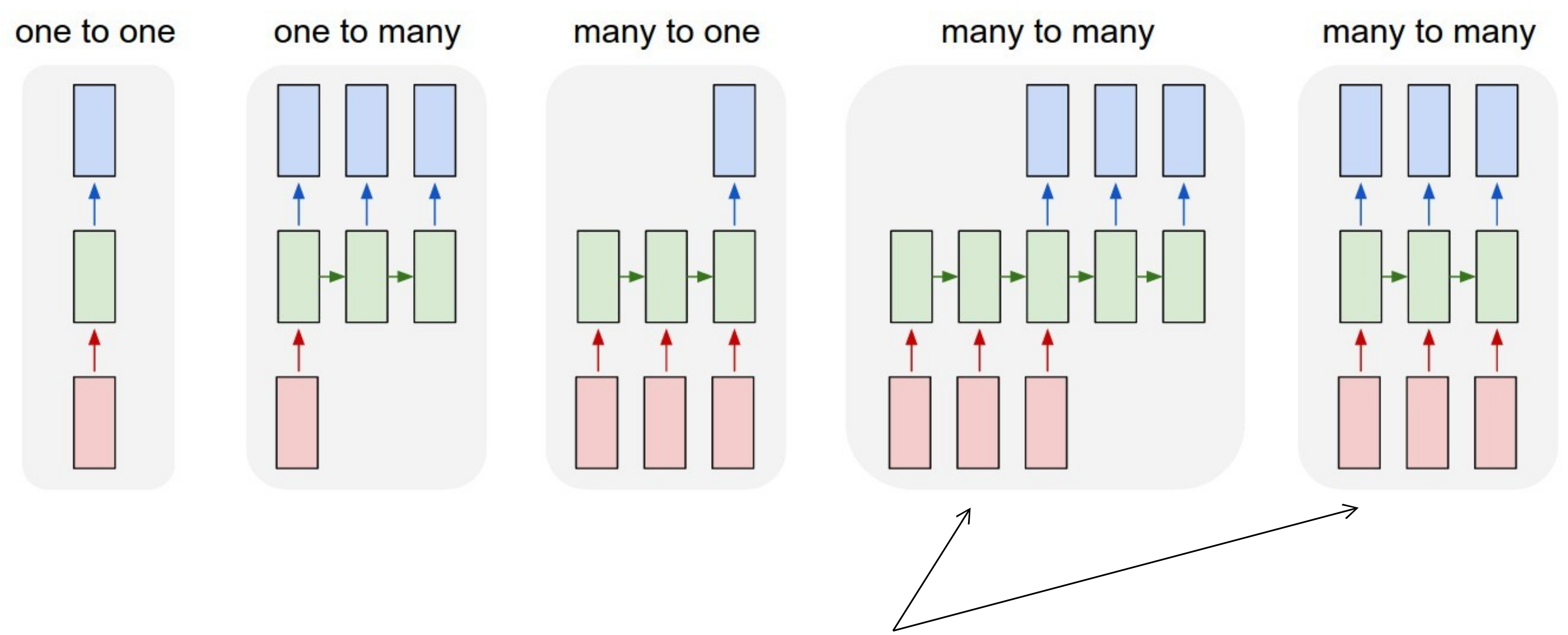
Image captioning,
Text generation

Process sequences



Action recognition
Sentiment classification

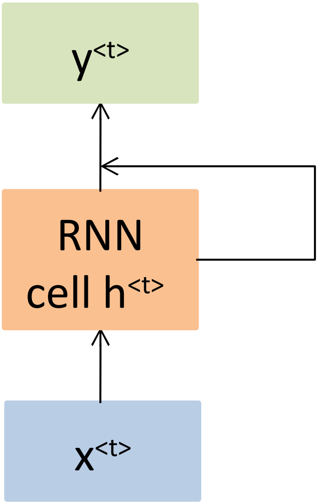
Process sequences



Text translation
Video classification at frame level

RNN cell

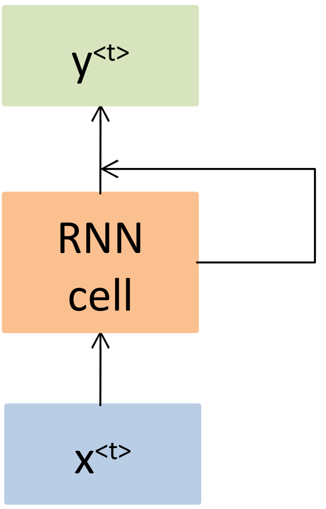
- **Recurrent core cell**
- Input: $x^{<t>}$
- Internal hidden state: $h^{<t>}$
  - updated each time an input $x^{<t>}$ is fed to the cell
- Output: $y^{<t>}$
  - at some time steps



RNN cell

$$h^{\text{¿} t > \text{¿} = f_{w} \text{¿} \text{¿}}$$

- updated hidden state
- input at time step t
-previous hidden state



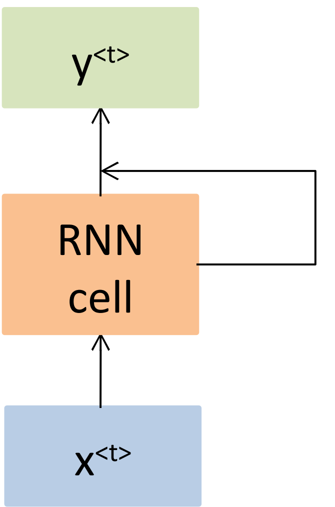
RNN cell

$$h^{¿t>¿}=f_{W}¿¿$$

- updated hidden state
- input at time step t
-previous hidden state

$$h^{¿t>¿}=\tanh ¿¿$$

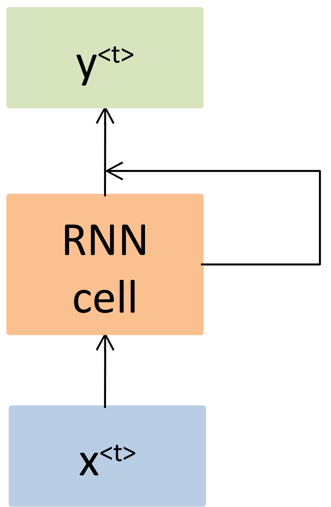
nonlinearity



RNN cell

$$h^{¿t>¿}=f_W ¿ ¿$$

- updated hidden state
- input at time step t
-previous hidden state

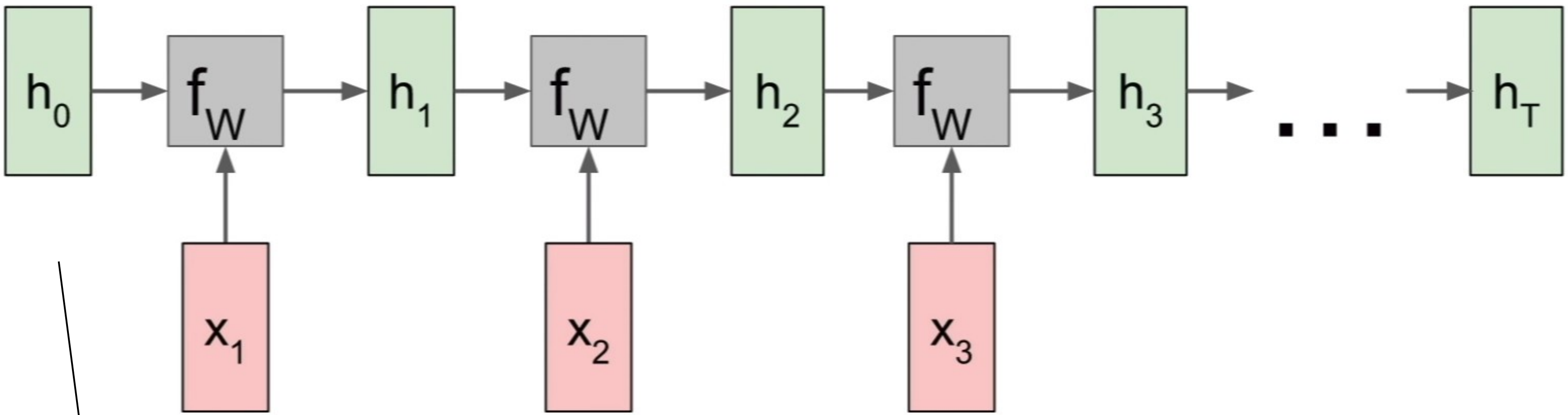


$$h^{¿t>¿}=\tanh ¿ ¿$$

: make a decision based on each time step

$$y^{¿t>¿}=W_y h^{¿t>¿¿}¿$$

RNN computational graph



Initialized to 0 in most contexts

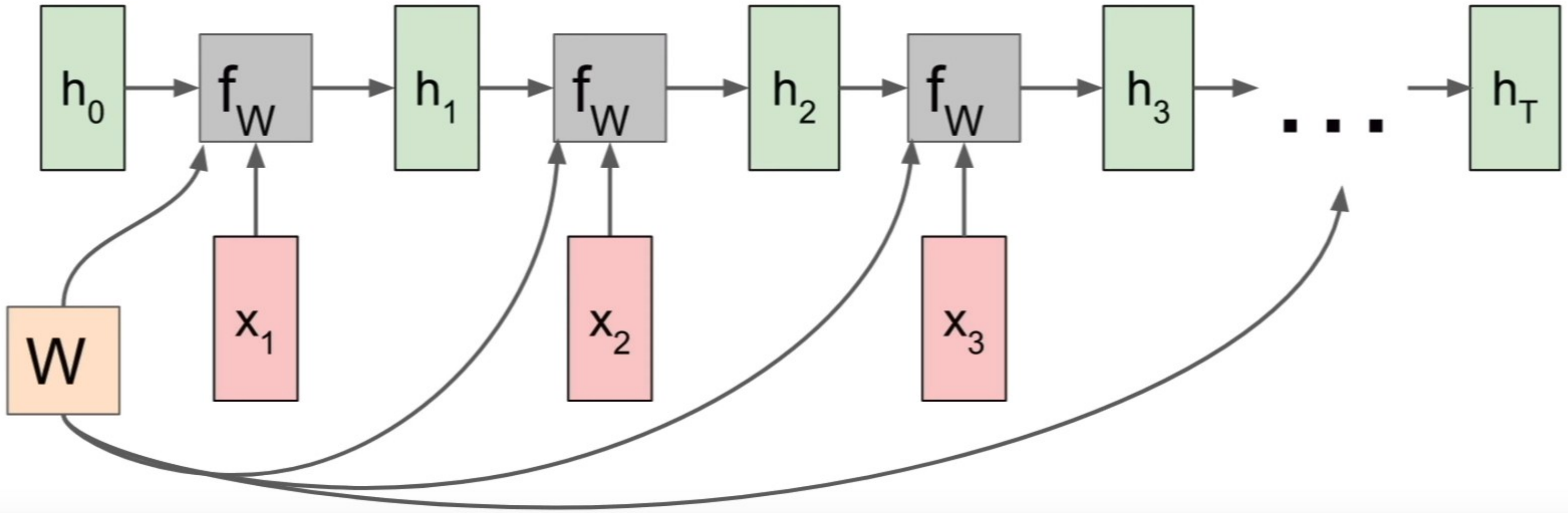
https://calvinfeng.gitbook.io/machine-learning-notebook/supervised-learning/recurrent-neural-network/recurrent_neural_networks

RNN computational graph

We use the same set of weight for every time step of the computation.

Backprop

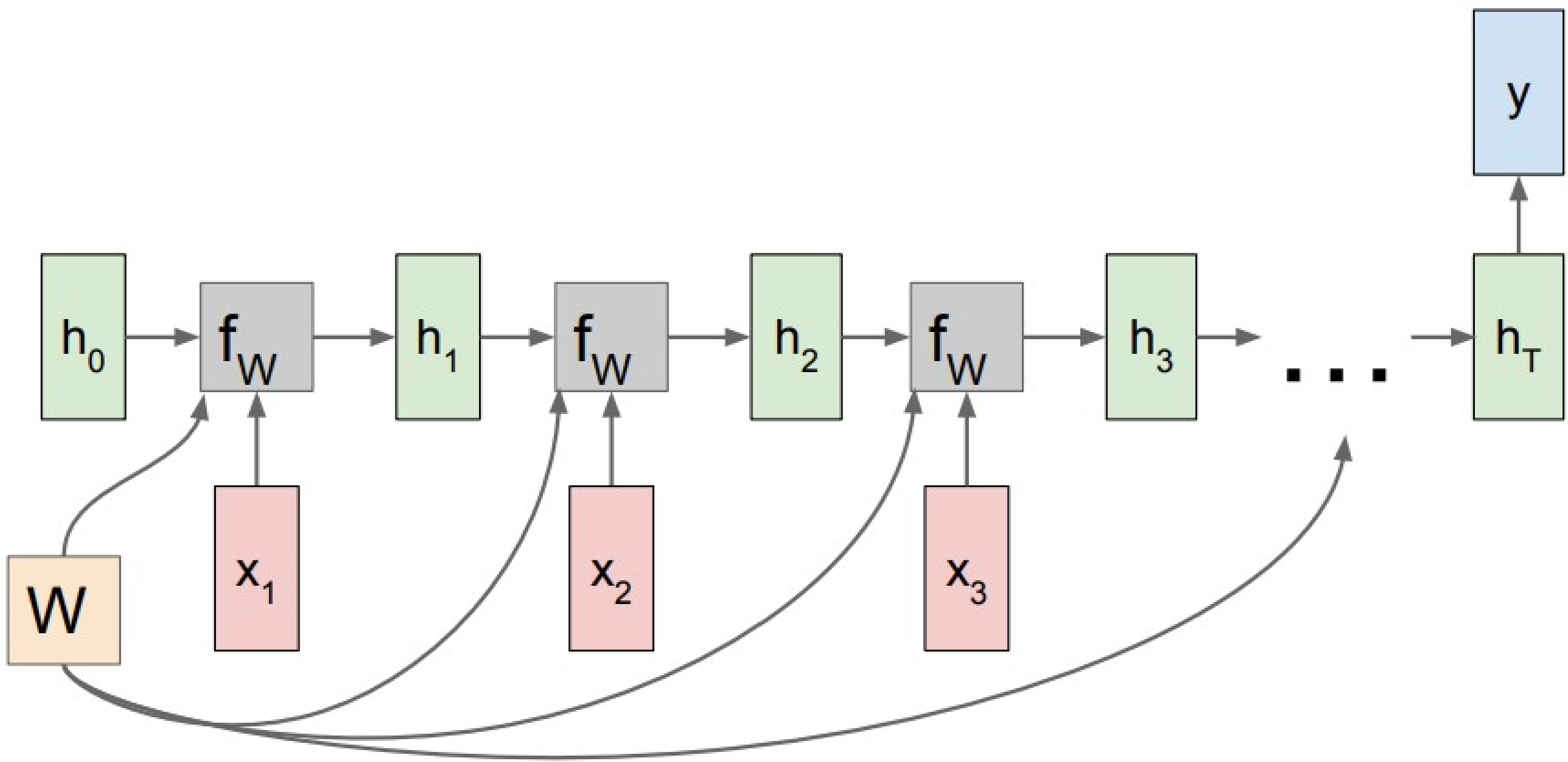
- Separate gradient for each W (from each of the timestamps)
- Total gradient: sum of the timestamp gradients



https://calvinfeng.gitbook.io/machine-learning-notebook/supervised-learning/recurrent-neural-network/recurrent_neural_networks

RNN computational graph

Many to one

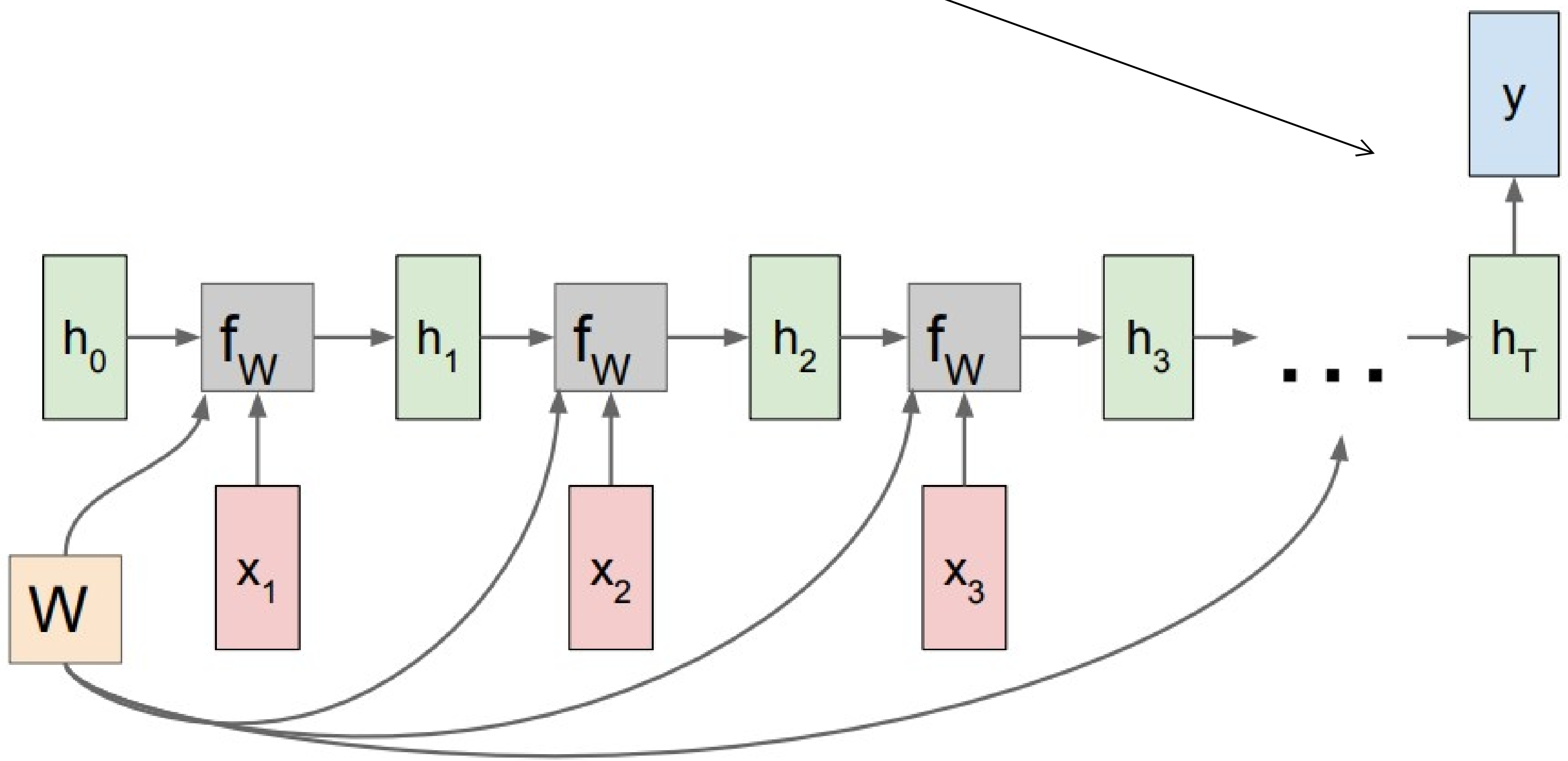


https://calvinfeng.gitbook.io/machine-learning-notebook/supervised-learning/recurrent-neural-network/recurrent_neural_networks

RNN computational graph

Many to one

Make decision on the last hidden state
(it summarizes the entire sequence)

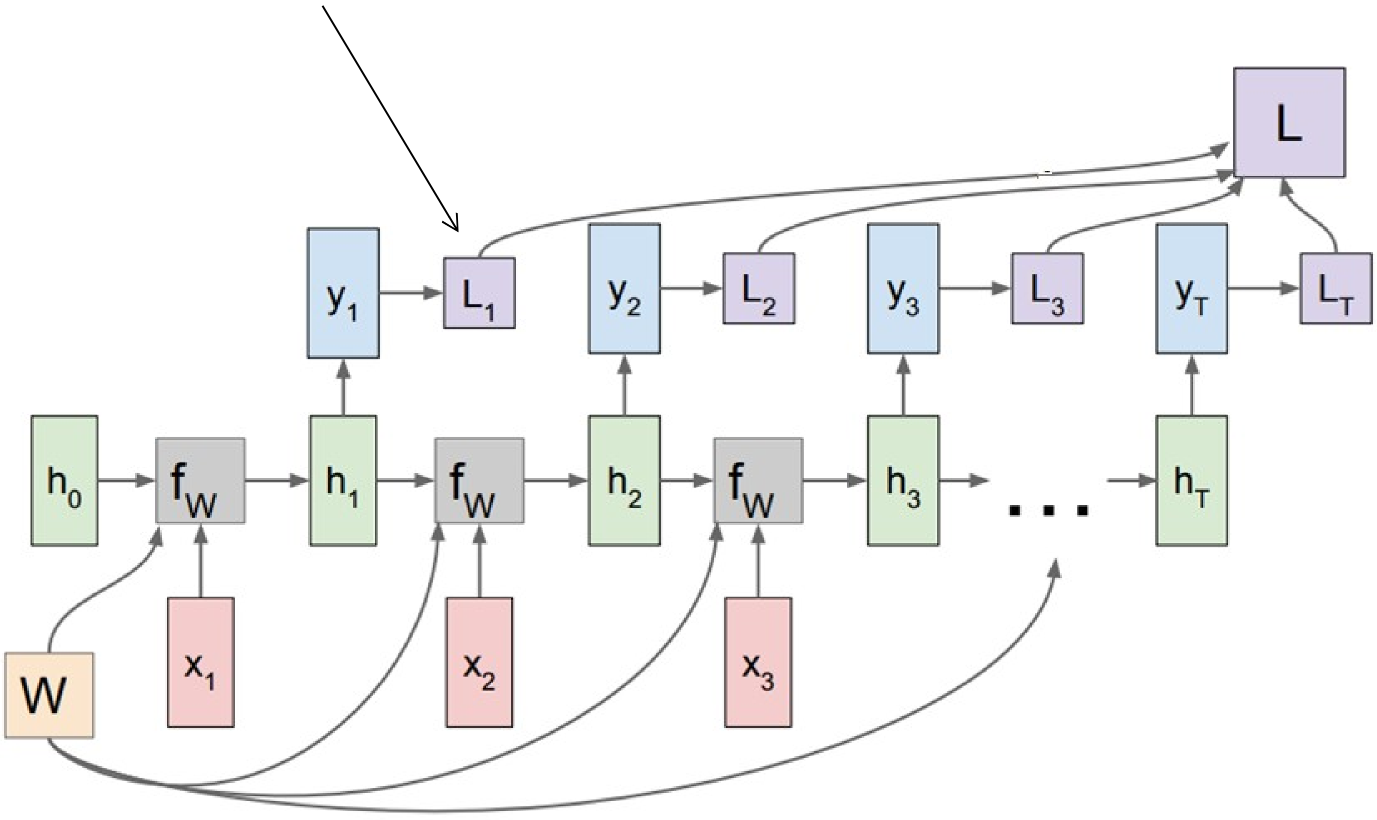


https://calvinfeng.gitbook.io/machine-learning-notebook/supervised-learning/recurrent-neural-network/recurrent_neural_networks

RNN computational graph

Many to many

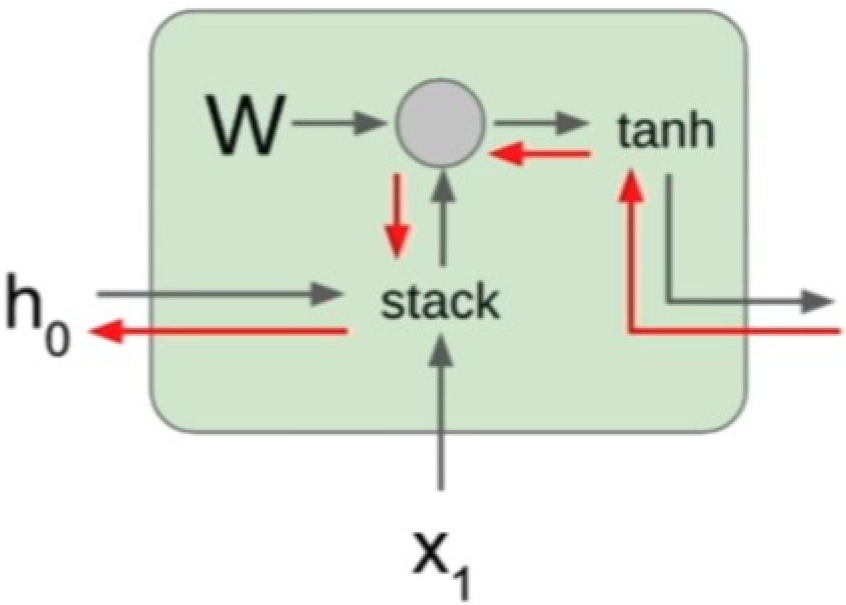
Compute loss at each timestamp



https://calvinfeng.gitbook.io/machine-learning-notebook/supervised-learning/recurrent-neural-network/recurrent_neural_networks

RNN backward pass

During backpropagation from the next state to the current state we pass through a matrix multiplication step

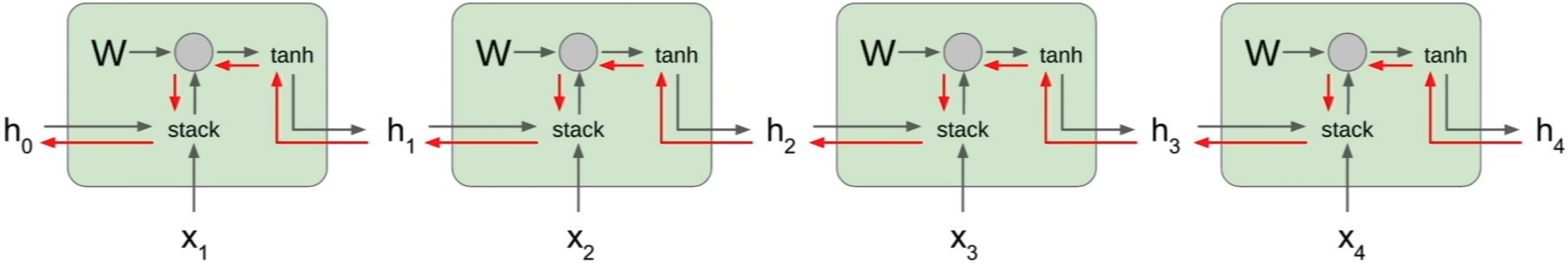


$$h_t = tanh\left(W_{hh}h_{t-1} + W_{xh}x_t\right)$$

$$h_t = tanh\left(\left(W_{hh} W_{xh}\right)\begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)$$

$$h_t = tanh\left(W\begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)$$

RNN backward pass



At each cell we multiply by many factors of the weight matrix W

> 1: exploding gradients (**gradient clipping**): scale the value of the gradient if it is too big
< 1: vanishing gradients

Long short term memory, 1997

- Alleviate the problems of vanishing and exploding gradient

RNN

$$h_t = \tanh\left(W\begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)$$

LSTM

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot tanh(c_t)$$

Two hidden states:
c – cell state
h – hidden state

Long short term memory

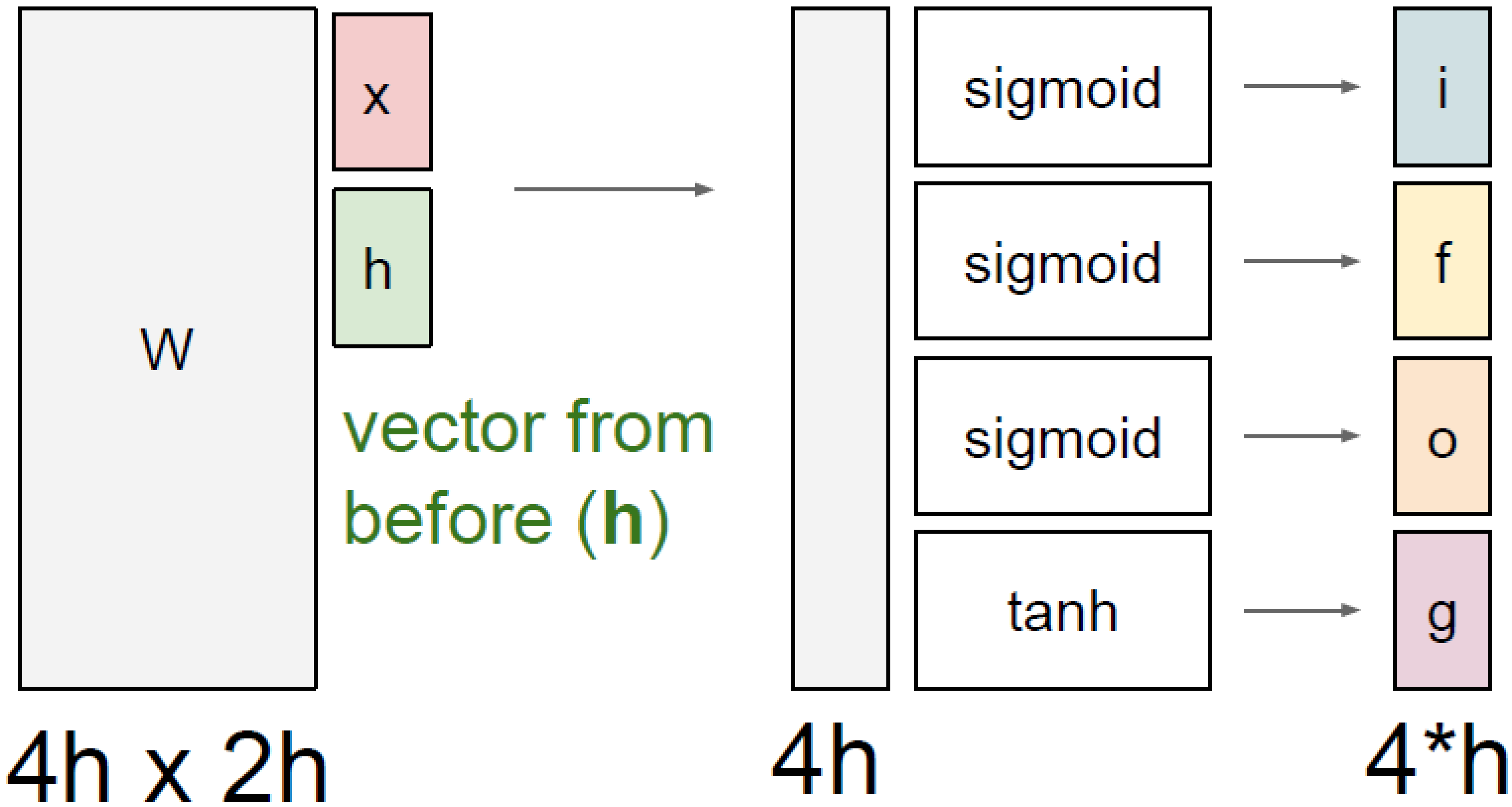
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot tanh(c_t)$$

Concatenate previous hidden state cell with the current input and multiply them with a bigger weight matrix to compute four gates

LSTM cell



W
x
h
vector from
before (h)
sigmoid
sigmoid
sigmoid
tanh
i
f
o
g
4h x 2h
4h
4*h

Long short term memory

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot tanh(c_t)$$

Concatenate previous hidden state cell with the current input and multiply them with a bigger weight matrix to compute four gates

forget gate: whether to erase cell (how much we want to forget from the previous cell state?)
input gate: whether to write to cell (how much to input into our cell?)
gate gate: how much to write to cell (how much to write into our cell?)
output gate: how much to reveal cell (how much to reveal from ourselves to the outside?

Long short term memory

Sigmoid: [0, 1]
tanh: [-1, 1]

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

Element wise multiplication: forget that element of the cell (0), or remember it (1)

$$h_t = o \odot tanh(c_t)$$

forget gate: whether to erase cell (how much we want to forget from the previous cell state?)
input gate: whether to write to cell (how much to input into our cell?)
gate gate: how much to write to cell (how much to write into our cell?)
output gate: how much to reveal cell (how much to reveal from ourselves to the outside?

Long short term memory

Sigmoid: [0, 1]
tanh: [-1, 1]

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

Element wise multiplication: for each element of the cell state, do we want to write to (1) it or not (0)?

$$h_t = o \odot tanh(c_t)$$

forget gate: whether to erase cell (how much we want to forget from the previous cell state?)
input gate: whether to write to cell (how much to input into our cell?)
gate gate: how much to write to cell (how much to write into our cell?)
output gate: how much to reveal cell (how much to reveal from ourselves to the outside?

Long short term memory

Sigmoid: [0, 1]
tanh: [-1, 1]

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot tanh(c_t)$$

Element wise multiplication: the candidate value that we might consider writing to the current cell state [-1, 1]
Independent scaler values, incremented of decrement by 1

forget gate: whether to erase cell (how much we want to forget from the previous cell state?)
input gate: whether to write to cell (how much to input into our cell?)
gate gate: how much to write to cell (how much to write into our cell?)
output gate: how much to reveal cell (how much to reveal from ourselves to the outside?

Long short term memory

Sigmoid: [0, 1]
tanh: [-1, 1]

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot tanh(c_t)$$

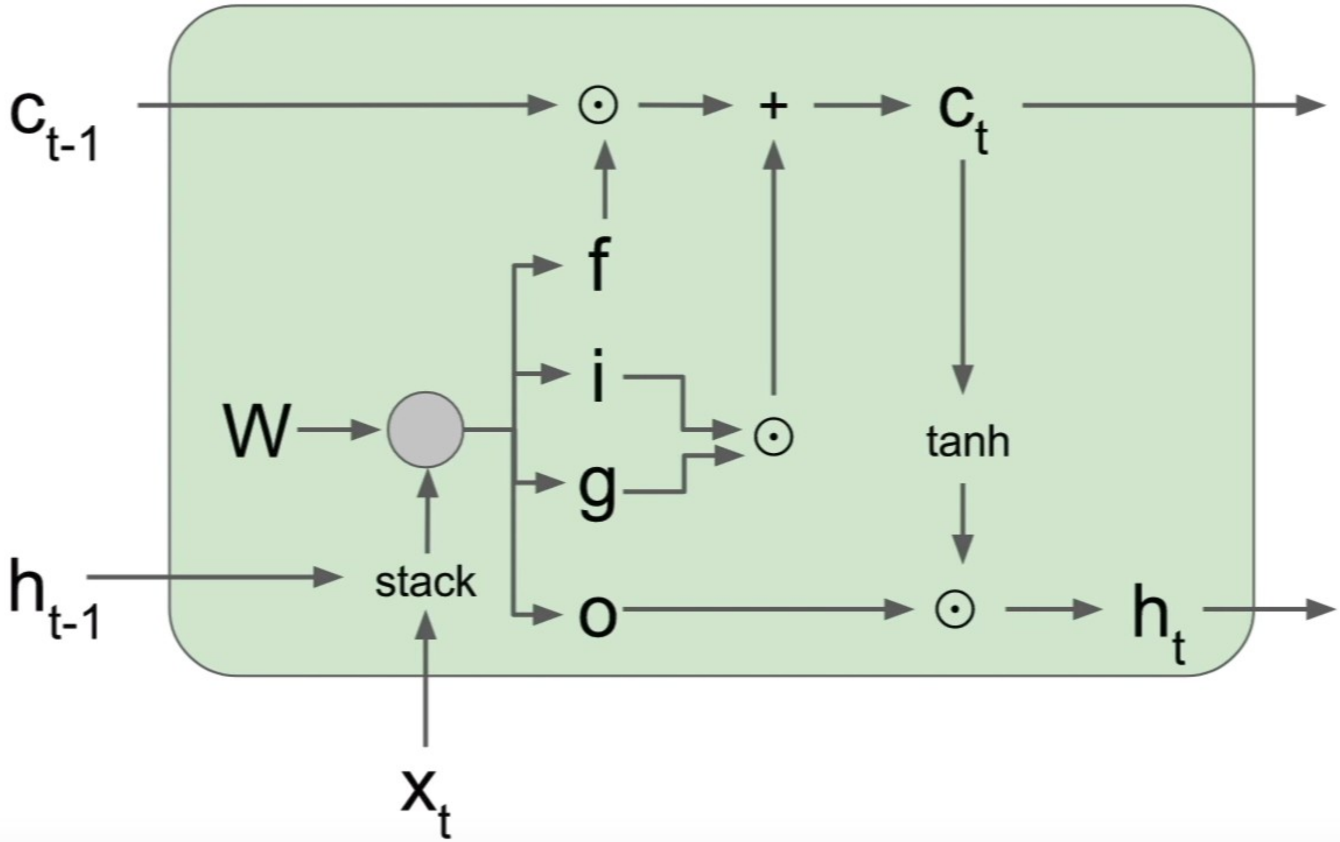
Use the cell state to compute a hidden state which we will reveal to the outside world

Sigmoid: for each element of out cell state, do we want to reveal it or not?

forget gate: whether to erase cell (how much we want to forget from the previous cell state?)
input gate: whether to write to cell (how much to input into our cell?)
gate gate: how much to write to cell (how much to write into our cell?)
output gate: how much to reveal cell (how much to reveal from ourselves to the outside?

LSTM cell

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$



$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot tanh(c_t)$$