AI - Lab

6. NLP

Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Prof. Dr. Patrick Baier

WS23/24

Overview

Natural Language Processing (NLP) allows machines to break down and interpret human language. It's at the core of tools we use every day – from translation software, chatbots, spam filters, and search engines, to grammar correction software, voice assistants, and social media monitoring tools.

Source: hhttps://monkeylearn.com/natural-language-processing/

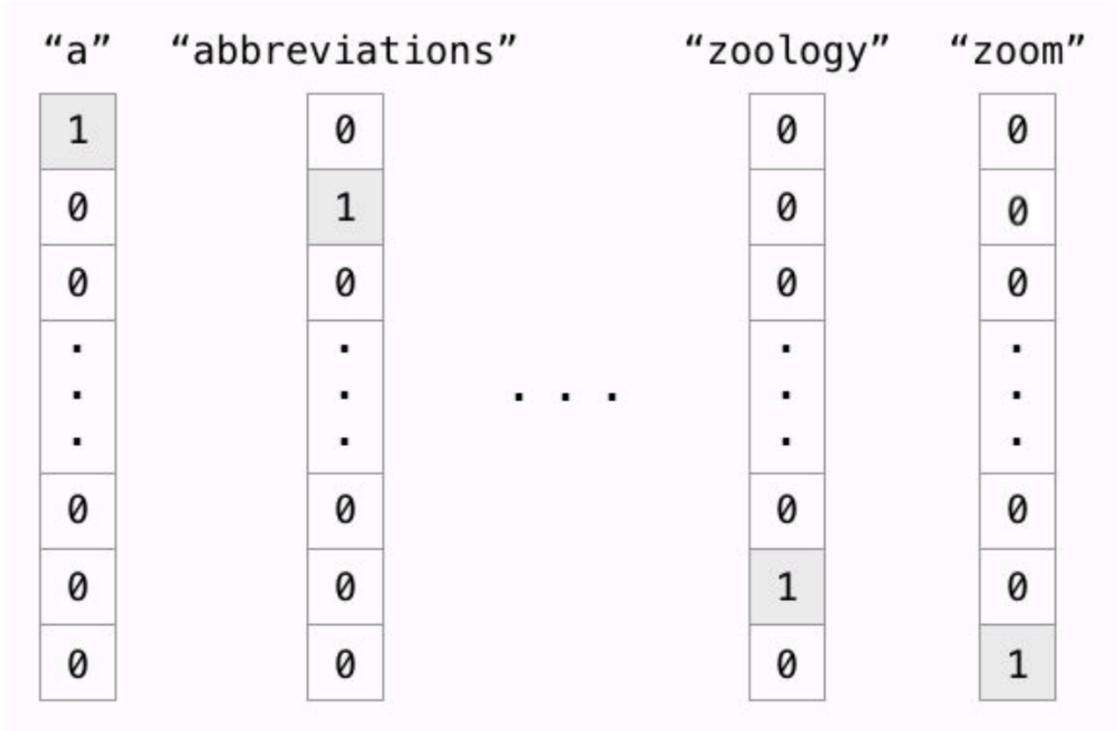
- Common Tasks: Text Classification, Part-of-Speech tagging, Named-entity-recognition, Word embeddings
- Applications: Chatbots, Language Models, Text Summarization, Language Translation, Audio Transcription, etc.

Text Classification

- Easiest NLP task that allows us to study some basic NLP principles.
- Input: A text snippet.
- Output: What category is the text from?
- Example: TweetEval
  „TweetEval consists of seven heterogenous tasks in Twitter, all framed as multi-class tweet classification.“
  - First task: Emotion Recognition, 4 labels: anger, joy,sadness, optimism
  - Example:
    Tweet: „My roommate: it's okay that we can't spell because we have autocorrect. #terrible #firstworldprobs“
    Label: Anger

One-Hot Encoding

- How do we input text into a neural network?
- One solution: One hot encoding of all available words



Word Embeddings

dog	-0.4	0.37	0.02	-0.34
cat	-0.15	-0.02	-0.23	-0.23

- Better solution: Encode words with embeddings
- Word embeddings present words with fixed size vectors in which words with similar meaning have a similar representation in the vector space.
- There are used as input of NLP tasks instead of one-hot-encodings since they improve performance of a lot of NLP tasks.
- There are many forms of pre-trained word embeddings (Word2Vec, GloVe, Bert, …) that you will learn about in the lecture.
- We can also train the a new word embedding as part of a downstream task (e.g. text classification). See next slide.

Embedding Layer

dog	-0.4	0.37	0.02	-0.34
cat	-0.15	-0.02	-0.23	-0.23

- PyTorch has an embedding layer which we can use to learned embeddings during training (no pre-trained embedding).

```
self.word_embeddings = nn.Embedding(vocab_size, embedding_dim)
```

Number of words in the vocabulary

Dimension of the embedding

- The embedding layer accepts a list of integers (each integer represents a word) as input and converts them to an embedding.

```
def forward(self, inputs):
    embeds = self.embeddings(inputs).view((1, -1))
    out = F.relu(self.linear1(embeds))
```

Embedding Layer

dog	-0.4	0.37	0.02	-0.34
cat	-0.15	-0.02	-0.23	-0.23

- Hence, all words in the vocabulary mused be matched to an unique integer:

```
word_to_ix = {}
for sent, tags in training_data:
    for word in sent:
        if word not in word_to_ix:
            word_to_ix[word] = len(word_to_ix)
```

- A given sequence of text, must then be converted to their corresponding integer before it is fed in to the model.

```
def prepare_sequence(seq, to_ix):
    idxs = [to_ix[w] for w in seq]
    return torch.tensor(idxs, dtype=torch.long)
```

- See tutorial Word Embeddings in PyTorch for more details.

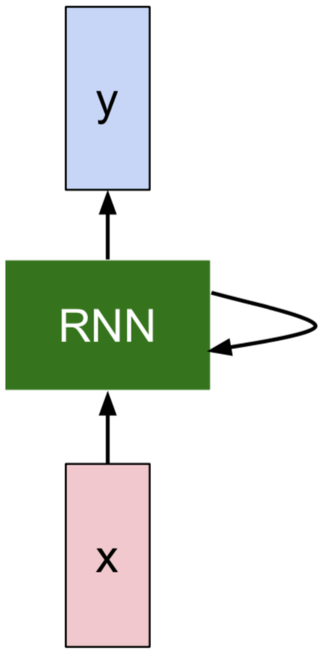
Sequential Networks

- If we have sequential input (like a sentence) a normal neural network is not suited.
  - The input size varies from sentence to sentence (a neural network has fixed number of input neurons).
  - There is no temporal correlation between input neurons in a neural network.
- The solution to this are sequential networks like RNNs, LSTMs or GRU.
- Basic principle of all of them:
  - Network predicts in several time steps.
  - In every time step, a new word is given as input to the network.
  - Network has a memory to transfer knowledge between time stamps.

Image Source: CS231

Recurrent Neural Networks

Basic working principle of an RNN:



Every time step i:

1. Read input x_i.
2. Update internal RNN state, based on input and last state.
3. Calculate output y_i.

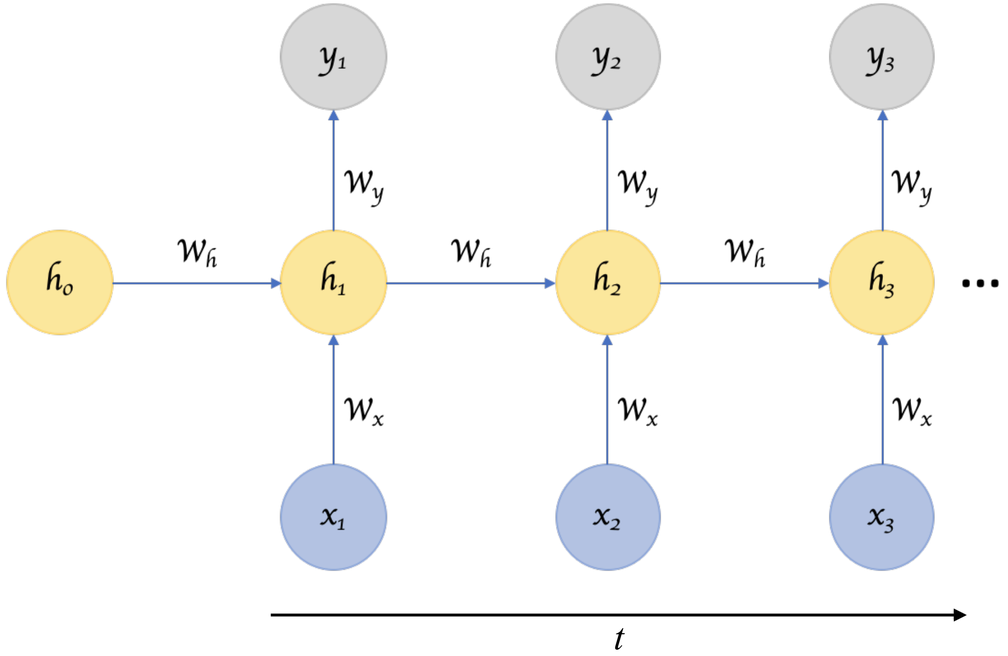
Example: Text Classification

- Input x_i is word embedding of word i in the sentence.
- Output y_n is the label for the sentence.
- There are n time steps, while n is the number of words in the sentence.

Image Source: CS231

Recurrent Neural Networks

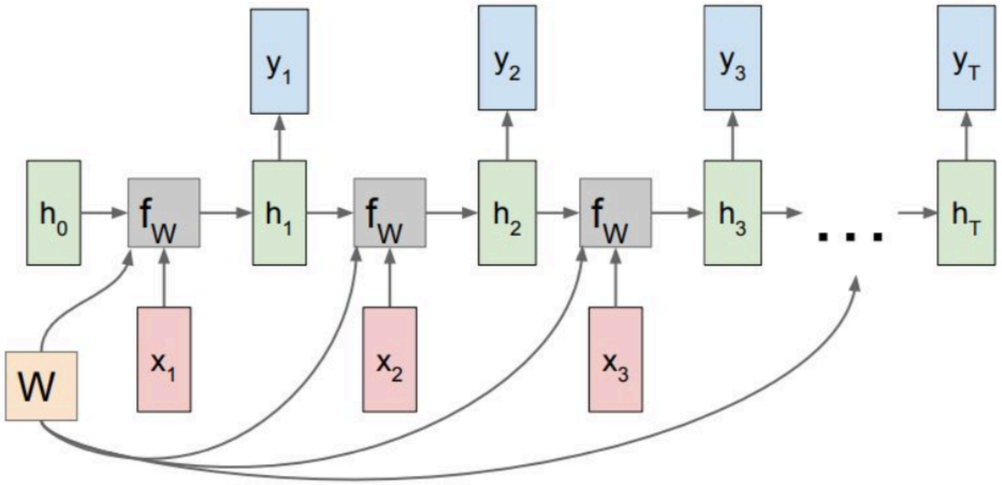
Un-rolled view for several time steps:



- RNNs use a hidden state that carries historical information about previous inputs.
- The hidden state h_t at time t is computed from the previous hidden state h_{t-1} and the current input x_t.
- First hidden state h_0 is not calculated but given. If no prior information is given, this is usually set to zero.

Image: https://towardsdatascience.com/recurrent-neural-networks-d4642c9bc7ce

Recurrent Neural Networks



$$h_t = f_W(h_{t-1}, x_t)$$

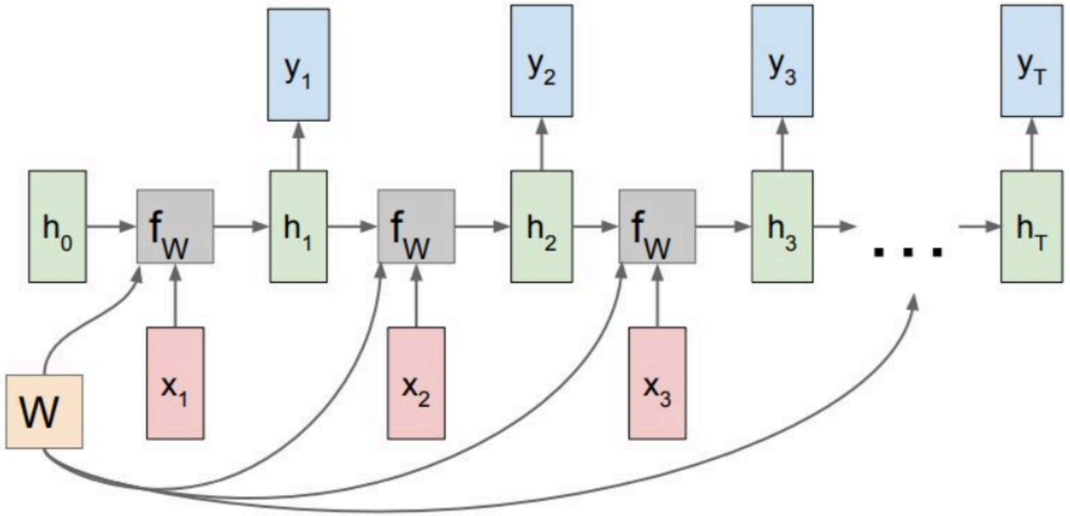
new state — some function with parameters W — old state — input vector at some time step

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

$$y_t = W_{hy} h_t$$

Image Source: CS231

Recurrent Neural Networks

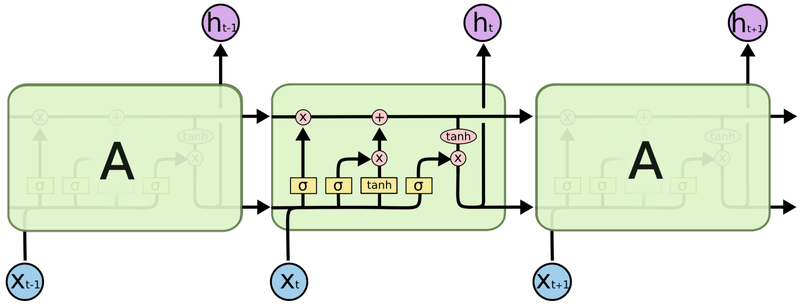


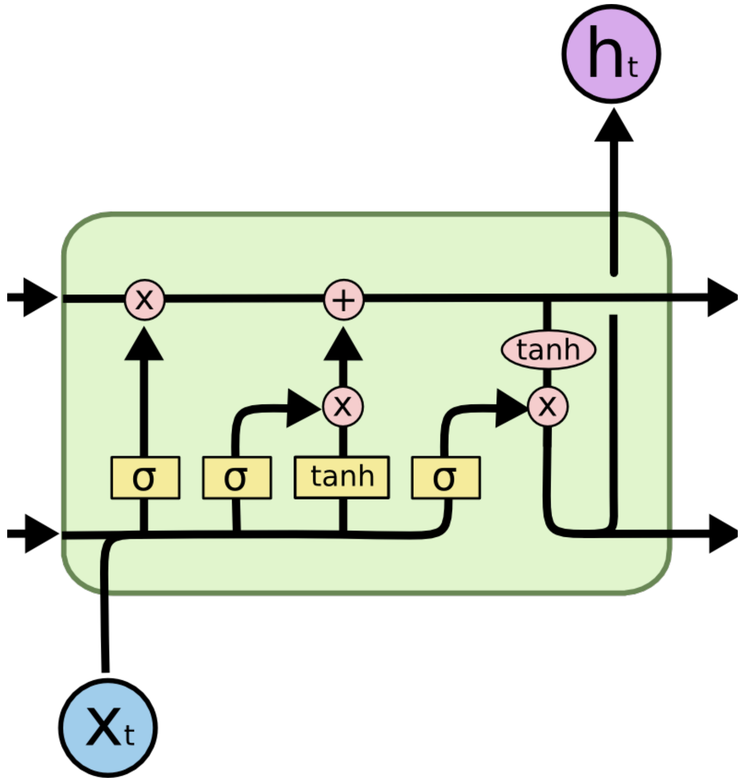
- Important: The <u>same</u> weights W_{hh}, W_{xh} and W_{hy} are used at every time step.
- More details (especially the training of RNNs) will be covered in the AI lecture.

Image Source: CS231

LSTM - Long Short-Term Memory

- Paper: “Long Short-Term Memory”, Hochreiter & Schmidhuber 1997.
- LSTM is a modification of the RNN architecture that helps to keep long sequence relationships.
- Main idea: Instead of one hidden state there is a “information highway” with unrestricted gradient flow.
- More information of how an LSTM works in the AI lecture.
- We use it for now as a black box that does the same as an RNN (but much better).





LSTMs in PyTorch

- LSTMs have their own class in PyTorch. (A good tutorial is here).
- There are two ways to feed an LSTM:

  1. Step-wise: Feed individual time steps to LSTM and carry on hidden state:

```
lstm = nn.LSTM(3, 3)  # Input dim is 3, output dim is 3
inputs = [torch.randn(1, 3) for _ in range(5)]  # make a sequence of length 5

# initialize the hidden state.
hidden = (torch.randn(1, 1, 3),
          torch.randn(1, 1, 3))
for i in inputs:
    # Step through the sequence one element at a time.
    # after each step, hidden contains the hidden state.
    out, hidden = lstm(i.view(1, 1, -1), hidden)
```

LSTMs in PyTorch

- LSTMs have their own class in PyTorch. (A good tutorial is here).
- There are two ways to feed an LSTM:

2. Sequence-at-once: Feed whole sequence to LSTM:

```
lstm = nn.LSTM(3, 3)  # Input dim is 3, output dim is 3
inputs = [torch.randn(1, 3) for _ in range(5)]  # make a sequence of length 5
```

```
inputs = torch.cat(inputs).view(len(inputs), 1, -1)
hidden = (torch.randn(1, 1, 3), torch.randn(1, 1, 3))  # clean out hidden state
out, hidden = lstm(inputs, hidden)
```

`out`: all of the hidden states throughout the sequence.
`hidden`: most recent hidden state.

LSTMs in PyTorch

- Typical structure:
  - Embedding layer: Converts categorical values (such as words) into embeddings.
  - LSTM: Takes embedding as input and outputs hidden state.
  - FC-layer: Takes hidden state as input and outputs class values.

```
self.word_embeddings = nn.Embedding(vocab_size, embedding_dim)

# The LSTM takes word embeddings as inputs, and outputs hidden states
# with dimensionality hidden_dim.
self.lstm = nn.LSTM(embedding_dim, hidden_dim)

# The linear layer that maps from hidden state space to tag space
self.hidden2tag = nn.Linear(hidden_dim, tagset_size)
```

Tensor Dimensions

The tensor that goes into an LSTM has three dimensions:

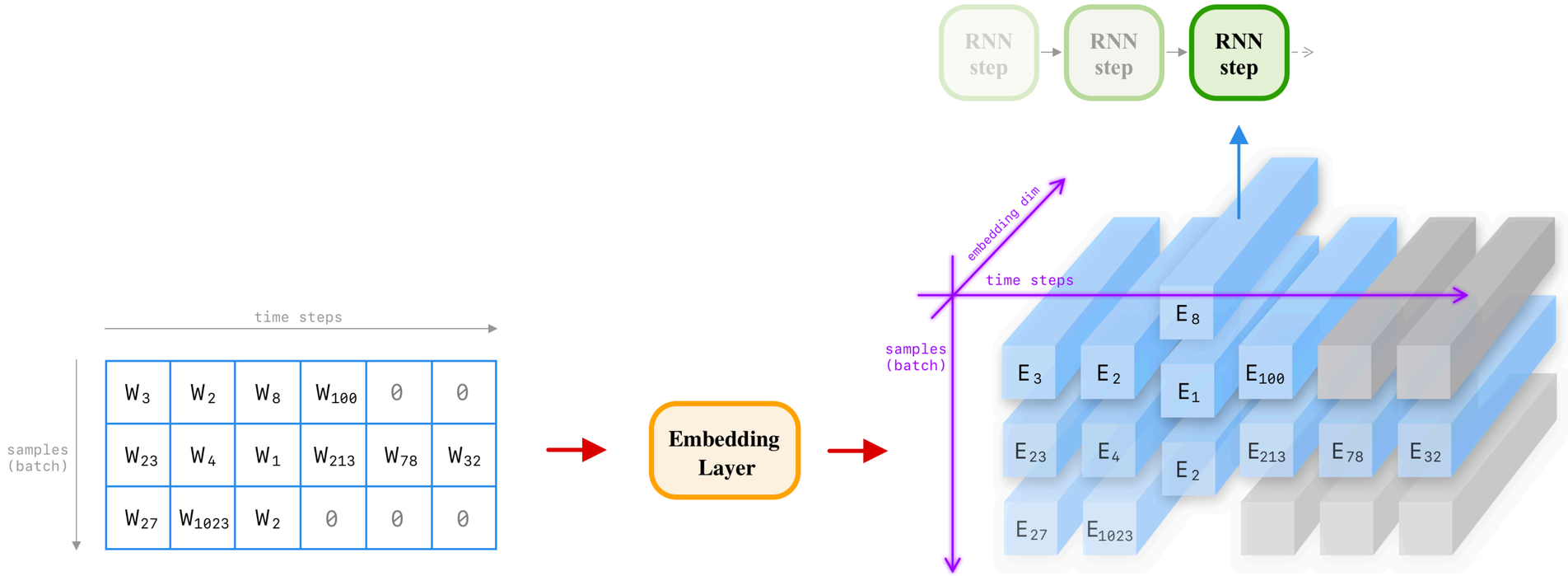


Image: https://iust-deep-learning.github.io/972/static_files/assignments/assignment_05_preview.html

GRU

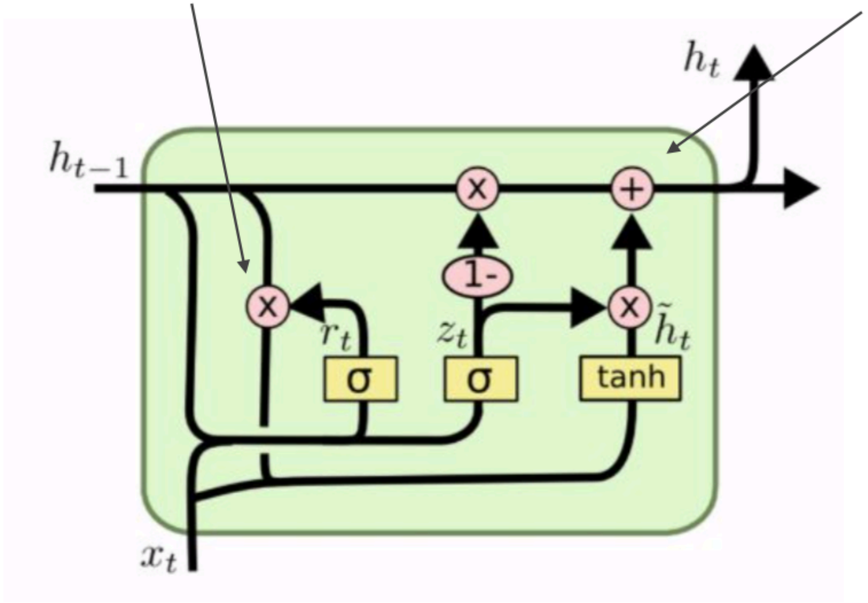
- Paper: “Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation”, Cho et. al., 2014.
- GRU is a newer architecture that is a simplification of LSTM.
- Adds two gates: “update” and “reset” with trainable parameters.
- Let the network learn when to open and close those gates, i.e. update the “highway” with new information.
- Therefore, instead of updating the hidden state at every RNN cell, the network can learn itself when to update the hidden state with new information.

GRU - Cell Diagram

$$
\begin{aligned}
z_t &= \sigma_g(W_z x_t + U_z h_{t-1} + b_z) \\
r_t &= \sigma_g(W_r x_t + U_r h_{t-1} + b_r) \\
h_t &= (1 - z_t) \circ h_{t-1} + z_t \circ \sigma_h(W_h x_t + U_h(r_t \circ h_{t-1}) + b_h)
\end{aligned}
$$

Which states do we want to forget

Which information do we want to pass on



Variables

- x_t: input vector
- h_t: output vector
- z_t: update gate vector
- r_t: reset gate vector
- W, U and b: parameter matrices and vector

Activation functions

- σ_g: The original is a sigmoid function.
- σ_h: The original is a hyperbolic tangent.

- Combines forget and input gates into a single update gate.
- Merges the cell state and hidden state.

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

Comparison LSTM vs. GRU

- Vanilla RNNs are almost never used because of the issues we discussed.
- The choice is between GRUs and LSTMs.
- As you saw, LSTMs are almost 20 years older than GRUs.
- GRU is clearly a simplification of LSTM.
- GRU is computationally easier.
- LSTM is more powerful.

Assignment 6

Train a text classification on the TweetEval emotion recognition dataset using LSTMs and GRUs.

- Follow the example described here. Use the same architecture, but:
  - only use the last output of the LSTM in the loss function
  - use an embedding dim of 128
  - use a hidden dim of 256.
- Use spaCy to split the tweets into words.
- Limit your vocabulary (i.e. the words that you converted to an index) to the most frequent 5000 words and replace all other words with an placeholder index (e.g. 1001).
- Evaluate the accuracy on the test set.
- Note: If the training takes to long, try to use only a fraction of the training data.
- Do the same, but this time use GRUs instead of LSTMs.

Assignment 6: Bonus

- The training is quite slow, since we do not train in batches but line by line. Providing the text lines in batches to the LSTM, will not work out of the box, because all tensors in one batch need to have the same length.
- The `DataLoader` class can be created with a `collate_fn` function that can be used to bring all lines in one batch to the same size:
  - Determine the longest tensor in the current batch.
  - Fill up all tensors that are shorter with an unused padding value (e.g. -1).
- Task: Put the data into a DataLoader, pad the lines in one batch to the same length and train in batches on the GPU. Compare the training time to the line-wise training.

Submission Guidelines

- Send your solutions as notebook file (.ipynb) to me.
- Use the following e-mail subject: [KI-LabWS23] Assignment 6 - <teamname>
- Deadline is next Thursday 23:59.

Final words

- You can start now working on the assignment.
- It is up to you and your partner how you organize your time working on this.
- I will be around until the end of the slot at 1pm for questions.
- Of course, questions can also be asked all time in the chat (but please do not post solutions!).

- Any more questions now?
- Happy Coding!