Neural Network Basics

Jindřich Libovický, Jindřich Helcl

February 20, 2019

EUROPEAN UNION
European Structural and Investment Fund
Operational Programme Research,
Development and Education

Charles University
Faculty of Mathematics and Physics
Institute of Formal and Applied Linguistics

Outline

Deep Learning in NLP

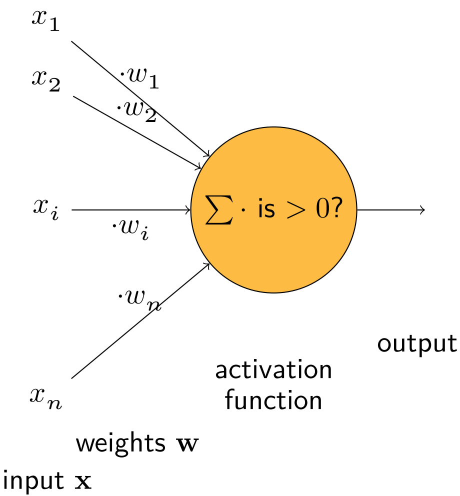
- NLP tasks learn end-to-end using deep learning — the number-one approach in current research
- State of the art in POS tagging, parsing, named-entity recognition, machine translation, ...
- Good news: training without almost any linguistic insight
- Bad news: requires enormous amount of training data and really big computational power

What is deep learning?

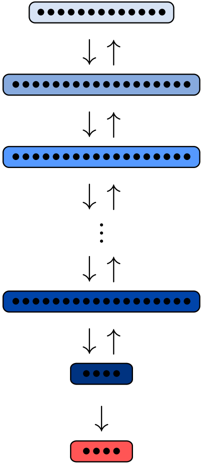
- Buzzword for machine learning using neural networks with many layers using back-propagation
- Learning of a real-valued function with millions of parameters that solves a particular problem
- Learning more and more abstract representation of the input data until we reach such a suitable representation for our problem

Neural Networks Basics

Single Neuron



Neural Network



$$x$$

$$\downarrow$$

$$h_1 = f(W_1 x + b_1)$$

$$\downarrow$$

$$h_2 = f(W_2 h_1 + b_2)$$

$$\downarrow$$

$$\vdots$$

$$\downarrow$$

$$h_n = f(W_n h_{n-1} + b_n)$$

$$\downarrow$$

$$o = g(W_o h_n + b_o)$$

$$\downarrow$$

$$E = e(o, t) \quad \rightarrow \quad \frac{\partial E}{\partial o}$$

$$\uparrow$$

$$\frac{\partial E}{\partial W_o} = \frac{\partial E}{\partial o} \cdot \frac{\partial o}{\partial W_o}$$

$$\uparrow$$

$$\uparrow$$

$$\uparrow$$

$$\uparrow$$

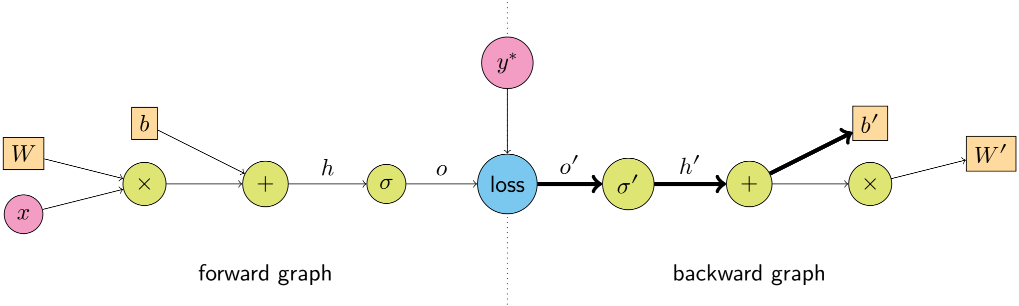
$$\uparrow$$

Implementation

Logistic regression:

$$y = \sigma\left(Wx + b\right) \tag{1}$$

Computation graph:



Representing Words

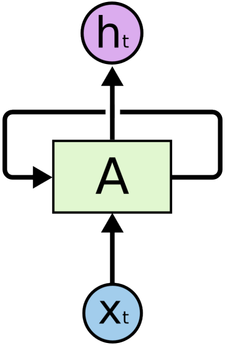
Discrete vs. Continous

Representing Sequences

Representing Sequences

Recurrent Networks

…the default choice for sequence labeling

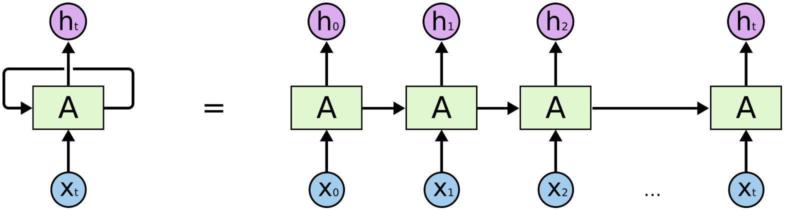


- inputs: $x_{,} \ldots, x_T$
- initial state $h_0 = \mathbf{0}$, a result of previous computation, trainable parameter
- recurrent computation: $h_t = A(h_{t-1}, x_t)$

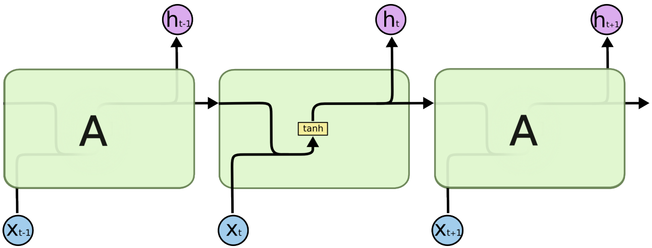
RNN as Imperative Code

```
def rnn(initial_state, inputs):
  prev_state = initial_state
  for x in inputs:
    new_state, output = rnn_cell(x, prev_state)
    prev_state = new_state
    yield output
```

RNN as a Fancy Image



Vanilla RNN



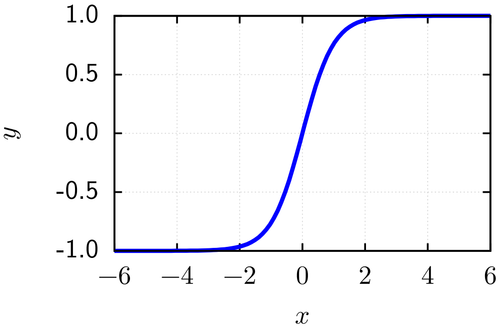
$$h_t = \tanh\left(W[h_{t-1}; x_t] + b\right)$$

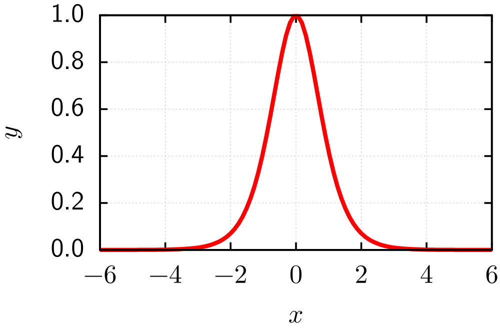
- cannot propagate long-distance relations
- vanishing gradient problem

Vanishing Gradient Problem (1)

$$\tanh x = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

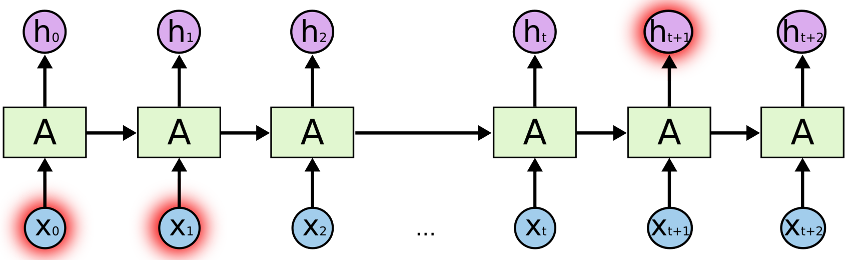
$$\frac{\mathrm{d} \tanh x}{\mathrm{d}x} = 1 - \tanh^2 x \in (0, 1]$$





Weight initialized $\sim \mathcal{N}(0, 1)$ to have gradients further from zero.

Vanishing Gradient Problem (2)

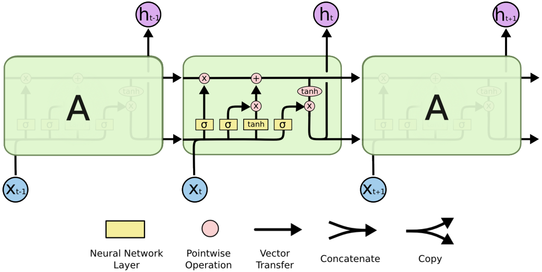


$$\frac{\partial E_{t+1}}{\partial b} = \frac{\partial E_{t+1}}{\partial h_{t+1}} \cdot \frac{\partial h_{t+1}}{\partial b} \quad \text{(chain rule)}$$

$$
\begin{aligned}
\frac{\partial h_t}{\partial b} &= \frac{\partial \tanh \overbrace{(W_h h_{t-1} + W_x x_t + b)}^{=z_t \text{ (activation)}}}{\partial b} \qquad (\tanh' \text{ is derivative of } \tanh) \\
&= \tanh'(z_t) \cdot \left(\frac{\partial W_h h_{t-1}}{\partial b} + \underbrace{\frac{\partial W_x x_t}{\partial b}}_{=0} + \underbrace{\frac{\partial b}{\partial b}}_{=1} \right) \\
&= \underbrace{W_h}_{\sim \mathcal{N}(0,1)} \underbrace{\tanh'(z_t)}_{\in (0;1]} \frac{\partial h_{t-1}}{\partial b} + \tanh'(z_t)
\end{aligned}
$$

Long Short-Term Memory Networks

LSTM = Long short-term memory

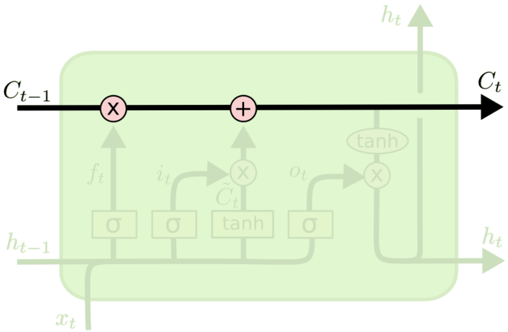


Control the gradient flow by explicitly gating:

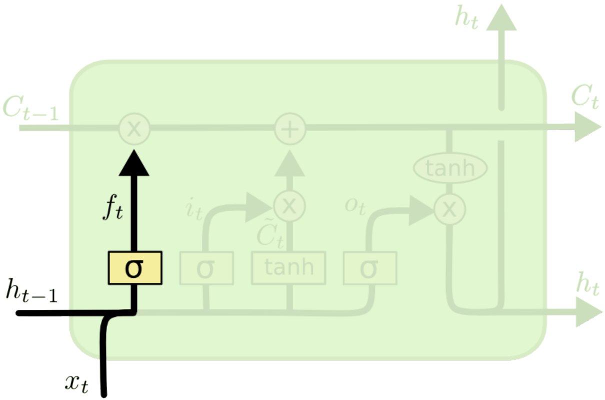
- what to use from input,
- what to use from hidden state,
- what to put on output

LMST: Hidden State

- two types of hidden states
- h_t — "public" hidden state, used an output
- c_t — "private" memory, no non-linearities on the way
- direct flow of gradients (without multiplying by ≤ 1 derivatives)



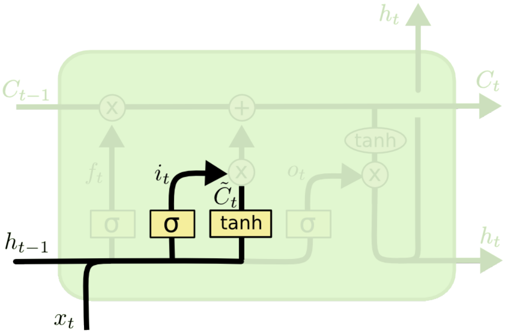
LSTM: Forget Gate



$$f_t = \sigma\left(W_f[h_{t-1}; x_t] + b_f\right)$$

- based on input and previous state, decide what to forget from the memory

LSTM: Input Gate

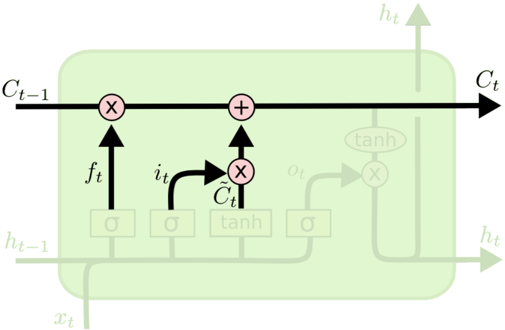


$$i_t = \sigma\left(W_i \cdot [h_{t-1}; x_t] + b_i\right)$$

$$\tilde{C}_t = \tanh\left(W_c \cdot [h_{t-1}; x_t] + b_C\right)$$

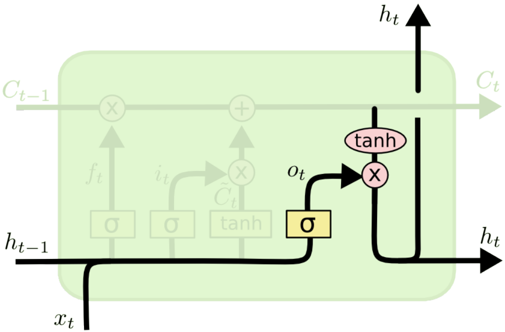
- $\tilde{C}$ — candidate what may want to add to the memory
- i_t — decide how much of the information we want to store

LMST: Cell State Update



$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$

LSTM: Output Gate



$$o_t = \sigma\left(W_o \cdot [h_{t-1}; x_t] + b_o\right)$$

$$h_t = o_t \odot \tanh C_t$$

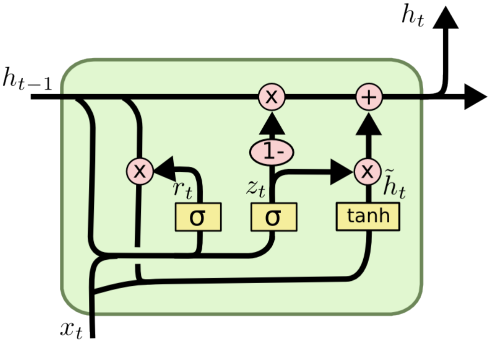
Here we are, LSTM!

$$
\begin{aligned}
f_t &= \sigma\left(W_f[h_{t-1}; x_t] + b_f\right) \\
i_t &= \sigma\left(W_i \cdot [h_{t-1}; x_t] + b_i\right) \\
o_t &= \sigma\left(W_o \cdot [h_{t-1}; x_t] + b_o\right) \\
\tilde{C}_t &= \tanh\left(W_c \cdot [h_{t-1}; x_t] + b_C\right) \\
C_t &= f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \\
h_t &= o_t \odot \tanh C_t
\end{aligned}
$$

Question How would you implement it efficiently?
Compute all gates in a single matrix multiplication.

Gated Recurrent Units

update gate	$z_t = \sigma(x_t W_z + h_{t-1} U_z + b_z) \in (0,1)$
remember gate	$r_t = \sigma(x_t W_r + h_{t-1} U_r + b_r) \in (0,1)$
candidate hidden state	$\tilde{h}_t = \tanh\left(x_t W_h + (r_t \odot h_{t-1}) U_h\right) \in (-1,1)$
hidden state	$h_t = (1 - z_t) \odot h_{t-1} + z_t \cdot \tilde{h}_t$



LSTM vs. GRU

- GRU is smaller and therefore faster
- performance similar, task dependent
- theoretical limitation: GRU accepts regular languages, LSTM can simulate counter machine

;

RNN in PyTorch

```
rnn = nn.LSTM(input_dim, hidden_dim=512, num_layers=1,
    bidirectional=True, dropout=0.8)
output, (hidden, cell) = self.rnn(x)
```

https://pytorch.org/docs/stable/nn.html?highlight=lstm#torch.nn.LSTM

RNN in TensorFlow

```
inputs = ... # float tf.Tensor of shape [batch, length, dim]
lengths = ... # int tf.Tensor of shape [batch]

# Cell objects are templates
fw_cell = tf.nn.rnn_cell.LSTMCell(512, name="fw_cell")
bw_cell = tf.nn.rnn_cell.LSTMCell(512, name="bw_cell")

outputs, states = tf.nn.bidirectional_dynamic_rnn(
    cell_fw, cell_bw, inputs, sequence_length=lengths)
```

https://www.tensorflow.org/api_docs/python/tf/nn/bidirectional_dynamic_rnn

Bidirectional Networks

- simple trick to improve performance
- run one RNN forward, second one backward and concatenate outputs

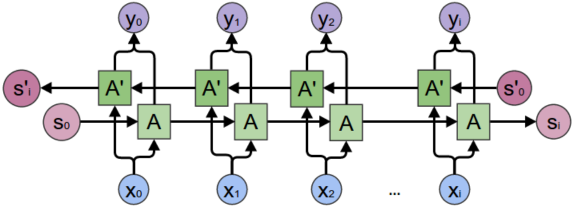


Image from: `http://colah.github.io/posts/2015-09-NN-Types-FP/`

- state of the art in tagging, crucial for neural machine translation

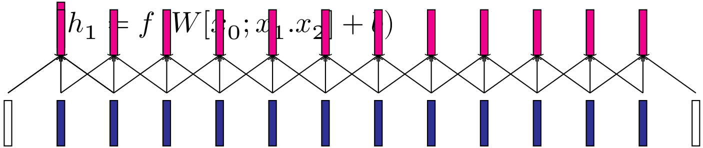
Representing Sequences

Convolutional Networks

1-D Convolution

≈ sliding window over the sequence

$$h_i = f\left(W\left[x_{i-1}; x_i; x_{i+1}\right] + b\right)$$



$x_0 = \vec{0}$ embeddings $\mathbf{x} = (x_1, \ldots, x_N)$ $x_N = \vec{0}$

pad with 0s if we want to keep sequence length

1-D Convolution: Pseudocode

```
xs = ... # input sequnce

kernel_size = 3 # window size
filters = 300 # output dimensions
strides=1     # step size

W = trained_parameter(xs.shape[2] * kernel_size, filters)
b = trained_parameter(filters)
window = kernel_size // 2

outputs = []
for i in range(window, xs.shape[1] - window):
    h = np.mul(W, xs[i - window:i + window]) + b
    outputs.append(h)
return np.array(h)
```

1-D Convolution: Frameworks

TensorFlow

```
h = tf.layers.conv1d(x, filters=300 kernel_size=3,
                     strides=1, padding='same')
```

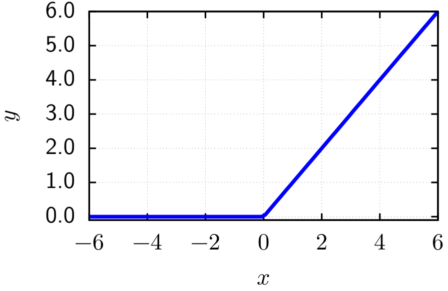
https://www.tensorflow.org/api_docs/python/tf/layers/conv1d

PyTorch

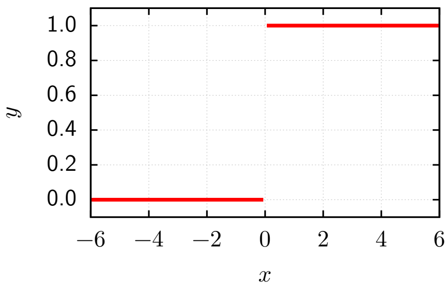
```
conv = nn.Conv1d(in_channels, out_channels=300, kernel_size=3, stride=1,
                 padding=0, dilation=1, groups=1, bias=True)
h = conv(x)
```

https://pytorch.org/docs/stable/nn.html#torch.nn.Conv1d

ReLU:



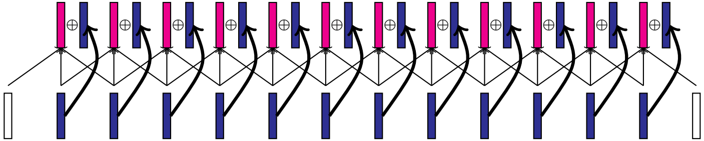
Derivative of ReLU:



faster, suffer less with vanishing gradient

Residual Connections

$$h_i = f\left(W\left[x_{i-1}; x_i; x_{i+1}\right] + b\right) + x_i$$



$x_0 = \vec{0}$ embeddings $\mathbf{x} = (x_1, \ldots, x_N)$ $x_N = \vec{0}$

Allows training deeper networks.

Why do you it helps?

Better gradient flow – the same as in RNNs.

Residual Connections: Numerical Stability

Numerically unstable, we need activation to be in similar scale $\Rightarrow$ layer normalization.
Activation before non-linearity is normalized:

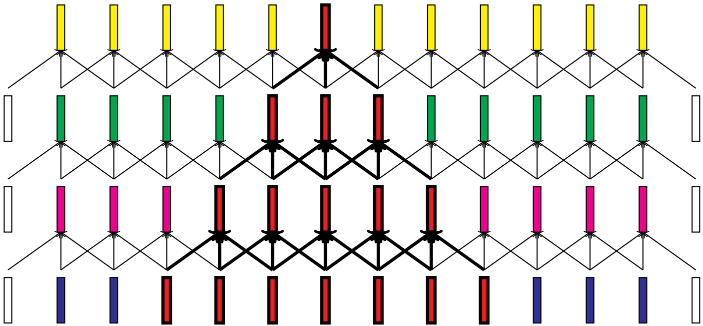
$$\bar{a}_i = \frac{g_i}{\sigma_i}(a_i - \mu_i)$$

...g is a trainable parameter, μ, σ estimated from data.

$$\mu = \frac{1}{H}\sum_{i=1}^{H} a_i$$

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

Receptive Field



$x_0 = \vec{0}$ embeddings $\mathbf{x} = (x_1, \ldots, x_N)$ $x_N = \vec{0}$

Can be enlarged by dilated convolutions.

Convolutional architectures

$+$

- extremely computationally efficient

$-$

- limited context
- by default no aware of n-gram order

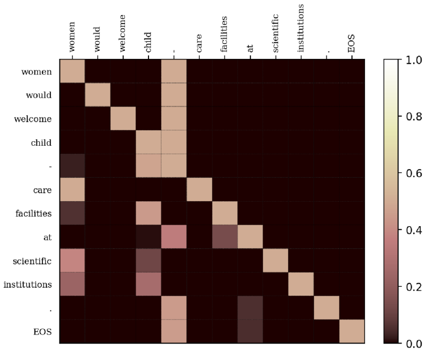
- max-pooling over the hidden states $=$ element-wise maximum over sequence
- can be understood as an $\exists$ operator over the feature extractors

Representing Sequences

Self-attentive Networks

Self-attentive Networks

- In some layers: states are linear combination of previous layer states
- Originally for the Transformer model for machine translation



- similarity matrix between all pairs of states
- $O(n^2)$ memory, $O(1)$ time (when paralelized)
- next layer: sum by rows

Multi-headed scaled dot-product attention

Single-head setup

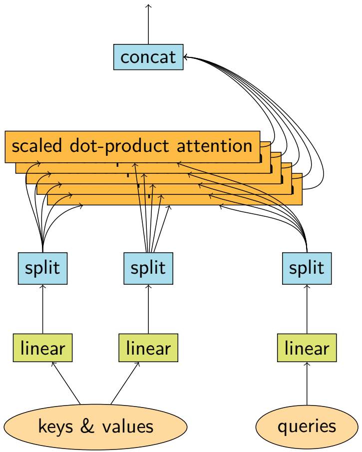
$$\mathrm{Attn}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V$$

$$h_{i+1} = \sum \mathrm{softmax}\left(\frac{h_i h_i^\top}{\sqrt{d}}\right)$$

Multihead-head setup

$$\mathrm{Multihead}(Q, V) = (H_1 \oplus \cdots \oplus H_h)W^O$$

$$H_i = \mathrm{Attn}(QW_i^Q, VW_i^K, VW_i^V)$$



Dot-Product Attention in PyTorch

```
def attention(query, key, value, mask=None):
    d_k = query.size(-1)
    scores = torch.matmul(query, key.transpose(-2, -1)) \
             / math.sqrt(d_k)
    p_attn = F.softmax(scores, dim = -1)
    return torch.matmul(p_attn, value), p_attn
```

Dot-Product Attention in TensorFlow

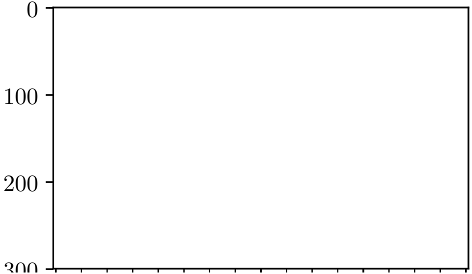
```
def scaled_dot_product(self, queries, keys, values):
    o1 = tf.matmul(queries, keys, transpose_b=True)
    o2 = o1 / (dim**0.5)

    o3 = tf.nn.softmax(o2)
    return tf.matmul(o3, values)
```

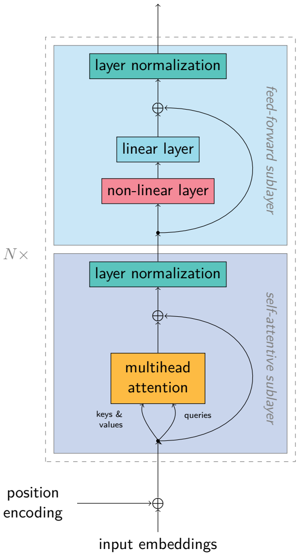
Position Encoding

Model cannot be aware of the position in the sequence.

$$\mathrm{pos}(i) = \begin{cases} \sin\left(\frac{t}{10^4}^{\frac{i}{d}}\right), & \text{if } i \mod 2 = 0 \\ \cos\left(\frac{t}{10^4}^{\frac{i-1}{d}}\right), & \text{otherwise} \end{cases}$$



Stacking self-attentive Layers



- several layers (original paper 6)
- each layer: 2 sub-layers: self-attention and feed-forward layer
- everything inter-connected with residual connections

Architectures Comparison

	computation	sequential operations	memory
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n \cdot d)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(n \cdot d)$
Self-attentive	$O(n^2 \cdot d)$	$O(1)$	$O(n^2 \cdot d)$

d model dimension, n sequence length, k convolutional kernel

Summary

1. Discrete symbols $\rightarrow$ continuous representation with trained embeddings
2. Architectures to get suitable representation: recurrent, convolutional, self-attentive
3. Output: classification, sequence labeling, autoregressive decoding …next time