COMP0188
Deep Representation and Learning

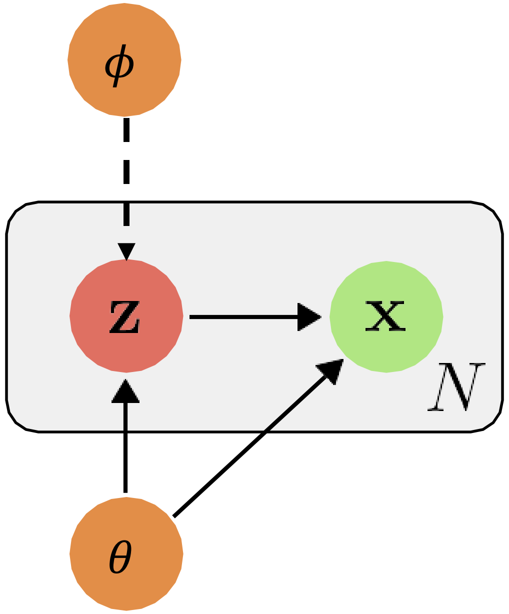
Joshua Spear
joshua.spear.21@ucl.ac.uk

Today

- Reviewing VAEs
- Brief introduction to tokenisation in NLP
- Attention
- Transformers
- Diffusion models

VAE update

VI with parameter learning



- ELBO:

$$\text{ELBO}(q_\phi(\text{z}|\text{x})) = -KL\left(q_\phi(z|x)||p_\theta(z)\right) + \mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x|z)\right]$$

- Optimise jointly with respect to the model parameters θ and the variational parameters ϕ
- Intuitively:
  - $KL(q_\phi(z|x)||p_\theta(z))$ is the divergence between the variational **posterior** and the **prior** over the latent variables and;
  - $\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)]$ is the expected log likelihood of the **reconstruction**
- Therefore, VAEs are **regularized autoencoders** where the form of the regulariser is **defined by the prior**!
- $p_\theta(z)$ is fixed and defined as $p_\theta(z) \sim N(0,1)$ (for this discussion)

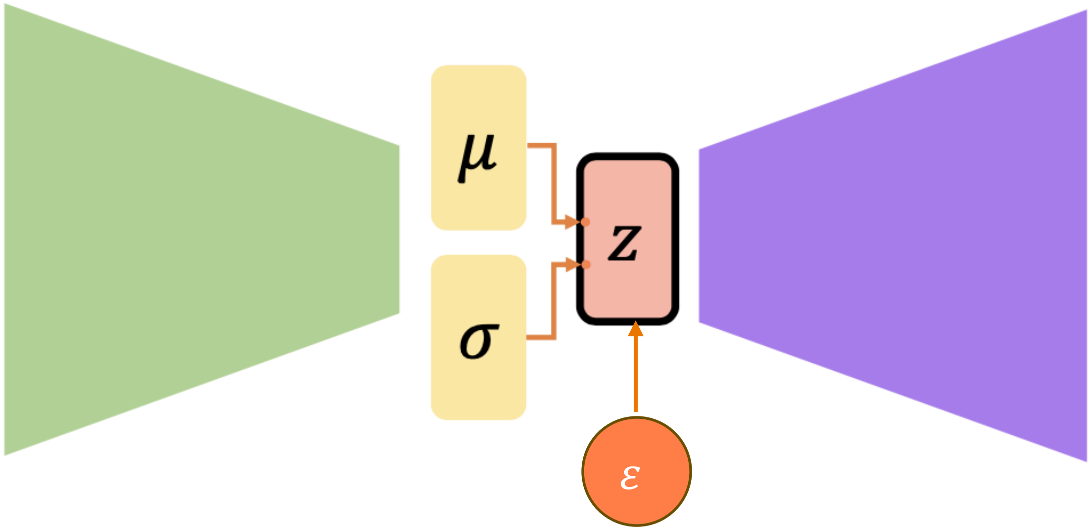
Reparameterisation trick

- Recall

$$p_\theta(z|x) = N(\mu, \sigma^2)$$

- Instead assume

$$p_\theta(z|x) = \mu + \sigma^2 \odot N(0,1)$$



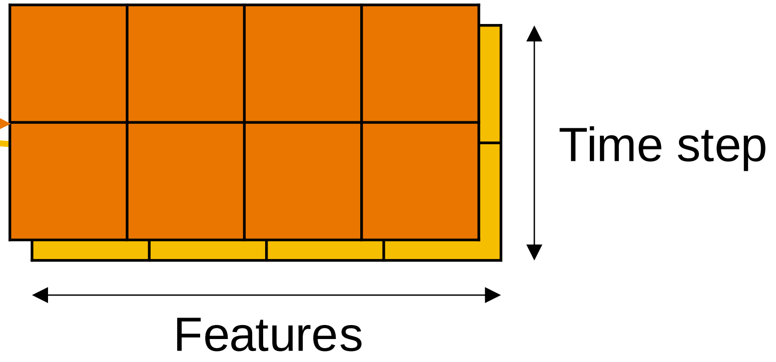
Sequence modelling with RNNs: A review

Timeseries tensor shapes

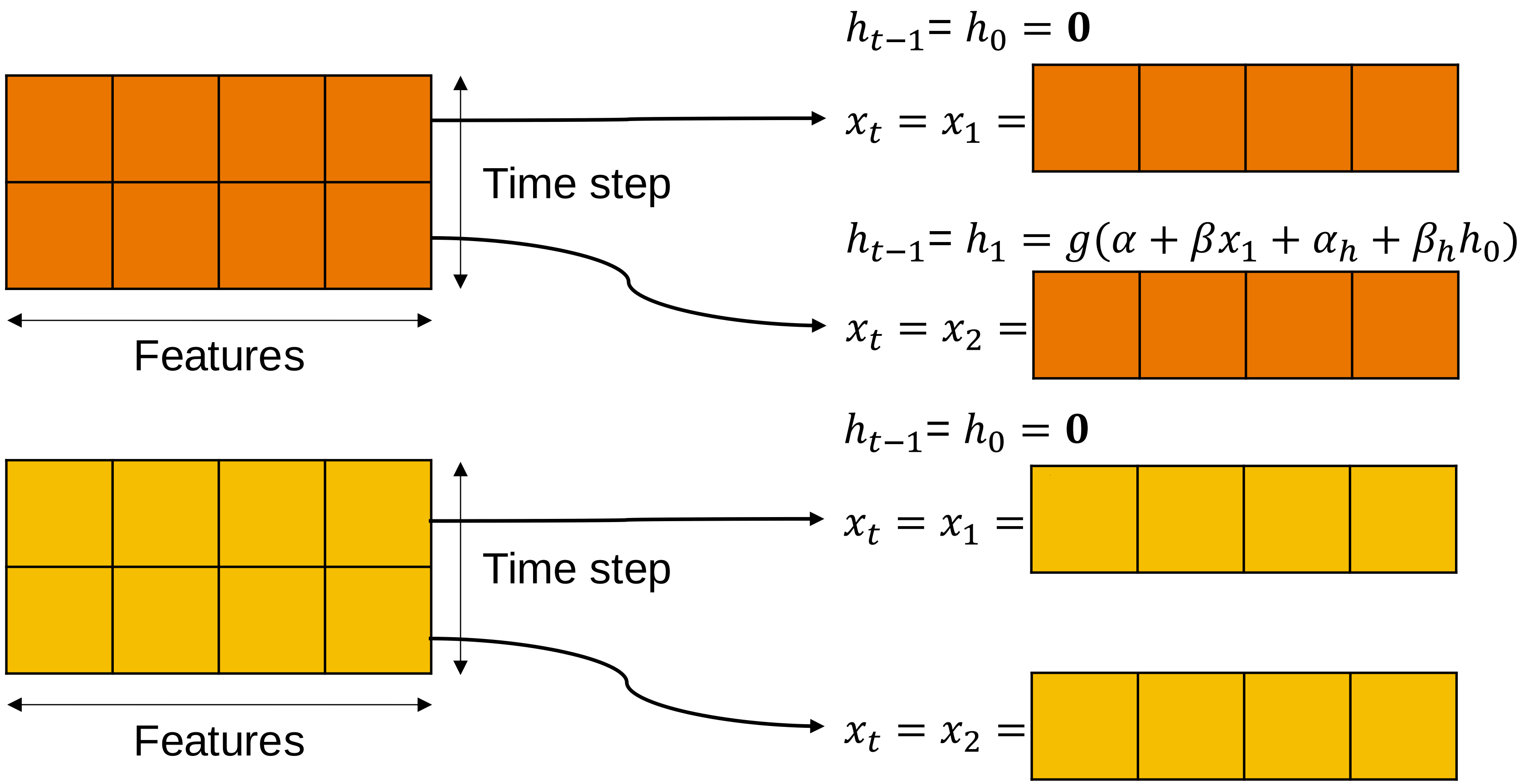
- Similarly to (black and white) images, timeseries add another dimension to the input

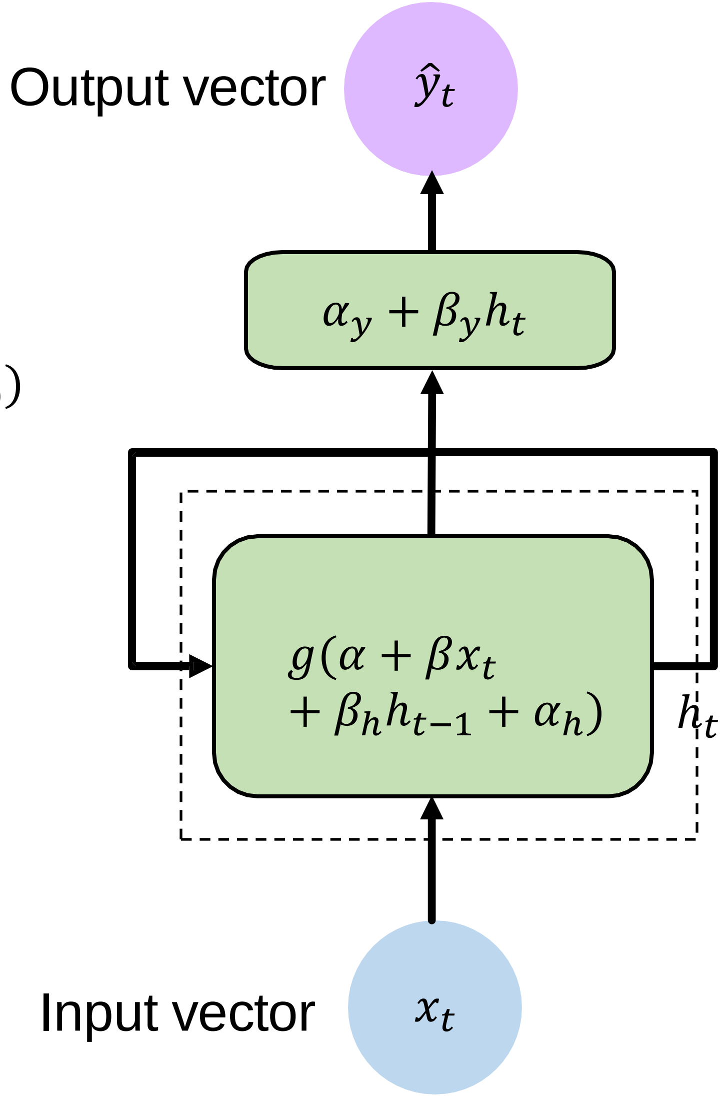
- For time series the input dimension will be:
  - **(batch size, time step, feature)**

	date	meantemp	humidity	wind_speed	meanpressure
0	2013-01-01	10.000000	84.500000	0.000000	1015.666667
1	2013-01-02	7.400000	92.000000	2.980000	1017.800000
2	2013-01-03	7.166667	87.000000	4.633333	1018.666667
3	2013-01-04	8.666667	71.333333	1.233333	1017.166667
4	2013-01-05	6.000000	86.833333	3.700000	1016.500000
5	2013-01-06	7.000000	82.800000	1.480000	1018.000000
6	2013-01-07	7.000000	78.600000	6.300000	1020.000000
7	2013-01-08	8.857143	63.714286	7.142857	1018.714286
8	2013-01-09	14.000000	51.250000	12.500000	1017.000000
9	2013-01-10	11.000000	62.000000	7.400000	1015.666667



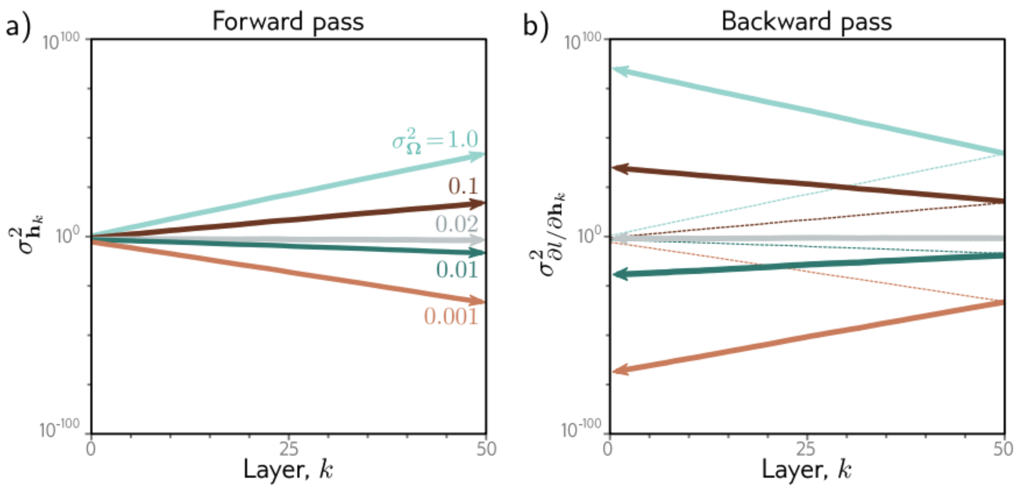
RNN


$h_{t-1} = h_0 = \mathbf{0}$
$x_t = x_1 =$
Time step
Features
$h_{t-1} = h_1 = g(\alpha + \beta x_1 + \alpha_h + \beta_h h_0)$
$x_t = x_2 =$
$h_{t-1} = h_0 = \mathbf{0}$
$x_t = x_1 =$
Time step
Features
$x_t = x_2 =$


Output vector
$\hat{y}_t$
$\alpha_y + \beta_y h_t$
$g(\alpha + \beta x_t$
$+ \beta_h h_{t-1} + \alpha_h)$
h_t
Input vector
x_t

Vanishing/exploding gradients

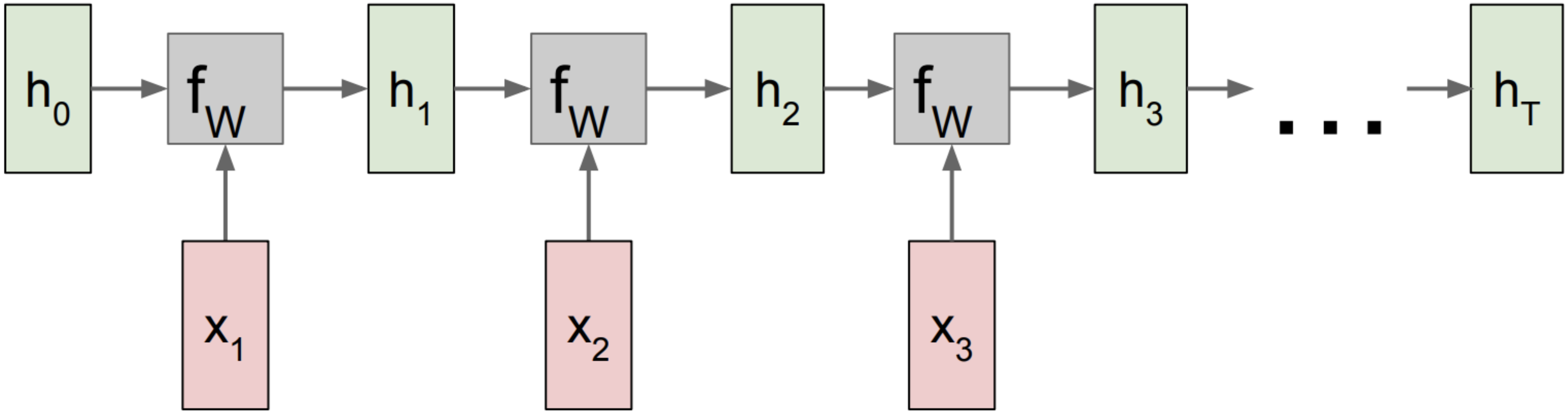
- Vanishing and exploding gradients occur when weight updates become **too small** or **too large**
- RNNs can be viewed as (broadly speaking), a very deep MLP where:
  - The time horizon denotes the “depth”
  - The same weight matrix, W, is used at each layer
- Since the **same** weight matrix is used, initialisation strategies such as He and Xiavier that rely on independent weights don’t apply!



We saw an example of this in **week 2** where, if the **weights were inappropriately initialised**, the **variance of the hidden state** could **explode/vanish**

Vanishing/exploding gradients

- Vanishing gradients are **desirable to some extent** as it is reasonable to assume that for timestep t, information at timestep $k\colon k < t$ is *more useful* than information at timestep $k'\colon k'' < k$
  - Is there a more **selective** way to learn historical relationships… (hint: yes! **Transformers**)?



Attention is all you need…

- Does not require bootstrapping of predictions
  - Timesteps can be processed in parallel
- Handles long term dependencies more effectively

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

Language embeddings (A brief introduction to tokenisation)

Numerical representations

- Machine learning models require **numerical representations**
- Defining **good** numerical representations is the ultimate goal of representation learning for language

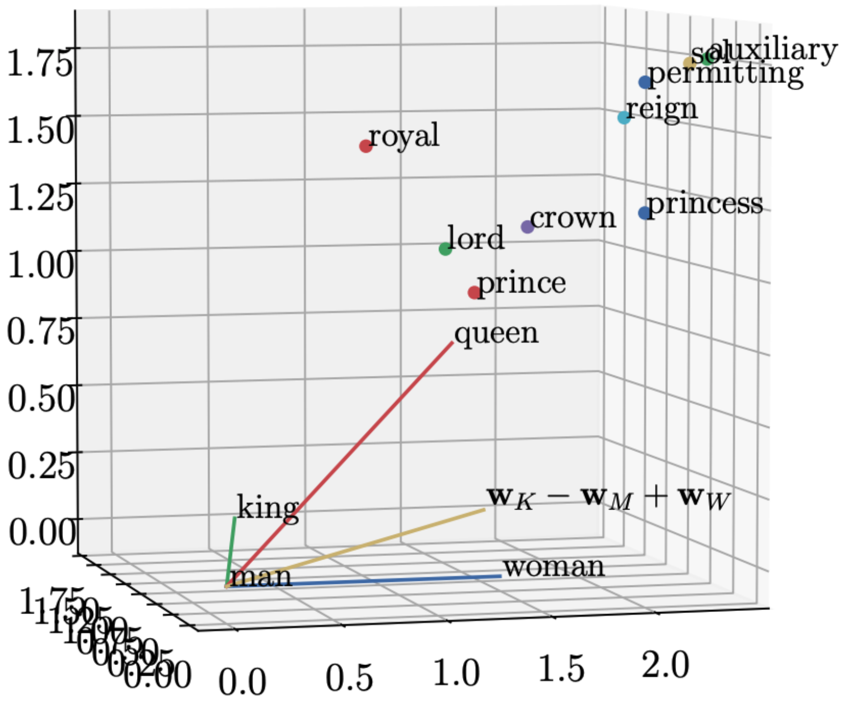


Figure 1. The relative locations of word embeddings for the analogy "*man* is to *king* as *woman* is to ..?". The closest embedding to the linear combination $\mathbf{w}_K - \mathbf{w}_M + \mathbf{w}_W$ is that of *queen*. We explain why this occurs and interpret the difference between them.

Tokenisation

- What defines a single input?
  - “The quick brown…”, [“The”, “quick”, “brown”, …], [“T”, “h”, “e”, “q”, “u”, “i”, “c”, “k”, …]
- Defining a **single input** is known as **tokenization**
  - Broadly speaking, optimal tokenisation defines **semantically meaningful** substrings of text
- BERT tokenizer (Word Piece)
  - “The quick brown fox” → [“The”, “quick”, “brown”, “fox”]
  - “Why the edits made under my username Hardcore Metallica Fan were reverted?” → [“Why”, “the”, “edit”, “##s”, “made”, “under”, “my”, “user”, “##name”, “Hard”, “##core”, “Metal”, “##lica”, “Fan”, “were”, “reverted”, “?”]
- https://www.youtube.com/watch?v=9vM4p9NN0Ts&t=2167s&ab_channel=StanfordOnline

Token embeddings

- Each embedding is mapped to a numeric ID:
  - “The quick brown fox” → {“The”: 1109, “quick”: 3613, “brown”: 3058, “fox”: 17594}
- Each numeric ID indexes a vector
  - torch.nn.Embedding is a trainable weight matrix

```python
class TokenEmbedding(nn.Module):

    def __init__(
        self,
        vocab_size:int,
        embed_size:int,
        ):
        super().__init__()
        self.embed_size = embed_size
        # (m, seq_len) --> (m, seq_len, embed_size)
        # padding_idx is not updated during training, remains as fixed pad (0)
        self.token = torch.nn.Embedding(vocab_size, embed_size, padding_idx=0)

    def forward(
        self,
        sequence:Float[torch.Tensor, "batch_size max_length"]
        ) -> Float[torch.Tensor, "batch_size max_length d_model"]:
        _token_embed = self.token(sequence)
        x = _token_embed
        return x

_tmp_te = TokenEmbedding(
    vocab_size=len(tokenizer.vocab),
    embed_size=32
)
for i in _tmp_te.token.named_parameters():
    print(i[0])
    print(i[1].shape)
```

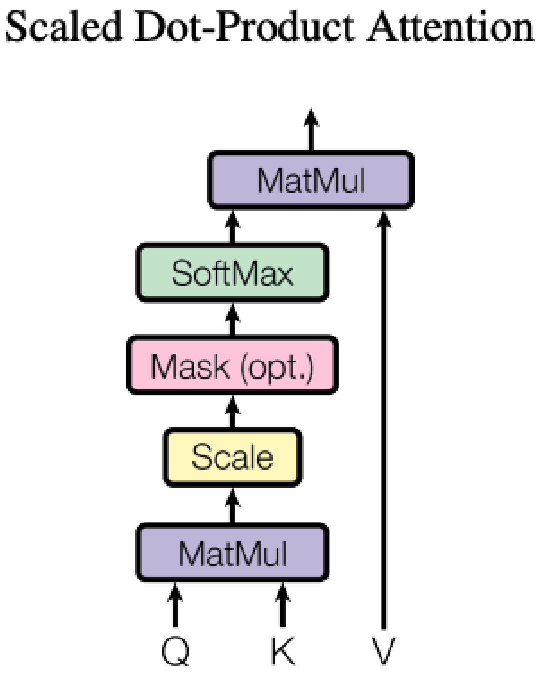
✓ 0.0s

Attention

Attention intuition

Scaled dot product attention

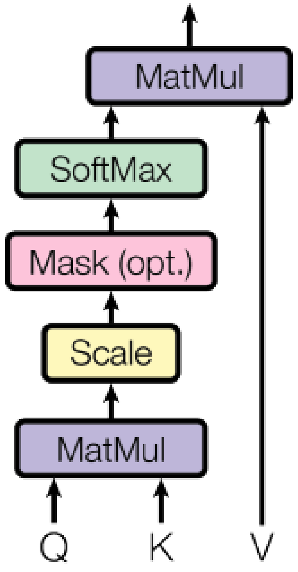
- Proposed in “Attention is all you need”
- Attention intuition:
  - Provide a set of input queries, Q
  - Map these to a set of (input) keys, K
    - Broadly speaking by computing how “close” the query is to a key, we are computing how “relevant” the key is to the query
  - Given some values V with one-to-one mapping to keys, up/down weight the magnitude of V by the previously calculated relevance



Scaled dot product attention

- Process:
  1. $\text{out}_1 = QK^T$
     - $Q \in \mathbb{R}^{d_Q}, K \in \mathbb{R}^{d_K} \Rightarrow \text{out}_1 \in \mathbb{R}$
  2. $\text{out}_2 = \frac{\text{out}_1}{\sqrt{d_K}} \in \mathbb{R}$
  3. $\text{out}_3 = \text{softmax}(\text{out}_2) \in \mathbb{R}$
  4. $\text{out}_4 = \text{out}_3 V$
     - $V \in \mathbb{R}^{d_V} \Rightarrow \text{out}_4 \in \mathbb{R}^{d_V}$
- In words:
  - Given a single query vector Q, and **single** key value pair K, V
  - How much attention should be paid to the value vector of the associated key

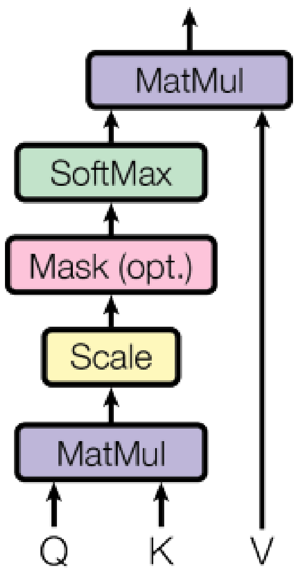
Scaled Dot-Product Attention



Scaled dot product attention

- Process:
  1. $\mathrm{out}_1 = QK^T$ ← Dot product can be considered a **measure of similarity**
     - $Q \in \mathbb{R}^{d_Q}, K \in \mathbb{R}^{d_K} \Rightarrow \mathrm{out}_1 \in \mathbb{R}$
  2. $\mathrm{out}_2 = \dfrac{\mathrm{out}_1}{\sqrt{d_K}} \in \mathbb{R}$
  3. $\mathrm{out}_3 = \mathrm{softmax}(\mathrm{out}_2) \in \mathbb{R}$ ← Given the similarity (defined in the softmax), how much should the **associated value be weighted**
  4. $\mathrm{out}_4 = \mathrm{out}_3 V$
     - $V \in \mathbb{R}^{d_V} \Rightarrow \mathrm{out}_4 \in \mathbb{R}^{d_V}$
- In words:
  - Given a single query vector Q, and **single** key value pair K, V
  - How much attention should be paid to the value vector of the associated key
- A single query and key-value is not so helpful…

Scaled Dot-Product Attention



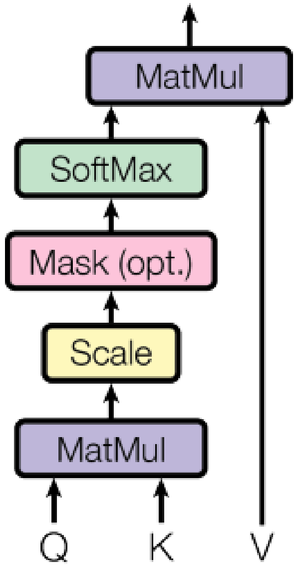
Scaled dot product attention

- Process for multiple queries and keys:
  1. $\text{out}_1 = QK^T$
     - $Q \in \mathbb{R}^{n_Q \times d_K}, K \in \mathbb{R}^{n_K \times d_K} \Rightarrow \text{out}_1 \in \mathbb{R}^{n_Q \times n_K}$
  2. $\text{out}_2 = \frac{\text{out}_1}{\sqrt{d_K}} \in \mathbb{R}^{n_Q \times n_K}$
  3. $\text{out}_3 = \text{softmax}(\text{out}_2) \in \mathbb{R}^{n_Q \times n_K}$
  4. $\text{out}_4 = \text{out}_3 V$
     - $V \in \mathbb{R}^{n_K \times d_V} \Rightarrow \text{out}_4 \in \mathbb{R}^{n_Q \times d_V}$

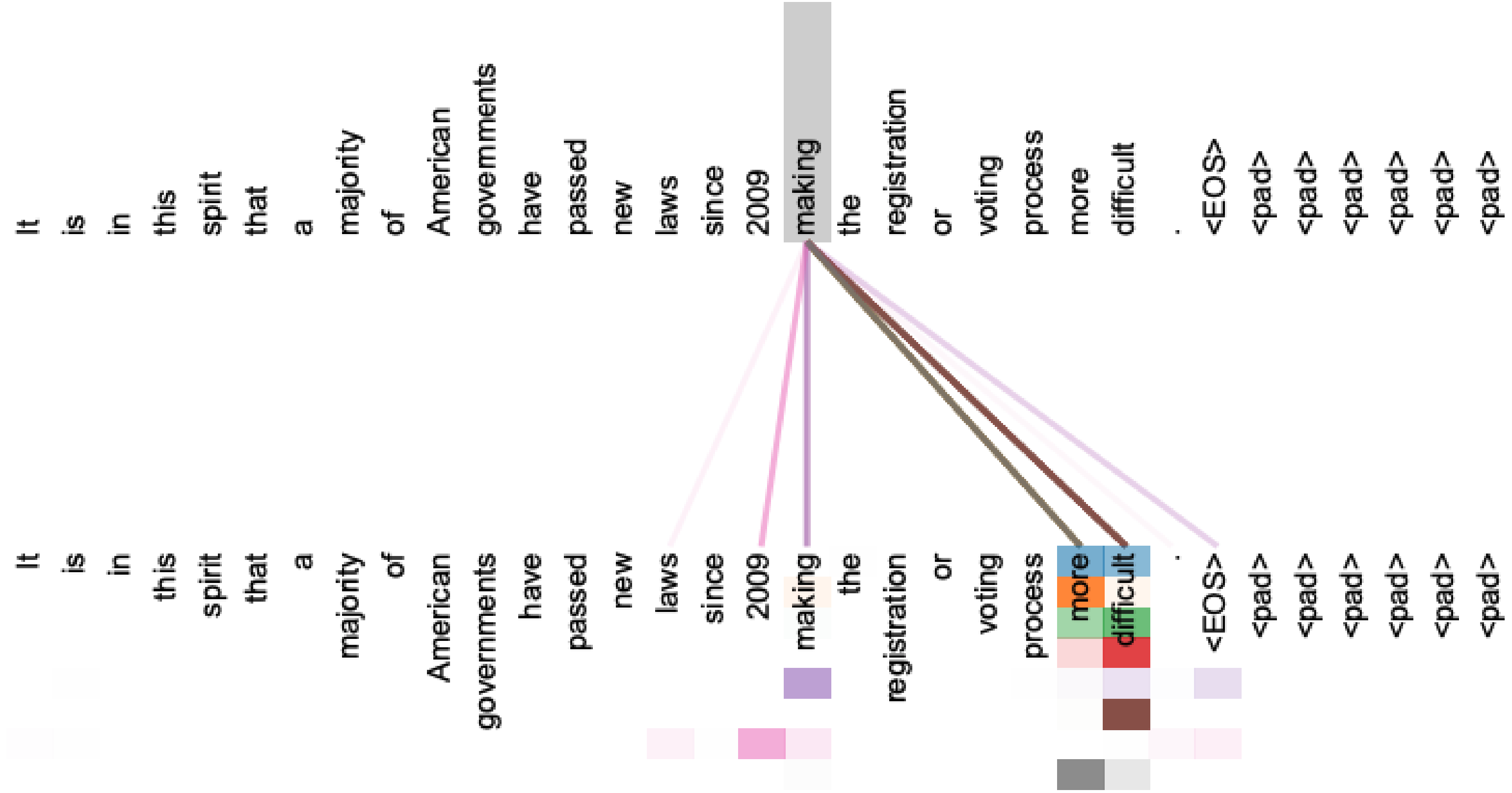
For each query (n_Q of them) there now exists a dot product for each key (n_K of them)

- Now, for each query (n_Q of them) there is a d_V associated "answer" vector
  - $V \in \mathbb{R}^{n_K \times d_V} \Rightarrow$ The same number of keys and values are required

Scaled Dot-Product Attention



Self attention



Self attention

- Scaled dot product attention: $\mathrm{out}_4 = \mathrm{softmax}(\frac{QK^T}{\sqrt{d_k}})V$
- In **self** attention, Q, K and V are **all** derived from the same input i.e., **self**
- Given an input, [“The”, “quick”, …]
  - Tokenize and convert to embeddings via lookup
  - $X = \begin{matrix} 0.5 & 0.2 \\ -1.2 & -1.4 \\ \ldots & \ldots \end{matrix}$ where $X \in \mathbb{R}^{d_s \times d_x}$ and x is a row vector from X
  - $Q = xW^Q$, $K = xW^K$, $V = xW^V$ such that $W^Q \in \mathbb{R}^{d_x \times d_K}$, $W^K \in \mathbb{R}^{d_x \times d_K}$, $W^V \in \mathbb{R}^{d_x \times d_V}$
- W^Q, W^K and W^V are **learnable** parameter matrices

Self attention

- Given an input, ["The", "quick", …]
  - Tokenize and convert to embeddings via lookup
  - $X = \begin{matrix} 0.5 & 0.2 \\ -1.2 & -1.4 \\ \dots & \dots \end{matrix}$ where $X \in \mathbb{R}^{d_S \times d_X}$ and x is a row vector from X
    - d_S defines the "sequence length" i.e., the number of tokens in the input
    - d_X defines the vector dimension for each token
  - Generally, set $d_Q = d_K = d_V$
  - $q = xW^Q$, $q = xW^K$, $v = xW^V$ such that $W^Q \in \mathbb{R}^{d_x \times d_Q}$, $W^K \in \mathbb{R}^{d_x \times d_K}$, $W^V \in \mathbb{R}^{d_x \times d_V}$
    - **IMPORTANT**: Linear projections are applied **individually** to each element of X
    - $q \in \mathbb{R}^{1 \times d_Q} \Rightarrow Q \in \mathbb{R}^{d_S \times d_Q}$ etc.

- Recall:

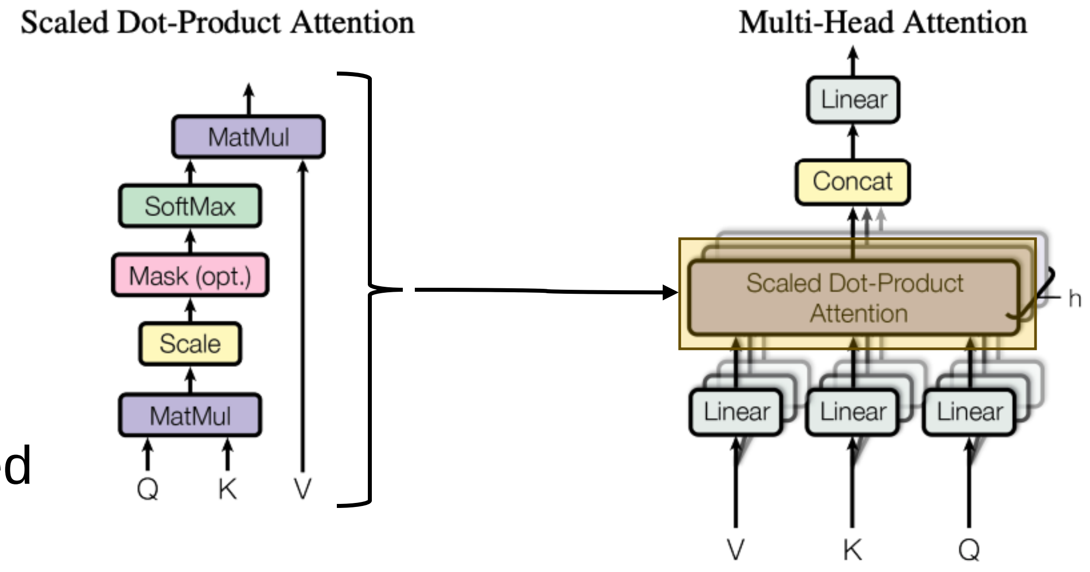
$$\mathrm{out}_4 = \mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

  - $QK^T \in \mathbb{R}^{d_S \times d_S} \Rightarrow$
    - QK^T defines the **relevance** between **every combination** of input token:

$$\mathrm{The} \times \mathrm{quick}$$

Multi-head attention

- h attention heads are computed in parallel
- Process
  - The dimensions of Q, K and V are **split** across the h heads
  - Each head outputs a matrix of dimension $\mathbb{R}^{d_S \times d_V/h}$
  - This means for a given query-key pair, **different softmax values** (amounts of attention) can be applied to different dimensions of V
- $\text{out} = \text{concat}(\text{head}_1, \text{head}_2, \dots)W^O$ where:
  - $\text{head}_i = \text{softmax}\left(\frac{Q_i K_i}{\sqrt{d_k}}\right) V_i, W^O \in \mathbb{R}^{hd_v \times d_o} \Rightarrow \text{out} \in \mathbb{R}^{d_S \times d_o}$
  - h is the number of scaled dot-product attention cells
  - $d_o = d_x$



Batched self attention

- Since this is SGD, we perform computations over batches
- Within batch computations are **independent** ⇒
  - The previous (multi head) self attention process is applied to each element of the batch **independently**
- Batch size 2:
  - $\underbrace{\text{[["The", "quick", …]}}_{\text{Batch element 1}}, \underbrace{\text{["I", "love", "ML"]]}}_{\text{Batch element 2}}$

Multi-head self attention (implementation)

```python
class MultiHeadSelfAttention(nn.Module):

    def __init__(
        self,
        d_model:int,
        n_heads:int = 1
        ) -> None:
        super().__init__()
        assert d_model % n_heads == 0
        self.__n_heads = n_heads
        self.__d_v = self.__d_k = int(d_model/n_heads)
        self.W_q = nn.Linear(d_model, d_model, bias=False)
        self.W_k = nn.Linear(d_model, d_model, bias=False)
        self.W_v = nn.Linear(d_model, d_model, bias=False)
        self.__norm = torch.sqrt(torch.tensor(d_model))
        self.soft = nn.Softmax()
        self.W_o = nn.Linear(d_model, d_model, bias=False)
```

```python
    def forward(
        self,
        x:Float[torch.Tensor, "batch_size max_length d_model"],
        ):
        # Linear regression is applied to only the last dimension i.e.,
        # this is equivalent to independantly applying the regression to each
        # element of each batch independendently
        # Output dimension:
        # Float[torch.Tensor, "batch_size max_length d_model"]
        batch_size, seq_len, embed_dim = x.shape
        q = self.W_q(x)
        k = self.W_k(x)
        v = self.W_v(x)
        # Split up by heads ->
        #(batch_size, max_length, heads, embedding_dim/heads)
        # permute ready for matmul ->
        #(batch_size, heads, max_length, embedding_dim/heads)
        q = q.view(
            batch_size,seq_len,self.__n_heads, self.__d_k
            ).permute(0,2,1,3)
        k = k.view(
            batch_size,seq_len,self.__n_heads, self.__d_k
            ).permute(0,2,1,3)
        v = v.view(
            batch_size,seq_len,self.__n_heads, self.__d_v
            ).permute(0,2,1,3)
        _t_1 = torch.matmul(q,torch.transpose(k, dim0=2, dim1=3))
        _t_2 = self.soft(_t_1/self.__norm)
        _t_3 = torch.matmul(_t_2,v)
        _t_4 = _t_3.permute(0,2,1,3).reshape(
            batch_size, seq_len, embed_dim
            )
        return self.W_o(_t_4)
```

Multi-head attention and long-term dependencies

- RNNs suffer:
  - From **vanishing gradients** $\Rightarrow$ struggle to model **long term dependencies**
  - **Slow computation** due to the **autogressive** processing of input data
    - Sequential operations scale as $O(n)$ where n is the sequence length
- Multi-head attention addresses both:
  - **Long term dependencies**: Modelled via attention. Don't need to go through $x_{t-1}, \ldots, x_{t-k}$ to model $g(x_t, x_{t-k-1})$
  - **Slow computation**: The dependencies between $x_t, \ldots, x_1$ are modelled **simultaneously**
    - Sequential operations scale as $\boldsymbol{O(1)}$**!**

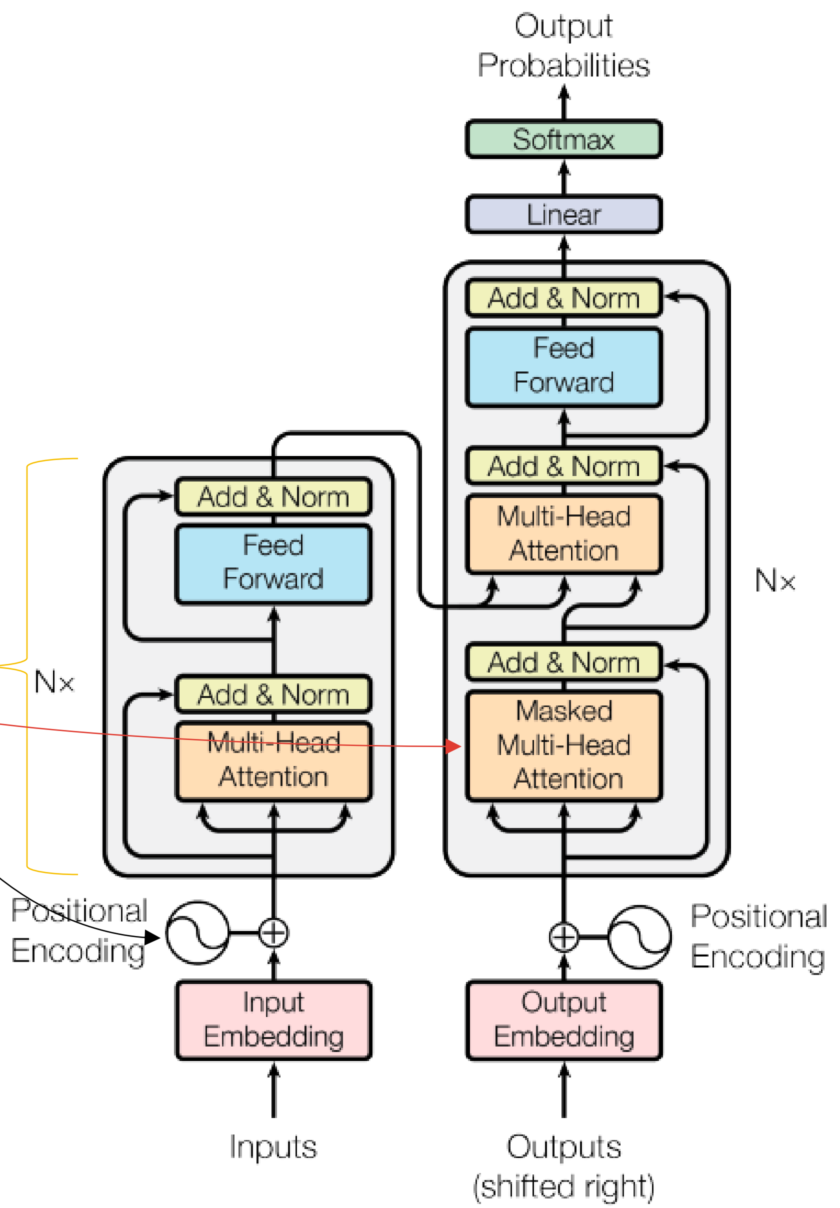
Transformer

Attention is all you need background

- Task: Translation
  - I.e., given a sentence in English, translate it to German
  - Training dataset contains **pairs** of (English, German)
  - At test time the model is provided an English and needs to **generate** a German translation
- Idea
  - Jointly embed the input and (masked) output string in the same latent space
  - Use only **multi-head attention** and **(small) feed forward networks**

Architecture

- Multi-head attention
- Positional encoding/Input embeddings
- Encoder cell
- Masking in the decoder cell

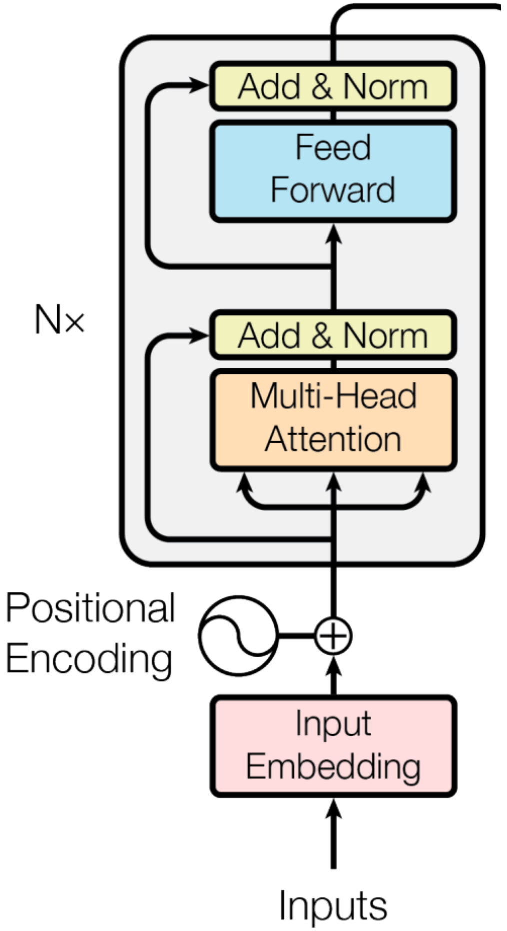


Transformer inductive bias

- **Invariance**:the output of function f is **unaffected** by a “transformation” of the input
  - $f(\rho x) = f(x)$
- **Equivariance**: The input and output are **affected in the same way** by a transformation on the input
  - $f(\rho x) = \rho f(x)$
- Transformers define **positional invariance**
  - [“The”, “quick”] or [“quick”, “The”] would define the **same** attention
  - This becomes less trivial with other contexts however, the principal remains

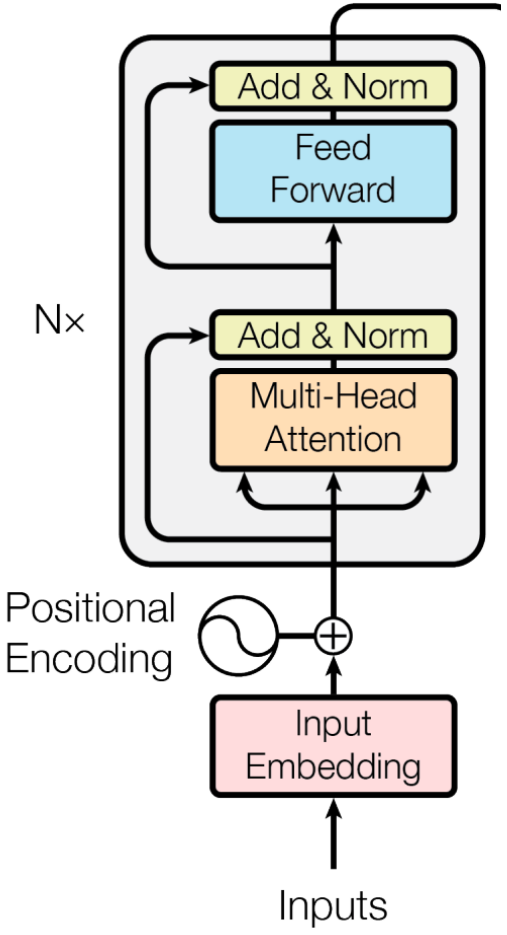
Transformer positional invariance

- Transformers define **positional invariance**
  - ["The", "quick"] or ["quick", "The"] would define the **same** attention
  - This becomes less trivial with other contexts however, the principal remains
- Positional invariance is **can be** useful however, for **sequences** we **don't want** positional invariance ⇒
  - Add **positional embeddings**
- Positional embeddings
  - $E_{(\text{pos},2i)} \sin\left({\text{pos}}/{10000^{2i/d_X}}\right)$
  - $E_{(\text{pos},2i+1)} \cos\left({\text{pos}}/{10000^{2i/d_X}}\right)$



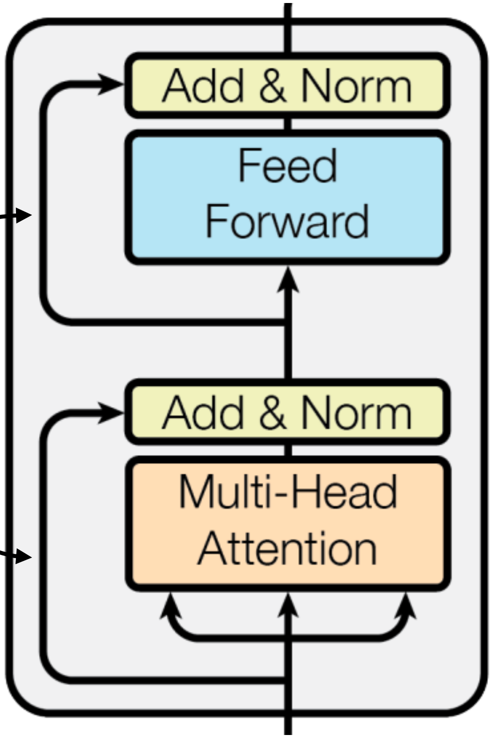
Transformer positional invariance

- Positional embeddings
  - $E_{(\mathrm{pos},2i)} \sin\left({\mathrm{pos}}/{10000^{2i/d_X}}\right)$
  - $E_{(\mathrm{pos},2i+1)} \cos\left({\mathrm{pos}}/{10000^{2i/d_X}}\right)$
- The input to the multi-head attention becomes
  - Tokenize and convert to embeddings via lookup
  - $X = \begin{matrix} 0.5 & 0.2 \\ -1.2 & -1.4 \\ \dots & \dots \end{matrix}$ where $X \in \mathbb{R}^{d_S \times d_X}$ and x is a row vector from X
  - Obtain positional embeddings $E \in \mathbb{R}^{1 \times d_X}$ (the positional embedding is constant across inputs)
  - $\tilde{X} = X + \tilde{E}$ where $\tilde{E} \in \mathbb{R}^{d_S \times d_X}$ such that each **row vector** is equal to E
- The Input embedding also contains a sentence identifier



Transformer encoder cell

- Transformer cell contains:
  - Multi-head self attention
  - Generally small (2 layer) feed forward network, taking $\mathrm{out} = \mathrm{concat}(\mathrm{head}_1, \mathrm{head}_2, \ldots)W^O$ as an input
  - Residual connections



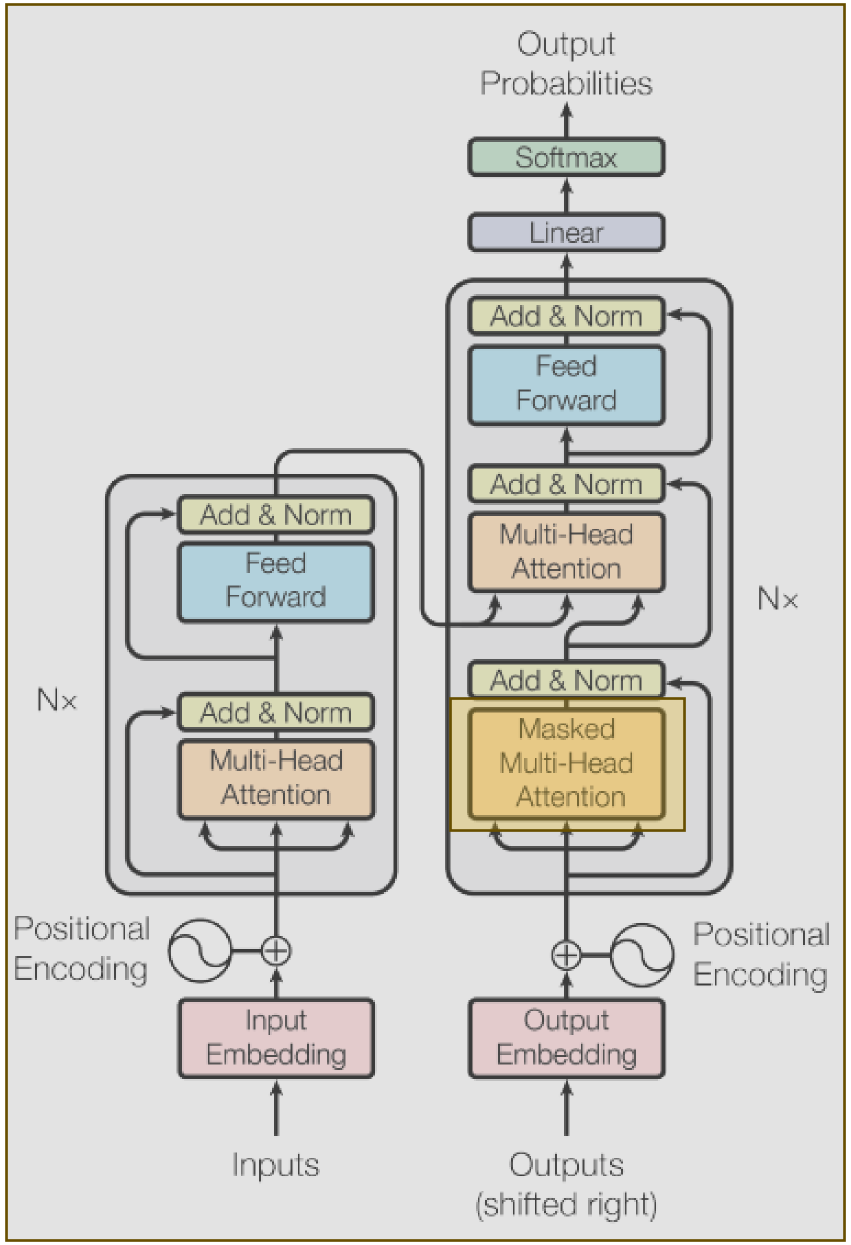

```python
class EncoderLayer(nn.Module):

    def __init__(
        self,
        d_model:int,
        n_heads:int,
        fc_dim:int,
        ) -> None:
        super().__init__()
        self.fc_1 = nn.Linear(
            d_model, fc_dim
        )
        self.relu = nn.ReLU()
        self.fc_2 = nn.Linear(
            fc_dim, d_model
            )
        self.mha = MultiHeadSelfAttention(
            d_model=d_model, n_heads=n_heads
        )
        self.layer_norm_1 = nn.LayerNorm(d_model, eps=1e-6)
        self.layer_norm_2 = nn.LayerNorm(d_model, eps=1e-6)

    def forward(self, x:torch.Tensor):
        mha_out = self.mha(x=x)
        resid_norm = self.layer_norm_1(mha_out+x)
        fc_out = self.fc_2(self.relu(self.fc_1(x)))
        return self.layer_norm_2(fc_out+resid_norm)
```

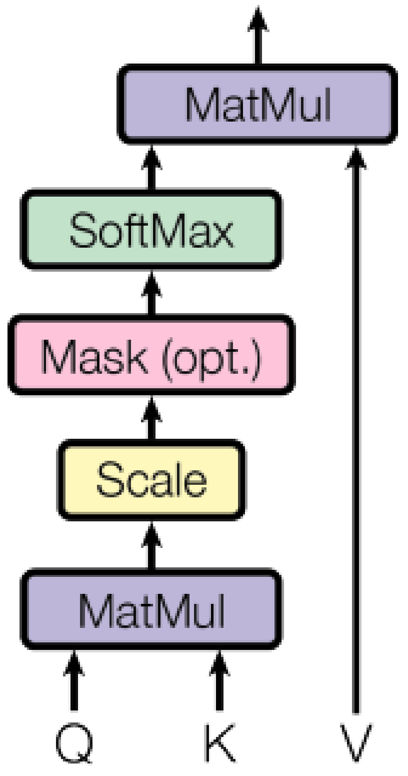
Masked attention

- Considering the test case:
  - Given an input in English: ”I love ML”
  - Translate to German
  - **Importantly**: The model would not have access to the translated output
  - However, **as the model generates text**, it can bootstrap previous predictions i.e.
    - First generate “Ich”
    - To generate “liebe”, the model can bootstrap its prediction of “Ich”



Masked attention

Scaled Dot-Product Attention



- Recall:
  - $QK^T \in \mathbb{R}^{d_S \times d_S} = \begin{matrix} 4.0 & -3.9 & 1.0 \\ 1.8 & -0.02 & -2.0 \\ -0.6 & 1.3 & 1.4 \end{matrix}$
  - Mask the top triangle of the QK^T attention matrix, setting these values to -1e9 (arbitrarily small value)
  - When passed to the softmax, this will result in the following:

$t+1$ value (-0.6) cannot influence t prediction

$$\begin{matrix} 0.05 & 0 & 0 \\ 0.9 & 0.5 & 0 \\ 0.004 & 0.85 & 0.87 \end{matrix} \times \begin{matrix} 1.4 & \ldots \\ -0.6 & \ldots \\ 0.2 & \ldots \end{matrix}$$

Dimension d^S (number of tokens)

Dimension d^S (number of tokens)

Dimension d^V (value vector dimension)

Note: Numbers used in this slide are random!

Training transformers

Training transformers

- When training transformers:
  - Require **a lot** of data
  - **Larger batch sizes** (than usual) are required
  - Similarly, **lower learning rates** (than usual) are required
- Fundamentally: SGD/Adam is not **as well behaved** as other NN’s (MLPs/RNNs/CNNs etc)
  - Require **tricks**

https://arxiv.org/abs/1804.00247

LayerNorm

- Previously defined batch norm:

$$\mathrm{BN}(x) = \frac{x - \mu(x)}{\sigma(x)}\beta + \gamma$$

  - Where μ and σ are defined **independently** for each **feature**
- Layer norm

$$\mathrm{LN}(x) = \frac{x - \mu(x)}{\sigma(x)}\beta + \gamma$$

  - Where μ and σ are defined **independently** for each **sample**
- Why layer norm?
  - Does not require as large batch sizes (as required by BN)
  - More appropriate for batches with sequences, particularly of different lengths

Attention is all you need tricks

- LayerNorm and Residual connections
- Adaptive learning rate: $d_x^{-0.5}\min(s^{-0.5}, sw^{-1.5})$ where s is algorithmic step and w is the number of warmup steps
  - Learning rate is proportional to the input embedding dimension $d_x^{-0.5}$. **Larger** the embedding, the **smaller** the learning rate
  - Learning rate increases linearly up to the warmup steps then decreases $\propto s^{-0.5}$
- Regularisation:
  - Dropout
  - Label smoothing (adding ε probability mass to incorrect labels in classification)

Attention is all you need tricks

- LayerNorm and Residual connections
- Adaptive learning rate: $d_x^{-0.5}\min(s^{-0.5}, sw^{-1.5})$ where s is algorithmic step and w is the number of warmup steps
  - Learning rate is proportional to the input embedding dimension $d_x^{-0.5}$. **Larger** the embedding, the **smaller** the learning rate
  - Learning rate increases linearly up to the warmup steps then decreases $\propto s^{-0.5}$
- Regularisation:
  - Dropout
  - Label smoothing (adding ε probability mass to incorrect labels in classification)

- ResNets enable **deeper** models
- Normalisation stabalises the residual connections
- https://arxiv.org/pdf/1901.09321

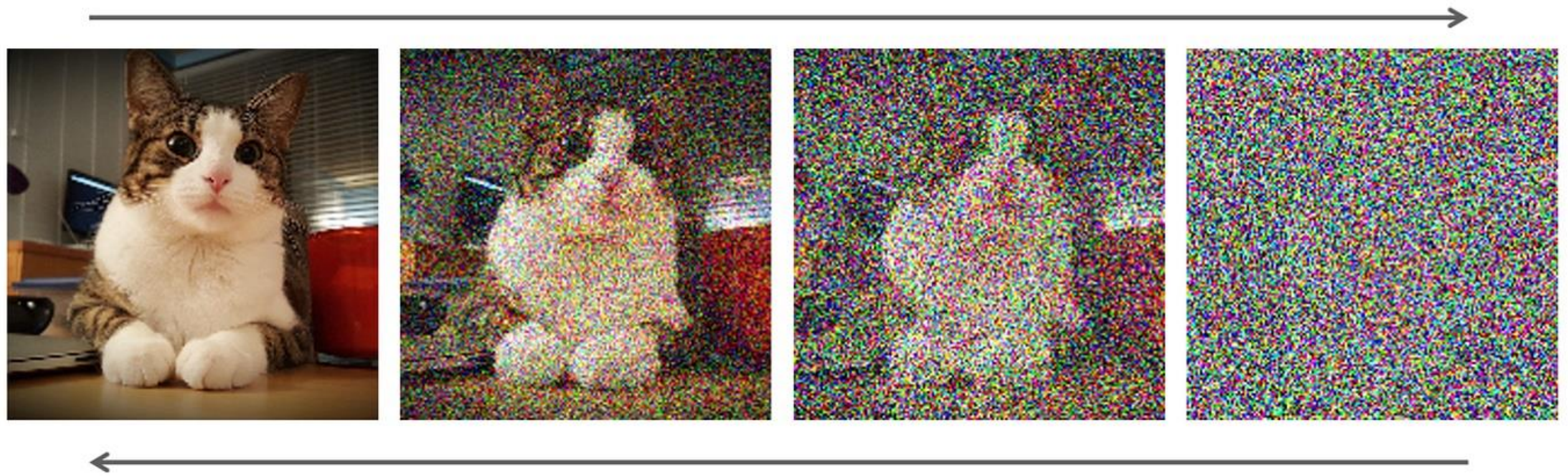
- At the **start** of training transfomers are **unstable**
  - Requiring **prohibitively** small learning rates
- Generally understood to be due to:
  - Layer Normalisation
  - (Potentially) amplified by Adam
- https://proceedings.mlr.press/v119/huang20f.html

- Prevent **overfitting**

Understanding diffusion (through VAEs)

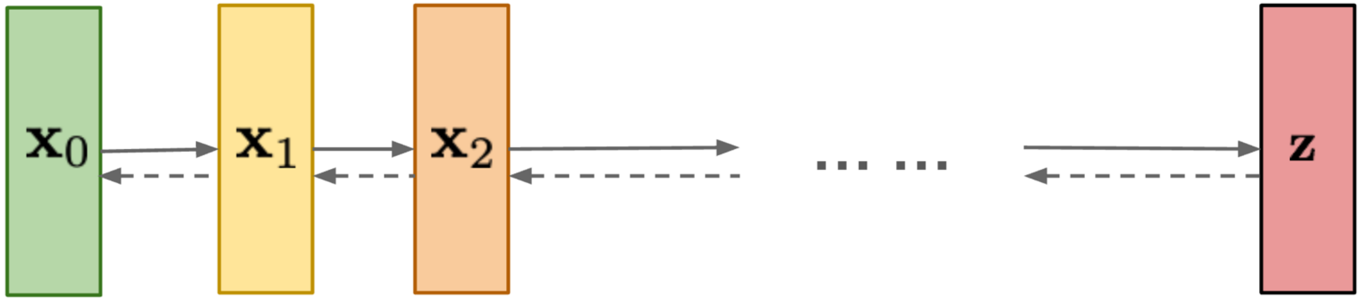
Why diffusion models

- GANs previously held SOTA image generation however:
  - Are notoriously difficult to train
  - Have been demonstrated to cover less of the generation space in comparison to explicit likelihood models
- **Diffusion models overcome this** (https://arxiv.org/pdf/2105.05233)



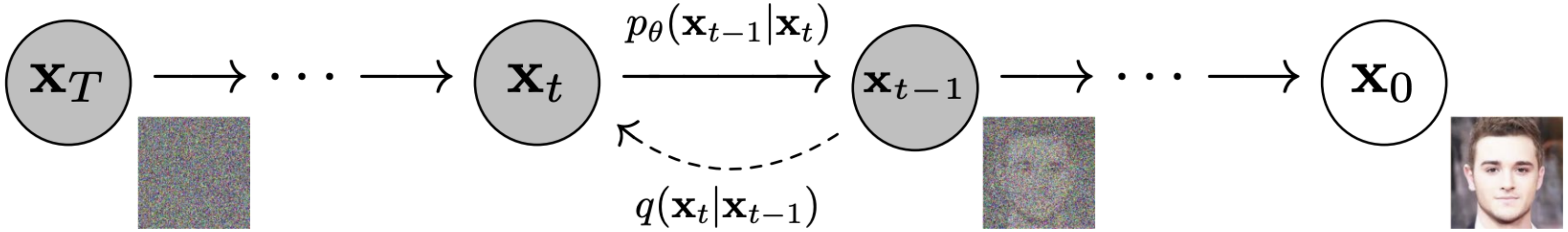
Denoising diffusion probabilistic models intuition

- Gradually add gaussian noise and then learn a decoder to reverse noise into images
- Generate images by sampling from Gaussian noise



https://arxiv.org/pdf/2006.11239

Denoising diffusion probabilistic models



- $p(x_0)$ defines the data distribution where $x_0 \in \mathcal{X}_0$ and is of dimension d
- $p_\theta(x_0) = \int p_\theta(x_{0:T}) dx_{1:T}$ is the proposed model where $\mathcal{X}_{1:T}$ is of dimension d^T i.e., each x_i is a latent variable of the same dimension as x_0
  - The same as VAEs, θ are the model parameters and q is the **variational posterior**

https://arxiv.org/pdf/2006.11239

VAE loss vs Diffusion loss

$$\textbf{VAE,}\ \boldsymbol{p_\theta(x) = \int p_\theta(x|z)p_\theta(z)dz}$$

$$\mathrm{ELBO}(q_\phi(\mathrm{z}|\mathrm{x}))=\mathbb{E}_{q_\phi(z|x)}(\log p_\theta(z,x)) - \mathbb{E}_{q_\phi(z|x)}(\log q_\phi(z|x)) \Rightarrow$$

$$L = -\underbrace{KL(q_\phi(z|x)||p_\theta(z))}_{\text{Posterior/Prior divergence}} + \underbrace{\mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x|z)\right]}_{\text{Expected log likelihood}}$$

$$\textbf{Diffusion,}\ \boldsymbol{p_\theta(x) = \int p_\theta(x_0|x_{1:T})p_\theta(x_{1:T})dx_{1:T}}$$

$$\mathrm{ELBO}(q(x_{1:T}|x_0))=\mathbb{E}_{q(x_{1:T}|x_0)}(\log p_\theta(x_{0:T})) - \mathbb{E}_{q(x_{1:T}|x_0)}(\log q(x_{1:T}|x_0)) \Rightarrow$$

Posterior/Prior divergence

Expected log likelihood

$$\boxed{\begin{gathered}L = -KL(q(x_T|x_0)||p_\theta(x_T)) - \\ \sum_{t>1} KL(q(x_{t-1}|x_t,x_0)||p_\theta(x_{t-1}|x_t)) + \mathbb{E}_{q(x_{1:T}|x_0)}\left[\log p_\theta(x_0|x_1)\right]\end{gathered}}$$

Diffusion variational parameters

- $p_\theta(x) = \int p_\theta(x_0|x_{1:T}) p_\theta(x_{1:T}) dx_{1:T}$
- $L = -KL(q(x_T|x_0)||p_\theta(x_T)) - \sum_{t>1} KL(q(x_{t-1}|x_t, x_0)||p_\theta(x_{t-1}|x_t)) + \mathbb{E}_{q(x_{1:T}|x_0)}[\log p_\theta(x_0|x_1)]$

Where is ϕ?!

Diffusion variational parameters

- $p_\theta(x) = \int p_\theta(x_0|x_{1:T})p_\theta(x_{1:T})dx_{1:T}$
- $L = -KL(q(x_T|x_0)||p_\theta(x_T)) - \sum_{t>1} KL(q(x_{t-1}|x_t, x_0)||p_\theta(x_{t-1}|x_t)) + \mathbb{E}_{q(x_{1:T}|x_0)}\,[\log p_\theta(x_0|x_1)]$
- "Forward process"/Variational posterior
  - $q(x_{1:T}|x_0) = \prod_{t=1}^{T} q(x_t|x_{t-1}) : q(x_t|x_{t-1}) \sim N\left(\sqrt{1-\beta_t}x_{t-1}, \beta_t \boldsymbol{I}\right)$
  - $\beta_1, \dots, \beta_T$ are the variational parameters **however**, these are considered **fixed** (in the simple case)
    - This is equivalent to a **very restrictive** $\boldsymbol{\mathcal{F}}$

Diffusion loss decomposition

- $L = -KL(q(x_T|x_0)||p_\theta(x_T)) - \sum_{t>1} KL(q(x_{t-1}|x_t, x_0)||p_\theta(x_{t-1}|x_t)) + \mathbb{E}_{q(x_{1:T}|x_0)}[\log p_\theta(x_0|x_1)]$
- $p_\theta(x_{t-1}|x_t)$ is the **neural network we are training**
  - $p_\theta(x_{t-1}|x_t) = N(\mu_\theta(x_t, t), \Sigma_\theta(x_t, t))$ with $\Sigma_\theta(x_t, t)$ fixed at $\sigma_t^2 \boldsymbol{I}$
  - Therefore, the neural network is modelling $\mu_\theta(x_t, t)$
- $L_{t-1} \propto \frac{1}{2\sigma_t^2}||\tilde{\mu}_t(x_t, x_0) - \mu_\theta(x_t, t)||^2$
  - Where $q(x_{t-1}|x_t, x_0) = N(\tilde{\mu}_t(x_t, x_0), \tilde{\beta}_t \boldsymbol{I})$ such that:
    - $\tilde{\mu}_t(x_t, x_0) = \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t} x_0 + \frac{\sqrt{\bar{\alpha}_t}(1-\bar{\alpha}_t)}{1-\bar{\alpha}_t} x_t$
    - $\tilde{\beta}_t = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t} \beta_t$

$\bar{\alpha}_t, \alpha_t$ and $\tilde{\beta}_t$ are **reparameterisations** of the **fixed variational parameters**, β_t

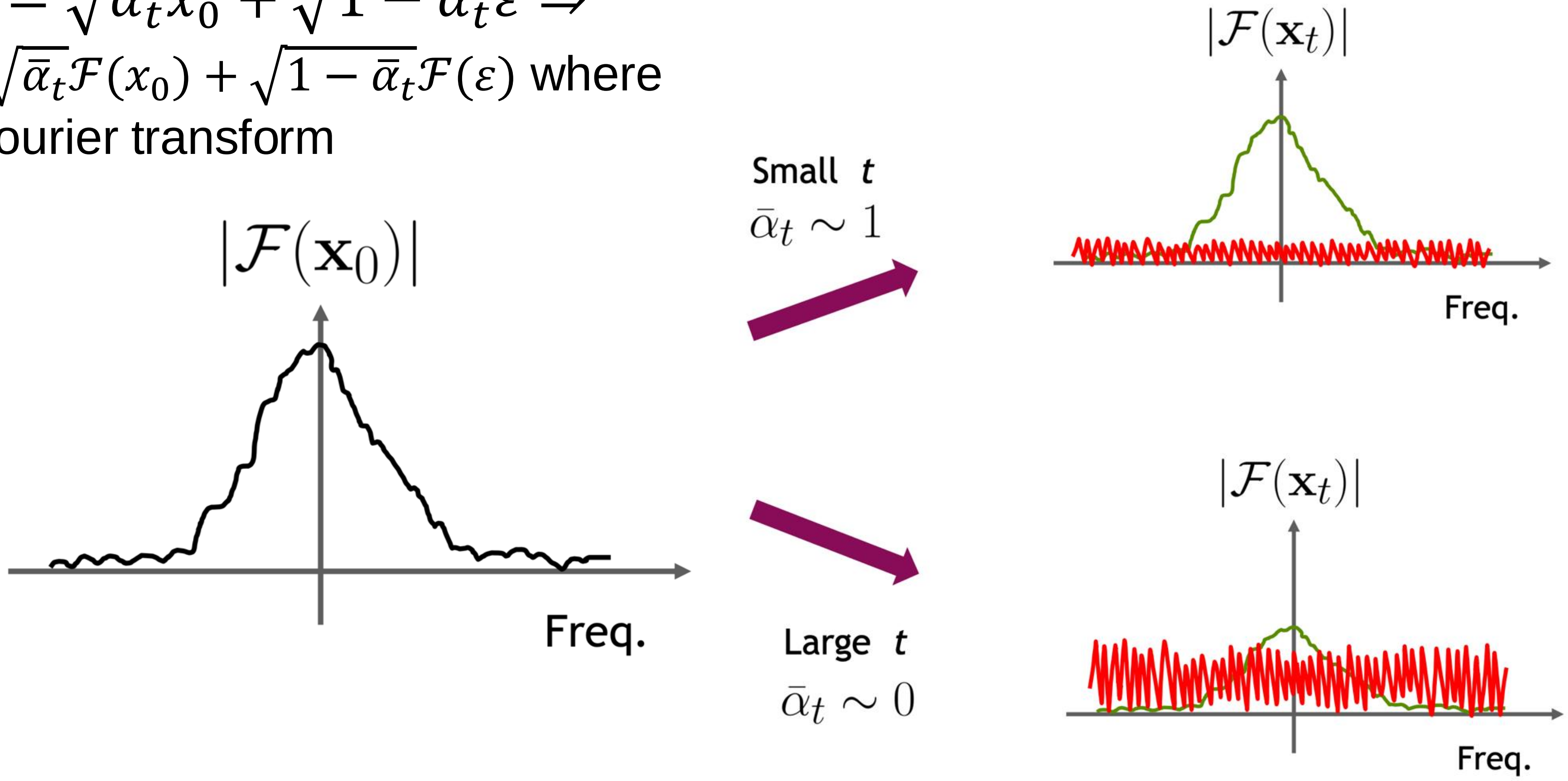
Diffusion simple loss through reparameterisation

$$L = \mathbb{E}_{t,x_0,\epsilon}\left[\underbrace{\frac{\beta_t^2}{2\sigma_t^2(1-\beta_t)(1-\bar{\alpha}_t)}}_{\lambda_t}\left\|\epsilon - \epsilon_\theta\left(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon, t\right)\right\|^2\right]$$

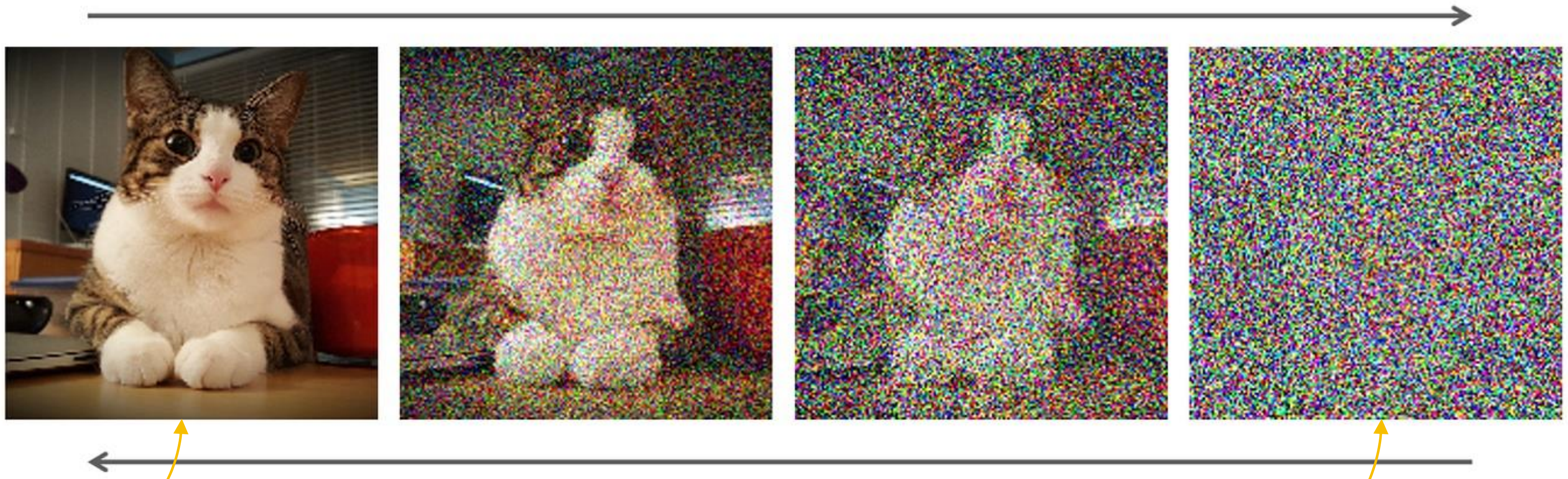
- Where ϵ is the noise added at time t and;
- ϵ_θ is a **neural network** instead of μ_θ
- The problem is **reparametersied** to predict the **noise** at step t rather than the **structure**
  - Empirically demonstrated to improve performance
- λ_t is often **fixed at 1** regardless of the step
  - Produces a biased estimator **however** empirically this produces better results as it weights latent variables closer to x_t more greatly

Understanding the forward process

- Recall $x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1-\bar{\alpha}_t}\varepsilon \Rightarrow$
  - $\mathcal{F}(x_t) = \sqrt{\bar{\alpha}_t}\mathcal{F}(x_0) + \sqrt{1-\bar{\alpha}_t}\mathcal{F}(\varepsilon)$ where $\mathcal{F}$ is the fourier transform



Content-detail tradeoff



Denoising model is specialized for generating high-frequency content

Denoising model is specialized for generating low-frequency content

Network architecture

- U-Net
  - Symmetric compression and decompression
  - Residual (as in ResNet) connections between compressed and decompressed layers of equal dimension

- Diffusion U-Net = U-Net +
  - Timestep embedding
  - Attention layer between during later compression/decompression stages