Neural Machine Translation in Linear Time

Simon Roth 11907576, Sebastian Moscicki 00968493, Nadin Weichenrieder 12230488

Introduction

- Intro to the Paper
- ByteNet Model
  - Masked, dilated Convolution
  - Residual Blocks (ReLU + MU)
  - ByteNet Structure
  - Stacking of Encoder/Decoder
  - Dynamic Unfolding
  - Embedding
- Experiment Reproduction
  - Character Level Translation
  - Character Prediction
- Conclusion

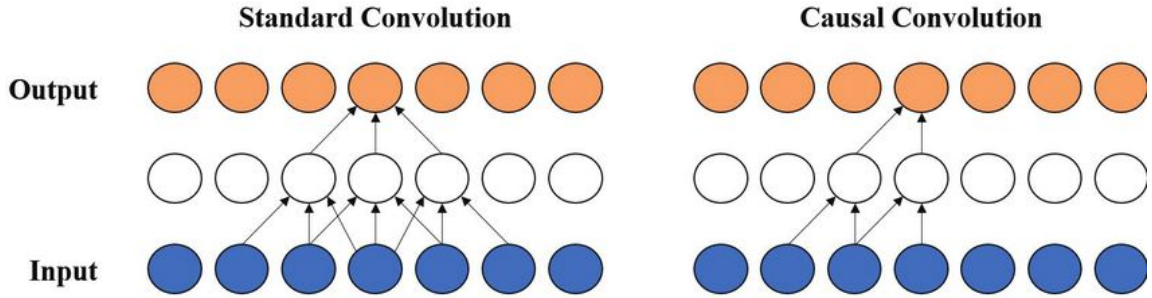
Neural Machine Translation in Linear Time

- Authors: Kalchbrenner et al., released March 2017 (before Transformers)
- Propose ByteNet
  - CNN Encoder/Decoder Architecture
  - Runs in Linear Time
- ByteNet achieved State of the Art translation (BLEU) scores on release
  - Beats Attention based RNNs
  - Less expensive to train
- Core Components
  - Residual Blocks
  - Dilation, Causal Convolution
  - Unfolding

ByteNet

Masked (Causal) Convolution

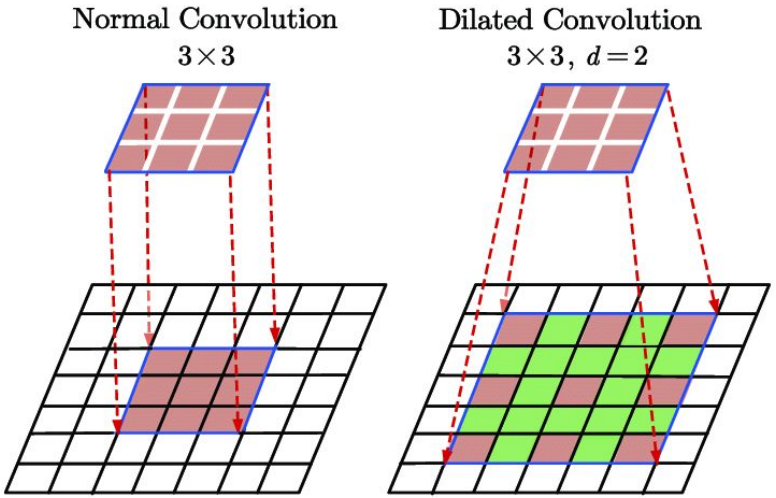
- One sided Padding for the input
  - All Padding on the left side of the input
- Disregard future elements of sequence
  - Used in the Decoder part of the ByteNet
- Padding size dependent on Kernel Size k and Dilation d
  - Compute as (k-1) * d



source: https://www.researchgate.net/figure/3-Comparison-of-a-standard-1D-convolution-and-a-causal-convolution_fig3_365138478

Dilated Convolution

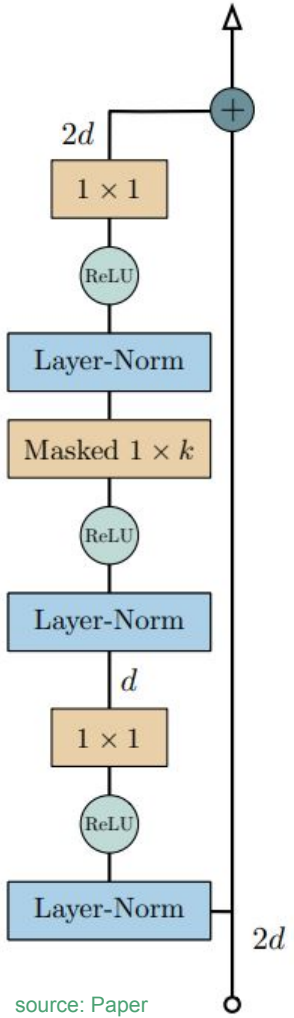
- “Leave out” elements during convolution
- Allows to quickly grow the receptive Field
  - Growth with doubling Dilation is exponential in network depth



source: https://www.researchgate.net/figure/The-illustration-of-the-2D-dilated-convolution-with-spatial-size-of-33-and-dilation_fig1_338780378

Residual Block - ReLU

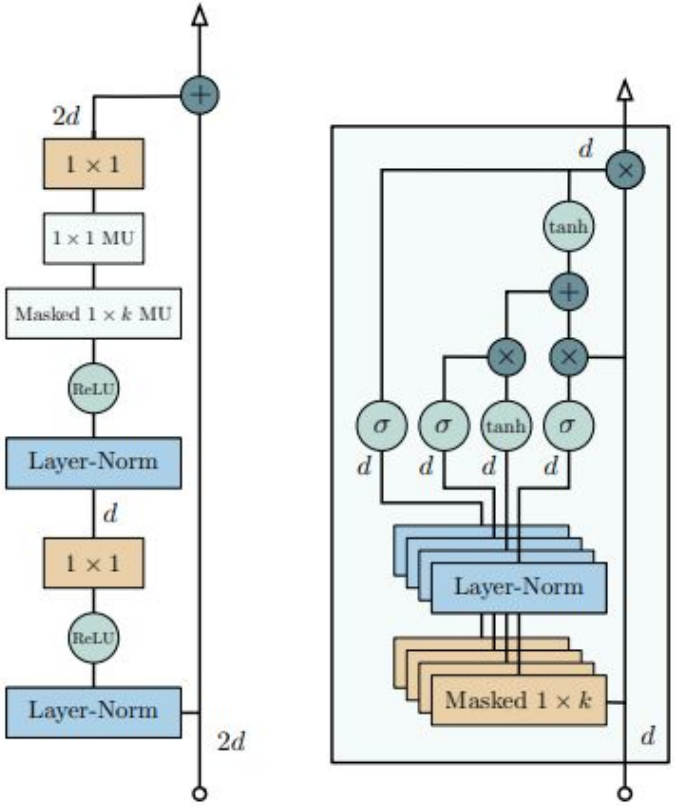
- Main component of Encoder/Decoder
  - x (Paper: 6) Sets of y (Paper: 5) Residual Blocks make up inner part
- Layers
  - Layer Normalization
  - ReLU
  - Conv1D
- Convolution Masked (Causal) in Decoder
  - In Encoder unmasked, but still padded
- Dilation doubles in every layer of each residual block
  - Start value is dilation = 1
  - up to a max value r (Paper: 16)
- Masked/unmasked Kernel Size in the Paper set to 3



source: Paper

Residual Block - MU

- Used for Character Prediction Task
- Layers
  - Input Layer Norm & Convolutions
  - Masked & unmasked Multiplicative Units
  - Output Convolution
- Masking same as in the ReLU block
- Dilation doubles in every layer of each block
  - Start value is dilation = 1
  - up to a max value r (Paper: 16)
- Masked/unmasked Kernel Size set to 3



source: Paper

ByteNet Structure

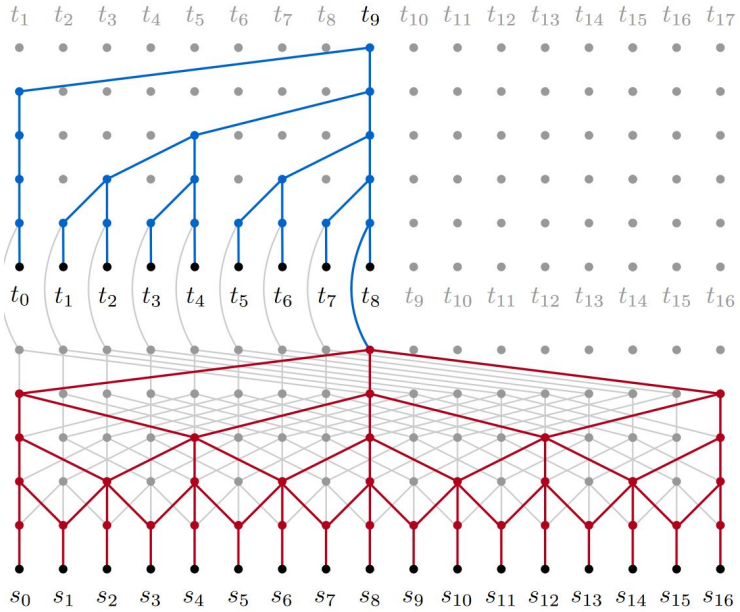
- Encoder
  - Embedding
  - x sets of y Residual Blocks
- Decoder
  - x sets of y Residual Blocks (with masked convolution)
  - Output convolution, ReLU (Dropout for Character Prediction) and Softmax

Encoder-Decoder Stacking

Encoder and Decoder are connected by stacking decoder on top of encoder

- **Encoder** processes input sequence and generates a context representation
- **Decoder** takes this context and target sequence to generate the output
  - automatically adapts to the output length

- Stacked architecture ensures efficient learning and generalization across tasks



Data Preparation and Model Techniques

Data Loader

- Loading and Tokenizing WMT19 Dataset with PyTorch and Transformers
- Objective: Loading and preparing data from the WMT19 dataset for machine learning tasks.
- Technologies: PyTorch, Transformers (Hugging Face), JSON, REST APIs.

Download and Processing

- Allowed downloading the entire WMT19 dataset into JSON files.
- Utilizing ThreadPoolExecutor for parallel download.
- Workers were used to enable parallel data loading, which significantly speeds up the process of preparing data for training.
- An offset was used to ensure that each worker processes a distinct segment of the data, to cap the download data and avoiding overlap of data

WMT19JSONLoader

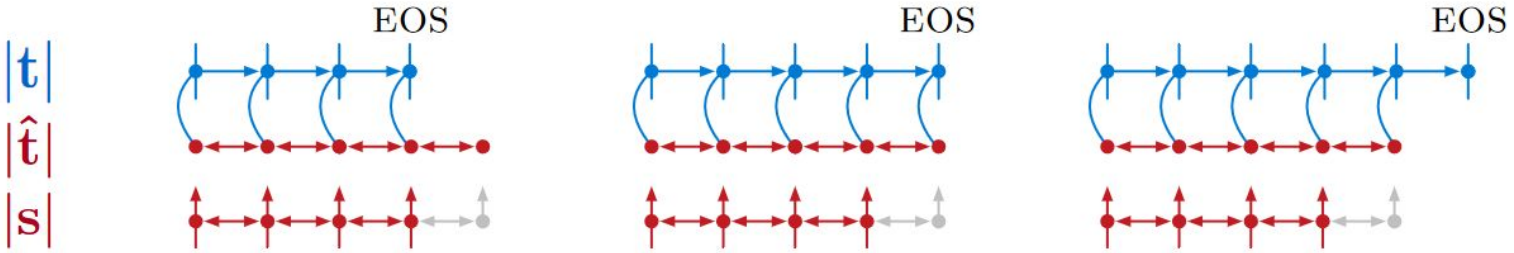
Responsible for Loading and Tokenizing from JSON

Functionalities:

- Loading data from a JSON file.
- Extracting out of the JSON the german and english rows
- Tokenizing source and target language texts.
- Using the ByT5 tokenizer for tokenization (Character-to-Character)
- Converting tokenized sequences into tensors.

Dynamic Unfolding in Sequence Modeling

- Dynamic unfolding optimizes sequence generation processes using encoder-decoder architectures.
- Key Parameters:
  - a: Scaling factor for source sequence length.
  - b: Offset for target sequence length.



Dynamic Unfolding: Target Length

Calculates target sequence length based on source sequence length.

The formula is:

$$|\hat{\mathbf{t}}| = a|\mathbf{s}| + b$$

Unfold:

Implements sequence unfolding using encoder output for decoder input.

Iterates until the decoder generates an end-of-sequence token, **or** until the target length is reached.

Unfolding Process Example

Example Scenario:

Example: "Hi all"

Decoder input: start_token => Decoder output: Hi

Decoder input: Hi => Decoder output: all

Decoder input: Hi all => Decoder output: end_of_sequence

Input-Embedding-Tensor

Objective: Embedding input tokens using lookup tables for sequence modeling.

Key Components:

- Vocabulary size: Denotes the total number of distinct tokens that can be embedded. Dictates the size and scope of the lookup table used to map token IDs to their corresponding embeddings.
- Size of embedding units: Determines the dimensionality of the embedding vectors stored in the lookup table. Each token ID is associated with an embedding vector of this specified size, capturing contextual meaning and relationships within sequences.

Input-Embedding-Tensor

Method Overview

- Functionality: The embed method facilitates the embedding of tokens into a vector space using a lookup table.
- Inputs: The method takes input token IDs, each intended to be mapped to its corresponding embedding vector.
- Outputs: Returns an embedded tensor containing representative embedding vectors for the input tokens.

Input-Embedding-Tensor Example

Vocabulary: ["Hallo", "**,** ", "wie", "gehts"]

- We want to embed these tokens into 3-dimension vectors
- We define a vocab size, in this case: Length of 4
- A embedding dimension of 3 is defined
- Words are assigned to IDs
- A embedding-layer is defined: The words are now transformed into numbers, more precise the embedded tensors
- Resulting tensor has length **n x 2d**

Input-Embedding-Tensor Result

Vocabulary: ["Hallo", ", ", "wie", "gehts"]

Result:

```
tensor([[-1.5470,  0.4647,  0.1201],
        [-0.8267, -0.4509,  0.4771],
        [ 0.0986,  1.1789,  0.1931],
        [-0.1981, -0.8969, -0.8885]], requires_grad=True)
```

Input-Embedding-Tensor Representation

tokens = ["Hallo", "wie"]

token-ids: [0, 2]

Embedded -Representation is:

```
tensor([[-1.5470,  0.4647,  0.1201],
        [ 0.0986,  1.1789,  0.1931]], grad_fn=<EmbeddingBackward0>)
```

Beam Search Decoder

Problem: Greedy Decoding selects the most probable next step but can lead to suboptimal solutions.

Goal: Improve prediction quality in sequential models aiming to find more optimal and coherent sequences of outputs

Beam Search: A heuristic algorithm that follows multiple possible paths simultaneously and keeps the best candidates.

Parameter: Beam Width (number of paths followed simultaneously)

Beam Search Decoder: How does it work

- Starts with the start token
- Initializes beams (paths) with the start token and an initial probability of 0
- At each step, the decoder calculates the probabilities of the next tokens for each beam
- Selects the top-K most probable tokens for each beam.
- beam_width is used to get the beam_width topk propabilities
- Retains the top-K beams based on cumulative probabilities.

Beam Search Decoder: Example Hello World

Start token: <s>

First step:

Candidates: <s> Hello, <s> Hi, <s> Good

Probabilities: 0.4, 0.35, 0.25 => Candidate **Hello**

Second Step:

Candidates: <s> World, <s> Globus, <s> Earth

Probabilities: 0.55, 0.15, 0.30 => Candidate **World**

Final beam: <s> Hello World

Experiments / Reproduction

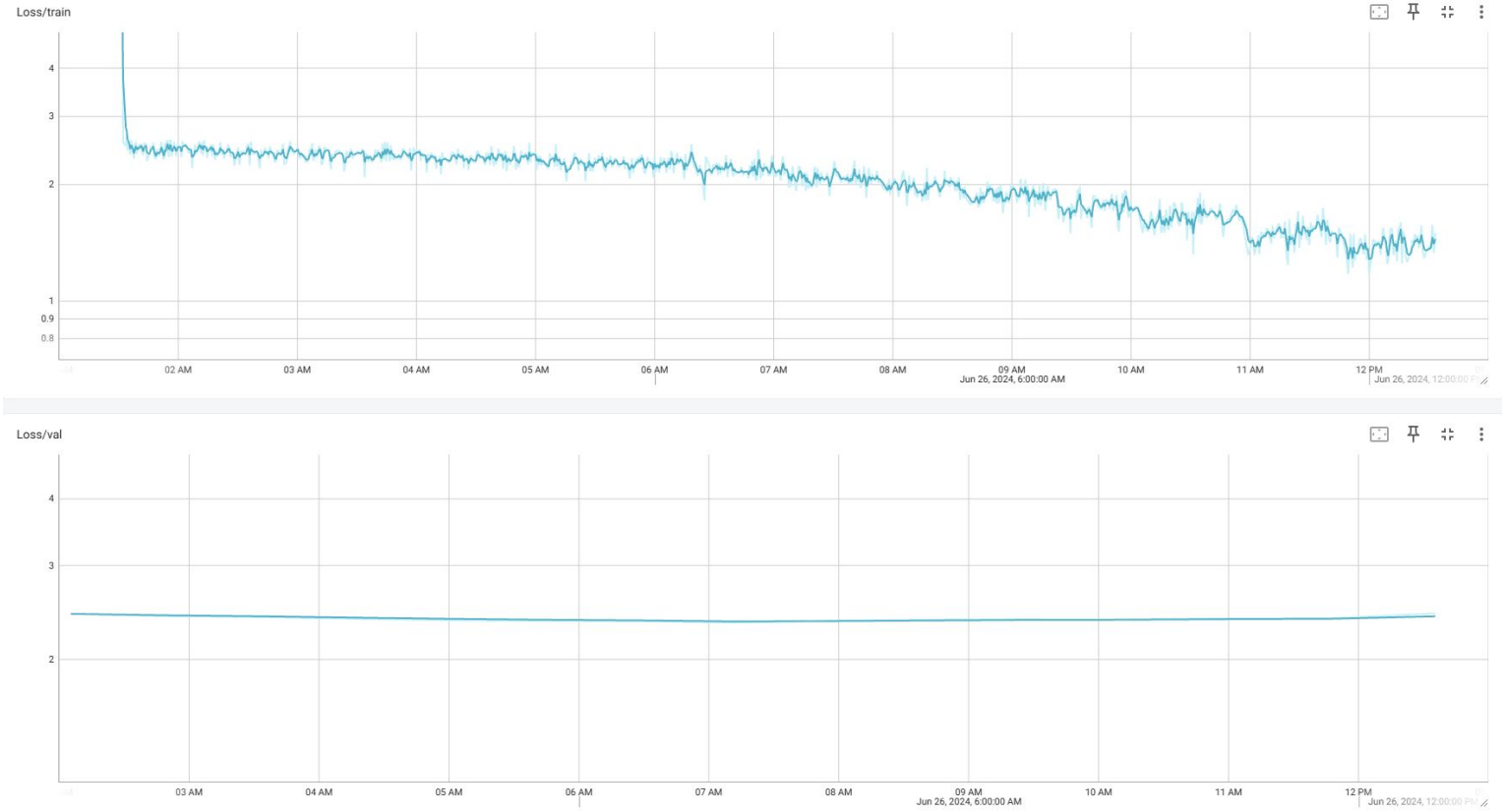
Experiment Setup

- All experiments run using either using Tesla T4 (Datalab) or local RTX 4070
- Params taken directly from the paper unless specified

Machine Translation Task

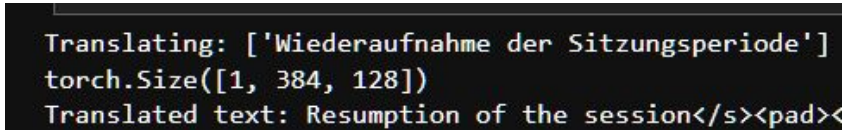
- Architecture in the Paper:
  - 30 ReLU residual blocks for Encoder and Decoder (6 sets with 5)
  - max dilation = 16, masked kernel size = 3
- Paper: WMT Dataset
- Our Architecture:
  - 15 ReLU residual blocks for Encoder and Decoder (3 sets with 5) due to Hardware limitations
  - same params
- Only possible for small subset of WMT
- Rather poor performance, model with very small subset performs well, but only on train data

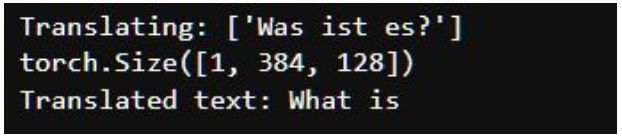
Machine Translation - Loss



Machine Translation - Results

- Train loss decreasing
  - Validation loss mostly stays the same -> likely overfitting due to small data subset
- Some simple sentences get translated rather well
  - Example:



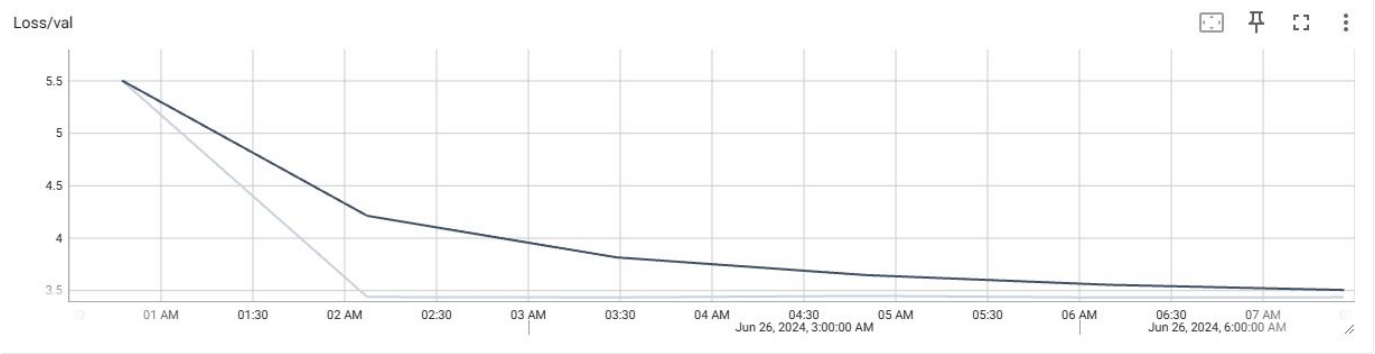


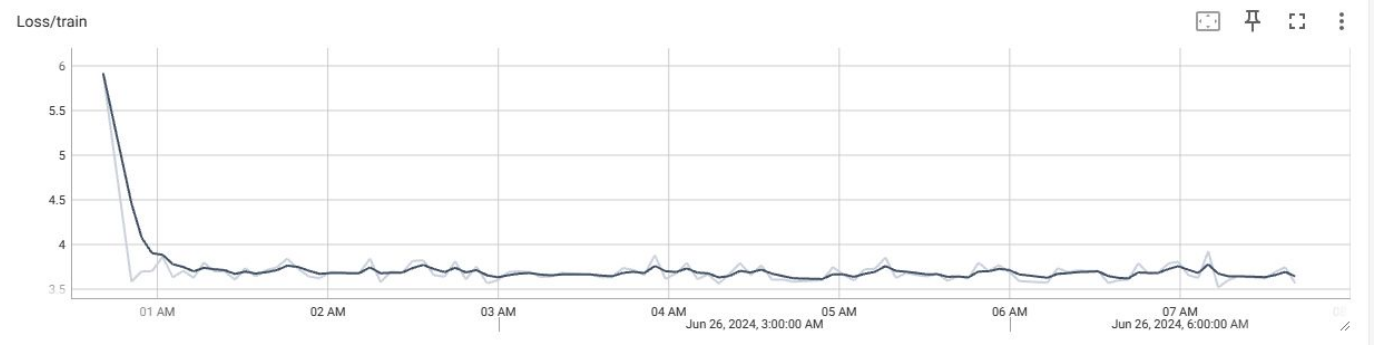
- However: Overfit!
- A lot of sentences/words do not translate well
- Experiments with different learning rates
  - Too high seems to completely break the experiment
  - Small changes have almost no effect on loss

Character Prediction Task

- Architecture:
  - Decoder only, 6 sets of 5 MU Residual blocks
  - Params generally same as for translation task
- Hunter Prize Wikipedia Dataset
- Our Approach:
  - 4 residual blocks, only 1 epoch on entire dataset

Character Prediction - Loss





Evaluation with the BLEU Score Task

- Approach in the Paper:
  - Encoder-Decoder Network
  - Character-level tokenization
  - Evaluation Metrics: BLEU Score, Bits per Character (BPC)
  - WMT Dataset 2014 and 2015 test sets

- Our Approach:
  - Encoder-Decoder Network
  - Average BLEU Score per Sentence Batch and Character
  - Evaluation Metrics: BLEU Score, Character-Level Accuracy
  - Subset of WMT Dataset 2019
  - Subset of WMT Dataset 2014

Evaluation with the BLEU Score Results

- Paper Results:
  - BLEU Score 22.85 (0.380 bits/character) and 25.53 (0.389 bits/character)

Average BLEU Score	wmt19	wmt 14
per Batch:	84.39 %	4.29 %
per Sentence:	85.71 %	5.40 %
per Character:	97.12 %	21.66 %

- Character-Level Accuracy
  - 98.72 % and 17.54 %

Reasons for low BLEU Score

- Significant different results
  - high values on wmt 19 (trainings dataset): good translations
  - low values on wmt 14: lower translation accuracy

- Translation on character level performs best with both datasets
  - to be expected because of our setup

- but why these results?
  - Limited dataset size for training
  - Potential overfitted model
  - Limited time for training
  - Differences in preprocessing and tokenization methods compared to the paper

Conclusion

Conclusion

- ByteNet experiments hard to reproduce
  - Limitations in Hardware lead to strong reduction in network size / Train set size
    - Still very long train times
  - Very little information / discussion / implementations on a lot of the parts of the ByteNet
  - Most existing information very outdated (and uses TensorFlow!)
  - Some parts (especially for the character prediction) quite vague in the Paper
- Future Work
  - Modernize ByteNet with developments since 2017
  - Compare with performance of modern Transformers

Thank you for your attention!