Lecture Notes for

Machine Learning in Python

Professor Eric Larson

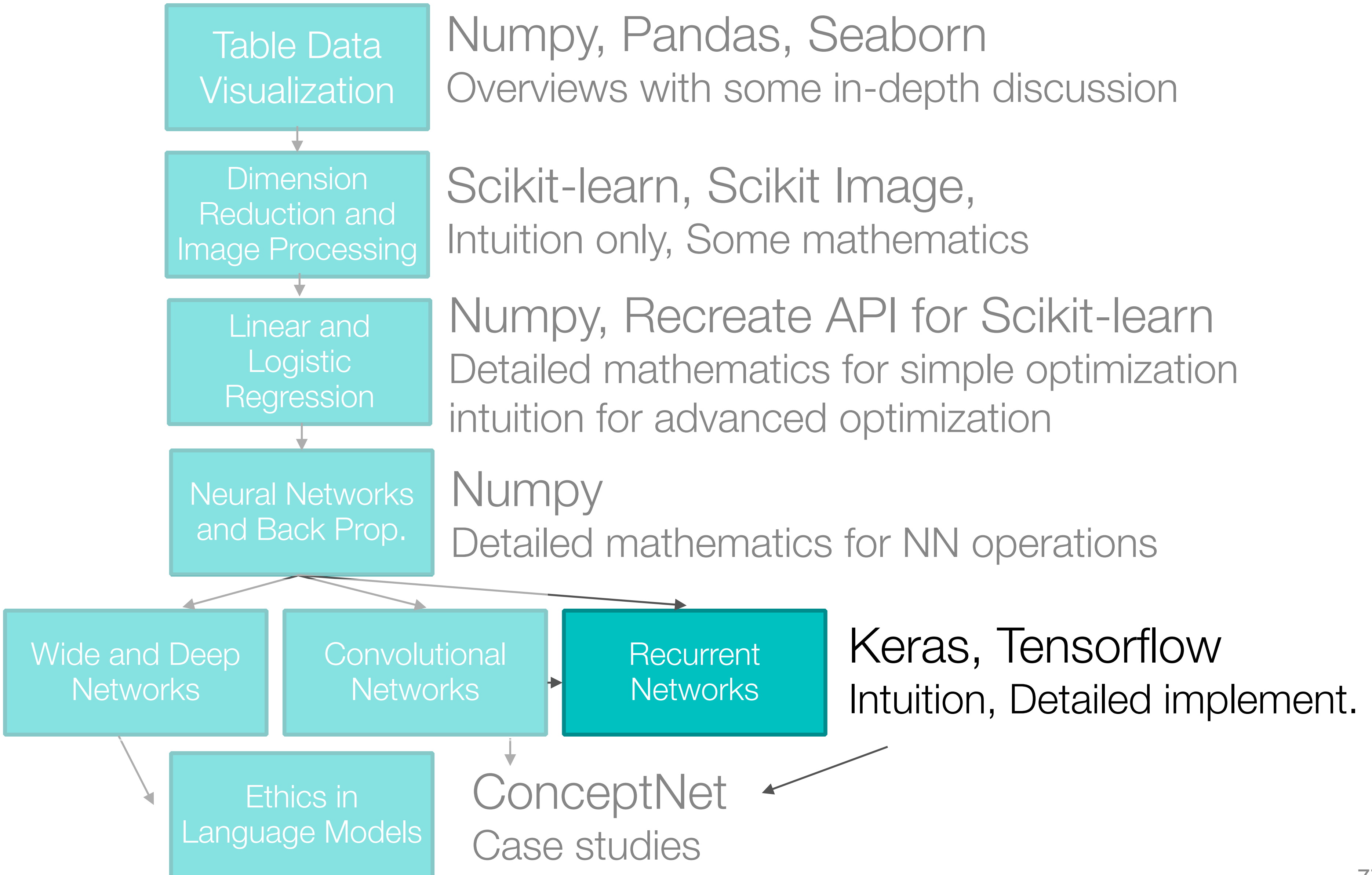
Seq-2-Seq and Transformers

Archived

Lecture Agenda

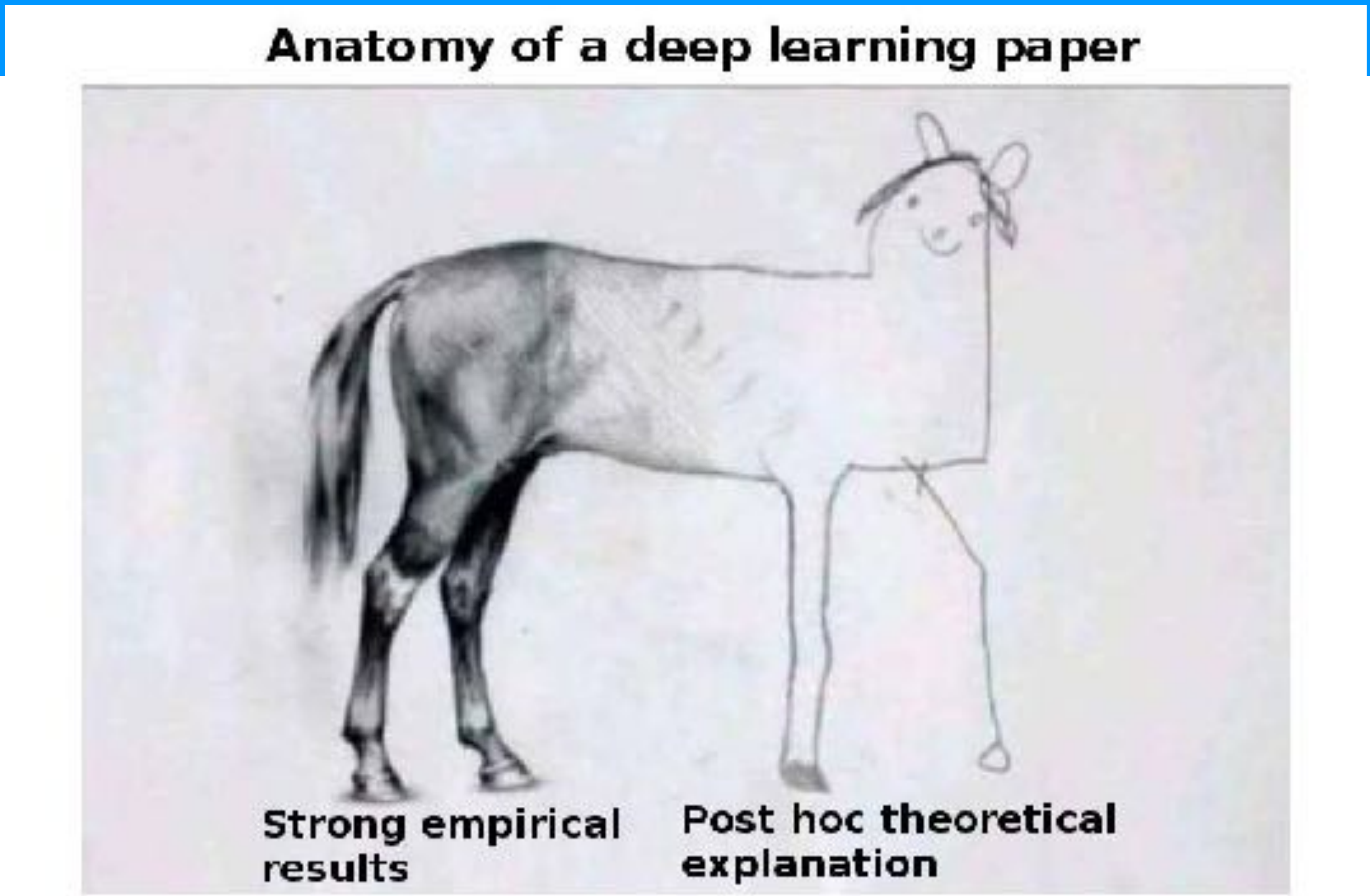
- Logistics
  - RNNs due **During Finals Time**
- Agenda
  - Sequence to sequence
  - Transformers

Class Overview, by topic



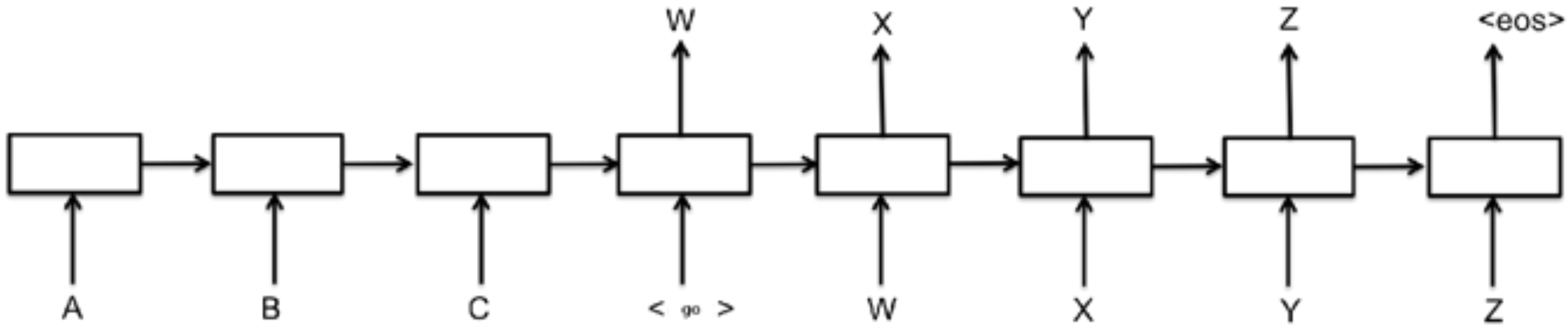
Last Time

Sequence to Sequence



Modeling Sequence to Sequence

Need to translate outputs of unknown size.



- Additional Vocabulary Special Casing:
    - <UNKNOWN>, for unknown input or characters not included in vocabulary
    - <EOS>, end of sentence
    - <GO>, start output sequence
    - <DONTCARE>, outputs before <GO> command

Sutskever et al. Sequence to Sequence Learning with Neural Networks, arXiv. 2014
https://arxiv.org/pdf/1409.3215.pdf

Modeling Sequence to Sequence

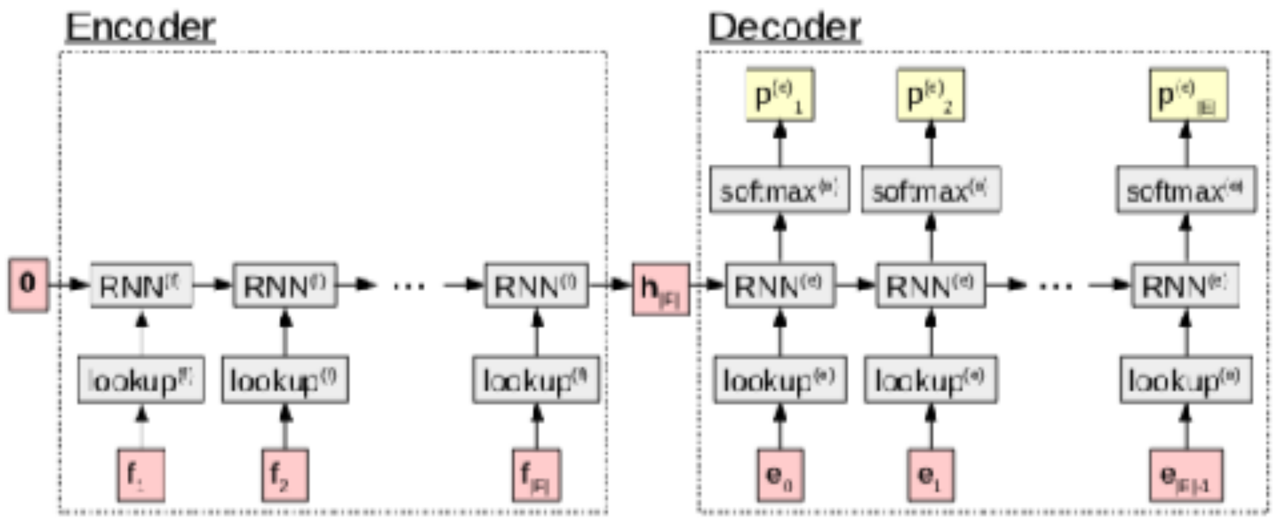


Figure 21: A computation graph of the encoder-decoder model.

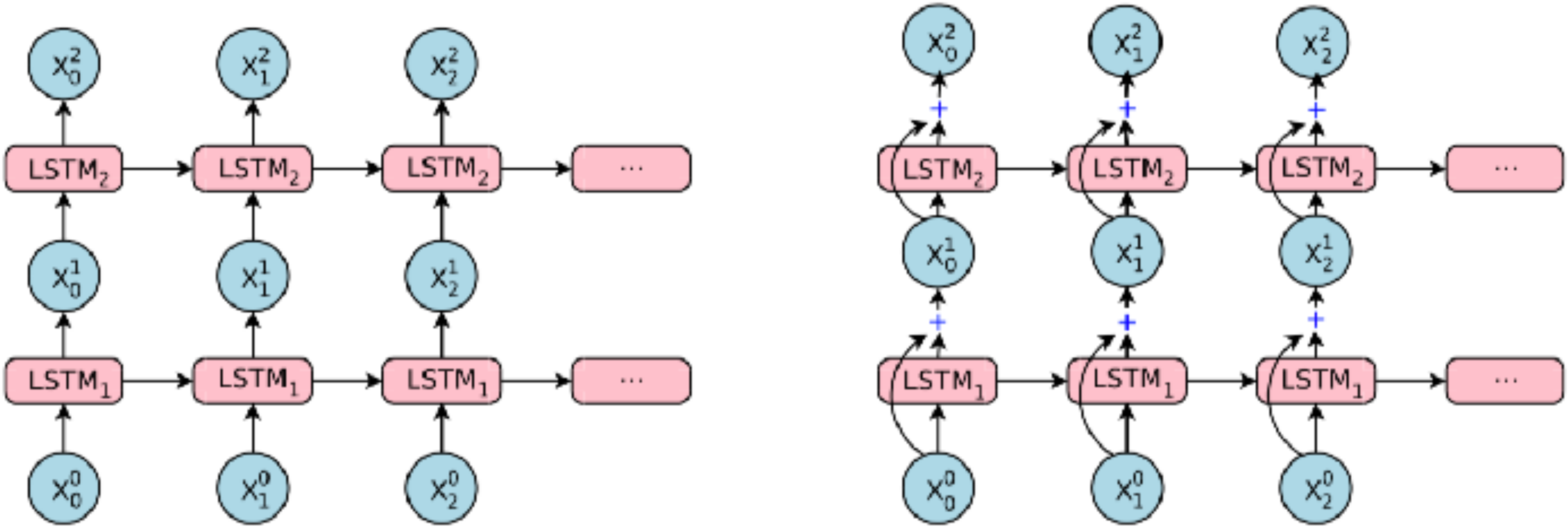
- **Training Process**: Give actual decoded letters for predicting next token
- **Decoding Process** can alter reliability of results:
  - Greedy Search, always choose most likely “next” symbol, seed
  - Keep list of “best” predictions for seeding (i.e., Beam Search)

Graham Neubig. 2017
Neural Machine Translation and
Sequence-to-sequence Models: A Tutorial
https://arxiv.org/pdf/1703.01619.pdf

https://github.com/m2dsupsdlclass/lectures-labs/blob/master/labs/07_seq2seq/Translation_of_Numeric_Phrases_with_Seq2Seq_rendered.ipynb

GNMT: Residuals

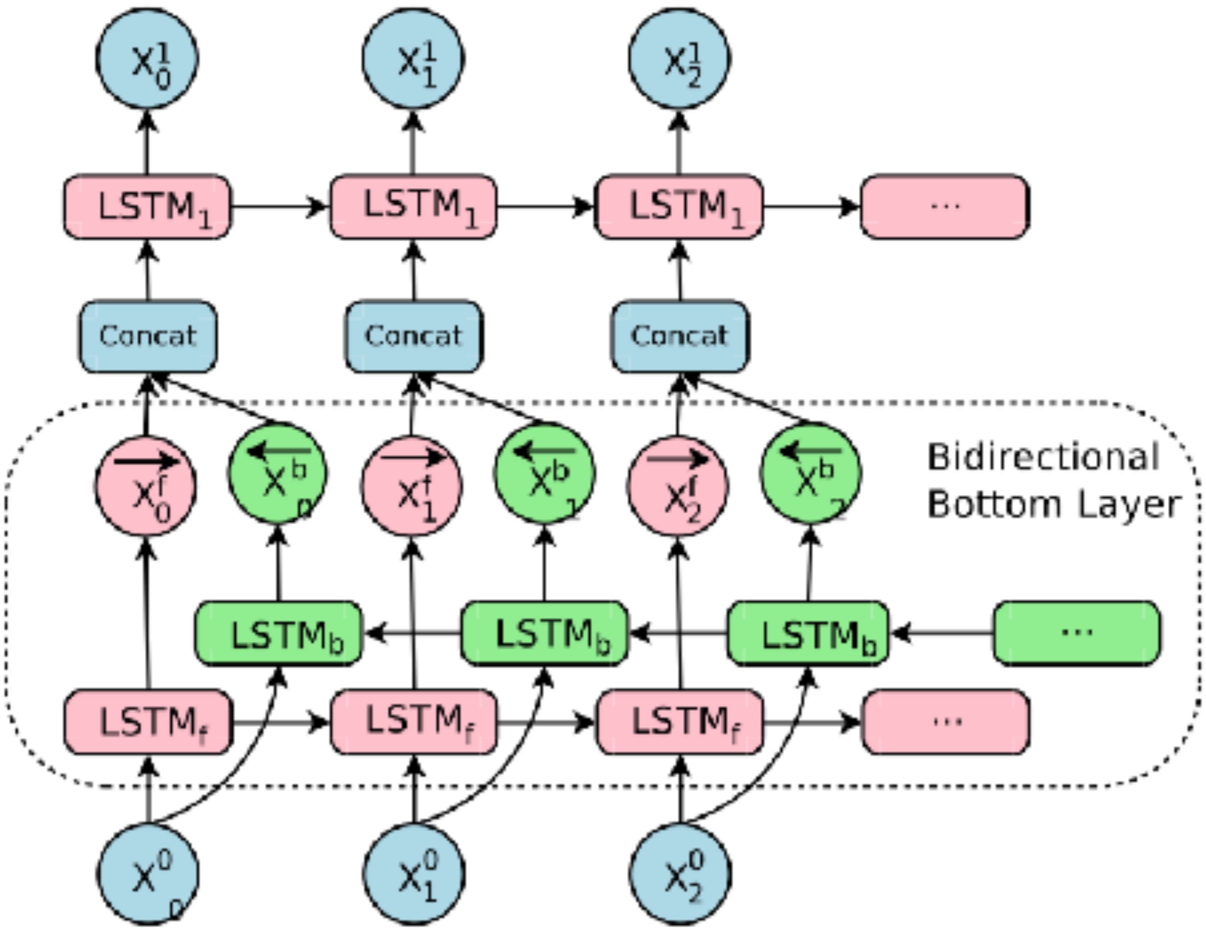
- Google, 2016



Google's Neural Machine Translation:
https://arxiv.org/pdf/1609.08144.pdf

GNMT: Bidirectionality

- Google, 2016



Google Neural Machine Translation:
https://arxiv.org/pdf/1609.08144.pdf

GNMT: Attention

- Google, 2016

$$s_t = AttentionFunction(\mathbf{y}_{i-1}, \mathbf{x}_t) \quad \forall t, \quad 1 \leq t \leq M$$

$$p_t = \exp(s_t) / \sum_{t=1}^{M} \exp(s_t) \qquad \forall t, \quad 1 \leq t \leq M$$

$$\mathbf{a}_i = \sum_{t=1}^{M} p_t . \mathbf{x}_t$$

where $\mathbf{x}_t$ is state of the t^{th} encoder
$\mathbf{y}_{i\text{-}1}$ is the state of the previous decoder
and $\mathbf{a}_i$ is the input for the i^{th} decoder

Google Neural Machine Translation:
https://arxiv.org/pdf/1609.08144.pdf

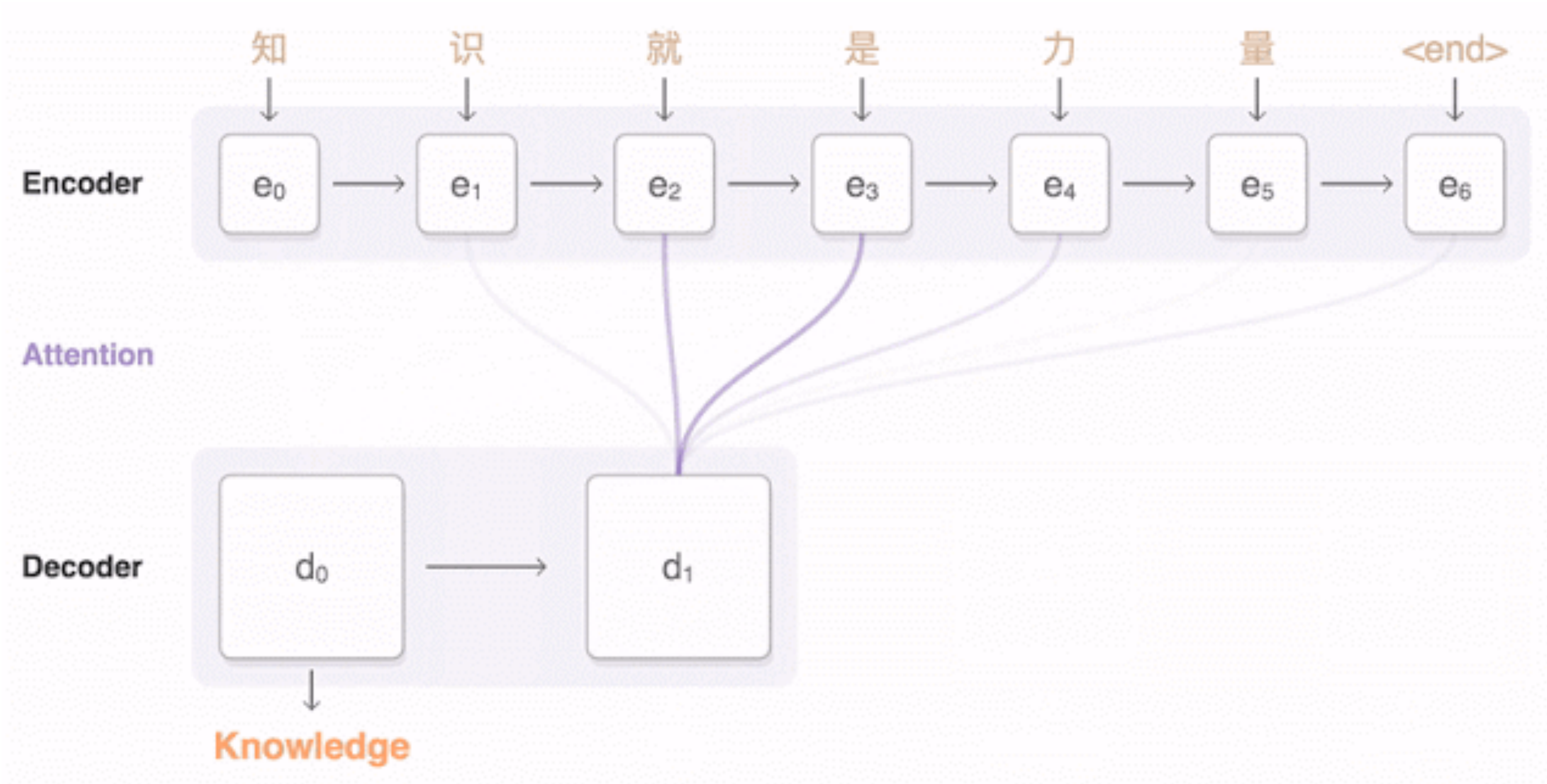
GNMT: Attention

$$s_t = AttentionFunction(\mathbf{y}_{i-1}, \mathbf{x}_t) \quad \forall t, \quad 1 \leq t \leq M$$

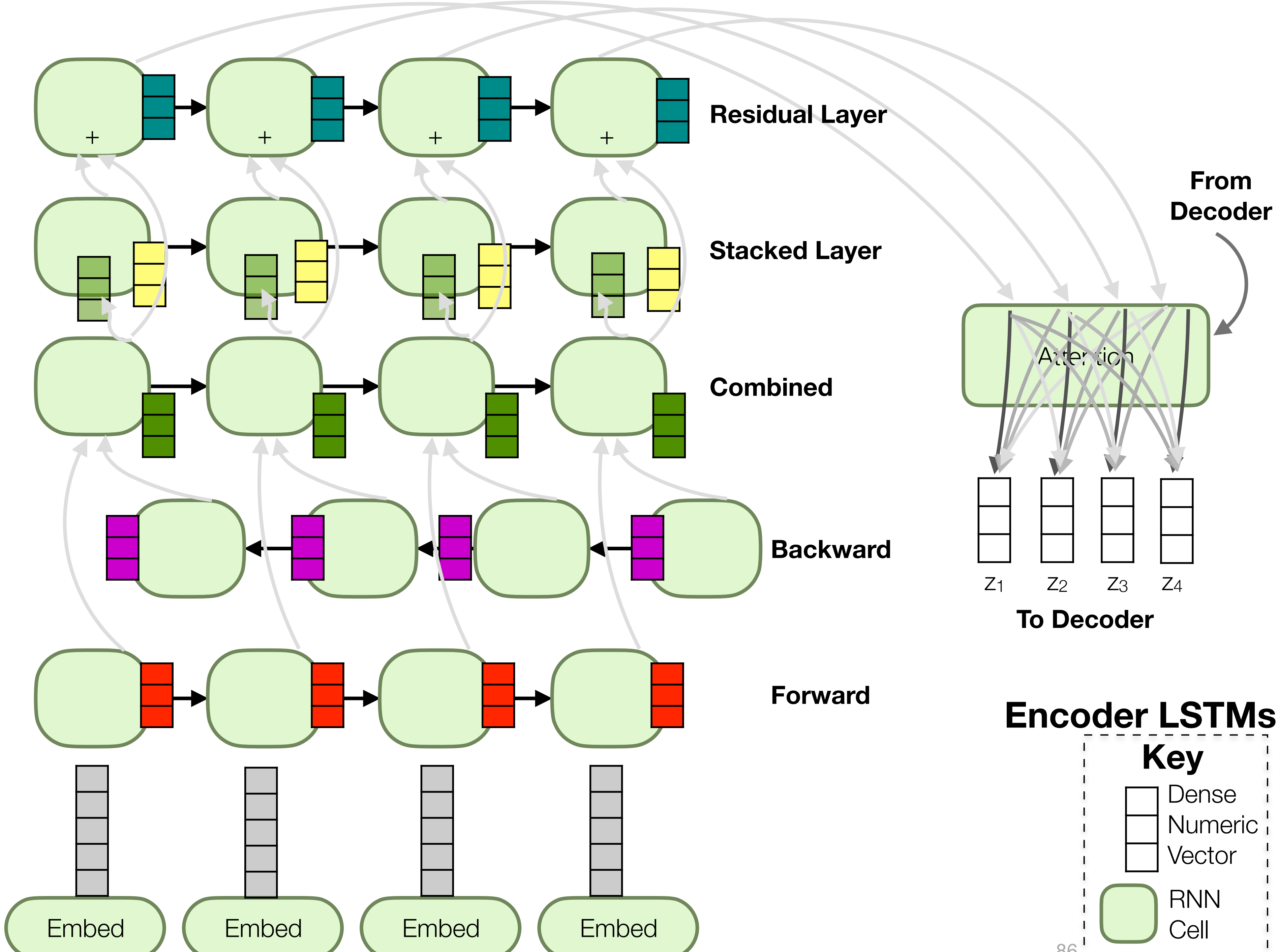
$$p_t = \exp(s_t) / \sum_{t=1}^{M} \exp(s_t) \qquad \forall t, \quad 1 \leq t \leq M$$

$$\mathbf{a}_i = \sum_{t=1}^{M} p_t . \mathbf{x}_t$$

- Google, 2016

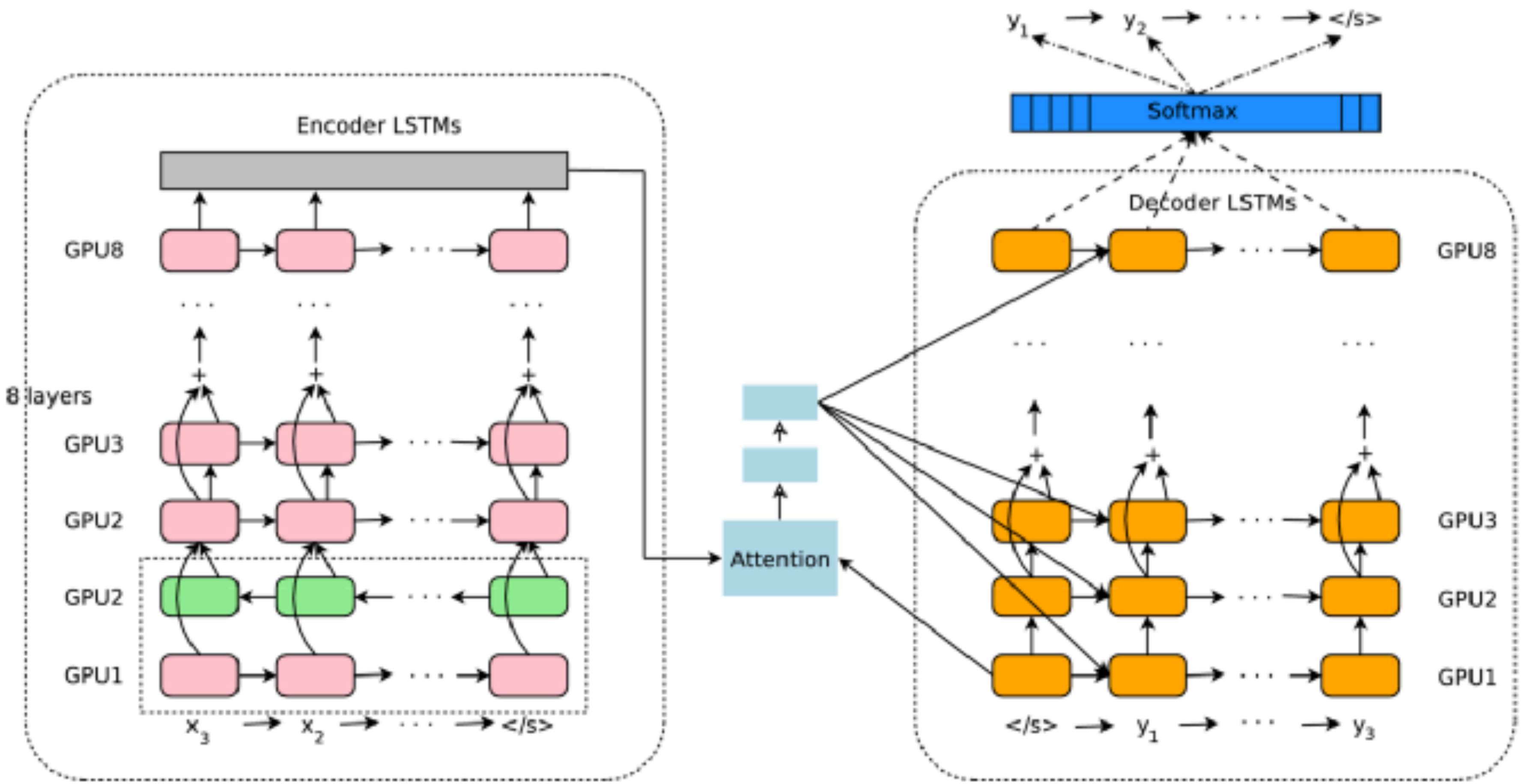


Google Neural Machine Translation:
https://arxiv.org/pdf/1609.08144.pdf
https://medium.com/@Synced/history-and-frontier-of-the-neural-machine-translation-dc981d25422d


Residual Layer
Stacked Layer
Combined
Backward
Forward
Embed
Embed
Embed
Embed
From Decoder
Attention
Z1
Z2
Z3
Z4
To Decoder
Encoder LSTMs
Key
Dense Numeric Vector
RNN Cell

GNMT: Putting it All together

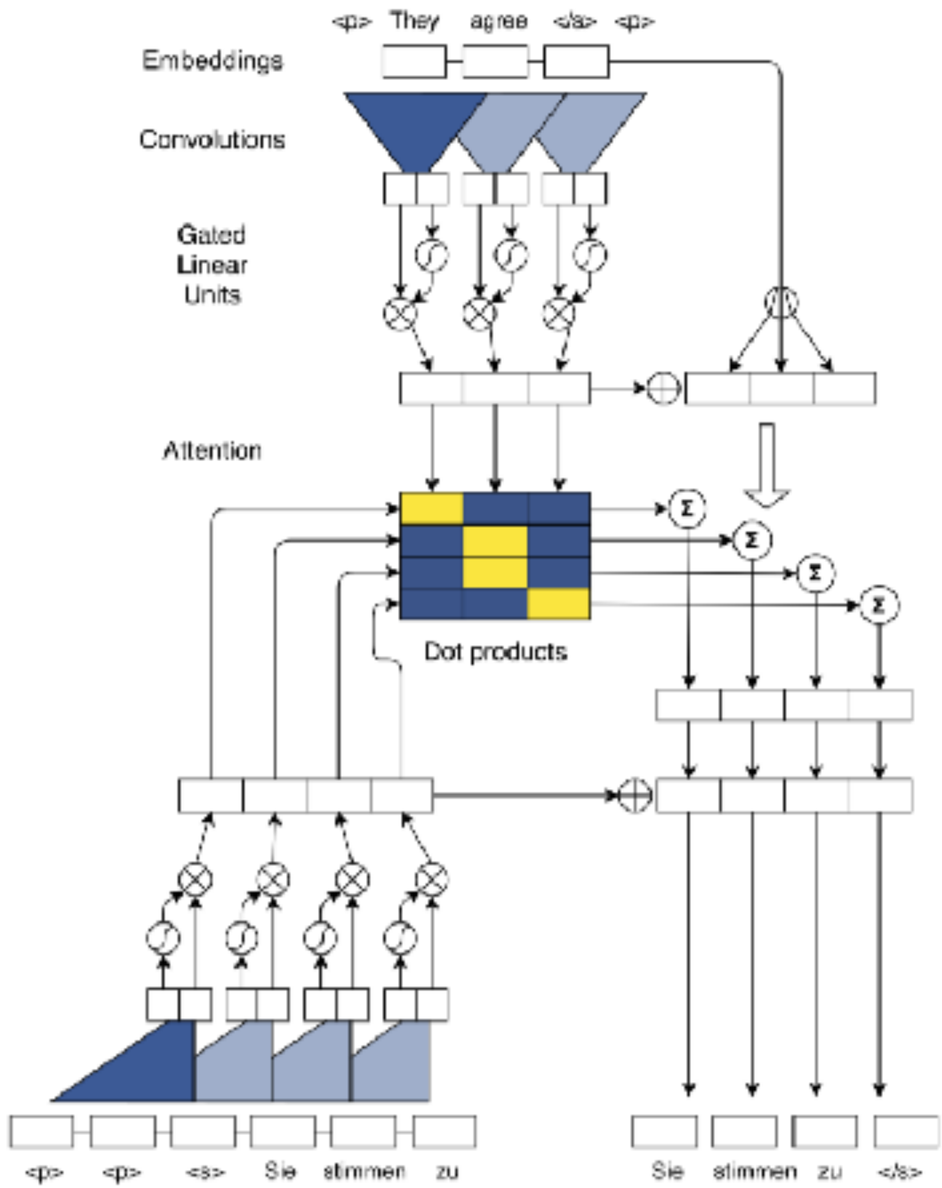
- Google, 2016



Google Neural Machine Translation:
https://arxiv.org/pdf/1609.08144.pdf

CNNs and RNNs

- Can translation also be done using only CNNs?
  - Yes, Facebook AI already did it,
  - 9 times faster than GNMT
  - Similar Performance
  - July, 2017



https://arxiv.org/pdf/1705.03122.pdf

Demo, if time

… from Olivier Grisel

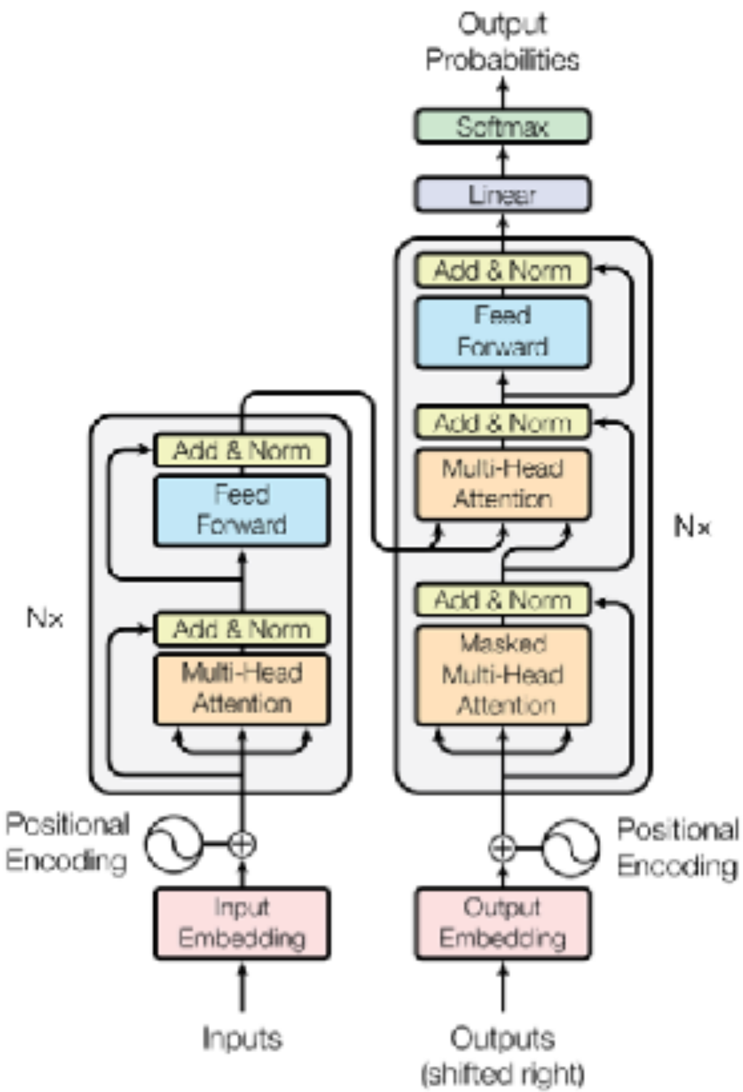
https://github.com/m2dsupsdlclass/lectures-labs/blob/master/labs/07_seq2seq/Translation_of_Numeric_Phrases_with_Seq2Seq_rendered.ipynb

Transformers

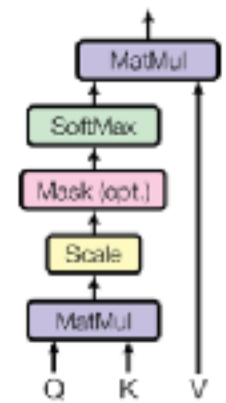
Attention is All You Need

- Well, its a good paper title, but not exactly accurate
- Problem: recurrent networks are not inherently parallelized or efficient at remembering
- Convolution needs many examples from all different word positions (after flattening)
- Filters are not resilient to long-term relationships
- Transformer Solution:
  - Build attention into model from the **beginning**
  - Compare all words to each other through **multi-headed** attention
  - Define a notion of “**position**”in the sentence

Transformer



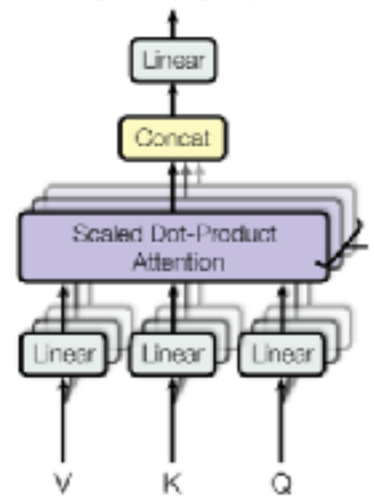
Scaled Dot-Product Attention



for each word

$$\mathrm{Attention}(Q,K,V) = \mathrm{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Multi-Head Attention

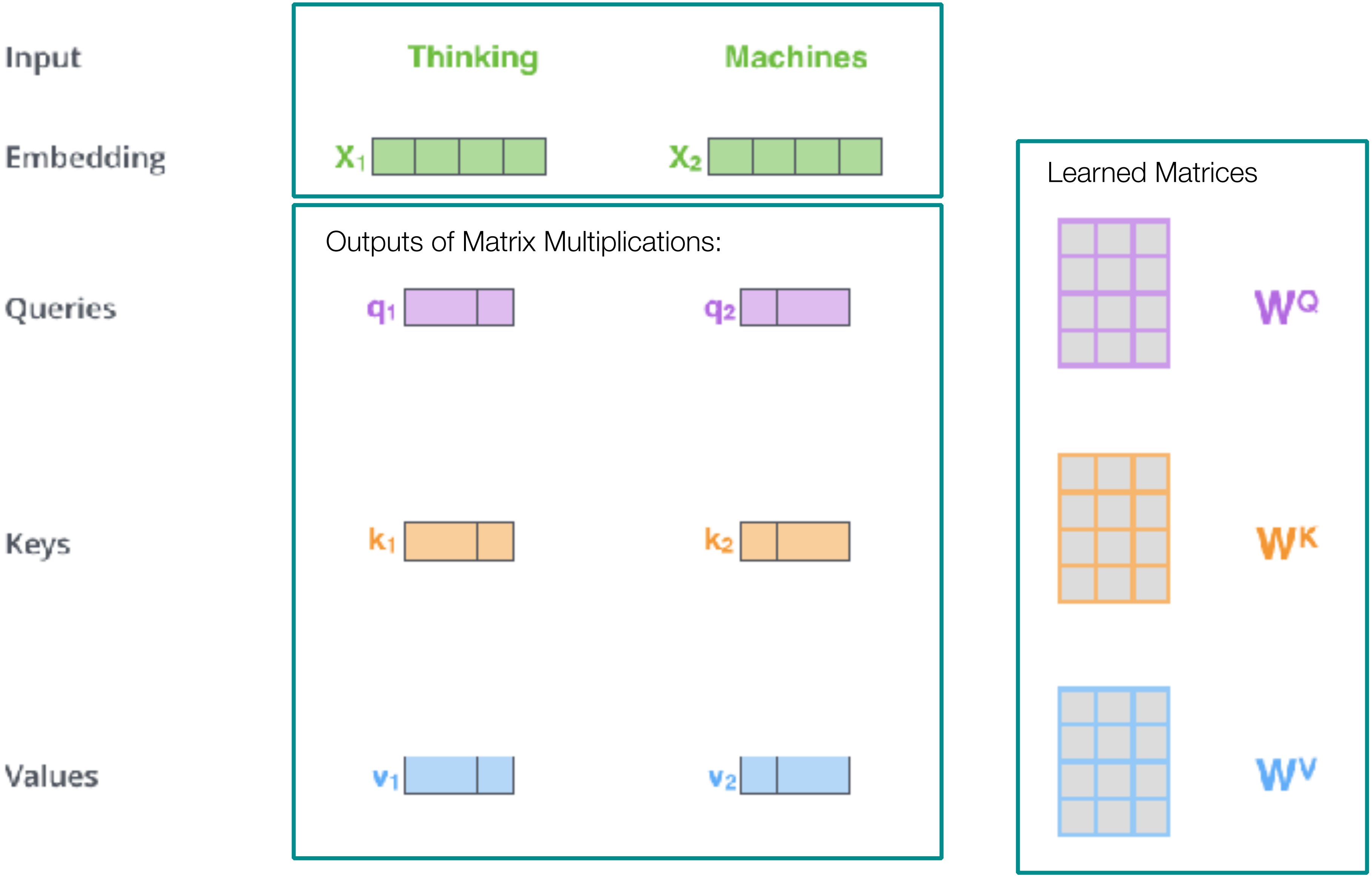


more than one
Q,K,V use in document

$$\mathrm{MultiHead}(Q,K,V) = \mathrm{Concat}(\mathrm{head}_1, ..., \mathrm{head}_h)W^O$$
$$\text{where } \mathrm{head}_i = \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

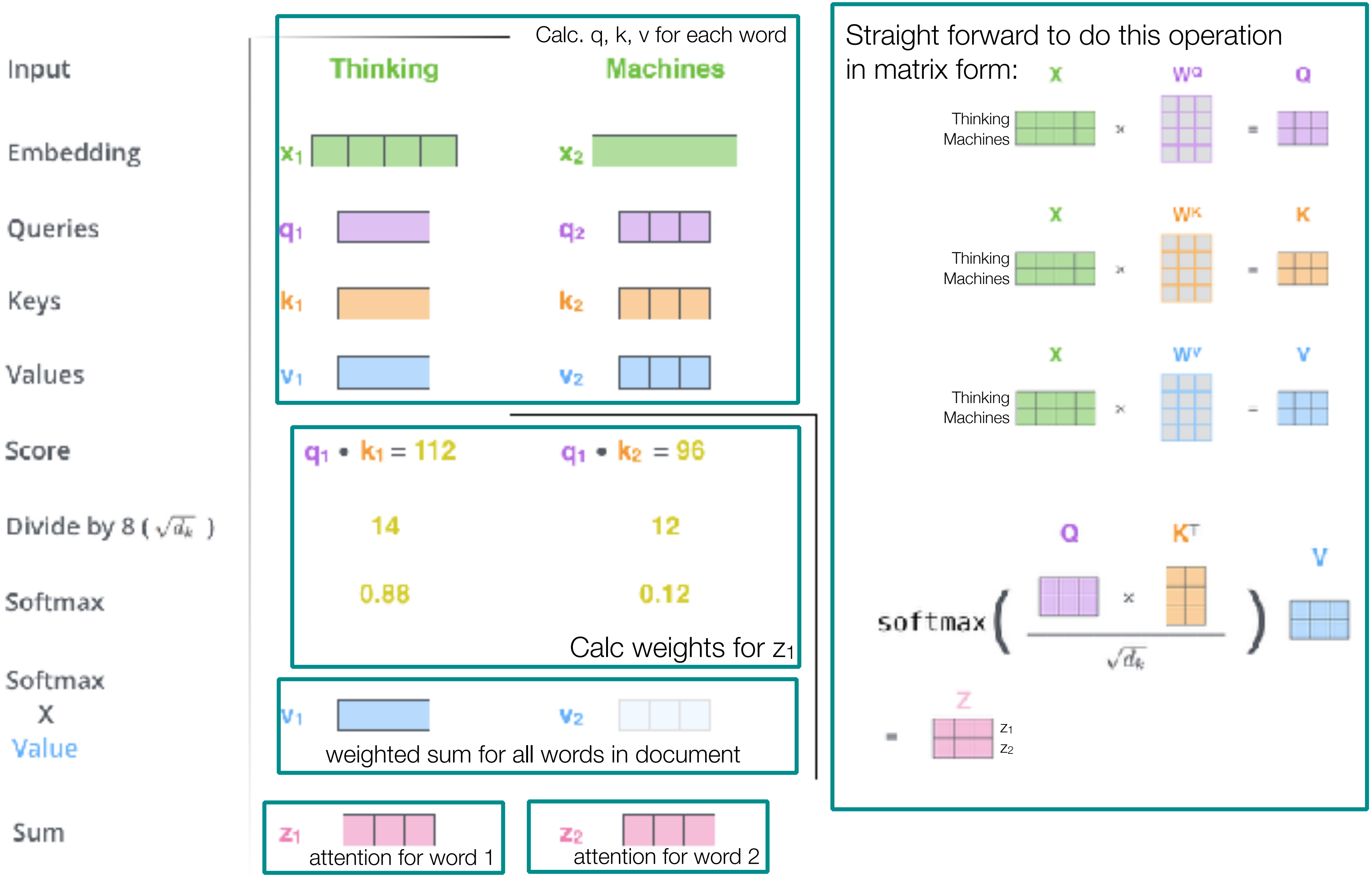
https://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf

Transformer: in more detail



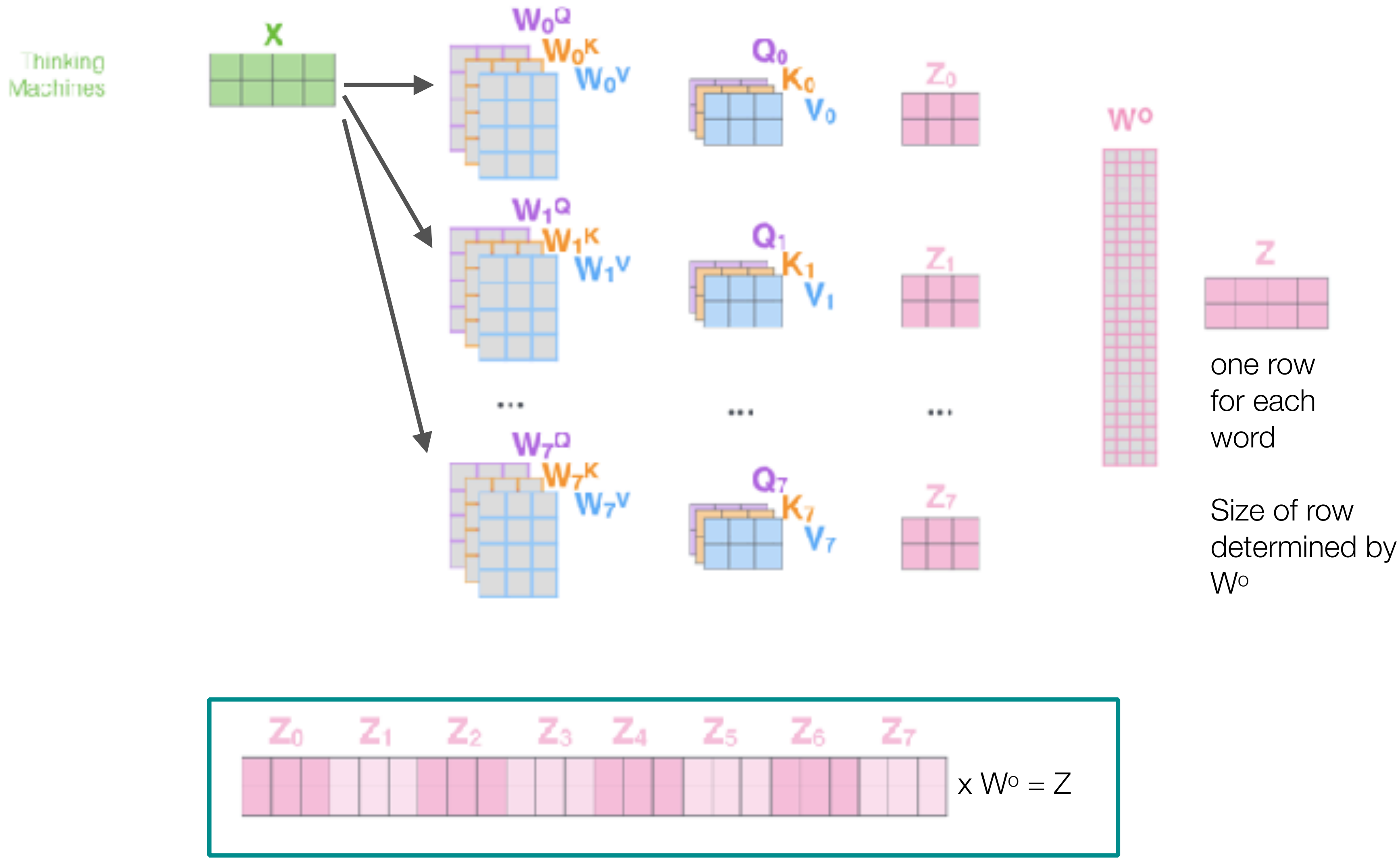
Excellent Blog on Transformers: http://jalammar.github.io/illustrated-transformer/

Transformer: in more detail



Excellent Blog on Transformers: http://jalammar.github.io/illustrated-transformer/

Transformer: Multi-headed Attention

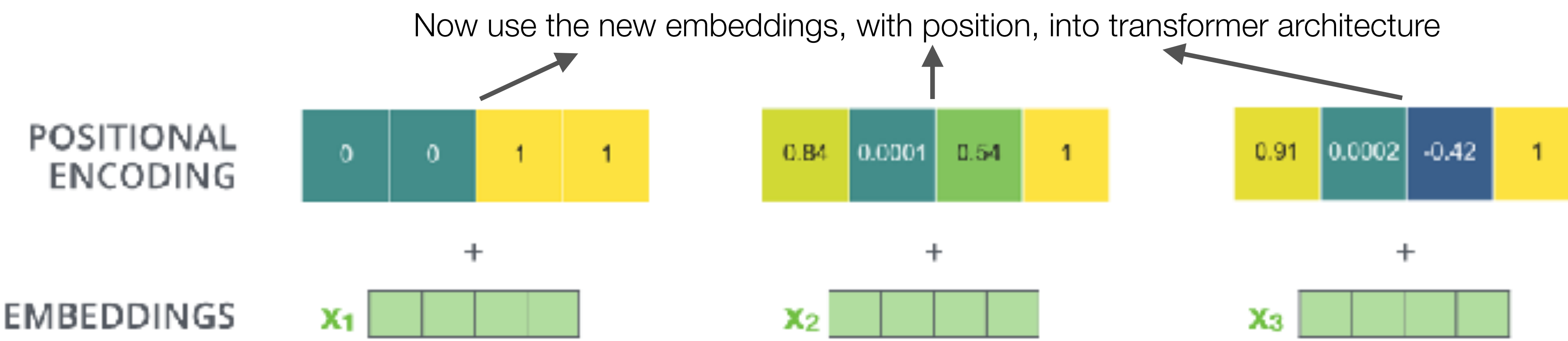


Excellent Blog on Transformers: http://jalammar.github.io/illustrated-transformer/

Transformer: Positional Encoding

- Objective: add notion of position to embedding
- Attempt in paper: add sin/cos to embedding
- But could be anything that encodes position

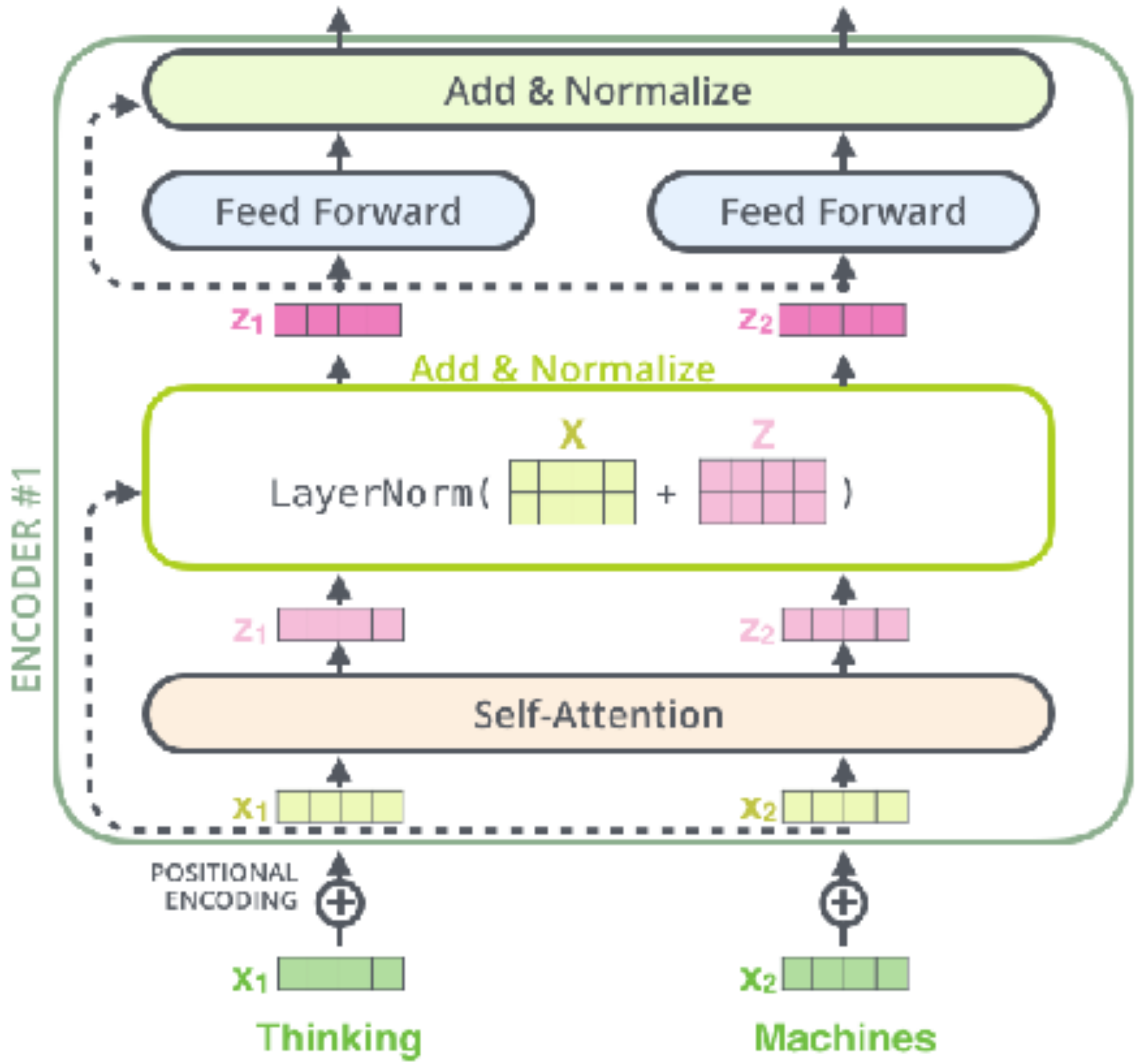
$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$



Hypothesis: Now the word proximity is encoded in the embedding matrix, with other pertinent information. Well, it does help… so it could be true that this is a good way to do it.

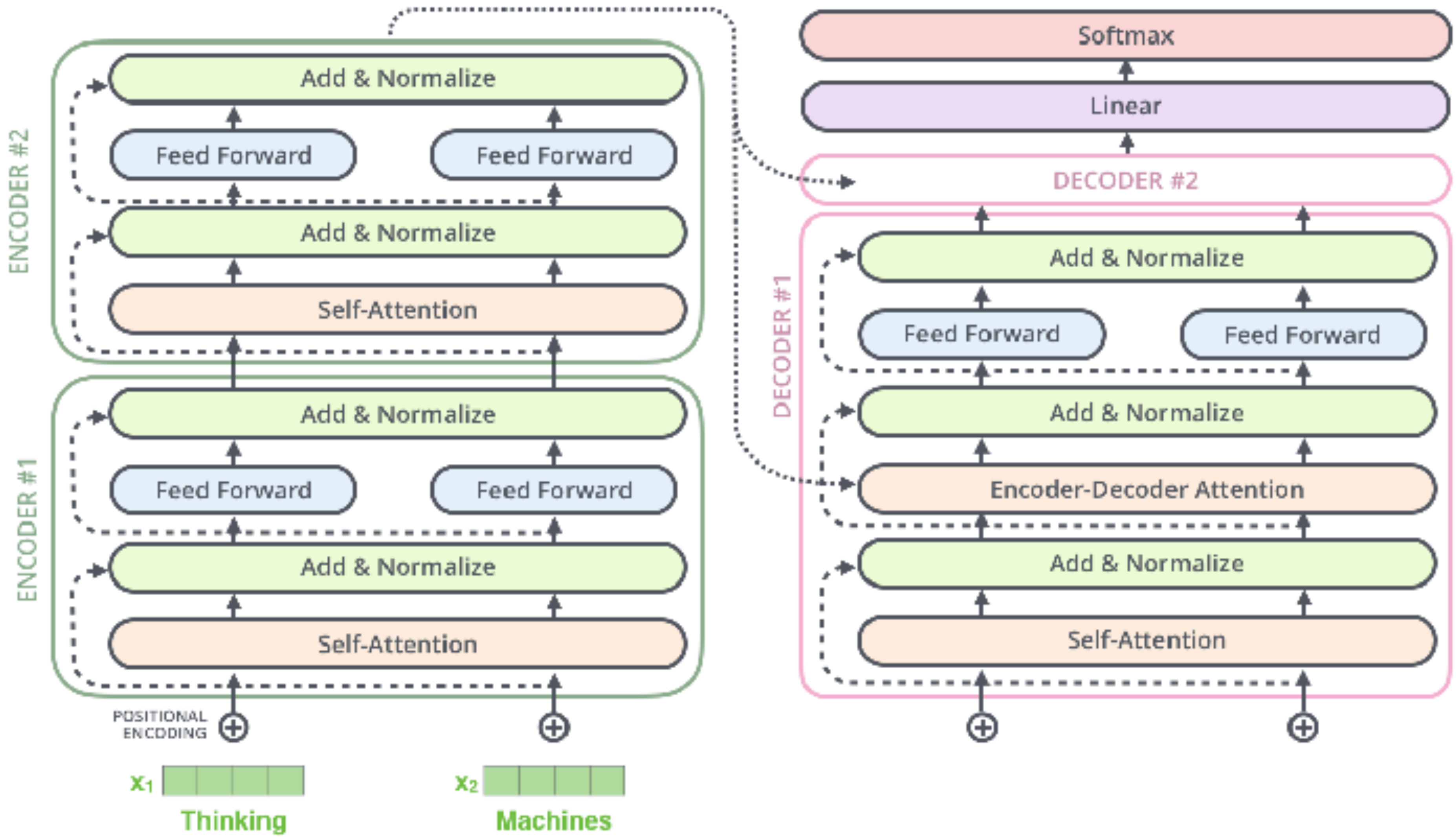
Excellent Blog on Transformers: http://jalammar.github.io/illustrated-transformer/

Transformer: Residual Connections



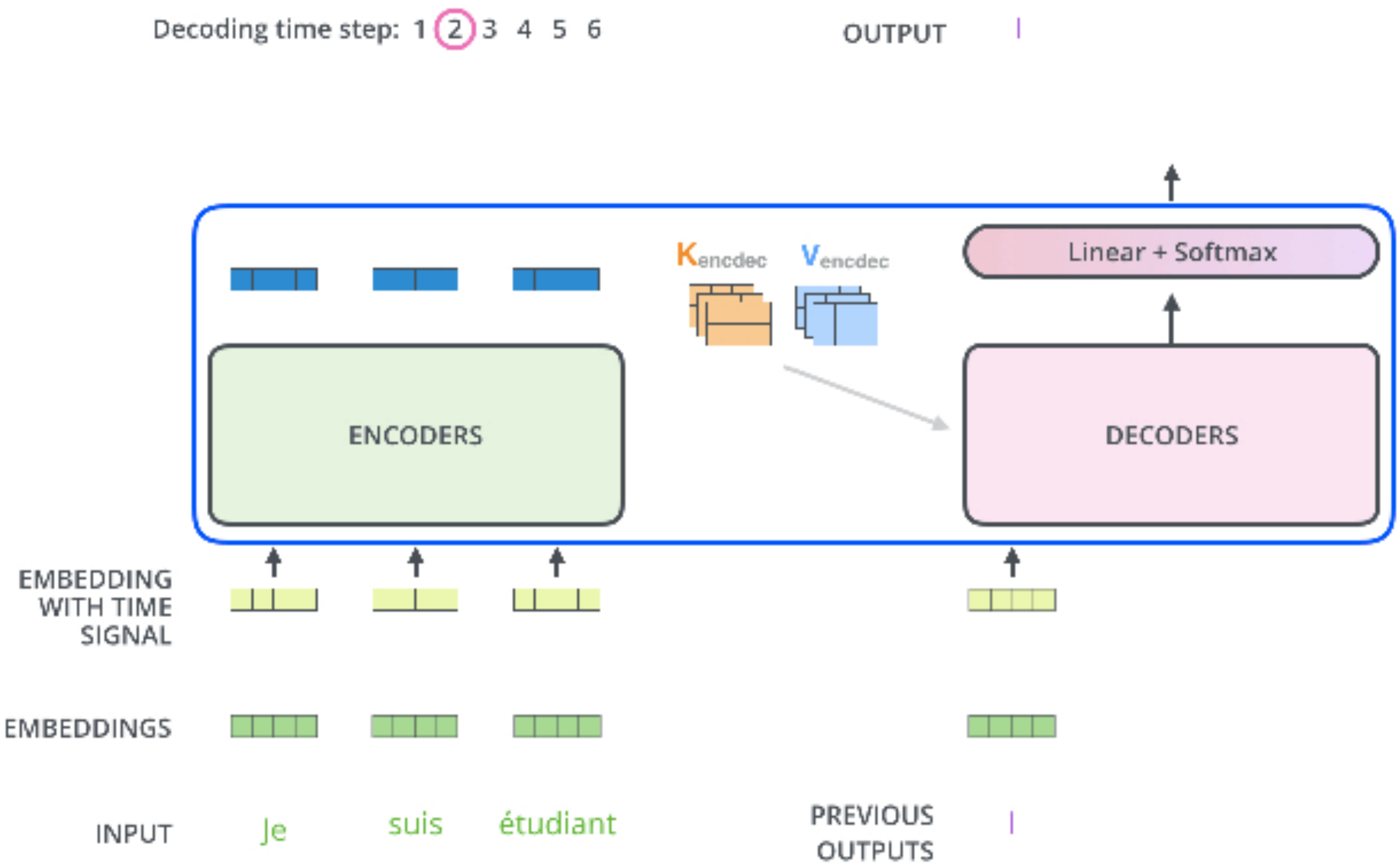
Excellent Blog on Transformers: http://jalammar.github.io/illustrated-transformer/

Transformer: Putting it all together



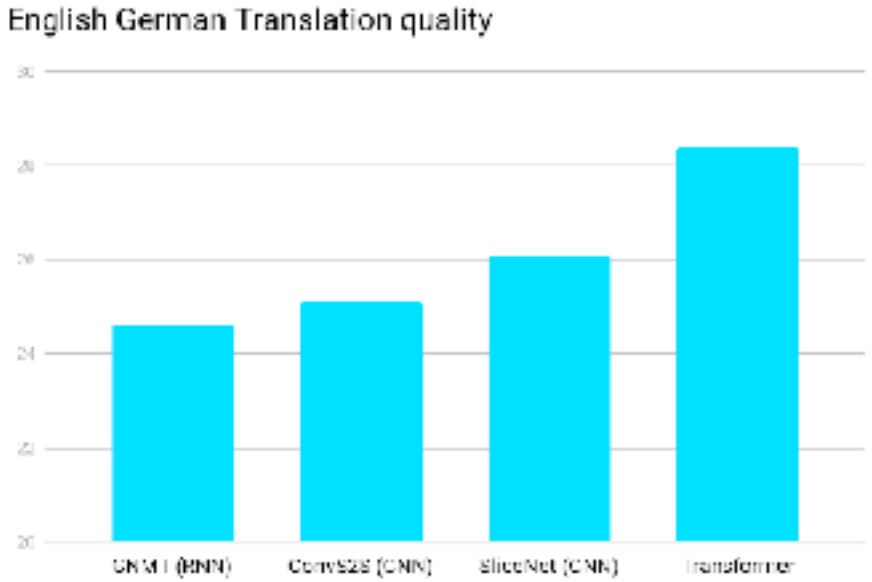
Excellent Blog on Transformers: http://jalammar.github.io/illustrated-transformer/

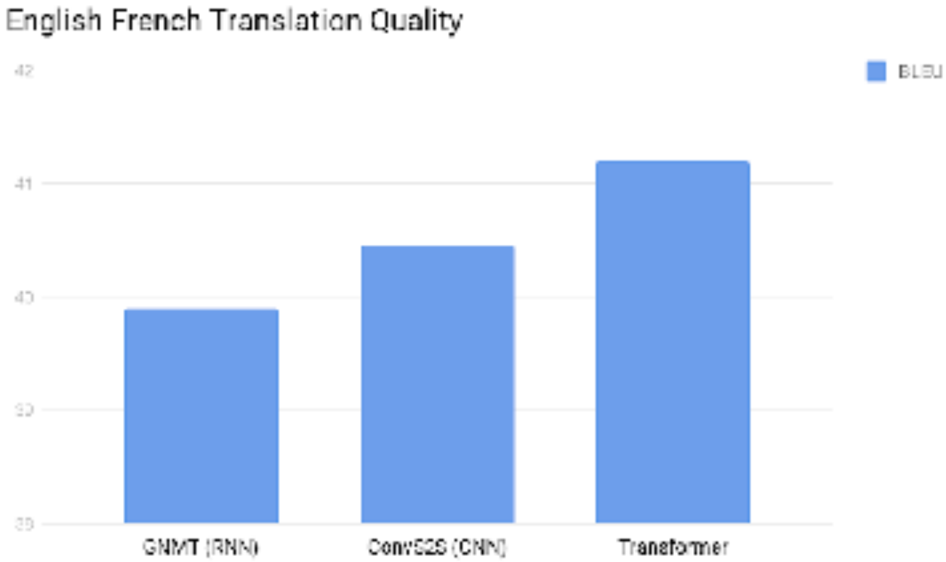
Transformer: Putting it all together



Excellent Blog on Transformers: http://jalammar.github.io/illustrated-transformer/

Results





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

Implementations:

- Not Native to Keras or Tensorflow, but many Open Source Implementations Exist
- Is Native to PyTorch

Next time

- Class Retrospective

TensorFlow

http://r2rt.com/recurrent-neural-networks-in-tensorflow-i.html

```
with tf.variable_scope('rnn_cell'):
    W = tf.get_variable('W', [num_classes + state_size, state_size])
    b = tf.get_variable('b', [state_size], initializer=tf.constant_initializer(0.0))

def rnn_cell(rnn_input, state):
    with tf.variable_scope('rnn_cell', reuse=True):
        W = tf.get_variable('W', [num_classes + state_size, state_size])
        b = tf.get_variable('b', [state_size], initializer=tf.constant_initializer(0.0))
    return tf.tanh(tf.matmul(tf.concat(1, [rnn_input, state]), W) + b)

state = init_state
rnn_outputs = []
for rnn_input in rnn_inputs:
    state = rnn_cell(rnn_input, state)
    rnn_outputs.append(state)
final_state = rnn_outputs[-1]

#logits and predictions
with tf.variable_scope('softmax'):
    W = tf.get_variable('W', [state_size, num_classes])
    b = tf.get_variable('b', [num_classes], initializer=tf.constant_initializer(0.0))
logits = [tf.matmul(rnn_output, W) + b for rnn_output in rnn_outputs]
predictions = [tf.nn.softmax(logit) for logit in logits]

# Turn our y placeholder into a list labels
y_as_list = [tf.squeeze(i, squeeze_dims=[1]) for i in tf.split(1, num_steps, y)]

#losses and train_step
losses = [tf.nn.sparse_softmax_cross_entropy_with_logits(logit,label) for \
          logit, label in zip(logits, y_as_list)]
total_loss = tf.reduce_mean(losses)
train_step = tf.train.AdagradOptimizer(learning_rate).minimize(total_loss)
```

recurrent networks

http://r2rt.com/recurrent-neural-networks-in-tensorflow-i.html

```
def train_network(num_epochs, num_steps, state_size=4, verbose=True):
    with tf.Session() as sess:
        sess.run(tf.initialize_all_variables())
        training_losses = []
        for idx, epoch in enumerate(gen_epochs(num_epochs, num_steps)):
            training_loss = 0
            training_state = np.zeros((batch_size, state_size))
            if verbose:
                print("\nEPOCH", idx)
            for step, (X, Y) in enumerate(epoch):
                tr_losses, training_loss_, training_state, _ = \
                    sess.run([losses,
                              total_loss,
                              final_state,
                              train_step],
                                  feed_dict={x:X, y:Y, init_state:training_state})
                training_loss += training_loss_
                if step % 100 == 0 and step > 0:
                    if verbose:
                        print("Average loss at step", step,
                              "for last 250 steps:", training_loss/100)
                    training_losses.append(training_loss/100)
                    training_loss = 0

    return training_losses
```

TensorFlow

http://r2rt.com/recurrent-neural-networks-in-tensorflow-i.html

```
def train_network(num_epochs, num_steps, state_size=4, verbose=True):
    with tf.Session() as sess:
        sess.run(tf.initialize_all_variables())
        for idx, epoch in enumerate(gen_epochs(num_epochs, num_steps)):
            training_state = np.zeros((batch_size, state_size))
            for X, Y in epoch:
                tr_losses, training_loss_, training_state, _ = \
                    sess.run([losses,
                              total_loss,
                              final_state,
                              train_step],
                             feed_dict={x:X, y:Y, init_state:training_state})
```

TensorFlow (simplified)

http://r2rt.com/recurrent-neural-networks-in-tensorflow-i.html

```
cell = tf.nn.rnn_cell.BasicRNNCell(state_size)
rnn_outputs, final_state = tf.nn.rnn(cell, rnn_inputs, initial_state=init_state)
```

```
loss_weights = [tf.ones([batch_size]) for i in range(num_steps)]
losses = tf.nn.seq2seq.sequence_loss_by_example(logits, y_as_list, loss_weights)
```

```
x = tf.placeholder(tf.int32, [batch_size, num_steps], name='input_placeholder')
y = tf.placeholder(tf.int32, [batch_size, num_steps], name='labels_placeholder')
init_state = tf.zeros([batch_size, state_size])

x_one_hot = tf.one_hot(x, num_classes)
rnn_inputs = tf.unpack(x_one_hot, axis=1)

cell = tf.nn.rnn_cell.BasicRNNCell(state_size)
rnn_outputs, final_state = tf.nn.rnn(cell, rnn_inputs, initial_state=init_state)
```

```
with tf.variable_scope('softmax'):
    W = tf.get_variable('W', [state_size, num_classes])
    b = tf.get_variable('b', [num_classes], initializer=tf.constant_initializer(0.0))
logits = [tf.matmul(rnn_output, W) + b for rnn_output in rnn_outputs]
predictions = [tf.nn.softmax(logit) for logit in logits]

y_as_list = [tf.squeeze(i, squeeze_dims=[1]) for i in tf.split(1, num_steps, y)]

loss_weights = [tf.ones([batch_size]) for i in range(num_steps)]
losses = tf.nn.seq2seq.sequence_loss_by_example(logits, y_as_list, loss_weights)
total_loss = tf.reduce_mean(losses)
train_step = tf.train.AdagradOptimizer(learning_rate).minimize(total_loss)
```