Efficient and Flexible Implementation of Machine Learning for ASR and MT

Albert Zeyer[1,2], Nick Rossenbach[1,2], Parnia Bahar[1,2], André Merboldt[1], Ralf Schlüter[1,2]

[1]RWTH Aachen University
[2]AppTek

Purpose of this Tutorial

- Overview of existing deep-learning frameworks for human language technology (HLT) related tasks
- Discuss core features of higher-level frameworks in context of sequence processing
- Discuss application oriented features in context of machine translation, speech recognition and other tasks
- Compare needs of researchers to requirements for production systems
- Compare our framework RETURNN to other frameworks

What this tutorial is not about:

- Designing full pipelines for HLT tasks
- Explaining how to write complete RETURNN setups

Target Audience

This tutorial is mostly targeting:

- Active framework and toolkit contributors
- Developers working on HLT related software
- Researchers interested in flexible neural architecture design

Prerequisite knowledge:

- Being familiar with sequence tasks and HLT
- Basic knowledge about deep learning platforms and frameworks (e.g. TensorFlow, PyTorch, MXNet...)
- Experience with designing and training neural networks

Terminology: Framework

Machine learning/deep learning software can have many different terms:

- TensorFlow:
  - "end-to-end open source machine learning platform" (Homepage / README)
  - "Open Source Machine Learning Framework" (GitHub description)
  - "open source software library for numerical computation using data flow graphs" (README 2015-2018)
- PyTorch:
  - "open source machine learning framework" (homepage)
  - "open source machine learning library" (Wikipedia)
- Keras:
  - "the Python deep learning API"
- OpenSeq2Seq, ESPNet, Fairseq:
  - "... toolkit ..."

We will discuss the conceptual and implementation aspects, not specific tools or scripts:
→ In this presentation we will stick to the term "framework"

Content and Speakers

Part 1: Implementation of Machine Learning for Sequence Processing

- Frameworks Overview
- Flexibility vs. Efficiency vs. Simplicity
- Concepts
- Recurrency
- Training
- Native Operations
- Conclusion

Part 2: Specific Models & Applications

- Introduction
- Machine translation (RNN/Transformer-based encoder-decoder-attention)
- Speech recognition (Hybrid HMM, Attention, other end-to-end approaches)
- Language modeling (RNN/LSTM, Transformer)
- End-to-end speech translation
- Text-to-speech

Conclusion

Content and Speakers

Part 1: Implementation of Machine Learning for Sequence Processing

Frameworks Overview (Speaker: Nick Rossenbach)

- Defining characteristics of deep-learning frameworks for HLT tasks
- Presenting a broad overview of existing frameworks and their differences

Framework Implementation (Speaker: Albert Zeyer)

- Introduction of RETURNN
- Core concepts of ML frameworks and RETURNN
- Requirements for sequence tasks
- Solutions and implementations

Content and Speakers

Part 2: Applications (1)

Machine Translation (Speaker: Parnia Bahar)

- Recurrent attention model with attention expansions
- Layer-wise pretraining
- Transformer networks
- Hybrid MT

Automatic Speech Recognition (Speaker: André Merboldt)

- Hybrid HMM and end-to-end networks
- Framework interface (RETURNN $\leftrightarrow$ RASR)
- Encoder pretraining
- Transducer model implementation details

Content and Speakers

Part 2: Applications (2)

Language Modeling (Speaker: Parnia Bahar)

- LSTM and Transformer language modeling
- Language model fusion
- Sampled Softmax

Speech Translation (Speaker: Parnia Bahar)

- Cascade system
- End-to-end modeling
- Transfer learning
- Multi-task learning

Text-to-Speech (Speaker: Nick Rossenbach)

- Tacotron-2 with GST
- Attention extensions
- Vocoder Integration

Part 1: Implementation of Machine Learning for Sequence Processing

Frameworks Overview

Part 1: Implementation of Machine Learning for Sequence Processing

Frameworks Overview

RETURNN

Flexibility vs. Efficiency vs. Simplicity

Concepts

Recurrency

Training

Native Operations

Conclusion

Deep Learning Backends

- Majority of ML applications implemented based on deep-learning platform / backend:
  - e.g. TensorFlow, PyTorch or MXNet
- Core features:
  - Providing mathematical operations with efficient hardware optimized implementations (CUDA, BLAS, ...)
  - Automatic gradient computation
- Extended by large amount of new features:
  - Neural network layer definitions
  - Graph and parameter management
  - Efficient data loading and processing
  - Loss and optimizer implementations
- For each backend exist framework extensions or wrappers that reduce programming overhead
  - Tensorflow: Keras/TF Addons (integrated), Sonnet
  - PyTorch: `torch.nn` (integrated), Lightning, Ignite
  - MXNet: Gluon (integrated)

Higher-Level Frameworks/APIs

- Features of higher-level frameworks
  - Simplifying code structure with model abstraction
  - Abstractions for data loading and training
  - Pre-defined complex layer modules, e.g. self-attention layer
  - Supporting multi-GPU / distributed training
  - Encapsulated training process with predefined parameters:
    - checkpoints, early stopping, profiling, etc...
- Frameworks designed for HLT tasks may have additional features
  - Task specific data (pre-)processing pipelines
  - Pre-defined model architectures, e.g. following encoder-decoder concepts
  - Example configurations for common tasks, e.g. LibriSpeech or WMT tasks
  - Pre-trained models to perform tasks on user-data

Frameworks Overview

Example Sequence-to-Sequence Frameworks

- Include finished implementations for HLT tasks
- Example configurations for quick model reproduction with nearly no effort

Framework Name	Backend	Language	Highlighted Features	Application Focus
OpenSeq2Seq	TensorFlow	Python	distributed training, powerful config	ASR/MT/TTS
Lingvo	TensorFlow	Python	module interface, scalability	ASR/IR/MT
Fairseq/Espresso	PyTorch	Python	customizable user plug-ins	MT/ASR
ESPNet	PyTorch & Chainer	Python	KALDI integration, many architectures	ASR/MT/TTS
Flashlight/Wav2letter++	ArrayFire/CUDA	C++	fast training and decoding	ASR
Sockeye	MXNet	Python	SOTA MT Architectures	MT

→ Differences in configuration flexibility

→ Some frameworks have modular components, some have rather fixed architectures

→ Frameworks may need internal code changes do add new models

→ Some frameworks (or toolkits) are designed for specific tasks, adding new tasks might require a lot of work.

Frameworks Overview

Flexibility and Features

There are different features, that are not covered by all frameworks:

- Some frameworks are specifically made for providing flexibility
  - OpenSeq2Seq, Lingvo, Fairseq: extendable classes for model, encoder, decoder, data, losses, etc...
  - OpenSeq2Seq: Python style configuration with possible user code imports
  - Lingvo: Everything is a class, even the actual experiment configuration
  - Fairseq: Supports user-folder that can implement new tasks, architectures, etc...

- Other frameworks are more focused on specific implementations
  - Wav2Letter++: Pre-defined Flashlight modules, simple sequential text-file format for S2S and CTC architectures
  - Sockeye: Originally designed for 3 fixed NMT architectures, now extended by more flexible Gluon API

- Frameworks might bring their own implementations
  - Flashlight/Wav2Letter++ implements its own functions based on C++, ArrayFire and CUDA
  - Lingvo implements custom C++ operations that are accessible via Python

Frameworks Overview: RETURNN

What is RETURNN

- RWTH extensible training framework for universal recurrent neural networks
- `https://github.com/rwth-i6/returnn`
- Python, based on TensorFlow, custom C++ / CUDA code
- Can be used as framework and a standalone tool
  - Framework (`import returnn` ...):
    - Similar: Keras, PyTorch Lightning
    - Can be used to write e.g. custom decoders, server applications, custom train loop, ...
    - Embed in existing software (e.g. RASR)
  - Tool (`rnn.py <config>` ...):
    - Similar: ESPNet, OpenSeq2Seq, Fairseq, ...
    - Write config, run given tool for training, decoding, ... (predefined but flexible tasks)
    - Arbitrary architectures and training schemes can be defined, just as flexible as framework
- Very generic, but some of our main applications are:
  - Automatic speech recognition (ASR): hybrid HMM-NN models
  - ASR / translation encoder-decoder-attention models, other end-to-end models
  - Language modeling
- Goals: High flexibility & simplicity & high training speed & high decoding speed

History of RETURNN

- 2013: Start of project (Theano backend), by Patrick Doetsch and Paul Voigtlaender
- Dec 2016: Start of TensorFlow support (TF 0.12.0), `Data` class
- Mar 2017: "RETURNN: ...", ICASSP [Doetsch & Zeyer+ 17]
- May 2017: Flexible `RecLayer`, `ChoiceLayer`, encoder-decoder attention, beam search
- Jul 2018: "RETURNN as a Generic Flexible Neural Toolkit ...", ACL [Zeyer & Alkhouli+ 18]
- Sep 2018: "... Attention Models for Speech Recognition", Interspeech [Zeyer & Irie+ 18]
- Oct 2018: "Neural Speech Translation at AppTek", IWSLT [Matusov & Wilken+ 18]
- Dec 2018: "... Analysis on Attention Models", IRASL/NeurIPS [Zeyer & Merboldt+ 18]
- May 2019: "... 2D Sequence-to-Sequence Models ...", ICASSP [Bahar & Zeyer+ 19a]
- Sep 2019: "... Local Monotonic Attention Variants", Interspeech [Merboldt & Zeyer+ 19]
- Sep 2019: "Language Modeling with Deep Transformers", Interspeech [Irie & Zeyer+ 19]
- Nov 2019: "... SpecAugment for ... Speech Translation", IWSLT [Bahar & Zeyer+ 19b]
- Dec 2019: "... Transformer and LSTM ... for ASR", ASRU [Zeyer & Bahar+ 19]
- Oct 2020: "... Improved Neural Transducer", Interspeech [Zeyer & Merboldt+ 20]

Usage as a Tool

- Call `rnn.py <config> [-other options]`
- Training, forwarding, eval, beam search / decoding, etc (`task` option)
- Config formats: Python or JSON (with comments)

- Config and/or command line options define:
  - Datasets including feature extraction (for train, cv, eval, ...)
  - Batching / chunking
  - Model/network topology (independent from search or training), including losses (for eval or training), constraints (L2, ...) and regularization (dropout, ...)
  - Training optimizer (SGD, Adam, momentum, ...)
  - Learning rate scheduling logic (Newbob, or constant decay, ...)
  - Pretraining logic
  - ... and many other things, lots of hooks
- Almost as flexible as framework usage

Flexibility vs. Efficiency vs. Simplicity

Part 1: Implementation of Machine Learning for Sequence Processing
- Frameworks Overview
- Flexibility vs. Efficiency vs. Simplicity
- Concepts
- Recurrency
- Training
- Native Operations
- Conclusion

Flexibility vs. Efficiency vs. Simplicity

Features:

- Simplicity
  - Writing config / code is simple & straight-forward (setting up experiment, defining model)
  - Debugging in case of problems is simple
  - Reading config / code is simple (defined model, training, decoding all becomes clear)
- Flexibility
  - Allow for many different kinds of experiments / models
- Efficiency
  - Training speed
  - Decoding speed

All items are important for research, decoding speed is esp. important for production.

Examples:

- CUDA/C++: Very flexible but not simple.
- Pure TensorFlow: Very flexible, not so simple, often not optimal efficiency
- ESPNet: Simple for supported models, needs extra effort for new models

Features imply trade-off.

Concepts

Part 1: Implementation of Machine Learning for Sequence Processing

Frameworks Overview

Flexibility vs. Efficiency vs. Simplicity

Concepts

- Tensors, Named Tensors, RETURNN `Data`
- Layers
- Network Topology & Construction
- Dataset Pipeline
- Training Execution Guide
- Decoding / Beam Search

Recurrency

Training

Native Operations

Conclusion

Concepts: Tensors, Named Tensors, RETURNN `Data`

Tensors

- Tensor $\equiv$ N-D array, generalization of scalar (0D), vector (1D), matrix (2D)
- Tensor is the atomic unit of all modern ML frameworks

Example tensors are of shape / format:

- $[\text{Batch}, \text{Time}, \text{Feature}]$: audio, or any sequence, float32 (dense)
- $[\text{Time}, \text{Batch}, \text{Feature}]$: more efficient for RNNs
- $[\text{Batch}, \text{Time}]$: e.g. class indices for each frame, int32 (sparse)
- $[\text{Batch}]$: e.g. class indices for the whole seq (speaker id or so)
- $[\text{Batch}, \text{Width}, \text{Height}, \text{Feature}|\text{Channel}]$: image
- $[\text{Batch}, \text{Feature}|\text{Channel}, \text{Width}|\text{Time}, (\text{Height})]$: more efficient for CNNs

Tensor Meta Properties

Meta properties about a tensor:

- Shape
  - What axes are there, in what order
  - What are their dimensions
- Identification of axes/dimensions (or: names / labels)
  - Batch
  - Input sequence / time
  - Output / target sequence
  - Feature dimension
- Sequence lengths
  - Input
  - Targets
  - Width/height of an image, or any 2D data
- Data type (float, int, string, ...)
- Categorical data, i.e. class indices (int) (sparse $\equiv$ one-hot vector) $\rightarrow$
  - How many classes (dimension)
  - Maybe some vocabulary

Review of Previous Attempts: Named Tensors, Axes have Names

- 2008, Pandas for Python, `DataFrame`: Labelled tabular data
- May 2014, xarray for Python: N-D labeled arrays
- Feb 2015, AxisArrays.jl for Julia: Each dimension can have a named axis
- Nov 2016, LabeledTensor for TensorFlow: Semantically meaningful dimensions
- Dec 2016, RETURNN / TensorFlow backend, `Data` wraps `tf.Tensor`
- Oct 2018, Tensor Shape Annotation Library (tsalib) for TF/PyTorch/NumPy: Named dimensions, e.g. `'btd'`
- Jan 2019, HarvardNLP / Alexander Rush, NamedTensor for PyTorch
- Oct 2019, PyTorch has official support for named tensors: E.g. `torch.zeros(2, 3, names=('N', 'C'))`

- Axes have names
  - Just a string, e.g. "channel"
  - Explicitly specified
- RETURNN `Data` axes and dimensions are similar but different:
  - Predefined with special meaning (e.g. batch)
  - Predefined by dataset & implicit by layers
- RETURNN `Data` contains much more meta information

RETURNN `Data` Axes / Dimensions

- Dynamic axes have sequence lengths per batch-dim
- Batch axis (B) is special
- Optionally some default time / dynamic axis, and feature-dim axis

- Dynamic axes defined/stored as global unique tag (`DimensionTag` object)
- Usually no need to manually/explicitly define dimension tags:
- Dynamic axis tags are predefined by dataset
  - E.g. input ("data") x_1^T:
    - `Data` x has shape `[B,T|'time:var:extern_data:data',F|40]` (40 dim. MFCC)
    - $T \equiv$ `DimensionTag 'time:var:extern_data:data'`
  - Targets ("classes") y_1^N:
    - `Data` y has shape `[B,T|'time:var:extern_data:classes']` (int32 class indices)
    - $N \equiv$ `DimensionTag 'time:var:extern_data:classes'`
- Other dynamic axes results from resizing operations (pooling, rescaling, slicing, prefix/postfix, padding, ...)
  - E.g. `PoolLayer 'pool1'` with 1D pooling results in `DimensionTag 'spatial:0:pool1'`

RETURNN Data Examples

- Input: `Data(name='data', shape=[B,T|'time:var:extern_data:data',F|40])`
- Targets: `Data(name='classes', shape=[B,T|'time:var:extern_data:classes'],`
  `dtype='int32', sparse=True, dim=1030, vocab=...,`
  `available_for_inference=False)`

- LSTM output: `Data(name='lstm0_fw_output', shape=[T|'time:var:extern_data:data',B,F|1024]`
- Encoder output (downsampled):
  `Data(name='encoder_output', shape=[T|'spatial:0:lstm1_pool',B,F|2048])`

- Attention weights, 8 heads, on downsampled encoder:
  - Inside decoder recurrent loop:
    `Data(name='att_weights_output', shape=[B,F|8,T|'spatial:0:lstm1_pool'])`
  - Outside decoder recurrent loop, in training (using target sequence lengths for decoder):
    `Data(... shape=[T|'time:var:extern_data:classes',B,F|8,'spatial:0:lstm1_pool'])`

RETURNN `Data` Object

The `Data` object contains:

- `tf.Tensor` reference (optional, we can just use `Data` as a template)
- (Batch) shape of tensor
- Marking of special axes:
  - Batch-dim axis
  - (Default) time-dim axis (or any spatial axis) (can be of variable length)
  - (Default) feature-dim axis
- (Sequence) lengths of dynamic axes (e.g. time axis) (per batch, shape [Batch])
- `DimensionTag`s (esp. for dynamic axes)
- Data type (`dtype`: float, int, any type supported by TF)
- Sparse $\rightarrow$ dtype is int, values are class indices ($\equiv$ one-hot vector)
- Dimension: number of classes, or feature dim.
- Vocabulary of classes
- Availability at decoding time
- Beam information from last choice in beam search

RETURNN `Data` Usage

- Used everywhere in RETURNN, since the very beginning of TensorFlow backend implementation
- Layer inputs / outputs are `Data`

- Most efficient tensor format depends on operation:
  - Audio, or any sequence
    - Standard format: [Batch, Time, Feature]
    - More efficient for RNNs: [Time, Batch, Feature]
  - Image
    - Standard format: [Batch, Width, Height, Feature|Channel]
    - More efficient for CNNs: [Batch, Feature|Channel, Width|Time, (Height)]
- Layers convert input to most efficient format, output as-is
- Difficult (or easy to get wrong) without named tensors / `Data`

Example for Convolution: NHWC vs. NCHW

- 2D convolution / pooling in CUDA, TensorFlow, PyTorch allows input formats:
  - NHWC: [Batch, Width, Height, Channel]
  - NCHW: [Batch, Channel, Width, Height]

- RETURNN training efficiency (time for subepoch) on Switchboard ASR (300h) for Hybrid HMM ResNet model:

Fused BatchNorm	Data format	Final pooling	Time [H:MM]
no	NHWC	`pool`	2:14
no	NCHW	`pool`	0:54
yes	NCHW	`pool`	0:39
yes	NCHW	`reduce_mean`	0:37

  - $\sim 2.5\times$ faster with right tensor format

Concepts: Layers

- NN literature and most frameworks encapsulate common operations in layers (modules)
  - Keras ("layers"): `layers.Dense`, `layers.Conv1D`, ...
  - PyTorch `torch.nn` ("modules"): `nn.Linear`, `nn.Conv1d`, ...
  - RETURNN ("layers"): `LinearLayer`, `ConvLayer`, ...
- NN / model defined by interconnected layers, as directed graph

- Example: Feed-forward (Linear) layer with (optional) (non-linear) activation function (f):
  - Input: $x \in \mathbb{R}^{\cdots \times N_{\text{in}}}$
  - Output: $\mathrm{Linear}(x) \in \mathbb{R}^{\cdots \times N_{\text{out}}}$
  - Parameters (trainable): Weight matrix $W \in \mathbb{R}^{N_{\text{in}} \times N_{\text{out}}}$, bias $b \in \mathbb{R}^{N_{\text{out}}}$
  - Operation: $\mathrm{Linear}(x) := f(xW + b)$
  - Configuration:
    - $f \in \{\mathrm{identity}, \tanh, \mathrm{sigmoid}, \mathrm{relu}, ...\}$
    - $N_{\text{out}} \in \mathbb{N}$
    - $N_{\text{in}} \in \mathbb{N}$ (often implicitly defined by input)
    - Optional: Parameter initialization (random, ...)
    - Optional: Regularization (dropout, L2, ...)
    - Optional: Extra losses

Concepts: Layers

Layers in Frameworks

Layers are of different complexity

(RETURNN/PyTorch examples, similar in all frameworks)

- Lower level, very atomic:
  - `ReLU`, `Tanh`, ...
  - `ReduceLayer`: wraps `tf.reduce_min|max|...`
  - `ConstantLayer`: wraps `tf.constant`
  - `VariableLayer`: wraps `tf.Variable` (parameter)
  - `DotLayer`: wraps `tf.matmul`
  - Difference `tf.matmul` vs. `DotLayer` in RETURNN: `Data`, like `einsum` + named axes
- Higher level:
  - `LinearLayer`: matmul, addition, params, activation
  - `ConvLayer`: `tf.nn.convolution`, addition, params, activation
  - `DropoutLayer`
  - `BatchNormLayer`
  - `SelfAttentionLayer`
  - `LSTM`
  - `Transformer`

Problems with Higher Level Layers in Frameworks

- Details hidden, often users do not even know about them
  - Might not be a problem, arguable
- Lots of variations exists (e.g. LSTM / Transformer)
  - Already old (e.g. LSTM): mostly settled, not too much a problem, just use default / "standard" variant
  - Under active research (e.g. Transformer): Will likely change, further options added, becomes messy
    - Only support "standard" (clean, few config options), or lots of options, often not variant you want
- Research on new variations:
  - Need to add further option to layer: `my_custom_feature=...`, becomes messy
  - Need to edit internal framework code
    - Own fork, diverges from main framework code, cannot easily collaborate anymore
  - Reimplement (copy), add new custom layer: `MyCustomTransformer`, ok but extra effort later
    - When should it be adopted in framework, and how?
  - If it ends up in framework
    - Lots of options / variations will not be used in future

  → Becomes complex and bloated

  → Rotten code, unmaintained code

RETURNN Layer Conventions

- Should be generic / reusable in many different contexts (avoid rotten code)
- Simple functionality, only apply a single function (atomic) (ideally...)
  - Avoid layers which becomes more and more complex over time (more and more options)
  - Exception: Very common functions are also combined (linear/conv + activation, LSTM, ...)
  - Historically grown, also some bad examples
  - Good examples: `LinearLayer`, `ConvLayer`, `PoolLayer`, `RecLayer`, `DropoutLayer`, `MergeDimsLayer`, `PadLayer`, `ReduceLayer`, `EvalLayer`, `DotLayer`, ...
  - Bad examples: `AllophoneStateIdxParserLayer`, `NeuralTransducerLayer`, `NoiseEstimationByFirstTFramesLayer`, ...
    - `SelfAttentionLayer`: turned out as bad example as well...
- Should be flexible on input, e.g. `LinearLayer` accepts:
  - Any shape: (B,T,F), (B,F), (T,B,F), (B,F,W,H), ...
  - Multiple sources (common behavior: concatenate in feature dim.)
  - Sparse (int, class indices): Lookup / embedding (num. classes needed here)
  - Dense (input has feature dim): Matrix multiplication
- Automatically transforms input when more efficient for operation
  - E.g. `RecLayer` converts to (T,B,F) and then outputs as (T,B,F)
- Used to define the model for training and recognition/decoding, including beam search

Layer Usage Experience (e.g. to use Transformer Variant)

- Lingvo / Fairseq
  - Base model often implemented in toolkit code
  - User code often derives from base class, often multiple inheritances
  - Details are only understandable by following class inheritance, reading framework code
- ESPNet
  - Models implemented in toolkit code
  - In config you just write e.g. `encoder:transformer` (very simple for supported models)
  - Modification of Transformer requires to edit framework code
  - Details are only understandable by reading framework code
- RETURNN
  - Whole model defined in config, no derivation of base models
    - Quite verbose
    - Details becomes clear just from looking at config (model & training & recognition)
    - No need to look at internal framework code / documentation (ideally...)
  - Most layers well tested, often used (because atomic or very common)
  - No rotten code in RETURNN (ideally..., at least for recommended layers)

RETURNN Layer Definition

- Without calculation, resolve layer dependencies
  - Can be optional, e.g. dep. only used at decoding → allows automatic optimization
  - Classmethod `transform_config_dict(...)`
- Without calculation, get `Data` template (without `tf.Tensor`)

→ Needed e.g. for recurrency (`RecLayer`)
  - Classmethod `get_out_data_from_opts(**kwargs)`
    - `kwargs`: Inputs (other layers, or attribs / options)
  - Return output (`Data` without `tf.Tensor`)
- Construct layer with calculation, get `Data` (with `tf.Tensor`)
  - `__init__(**kwargs)`
    - `kwargs`: Inputs (other layers, or attribs / options) (same as `get_out_data_from_opts`)
  - Set `self.output` (`Data` with `tf.Tensor`)
- Optional:
  - Parameters
  - Losses
  - Constraints (L2, ...)
  - Sub layers, subnetwork (`SubnetworkLayer`, `RecLayer`, `SplitLayer`, ...)
  - Hidden state (inside `RecLayer`)

RETURNN Network Topology / Model Definition Example

Example of 2 layer BLSTM model:

```
network = {
  "lstm0_fw": {"class":"rec", "unit":"lstm", "n_out":500, "direction":1,  "from": "data"},
  "lstm0_bw": {"class":"rec", "unit":"lstm", "n_out":500, "direction":-1, "from": "data"},

  "lstm1_fw": {"class":"rec", "unit":"lstm", "n_out":500, "direction":1,  "from": ["lstm0_fw", "lstm0_bw"]},
  "lstm1_bw": {"class":"rec", "unit":"lstm", "n_out":500, "direction":-1, "from": ["lstm0_fw", "lstm0_bw"]},

  "output":   {"class": "softmax", "loss": "ce", "from": ["lstm1_fw", "lstm1_bw"]}
}
```

- Consists of layers which are interconnected in any possible way
- Defined in config as a `dict`
  - key: (`str`) layer name
  - value: (`dict`) `kwargs` for layer
- Some `kwargs` have special handling:
  - `"class"`: (`str`) Python layer class
  - `"from"`: (`str|list[str]`) source(s) as layer name(s)
  - Layer classes can override/extend special handling
- **Defines model for decoding, and also for training**
  - Also for more complex models: encoder-decoder, transducer, ...

RETURNN Network Construction Example

Example:

```
network = {
    "conv0": {"class": "conv", "filter_size": [5], "padding": "same", "n_out": 100, "from": "data"},
    "lstm1": {"class": "rec", "unit": "lstm", "n_out": 500, "from": "conv0"},
    "lstm2": {"class": "rec", "unit": "lstm", "n_out": 500, "from": "lstm1"},
    "output": {"class": "softmax", "loss": "ce", "from": "lstm2"}
}
```

- Start constructing `"output"` layer
- Pop out `"class"` from the kwargs dict, get Python layer class (e.g. `SoftmaxLayer`, `RecLayer`, ...)
- Pop out `"from"`, collect list of layers, or recursively construct them;
  then add list of layers as `"sources"` back to kwargs dict;
  `"data"` is implicit layer for input data by dataset
- Basically call `layer_class(**kwargs)`

Concepts: Dataset Pipeline

Dataset Aspects, and Dataset Pipeline

- Supervised / unsupervised training needs data
  - Supervised: Input / output pairs
  - Unsupervised (or self-supervised): Just input (e.g. text to train language model)
  - Multiple losses might require extra information
    - Speech: e.g. speaker id
- Generic interface:
  - Multiple data streams (TF "feature columns") of different type / format
  - Not all available during decoding
- Preprocessing / feature extraction (MFCC or so)
- Mini-batch preparation logic
  - Shuffling the dataset (per epoch, or on-the-fly)
  - Take whole sequences, or chunks
  - Padding (try to minimize)
- (Reinforcement learning: Interactive environment + reward signal)

Concepts: Dataset Pipeline

Datasets in Frameworks

- TensorFlow: `tf.data.Dataset`, lots of batching/sorting logic in `tf.data`
- PyTorch: `torch.utils.data.Dataset` and `torch.utils.data.DataLoader`

- RETURNN generic `Dataset` interface:
  - Data streams (e.g. source, target, ...)
  - Get next utterance:
    - Values for all data keys, matching to the shape/dtype
    - Or end-of-epoch
  - Initialize new epoch
    - One epoch: arbitrarily defined by the dataset, also partition epoch
  - Control sorting / shuffling of utterances

RETURNN Dataset Usage Experience

- Existing implementations:
  - `ExternSprintDataset`: RASR (Sprint) corpus & feature extraction
  - `HDFDataset`: data stored in HDF files
  - `OggZipDataset`: audio as Ogg in Zip files + transcriptions
  - `LibriSpeechCorpus`: specific for LibriSpeech
  - ...
  - `MetaDataset`: combine multiple datasets
  - Create artificial data on-the-fly...
- New corpora, possible options:
  - Convert such that e.g. `HDFDataset` or `OggZipDataset` can read it
  - Write own dataset implementation
    - Use it directly
    - `hdf_dump` script can convert it to `HDFDataset`

→ Similar as other frameworks

Concepts: Training Execution Guide

Standard for most frameworks (including RETURNN):

1. Setup (load config, load/setup TF, ...)
2. Construct network for training (TF computation graph)
3. Construct optimizer for losses, gradients (TF computation graph)
4. Randomly initialize params (or partly import)
5. Maybe load model checkpoint file and load params
6. Train one epoch (epoch defined by dataset)
7. Cross validation
8. Learning rate scheduling update
9. Save model checkpoint
10. Repeat with next epoch

Variations:

- No epochs but just count train steps
- More complex train loops
  - Multiple losses, decoupled loops (e.g. GAN)
  - Reinforcement learning: rollouts, replay buffer, ...

Concepts: Decoding / Beam Search

Why Special Logic Needed for Decoding?

- Maybe not relevant for image object recognition, just use argmax

- Sequence tasks like ASR/MT need non-trivial search for recognition/translation
- Search space is too big (sequences: N, y_1^N) $\Rightarrow$ pruning needed $\Rightarrow$ beam search

Framework Properties of Beam Search

- Decoding/search can be external
  - RETURNN: Interfaces on multiple levels, embedded in other frameworks (RASR decoder)
- Direct support for beam search
  - RETURNN
  - ESPNet
  - Lingvo
  - ... (all frameworks supporting "end-to-end" models)
- Generic: possibly multiple stochastic (maybe latent) variables to search over, custom dependency graphs
  - RETURNN: Any number of stochastic variables, any dependency graph (unique feature)
  - Most other frameworks: Single hardcoded stochastic variable
- How to define model with/for training vs. decoding?
  - RETURNN: Single definition for training and decoding / beam search (unique feature)
  - Most other frameworks: Separate implementation for training & decoding

Recurrency

Part 1: Implementation of Machine Learning for Sequence Processing

- Frameworks Overview
- Flexibility vs. Efficiency vs. Simplicity
- Concepts
- Recurrency
  - Beam Search Decoding
  - Automatic Optimization
  - Example: Attention-based encoder-decoder model
  - Example: Decoupled LSTM and Transformer
  - Minimum Expected Risk Training
  - Multiple Stochastic Variables
- Training
- Native Operations
- Conclusion

Recurrency in RETURNN

Recurrency

$=$

Step-by-step execution,
where current step depends on previous step

- RNN, LSTM
- Beam search
- Any dynamic programming calculation (e.g. forced alignment)

RETURNN `RecLayer`

- Some predefined units (LSTM, ...) (using efficient native implementation)
- Can define any possible recurrent formula, via subnetwork, applied frame by frame
- Generic wrapper for `tf.while_loop` (while loop: run some body iteratively while some condition is true)
- Use `"prev:layer"` to access layer output from previous frame
- Example (LSTM):

```
network = {
 "lstm": {"class": "rec", "from": "data", "unit": {
   "input": {"class": "copy", "from": ["prev:output", "data:source"]},
   "input_gate":{"class":"linear","from":"input","activation":"sigmoid","n_out":10},
   "forget_gate":{"class":"linear","from":"input","activation":"sigmoid","n_out":10},
   "output_gate":{"class":"linear","from":"input","activation":"sigmoid","n_out":10},
   "cell_in": {"class":"linear", "from":"input", "activation":"tanh", "n_out":10},
   "c": {"class":"eval", "from":["input_gate", "cell_in", "forget_gate", "prev:c"],
        "eval": "source(0) * source(1) + source(2) * source(3)"},
   "output": {"class": "eval", "from": ["output_gate", "c"],
             "eval": "source(0) * source(1)"},
 }},
 "output": {"class": "softmax", "loss": "ce", "from": "lstm"}
}
```

- Automatic optimization (move independent layers outside of loop)
- Beam search and manage stochastic (latent) variables

RETURNN Beam Search Decoding

Example (simple encoder decoder model, no attention):

```
network = {
  "input": {"class": "rec", "unit": "nativelstm2", "n_out": 20},  # encoder
  "input_last": {"class": "get_last_hidden_state", "from": "input", "n_out":40},

  "output": {"class": "rec", "from": [], "target": "classes", "unit": {  # decoder
    "embed": {"class":"linear", "activation":None, "from":"output", "n_out":10},
    "s": {"class": "rec", "unit": "nativelstm2", "n_out": 20, "from": "prev:embed", "initial_state": "base:input_last"},
    "p": {"class":"softmax", "from":"s", "target": "classes", "loss": "ce"},
    "output": {"class":"choice", "from":"p", "target":"classes", "beam_size":8}
  }}
}
```

`ChoiceLayer` $\equiv$ stochastic (opt. latent) variable:

Search flag enabled:

- Select N best (in this frame) (`tf.nn.top_k`)
- Beam (N hyps) hidden in batch dim, Batch$'$:= Batch $\cdot N$ (all other layers just work as usual)
- Output shape (per frame): [Batch$'$], int32

Search flag disabled (default in training):

- Returns ground truth labels
- Output shape (per frame): [Batch], int32
- No dependency on `"p"` or anything in the rec layer

RETURNN Beam Search Decoding Internals

- `SearchChoices`:
  - Stores scores of current hyps in the beam (`beam_scores`)
  - Reference to layer where last choice was taken
  - Indices of source beam hyps (`src_beams`)
  - `TFNetwork.get_search_choices(sources, ...)`
  - `SearchChoices.translate_to_common_search_beam`
- `RecLayer` remembers: `src_beams` for each frame, class indices, final beam scores
- `RecLayer` does another inverse `tf.while_loop` to backtrack the final beam

RETURNN Network Construction, Mode Flags and `RecLayer`

- Mode flags: training, beam search, ...
- TF computation graph construction different depending on mode flags
  - E.g. dropout used in training only
- Layer dependencies depending on mode flags:
  - `prev:output` is ground truth label in training (no dependencies) or predicted at decoding
  - Huge effect in `RecLayer`, e.g. Transformer training fully parallel via automatic optimization
  - Example (encoder decoder, no attention), automatic optimization:

```
network = {
  "input": {"class": "rec", "unit": "nativelstm2", "n_out": 20},  # encoder
  "input_last": {"class": "get_last_hidden_state", "from": "input", "n_out":40},

  "output": {"class": "rec", "from": [], "target": "classes", "unit": {  # decoder
    "embed": {"class":"linear", "activation":None, "from":"output", "n_out":10},
    "s": {"class": "rec", "unit": "nativelstm2", "n_out": 20, "from": "prev:embed", "initial_state": "base:input_last"},
    "p": {"class":"softmax", "from":"s", "target": "classes", "loss": "ce"},
    "output": {"class":"choice", "from":"p", "target":"classes", "beam_size":8}
  }}
}
```

    - In training: all sub layers outside loop
    - With opt.: 8 sec/epoch. Without: 14 sec/epoch.

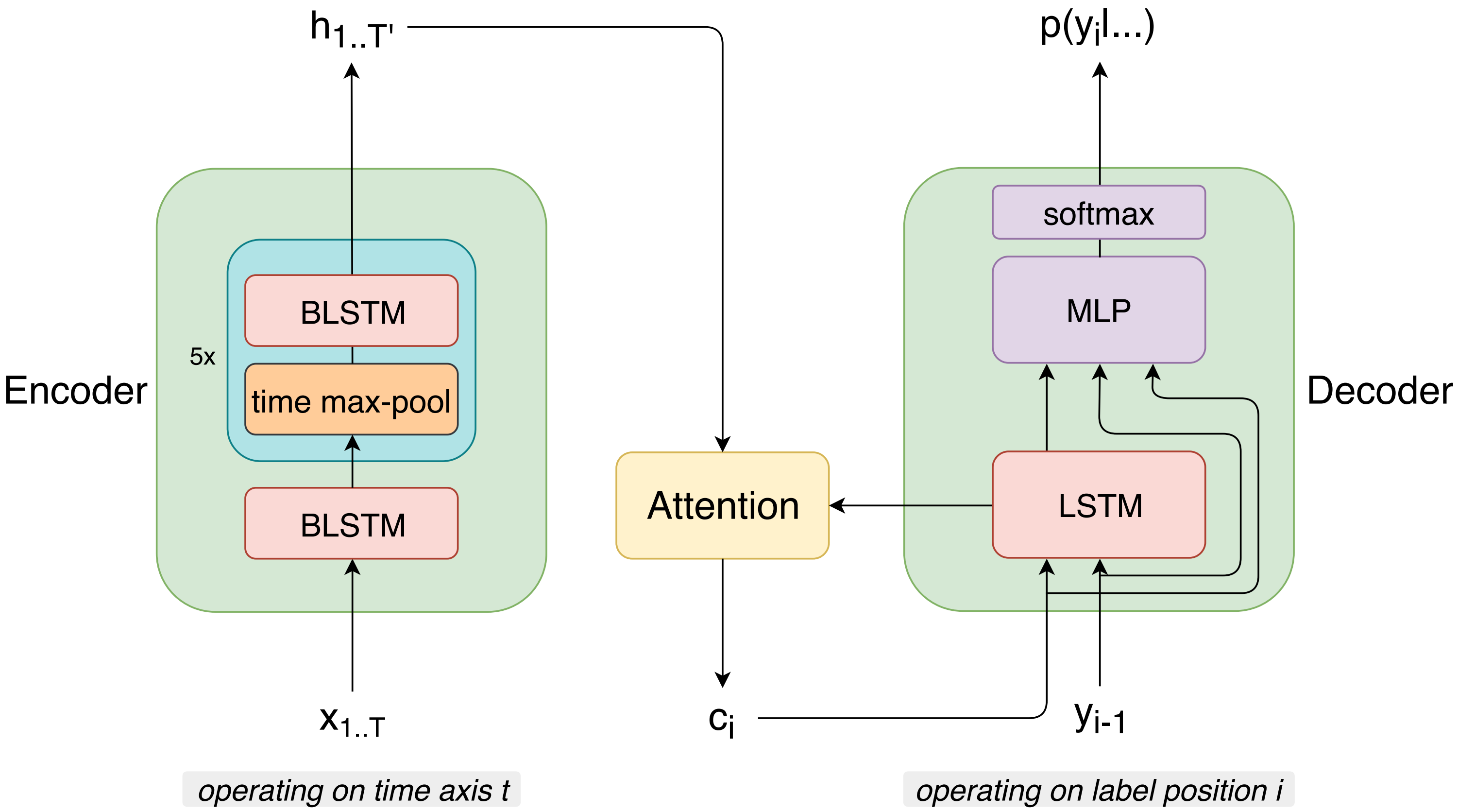
User Model Definition for Training & Decoding

- Model defined as used at decoding time (single definition for training & decoding)
  - Auto-regressive model: Predict next label (stochastic var.), given history
  - Using `ChoiceLayer` (decoding, but then also training)
- No separate logic / code in user config of model for training vs. decoding needed
  - Separate logic (training vs. decoding) would be more effort for a new model
  - Separate logic can lead to bugs / inconsistencies

- Automatic optimization for training (e.g. teacher forcing: previous label $\equiv$ ground truth)
  - Still as efficient as possible

Recurrency: Example: Attention-based encoder-decoder model

Encoder-Decoder Attention Architecture



[Zeyer & Irie+ 18]

Recurrency: Example: Attention-based encoder-decoder model

Decoder

- Output y_i and end-of-sentence condition:

```
"output": {'class': 'choice', 'target': 'classes', 'beam_size': 12, 'from': "output_prob", "initial_output": 0},
"end": {"class": "compare", "from": "output", "value": 0},
```

- Output prediction probability: $p(y_i|y_1^{i-1}, x_1^T) = \mathrm{softmax}(\mathrm{linear} \circ \mathrm{maxout} \circ \mathrm{linear}(s_i, y_{i-1}, c_i))$

```
"embed": {'class': 'linear', 'activation': None, "with_bias": False, 'from': "output", "n_out": 621, "initial_output": 0},
"readout_in": {"class": "linear", "from": ["s", "prev:embed", "att"], "activation": None, "n_out": 1000},
"readout": {"class": "reduce_out", "mode": "max", "num_pieces": 2, "from": "readout_in"},
"output_prob": {"class": "softmax", "from": "readout", "dropout": 0.3, "target": "classes", "loss": "ce", "loss_opts": {"label_smoothing": 0.1}}
```

- Decoder state: $s_i = \mathrm{LSTMCell}(s_{i-1}, y_{i-1}, c_{i-1}) \in \mathbb{R}^{D_{\mathrm{dec}}}$

```
"s": {"class": "rec", "unit": "NativeLstm2", "from": ["prev:embed", "prev:att"], "n_out": 1000},
```

- Attention context vector: $c_i = \sum_{t=1}^{T'} \alpha_{i,t} h_t \in \mathbb{R}^{D_{\mathrm{enc}}}$

```
"att0": {"class": "generic_attention", "weights": "att_weights", "base": "base:encoder"},  # (B, H, V)
"att": {"class": "merge_dims", "axes": "static", "from": "att0"},  # (B, H*V)
```

- Attention weights: $\alpha_i = \mathrm{softmax}_t(e_i) \in \mathbb{R}^{T'}$, normalized over time

```
"att_weights": {"class": "softmax_over_spatial", "from": "energy"},  # (B, enc-T, H)
```

- Attention energies: $e_{i,t} = v^\top \tanh(W[s_i, h_t]) \in \mathbb{R}$

```
"enc_ctx": {"class": "linear", "activation": None, "from": "base:encoder", "n_out": EncKeyTotalDim},  # (B, enc-T, K)
"s_transformed": {"class": "linear", "activation": None, "with_bias": False, "from": "s", "n_out": EncKeyTotalDim},  # (B, K)
"energy_in": {"class": "combine", "kind": "add", "from": ["enc_ctx", "s_transformed"], "n_out": EncKeyTotalDim},  # (B, enc-T, K)
"energy_tanh": {"class": "activation", "activation": "tanh", "from": "energy_in"},
"energy": {"class": "linear", "activation": None, "with_bias": False, "from": "energy_tanh", "n_out": AttNumHeads},  # (B, enc-T, H)
```

`RecLayer` Automatic Optimization in Training

- `RecLayer` represents while loop, iterating some calculation step-by-step
  - Used to define decoder of an encoder-decoder model
- Some calculations do not need to be in the same loop → optimize out of loop
- Showing optimized layers, and shape of layers

```
Rec layer sub net:
  Input layers moved out of loop:
    output    -- ground truth targets, [B,T|'time:var:extern_data:classes'], int32, dim 1030
    embed     -- [B,T|'time:var:extern_data:classes',F|621]
    enc_ctx   -- [T|'spatial:0:lstm1_pool',B,F|1024]
  Layers in loop:
    s              -- [B,F|1000]
    s_transformed  -- [B,F|1024]
    energy_in      -- [T|'spatial:0:lstm1_pool',B,F|1024]
    energy_tanh    -- [T|'spatial:0:lstm1_pool',B,F|1024]
    energy         -- [T|'spatial:0:lstm1_pool',B,F|1]
    att_weights    -- [B,F|1,T|'spatial:0:lstm1_pool']
    att0           -- [B,1,F|2048]
    att            -- [B,F|2048]
  Output layers moved out of loop:
    readout_in   -- [T|'time:var:extern_data:classes',B,F|500]
    readout      -- [T|'time:var:extern_data:classes',B,F|1000]
    output_prob  -- [T|'time:var:extern_data:classes',B,F|1030]
  Unused layers:
    end
```

Recurrency: Example: Attention-based encoder-decoder model

`RecLayer` Automatic Optimization in Decoding

- `RecLayer` represents while loop, iterating some calculation step-by-step
  - Used to define decoder of an encoder-decoder model
- Some calculations do not need to be in the same loop $\rightarrow$ optimize out of loop
- Showing optimized layers, and shape of layers

```
Rec layer sub net:
  Input layers moved out of loop:
    enc_ctx  -- [T|'spatial:0:lstm1_pool',B,F|1024]
  Layers in loop:
    s              -- [B,F|1000], beam 'prev:output', beam size 12
    s_transformed  -- [B,F|1024], beam 'prev:output', beam size 12
    energy_in      -- [B,T|'spatial:0:lstm1_pool',F|1024], beam 'prev:output', beam size 12
    energy_tanh    -- [B,T|'spatial:0:lstm1_pool',F|1024], beam 'prev:output', beam size 12
    energy         -- [B,T|'spatial:0:lstm1_pool',F|1], beam 'prev:output', beam size 12
    att_weights    -- [B,F|1,T|'spatial:0:lstm1_pool'], beam 'prev:output', beam size 12
    att0           -- [B,1,F|2048], beam 'prev:output', beam size 12
    att            -- [B,F|2048], beam 'prev:output', beam size 12
    readout_in     -- [B,F|1000], beam 'prev:output', beam size 12
    readout        -- [B,F|500], beam 'prev:output', beam size 12
    output_prob    -- [B,1030], beam 'prev:output', beam size 12
    output         -- [B], int32, dim 1030, beam 'output', beam size 12
    end            -- [B], bool, beam 'output', beam size 12
    embed          -- [B,F|621], beam 'output', beam size 12
  Output layers moved out of loop:
    None
  Unused layers:
    None
```

Recurrency: Example: Decoupled LSTM and Transformer

Decoupled decoder LSTM [Hannun & Lee+ 19, Zeyer & Bahar+ 19, Pham & Xu+ 20]:

- Model like standard LSTM-based encoder-decoder as before,
  very simple variation, decoder LSTM does not depend on attention context:

```
"s": {"class": "rec", "unit": "nativelstm2", "from": ["prev:embed", ~~"prev:att"~~], "n_out": 1000}
```

- What recurrency is there in the layers?
  - Only `prev:output`, nothing else (ground truth in training)
  - LSTM `s` has internal hidden state, but this layer can be calculated independently now
- Automatic optimization → All layers moved out of loop, much faster

Transformer [Vaswani & Shazeer+ 17]:

- Remember: model definition like used at decoding, i.e. auto-regressively, same def. then also used for training
- What recurrency is there in the layers?
  - Only `prev:output`, nothing else (ground truth in training)
  - Self-attention can be interpreted as layer with hidden state during decoding
- Automatic optimization → All layers moved out of loop, much faster
  - Results in the standard training recipe for Transformers

Recurrency: Minimum Expected Risk Training

- Before: `ChoiceLayer` returns ground-truth in training
- This is flexible. Can use beam search during training.

⇒ Simple & flexible to implement all kinds of training variants

- Minimum Expected Risk Training
  - Min. expected WER training [Prabhavalkar & Sainath⁺ 17]
  - Max. expected BLEU training [Edunov & Ott⁺ 17]
- Optimal completion distillation (OCD) [Sabour & Chan⁺ 18]
- Maximum mutual information (MMI) [Michel & Schlüter⁺ 20]
- Scheduled sampling [Bengio & Vinyals⁺ 15]

- Example for min. expected WER training:

```
"encoder": ...,

"output": {"class": "rec", "unit": { ...
    "output_prob": {"class": "softmax", "from": "readout", "target": "classes"},
    'output': {'class': 'choice', 'target': 'classes', 'beam_size': 12, 'from': "output_prob", "initial_output": 0},
} ...},  # [T|'time:var:extern_data:classes',B], int32, dim 1030, beam 'output', beam size 12

"min_wer": {
    "class": "copy", "from": "output",
    "loss": "expected_loss",  # expect beam search results with beam scores
    "target": "classes",
    "loss_opts": {"loss": {"class": "edit_distance"}, "loss_kind": "error"}
},
```

Recurrency: Multiple Stochastic Variables

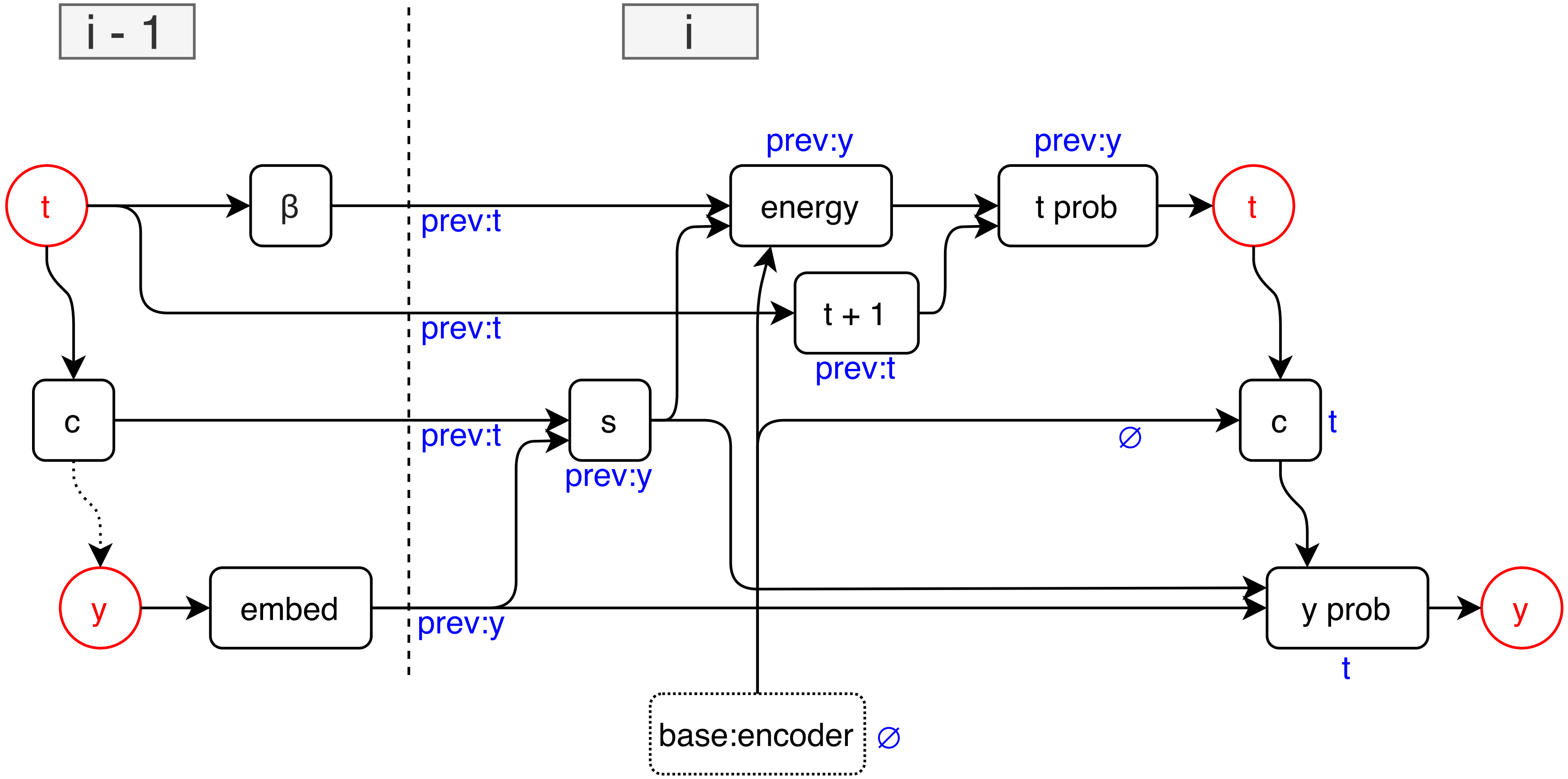
- Example: Hard attention, segmental model:
  - Input seq. x_1^T, output seq. y_1^N
  - New stochastic latent variable t_i

$$
\begin{aligned}
p(y_1^N \mid x_1^T) &= \sum_{t_1^N} p(y_1^N, t_1^N \mid x_1^T) \\
&= \sum_{t_1^N} \prod_{i=1}^{N} p(y_i, t_i \mid y_1^{i-1}, t_1^{i-1}, x_1^T) \\
&= \sum_{t_1^N} \prod_{i=1}^{N} p(y_i \mid y_1^{i-1}, t_1^{i}, x_1^T) \cdot p(t_i \mid y_1^{i-1}, t_1^{i-1}, x_1^T)
\end{aligned}
$$

  - Approximation: $\sum$ can be replaced by $\max$
- Tasks:
  - Forced alignment: Keep y fixed, search over t
  - Free search over both y and t
- RETURNN: One `ChoiceLayer` for y_i (as before), second `ChoiceLayer` for t_i

Recurrency: Multiple Stochastic Variables

Hard Attention Model



- Stochastic vars.: t, y
  - Dependencies:
    $t_{i-1} \to y_{i-1} \to t_i \to y_i$
    (prev:t, prev:y, t, y)
  - Different beams:
    ∅, prev:t, prev:y, t, y
- Need to combine sources from different beams
  - s: prev:t, prev:y
  - energy: ∅, prev:t, prev:y
  - t prob: prev:t, prev:y
  - c: ∅, t
  - y prob: prev:y, t

→ RETURNN `translate_to_common_search_beam`

Part 1: Implementation of Machine Learning for Sequence Processing

Frameworks Overview

Flexibility vs. Efficiency vs. Simplicity

Concepts

Recurrency

Training

- Pre-training
- Custom Training Pipeline
- Regularization
- Multi-GPU Training

Native Operations

Conclusion

Training in RETURNN

- Minimize some loss
  - Defined within the network, attached to a layer, based on the layer output
  - Not used at decoding time
  - Use some predefined losses
    - Cross Entropy
    - Min. expected WER
  - As many losses as you want
  - Loss scales as you want
  - Define any custom calculation as a loss (layer output itself is loss)
- Regularization
- Optimizer: Gradient descent, Adam, ...
- Scheduling (learning rate, regularization, ...)
- Pretraining (supervised, unsupervised, bootstrap alignment, ...)

Pre-training in RETURNN

- Loss / training: Supervised / unsupervised / custom
- Different network topology every epoch, e.g. start with one layer, add more and more
- Automatically copies over parameters from one epoch to the next as far as possible
  - Configurable
  - New weights are newly initialized (e.g. randomly)
  - If dimension increased, can copy over existing weights (grow in width / dim.)

- More generic support:
  - Write custom Python function in config which returns network topology for each pretrain epoch
  - Overwrite anything (loss, hyper params) for each epoch
  - Freeze some parameters (e.g. only train newly added layer)

Custom Training Pipeline in RETURNN

- Generalization of pretraining

- Example:
  1. Train small NN using frame-wise cross-entropy with linear alignment
  2. Calculate new alignment
  3. Train NN using frame-wise cross-entropy with new alignment
  4. Repeat with calculating new alignment (maybe increase NN size)

- Example:
  1. Train CTC model with CTC loss
  2. Calculate new alignment
  3. Train NN (e.g. transducer) using frame-wise cross-entropy with new alignment

Regularization in RETURNN

- Extra loss terms
  - L2
  - Other auxilary losses (supervised or unsupervised)
- Model variations
  - Dropout
  - Variational param noise [Graves 11, Watanabe & Hori+ 17]
  - Stochastic depth [Huang & Sun+ 16]
  - Data augmentation (e.g. SpecAugment) $\equiv$ extra layers on input

- Directly in the network definition
- Used in training only, or flexible / configurable

Training: Multi-GPU Training

- Implementations in RETURNN:
  - Custom engine (Theano)
  - Horovod (TF)
  - Distributed TF
- Dataset distribution
  - Sharded: slice of dataset
    - Load data centralized and distribute it (might be bottleneck)
    - All workers independent: Load all data and take slice (wasted IO, might be bottleneck)
    - All workers independent: Load slice directly (not easily possible)
  - Different random seed
    - All workers independent (no slicing, so no wasted IO)
- Parameter reduction
  - Collect and sum all gradients (needs sync for every update step)
  - Average parameters after N steps (needs sync only every N steps)
    - As fast as the slowest worker
  - Average parameters after T secs
    - Mix different hardware (GPU types) wich different speed
    - Variance in the data results in different speed

Native Operations

Part 1: Implementation of Machine Learning for Sequence Processing

- Frameworks Overview
- Flexibility vs. Efficiency vs. Simplicity
- Concepts
- Recurrency
- Training
- Native Operations
  - Framework
  - LSTM
- Conclusion

Native Operations: Framework

Motivation

- Speed up some important common calculations
  - LSTM [Hochreiter & Schmidhuber 97]
  - CTC loss [Graves & Fernández+ 06]
- Pure TensorFlow implementations can be suboptimal
  - TF ops almost always create copies, even SplitOp etc
    - Not a memory problem, as input tensor will get freed if not used further
    - Performance problem
  - Gradient might be suboptimal
    - Require too much memory (see automatic gradient checkpointing for a solution)
    - No automatic optimization
    - (Could be solved by custom TF gradient)
  - Memory can be too much distributed
    - Esp. problematic in loop: Separate tensor for every iteration
    - Much better to allocate it as consecutive / contiguous block
  - Overhead (calling individual TF ops, etc) (minor compared to the other points)

→ Write native (C++/CUDA) code

Native Code

Why is native code faster?

- Operate inplace on tensors
  - Solves all problems mentioned, no unnecessary copies
  - Can use consecutive tensor / memory
- Enforces custom gradient implementation

Problems with native code:

- Can be difficult, memory unsafe, needs more debugging
- Need multiple implementations: CPU (C++), GPU (CUDA)

Native Operations: Framework

Our Approach in RETURNN

- Some wrapper / helper code to simplify writing custom native op
- Abstractions to allow single code for CPU & GPU
  - Write kernel CUDA style, using `threadIdx`, `blockIdx`, etc
    - Kernel code must be flexible for amount of threads
    - Example, LSTM kernel, loop over dimensions, executed per time-frame:

```
int idx = threadIdx.x + blockDim.x * blockIdx.x;
while (idx < n_cells * n_batch) {
    int batch_idx = idx / n_cells;
    int cell_idx = idx % n_cells;
    ...
    idx += gridDim.x * blockDim.x;
}
```

  - On CPU
    - Custom `gridDim`, `blockDim`
    - Other CUDA-like wrappers

→ `NativeOp` framework
  - Already available for the Theano backend
  - Ported to TensorFlow
    - Directly support for all already prev. implemented ops (LSTM, Baum Welch aligner, ...)
  - Easy to port to other frameworks

RETURNN Native LSTM, CUDA Implementation

- Initial CUDA implementation by Paul Voigtlaender for Theano
- Theano `NativeOp`: general framework to write code both for CPU/GPU
- Theano `NativeLstm` based on `NativeOp` (CPU + GPU)
- TF `TFNativeOp`: porting `NativeOp` to TF, including `NativeLstm`
- `NativeLstm2`: rewrite, faster, more flexible, more options

TF LSTM implementations, can all be used in `RecLayer`:

- TF official: LSTMCell (StandardLSTM), BasicLSTMCell (BasicLSTM)
- TF contrib: LSTMBlockCell, LSTMBlockFusedCell
- TF contrib: CudnnLSTM (GPU only)
- Our natives: `NativeLstm`, `NativeLstm2`

Cell vs. fused:

- ...Cell does a single time-step, needs `tf.while_loop`, input (B,D)
  - input can be recurrent itself, e.g. attention context
- ...FusedCell, Cudnn, NativeLstm operate on the whole sequence
  - whole input sequence must be defined in advance, input (T,B,D)

LSTM Benchmark

- RETURNN native LSTM vs. CuDNN vs. TF LSTMBlock vs. pure TF
- 5 layer BLSTM, 500 dimensionin each direction,
  framewise CE training,
  GeForce GTX 1080 Ti, 8 CPU threads

Device	kernel	time
GPU	NativeLSTM	0:00:05.2728
GPU	CudnnLSTM	0:00:05.3645
GPU	LSTMBlockFused	0:00:09.3915
GPU	LSTMBlock	0:00:15.3071
GPU	StandardLSTM	0:00:17.8279
GPU	BasicLSTM	0:00:22.3976
CPU	NativeLSTM	0:05:09.6268
CPU	LSTMBlockFused	0:07:45.5984
CPU	StandardLSTM	0:08:02.5465
CPU	BasicLSTM	0:08:16.3543
CPU	LSTMBlock	0:08:18.1589

https://returnn.readthedocs.io/en/latest/tf_lstm_benchmark.html

Conclusion

Part 1: Implementation of Machine Learning for Sequence Processing
- Frameworks Overview
- Flexibility vs. Efficiency vs. Simplicity
- Concepts
- Recurrency
- Training
- Native Operations
- Conclusion

Conclusion

How do we get flexibility & simplicity & efficiency?

Important features of RETURNN:

- `Data`, dimension tags (aka. named tensor)
  - → Simpler to read & write code
  - → Simpler debugging
  - → Faster due to most efficient tensor format
- Generic `RecLayer` to define recurrent operations & `ChoiceLayer` (auto-regressive models, iterative ops, ...)
  - → Flexible and simple to define any possible model, operation, decision process (beam search), ...
- Single config / code / model definition for beam search, forwarding, training
  - → Simpler to read & write code, needs less debugging (no inconsistencies)
- Automatic optimization for recurrent definition
  - → Necessary such that single code for training & decoding is efficient
  - → Also more efficient in many other cases
  - → Simpler to read & write code (needs less tricks / simpler code to make it fast)
- Native ops, e.g. LSTM
  - → More efficient

Negative Experiences with RETURNN

Many aspects have historically grown

- Leads to strange / inconsistent APIs
- We try to keep backward compatibility to older setups
  - Might rely on strange behavior on some edge cases
  - Must keep behavior, makes code more complicated
  - Lots of test cases (but this is also a good thing)

- New ideas & concepts for TensorFlow backend
  - No intention to keep Theano and TF backend compatible, to allow for new possibilities
  - `use_tensorflow = True` in config to enable, i.e. old setups stay compatible
  - Only backend interface, datasets and other logic is still shared

  → Can be helpful to introduce new interface, while not breaking old setups

Negative Historically Grown Examples of RETURNN

- `Data`
  - Some of its features have grown:
    - Explicit dimension tags (although we distinguished them before, but more heuristically)
    - Beam information
  - Some design choices turned out not so helpful
    - `shape` is always without batch-dim, `batch_shape` is with (optional) batch-dim
    - Constructor is slightly unintuitive
  - (Complaints on high level. After all, very important feature.)
- Dataset
  - Originally only `HDFDataset`, API derived from single instance, then generalized
    - Next dataset: Total num sequences not known in advance, iterable-style dataset
  - API / behavior of functions was not well defined in all cases
    - Behavior defined later, but not consistent / correct in all cases
  - (Complaints on high level. Very useful concept & powerful API.)

Negative Historically Grown Examples of RETURNN (II)

- Automatic `Data` inference tricky with recursive dependencies
  - Example:

```
{"a": {"class": "linear", "activation": "relu", "from": "prev:b", "n_out": 10},
 "b": {"class": "linear", "activation": "relu", "from": "prev:a", "n_out": 20},
  "output": {"class": "copy", "from": "a"}}
```

  - Feature-dims are clear here (a: 10, b: 20)
  - What other axes/dimensions are there?
  - What DType?
  - (Maybe mostly resolved now.)
- `SelfAttentionLayer` too specific
  - Current options: `n_out`, `num_heads`, `total_key_dim`, `key_shift`, `forward_weights_init`, `attention_dropout`, `attention_left_only`, `initial_state`, `restrict_state_to_last_seq`, `state_var_lengths`
  - Too many already, and people want to add more for their special need, have their own forks
  - Some of these are not obvious, need to look at documentation & code to understand
  - Solution: Split up into more atomic buildings blocks

Future Plans and Ideas for RETURNN

- New more PyTorch / Keras like Python syntax instead of defining net dict
  - Just syntactic but might be more familiar, also auto-completion support by IDEs
  - Could introduce more clean behavior, solve some mentioned problems
    - Change some inconsistent names or defaults (or remove bad defaults)
    - Enforce more consistent explicit dimension tag usage
- Easy mechanism to reuse common parts (e.g. Transformer, SpecAugment, ...)
  - Not supposed to be part of main framework (yet), as they are under active research
  - Still common enough that you easily want to use them
  - One solution: E.g. imports like in Go, from other Git repos, fixed version
- More support for RETURNN as a framework
  - Usage similar to PyTorch or Keras, or maybe PyTorch Lightning
- More dynamic support for reinforcement learning
  - Interactive environment, extended dataset
  - Replay buffer / generic auxiliary database

Part 2: Specific Models & Applications

Introduction

Part 2: Specific Models & Applications

- Introduction
- Machine translation (RNN/Transformer-based encoder-decoder-attention)
- Speech recognition (Hybrid HMM, Attention, other end-to-end approaches)
- Language modeling (RNN/LSTM, Transformer)
- End-to-end speech translation
- Text-to-speech

Purpose of this Part

- Machine learning implementation using RETURNN as exemplary case
- Discuss application oriented features
  - Machine translation (MT)
  - Automatic speech recognition (ASR)
  - Language modeling (LM)
  - Speech to text translation (ST)
  - Text to speech (TTS)
- Examples of different methods and the ease of their configuration
- Comparison to other frameworks

Part 2: Specific Models & Applications

Introduction

Machine translation (RNN/Transformer-based encoder-decoder-attention)

- Recurrent Attention Model
- Attention Extension
- Layer-wise Pretraining
- Transformer Model
- Hybrid NMT

Speech recognition (Hybrid HMM, Attention, other end-to-end approaches)

Language modeling (RNN/LSTM, Transformer)

End-to-end speech translation

Text-to-speech

Introduction

Machine translation = translating text from one natural language into another

- Utilize neural networks, encoder-decoder attention architecture
  - Recurrent neural networks (RNNs)/Long Short-Term Memory (LSTM) [Bahdanau & Cho+ 15]
  - Self-attention [Vaswani & Shazeer+ 17]
  - Convolution [Gehring & Auli+ 17]
  - Combination of networks [Chen & Firat+ 18]
- From phrase based system to neural based approaches in the production
- Frameworks:
  - Tensor2Tensor [Vaswani & Bengio+ 18]
  - Sockeye [Hieber & Domhan+ 17, Domhan & Denkowski+ 20]
  - Fairseq [Ott & Edunov+ 19]
  - OpenNMT [Klein & Kim+ 17]
  - ...

MT

What MT requires from a framework:

- Flexibility, simplicity, scalability and extensibility as mentioned before
- Easy architecture setup to enable different models
  - Diverse architectures, recurrent, convolution, Transformer
  - Easy combination of provided building blocks (encoder, decoder, losses)
  - Different attention components, different positional encodings, etc.
- Data pipeline, preprocessing e.g. filtering, tokenization as well as postprocessing
- Model units: subwords, sentence pieces
- Fine-tuning, resume training
- Attention visualization for better explainability
- Generating synthetic data, back-translation
- ...

MT: Recurrent Attention Model

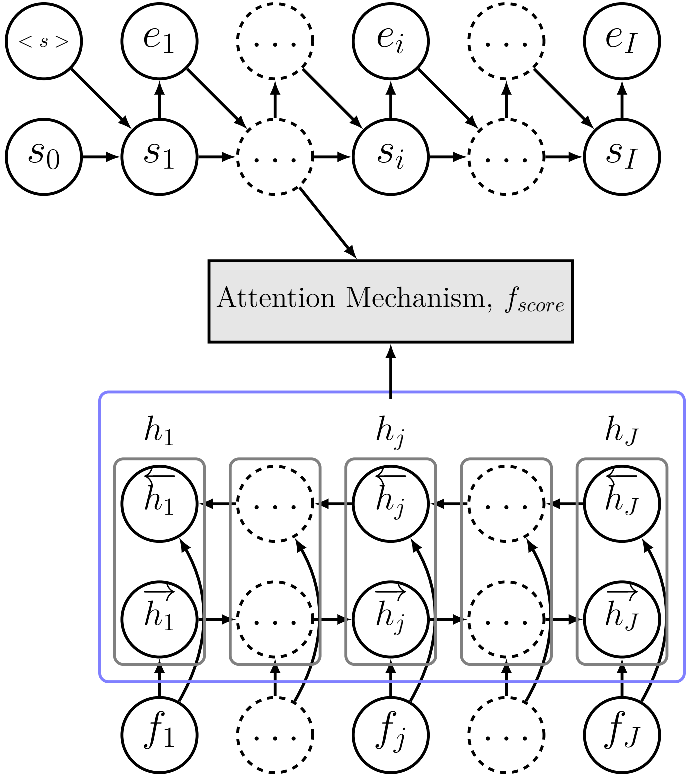
- Source sentence of J words f_1^J, target sentence of I words e_1^I
- Bidirectional LSTM in the encoder h_j, unidirectional LSTM in the decoder s_i

  - Unnormalized weights: $\alpha'_{i,j} = f_{score}(K, Q) = f_{score}(h_j, s_{i-1})$
    $= v^\top \tanh(W[h_j, s_{i-1}, e_{i-1}]) \in \mathbb{R}$
  - Attention weights: $\alpha_{i,j} = \text{softmax}_t(\alpha'_{i,j}) \in \mathbb{R}^J$
  - Weighted average: $c_i = \Sigma_{j=1}^J \alpha_{i,j} h_j$
  - Decoder state:

    $$s_i = \text{LSTM}\left(s_{i-1}, e_i, c_i\right) \quad \text{or}$$
    $$s_i = \text{LSTM}\left(s_{i-1}, e_{i-1}, c_i\right)$$

  - Probability distribution:

    $$p(e_i|e_1^{i-1}, f_1^J) = \text{softmax}\left(\text{linear}(s_{i-1}, e_{i-1}, c_i)\right) \quad \text{or}$$
    $$p(e_i|e_1^{i-1}, f_1^J) = \text{softmax}\left(\text{linear}(s_i, e_{i-1}, c_i)\right)$$



Network Construction

```
network = {
 # encoder
"source_embed": {"class": "linear", "activation": None, "with_bias": False, "n_out": 512},
"lstm0_fw" : { "class": "rec", "unit": "nativelstm2", "n_out" : 1024, "direction": 1, "from": ["source_embed"], "dropout": 0.3},
"lstm0_bw" : { "class": "rec", "unit": "nativelstm2", "n_out" : 1024, "direction": -1, "from": ["source_embed"], "dropout": 0.3},
"encoder": {"class": "copy", "from": ["lstm0_fw", "lstm0_bw"]}, # dim: 2048
"enc_ctx": {"class": "linear", "activation": None, "with_bias": True, "from": ["encoder"], "n_out": 1024},
 # decoder using recLayer
"output": {"class": "rec", "from": [], "unit": {
        "output": {"class": "choice", "target": "classes", "beam_size": 12, "from": ["output_prob"], "initial_output": 0},
        "end": {"class": "compare", "from": ["output"], "value": 0}, #end-of-sentence condition
        "target_embed": {"class": "linear", "activation": None, "with_bias": False, "from": ["output"], "n_out": 512, "initial_output": 0},
        "prev_s_transformed": {"class": "linear", "activation": None, "with_bias": False, "from":  ["prev:s"], "n_out": 1024, "dropout": 0.3},
         # additive attention mechanism
        "energy_in": {"class": "combine", "kind": "add", "from":  ["base:enc_ctx", "prev_s_transformed"], "n_out": 1024},
        "energy_tanh": {"class": "activation", "activation": "tanh", "from": ["energy_in"]},
        "energy": {"class": "linear", "activation": None, "with_bias": False, "from": ["energy_tanh"], "n_out": 1},
        "att_weights": {"class": "softmax_over_spatial", "from": ["energy"]},
        "context_att": {"class": "generic_attention", "weights": "att_weights", "base": "base:enc_ctx"},
        "s": {"class": "rnn_cell", "unit": "LSTMBlock", "from":  ["target_embed", "context_att"], "n_out": 1024, "dropout": 0.3},
        "readout": {"class": "linear", "from":  ["prev:s", "prev:target_embed", "context_att"], "activation": None, "n_out": 1024, "dropout": 0.3},
         # output layer using crossentropy loss
        "output_prob": {"class": "softmax", "from": ["readout"], "dropout": 0.3,
"target": "classes", "loss": "ce", "loss_opts": {"label_smoothing": 0.1}}
}, "target": "classes", "max_seq_len": "max_len_from("base:encoder")*3"},}
```

MT: Recurrent Attention Model

Network Construction

- First update decoder state, then predict output word

```
network = {
... # encoder the same
"output": {"class": "rec", "from": [], "unit": {
    ...
    "prev_s_transformed": {"class": "linear", "activation": None, "with_bias": False, "from": ["prev:s"], "n_out": 1024, "dropout": 0.3},
    "energy_in": {"class": "combine", "kind": "add", "from": ["base:enc_ctx","prev_s_transformed", "prev:target_embed"], "n_out": 1024},
    ...
    "s": {"class": "rnn_cell", "unit": "LSTMBlock", "from": ["prev:target_embed", "context_att"], "n_out": 1024, "dropout": 0.3},
    "readout": {"class": "linear", "from": ["s", "prev:target_embed", "context_att"], "activation": None, "n_out": 1024, "dropout": 0.3},
```

MT: Attention Extension

- Easy construction via config file
- Extensions over previous attention weights [Tu & Lu+ 16, Cohn & Hoang+ 16]
- Past alignments information as coverage β

$$\alpha'_{i,j} = v^\top \tanh\left(h_j, s_{i-1}, \beta\right)$$

- E.g. fertility concept with $N = 2$, $\beta = \frac{\text{sigmoid}(v_\phi^\top h_j)}{N} \Sigma_{k=1}^{i-1} \alpha_{i,j}$

```
network = {
...
"inv_fertility": {"class": "linear", "activation": "sigmoid", "with_bias": False, "from": ["encoder"], "n_out": 1},

"output": {"class": "rec", "from": [], "unit": {
        ...
         "weight_feedback": {"class": "linear", "activation": None, "with_bias": False, "from": ["prev:accum_att_weights"], "n_out": 1024, "dropout": 0.3},

        "energy_in": {"class": "combine", "kind": "add", "from": ["base:enc_ctx", "prev_s_transformed",  "weight_feedback"], "n_out": 1},
        ...
         "accum_att_weights": {"class": "eval", "from": ["prev:accum_att_weights", "att_weights", "base:inv_fertility"],
        "eval": "(source(0)+ source(1))*source(2)*0.5", "out_type": {"dim": 1, "shape": (None, 1)}},
        ...
```

MT: Attention Extension

- Past alignments information as a linguistic coverage β

$$\alpha'_{i,j} = v_a^\top \tanh\left(s_{i-1}, h_j, \beta\right)$$

- E.g. past context vector

$$\beta = c_{i-1}$$

```
"context_att_transformed": {"class": "linear", "activation": None, "with_bias": False, "from": ["prev:context_att"], "n_out": 1024},
"energy_in": {"class": "combine", "kind": "add", "from": ["base:enc_ctx", "prev_s_transformed",  "context_att_transformed"], "n_out": 1024},
```

- E.g. past context vector in the decoder

$$s_i = \mathrm{LSTM}\left(s_{i-1}, e_i, c_i, c_{i-1}\right)$$

```
"context_att_transformed": {"class": "linear", "activation": None, "with_bias": False, "from": ["prev:context_att"], "n_out": 1024, "dropout": 0.3},
"s": {"class": "rnn_cell", "unit": "LSTMBlock", "from": ["target_embed", "context_att",  "context_att_transformed"], "n_out": 1024},
"readout": {"class": "linear", "from": ["prev:s", "prev:target_embed", "context_att",  "context_att_transformed"], "activation": None, "n_out": 1024},
```

MT: Attention Extension

Other Attention Types

- Multiplicative: $f_{score}(Q, K) = K^\top W Q = h_j^\top W s_{i-1}$

```
"energy": {"class": "dot", "red1": −1, "red2": −1, "var1": "T", "var2": "T?", "from": ['base:enc_ctx', 'prev_s_transformed']},
"att_weights": {"class": "softmax_over_spatial", "from": ['energy'],"energy_factor": 1024 ** −0.5},
```

- Multi-head additive attention instead of single-head attention [Chen & et al 18]
- Both additive and multiplicative attention
- `SplitDimsLayer` to split a vector
- `MergeDimsLayer` to flatten a dimension

```
AttNumHeads = 8
network = {
...
"enc_value": {"class": "split_dims", "axis":"F","dims":( AttNumHeads, 1024), "from":["encoder"]},

"output":{"class":"rec","from":[],"unit":{
...
"energy":{"class":"linear","activation":None,"with_bias":False,"from": ["energy_tanh"],"n_out":( AttNumHeads),
"att_weights": {"class":"softmax_over_spatial","from": ["energy"]},  # (B, J, H)
"context_att0": {"class": "generic_attention", "weights": "att_weights", "base": "base:enc_value"},  # (B, H, V)
"context_att": {"class": ( "merge_dims", "axes": "except_batch", "from": ["context_att0"]},  # (B, H*V)
...
```

MT: Attention Extension

Attention Extension in Other Frameworks

Sockeye NMT

- More specific to MT task via options
  - `-rnn-attention-type`: attention types
  - `-rnn-attention-use-prev-word`: include the previous target token in attention calculation
  - `-rnn-attention-coverage-type`: model for updating coverage vectors and accumulate attention scores
  - `-rnn-attention-in-upper-layers`: pass the attention to the upper layers of the RNN decoder, similar to GNMT paper
  - `-rnn-attention-mhdot-heads`: enable multi-head dot attention inside RNN model
- Further extensions need to directly change the code

Fairseq

- More specific than RETURNN
- Many predefined architectures with hyperparameters, pretrained models
- Either use pre-written modules or extend it in the code
- Modification in `class AttentionLayer` and `class LSTMDecoder`

MT: Layer-wise Pretraining

- Faster convergence for deep networks, proven to help MT a lot [Zeyer & Alkhouli+ 18]
- Iteratively starts with a small model and adds new layers in the process
- Using `pretrain` with number of iterations `repetitions`
- Customize your own `construction_algo`

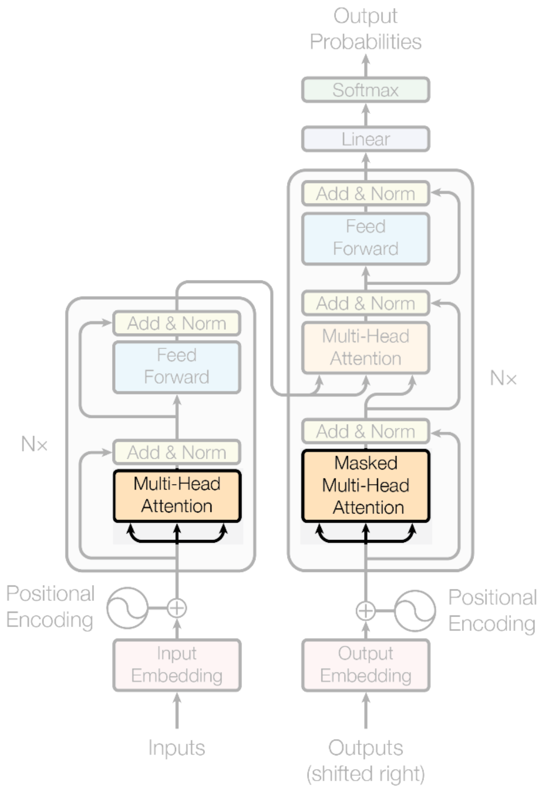
```
def custom_algo(idx, net_dict):
    orig_num_lstm_layers = 0
    while "lstm%i_fw" % orig_num_lstm_layers in net_dict:
        orig_num_lstm_layers += 1
    num_lstm_layers = idx + 1  # idx starts at 0. start with 1 layer
    if num_lstm_layers == orig_num_lstm_layers:
        return net_dict
    if num_lstm_layers >= orig_num_lstm_layers:
        return None
    # Leave the last layer as-is, but only modify its source.
    net_dict["encoder"]["from"] = ["lstm%i_fw" % (num_lstm_layers - 1), "lstm%i_bw" % (num_lstm_layers - 1)]
    return net_dict

pretrain = {"repetitions": 5, "construction_algo": custom_algo}
```

Self-Attention Blocks

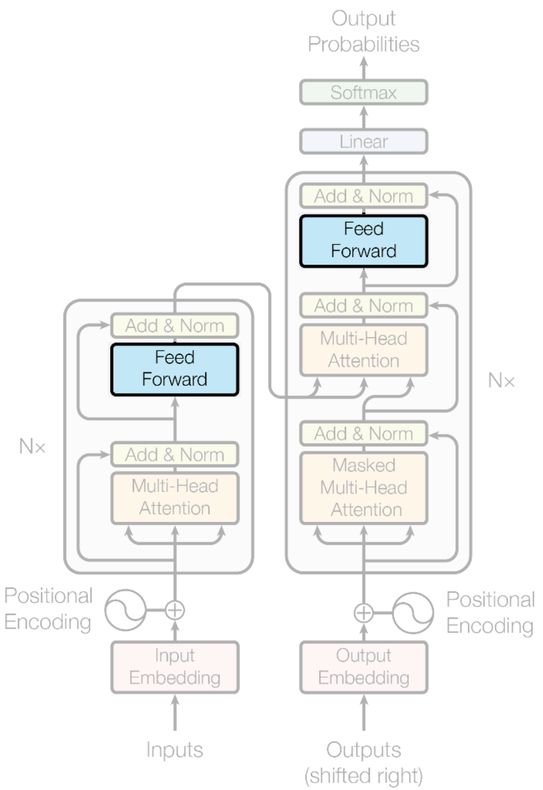
Three different inputs to attention block: key, value and query

- Self-attention in encoder
  - Input and output: sequence of features of shape $[B\times]\ T \times F$
  - This sequence is used as keys, values and queries: $K = V = Q$
- Masked self-attention in decoder
  - In decoder key and value positions right from the query positions have to be neglected (auto-regressive model)
    - Practical implementation: corresponding positions are set to $-\infty$ right before softmax operation
- In RETURNN:
  - Multi-headed self-attention: `"class": "self_attention"`
  - This layer also works within the recurrent sublayer (`class: "rec"`)
  - In training it is optimized to work fully parallel and not step-by-step



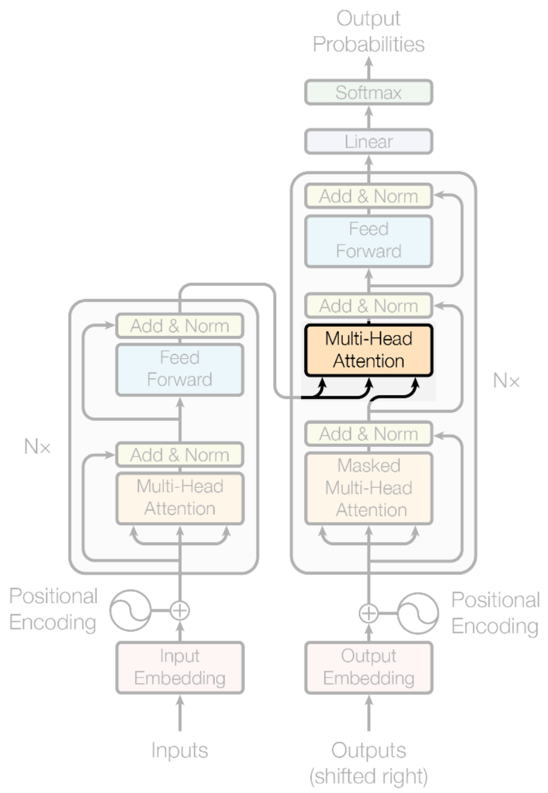
Feed-Forward Blocks

- FF$(x) = \max(0, xW_1 + b_1)W_2 + b_2$
- In RETURNN:
  - Just two linear layers: `"class": "linear"`
    - First with `"activation": "relu"`
    - Second with `"activation": None`



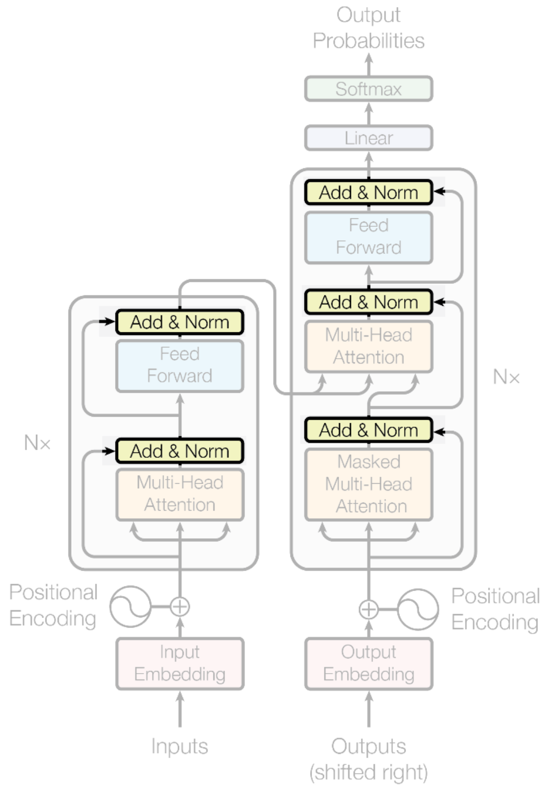
Encoder Decoder Attention Block

- Keys and values $(K = V)$ are the encoder output of the last layer
- Queries Q come from corresponding decoder layer
- In RETURNN:
  - In contrast to the multi-head self-attention the multi-head attention between encoder and decoder has to be built from more basic layers
    - `"class": "linear"`
    - `"class": "split_dims"`
    - `"class": "merge_dims"`
    - `"class": "dot"`
    - `"class": "softmax_over_spatial"`
    - `"class": "dropout"`
    - `"class": "generic_attention"`
  - The layers transforming the encoder outputs $(K = V)$ should be put outside the recurrent subnetwork



Preprocessing and Postprocessing Blocks

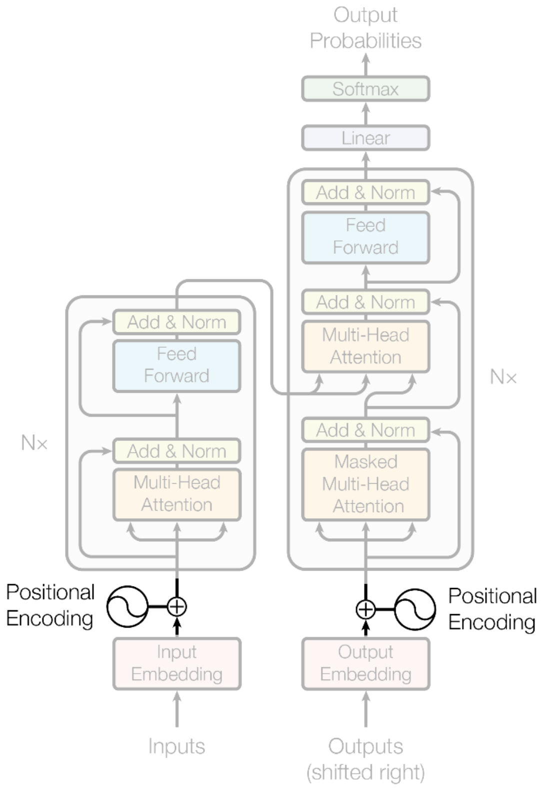
- Preprocessing step: layer normalization
- Postprocessing step: dropout and residual connection
- In RETURNN:
  - Layer normalization: `"class": "layer_norm"`
  - Dropout: `"class": "dropout"`
  - Residual connection: `"class": "combine"` with `"kind": "add"`
- Tensor2Tensor implementations
  - Preprocessing step: layer normalization
  - Postprocessing step: dropout and residual connection
- Fairseq implementations
  - Postprocessing step: dropout, residual connection and layer normalization
  - Tensor2Tensor setup activation: `encoder_normalize_before`
  - Different order needs code change, however easy



Positional Encoding

- Positional encoding is crucial for MT task
- An extension with relative positional encoding
  - Resolve the sequence length problem [Rosendahl & Tran+ 19]
- in RETURNN:
  - `"class": "positional_encoding"`
  - `"class": "relative_positional_encoding"`

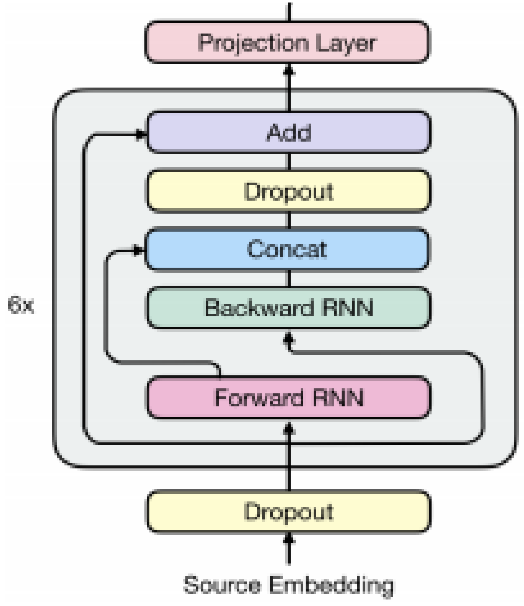
```
d[output + '_rel_pos'] = {
    "class": "relative_positional_encoding",
    "from": [output + '_att_laynorm'],
    "n_out": 512 // 8,
    "forward_weights_init": ff_init}
d[output + '_att_att'] = {
    "class": "_attention",
    "num_heads": 8,
    "total_key_dim": 512,
    "n_out": 512, "from": [output + '_att_laynorm'],
    "attention_left_only": False, "attention_dropout": 0.1,
    "forward_weights_init": ff_init,  "key_shift": output + '_rel_pos'}
```



RNMT+ Network Configuration - Encoder

- Transformer model typically trains faster while slower in search
- Hybrid NMT, e.g. combination of RNN and Transformer [Chen & Firat+ 18]
  - Receive information from different channels
  - Best setup: Transformer encoder, RNN decoder equipped with layernorm

```
# layer 0 — wrap the layer as a function to call it N times
"source_embed": {"class": "linear", "activation": None, "with_bias": False, "n_out": 1024},
"source_embed_drop": {"class": "dropout", "from": ["source_embed"], "dropout": 0.1},
"lstm0_fw" : { "class": "rec", "unit": "nativelstm2", "n_out" : 1024, "direction": 1, "from": ["source_embed_drop"]},
"lstm0_bw" : { "class": "rec", "unit": "nativelstm2", "n_out" : 1024, "direction": -1, "from": ["source_embed_drop"]},
"concat0": {"class": "copy", "from": ["lstm0_fw", "lstm0_bw"]},
"concat0_drop": {"class": "dropout", "from": ["concat0"], "dropout": 0.1},
"res0" : {"class": "combine", "kind": "add", "from": ["concat0_drop", "source_embed_drop"], "n_out": 1024},
"project_res0": {"class": "linear", "activation": None, "with_bias": True, "from": ["res0"], "n_out": 1024},
```



Hybrid NMT in Other Frameworks

Sockeye NMT

- `-encoder`
  - Type of encoder
  - `rnn, rnn-with-conv-embed, transformer, transformer-with-conv-embed, cnn`
- `-decoder`
  - Type of decoder
  - `rnn, transformer,cnn`
- Further modifications require direct changes to the code and therefore is deemed not flexible

Fairseq

- Slight modification to the code via `transformer.py` or `lstm.py`
- Instead use `class LSTMEncoder` or `LSTMDecoder`

Part 2: Specific Models & Applications
Introduction
Machine translation (RNN/Transformer-based encoder-decoder-attention)
Speech recognition (Hybrid HMM, Attention, other end-to-end approaches)
Hybrid HMM-NN & CTC (on phonemes)
Attention-based encoder-decoder models
Pretraining
Transducer Models
Language modeling (RNN/LSTM, Transformer)
End-to-end speech translation
Text-to-speech

Automatic Speech Recognition (ASR)

Task: Map **speech sequence** to corresponding **word sequence**

Evaluation: Word/Character Error Rate (WER/CER), latency, speed

Models:

- Hybrid HMM
- CTC/ASG, Lattice-free MMI
- Encoder-decoder attention approaches
- Transducer approaches
- Many more...

Frameworks:

- Mozilla DeepSpeech (15.2k Stars): CTC
- JHU ESPnet (2.9k Stars): CTC/Attention, Transducer, many more
- JHU Kaldi (9.5k Stars): GMM/DNN-HMM, seq. discr. training
- Facebook Wav2Letter++ (5.4k Stars): seq2seq with conv-nets
- Sequence processing frameworks discussed before

ASR

How a framework can help researchers, especially for ASR:

- Enable fast iteration speed (mentioned in first part: flexibility, simplicity, *etc.*)
- Custom user configuration/code:
  - Researchers can try out different ideas without touching stable code
  - User code can be merged back into main branch
- Provide building blocks (encoder, decoder, losses, beam search)

Important aspects of ASR:

- Input features: MFCC, log-Mel, Gammatone, *etc.*
- Target features: Sub-word units, characters, phonemes
- Frontend, before encoder: convolutional layers, data augmentation (e.g. SpecAugment [Park & Chan^{+} 19])
- Acoustic Model
- Language Model
  - How to integrate during training/recognition? (fusion techniques)

→ Dependency between all parts!

Model aspects

- Network size w.r.t. parameters
  - Mobile/Embedded or Server?
  - Quantization, teacher-student
- Recurrency ($\rightarrow$ Speed, Parallelizability)

Online capability:

- Model
  - For BLSTM:
    - Latency-controlled BLSTM (LC-BLSTM)
    - Frame-chunking
    - Unidirectional LSTM
  - For Transformer
    - Local context self-attention [Povey & Hadian$^+$ 18]
    - Block processing [Dong & Wang$^+$ 19, Tsunoo & Kashiwagi$^+$ 19]
  - For attention/decoding: hard attention, MoChA[Chiu* & Raffel* 18], CTC-synchronous, triggered attention ...
- Framework support for production setting

Models overview

Acoustic model / Encoder

- LSTM stack (bi-/unidirectional, GRU, variants)
- Transformer (maybe plus an LSTM stack)
- Convolutional (+variants)

Sequence model / Decoder

- Hybrid HMM-NN / CTC
- Attention-based encoder-decoder
  - Global soft attention [Bahdanau & Cho+ 15]
  - Self-attention (cf. Transformer) [Vaswani & Shazeer+ 17]
  - Local soft attention [Luong & Pham+ 15, Hou & Zhang+ 17, Merboldt & Zeyer+ 19]
  - Hard attention variants (MoChA, Latent attention [Bahar & Makarov+ 20])
- Transducer
  - Recurrent Neural Aligner (RNA)
  - Recurrent Neural Transducer (RNN-T)
  - Generalized variant [Zeyer & Merboldt+ 20]
- New approach?

RWTH ASR (RASR), relevant features for hybrid HMM-NN

- RASR is the RWTH speech recognition toolkit [Rybach & Hahn^{+} 11]
- HMM-GMM
- Forced aligner
- Context-dep. phone clustering (CART)
- Complex decoder / search
- Python APIs/interfaces for different tasks:
  - Various feature extraction modules (MFCC, Gammatone, ...)
  - Discriminative sequence training (MMI / MPE)
  - Hybrid HMM-NN (allows for plug-in any framework)
  - TensorFlow C++: load graph and checkpoints
- Much more ...

ASR: Hybrid HMM-NN & CTC (on phonemes)

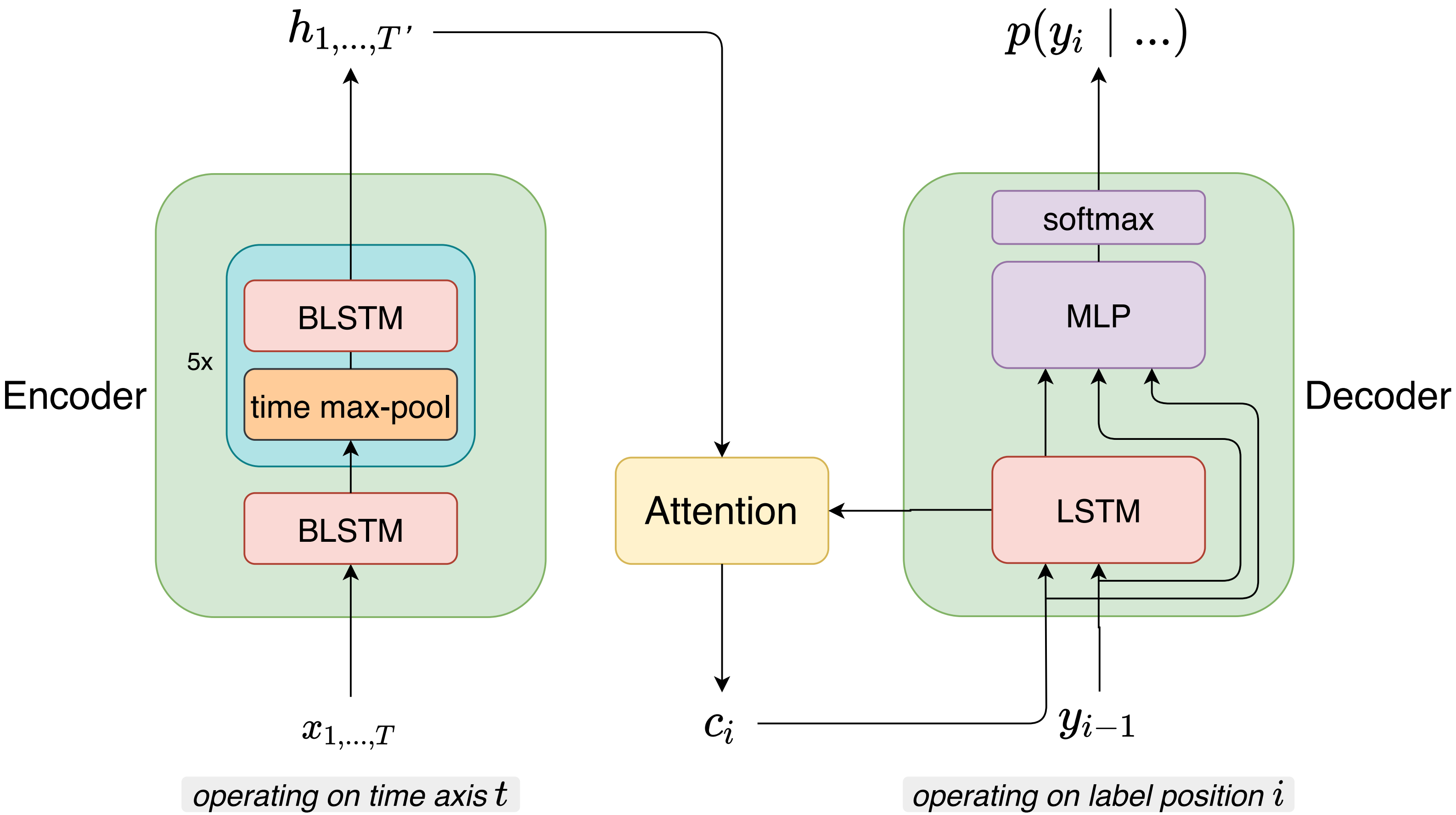
RETURNN ↔ RASR

RETURNN has many interfaces to RASR

- Feature extraction / dataset loading (including forced alignment) (`ExternSprintDataset`)
  - Used for frame-wise CE training of hybrid HMM-NN
- RASR decoding / search for recognition
  - RETURNN for acoustic model NN
    - Via Python interface (`SprintInterface`)
    - Or: Compile and store TF graph and use that directly (`compile-tf-graph` script)
  - RETURNN for language model NN
    - Compile and store TF graph and use in first-pass decoding (`compile-tf-graph` script)
- Discriminative sequence training
  - Generic interface, RASR gets posteriors, calculates loss & gradient (error signal)
- Extract finite state automaton (FSA) using pronunciation lexicon
  - Full-sum / CTC training (`FastBaumWelch` in RETURNN, accepts any FSA)

ASR: Attention-based encoder-decoder models

Model



ASR: Attention-based encoder-decoder models

Training and recognition

- During training: minimize

$$L := -\log p(y_1^N | x_1^T, \theta) = -\sum_{i=1}^{N} \log p(y_i | y_1^{i-1}, x_1^T)$$

  w.r.t. model parameters θ.
  - x, y input/target sequence from dataset
  - $\rightarrow$ Cross entropy

In recognition:

- Given is only x, θ.
- Find y such that we maximize the log-posterior $\log p(y_1^N | x_1^T, \theta)$
- $\rightarrow$ Beam search
  - Label synchronous
  - Usually fixed, small beam size compared to hybrid models

RETURNN: Single config, straight-forward, using `ChoiceLayer` for y_i

ASR: Pretraining

Training

Improve convergence:

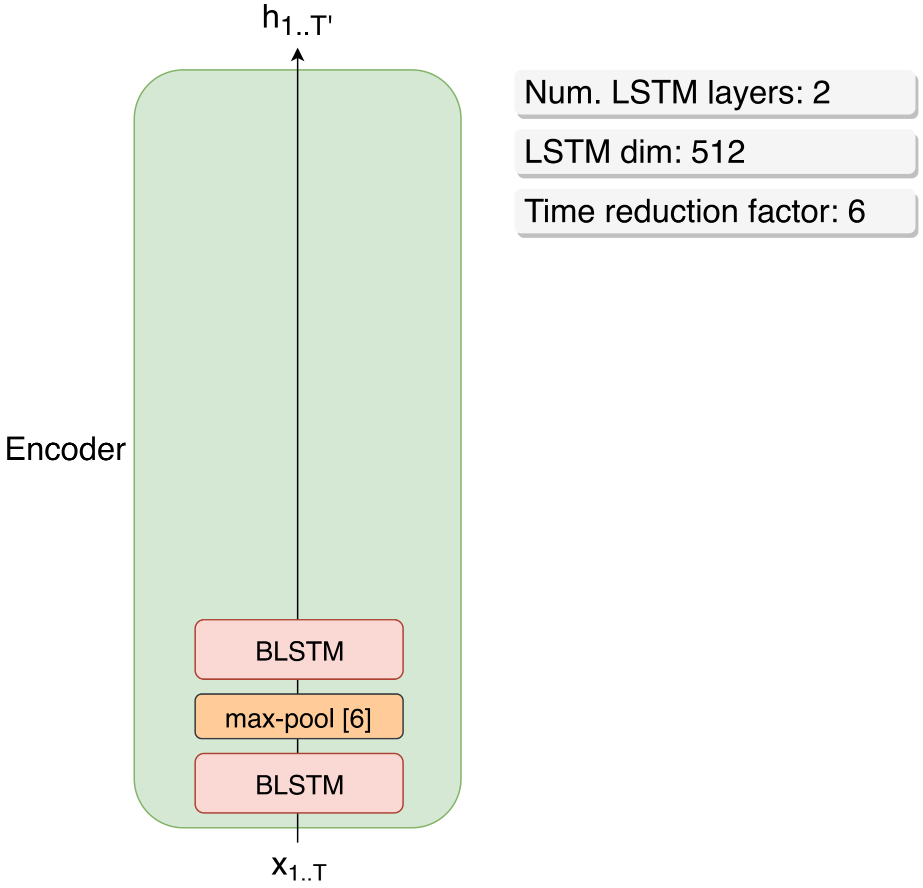
- Pretraining
  - Grow encoder/decoder
  - Learning rate warmup+scheduling
  - Initialize parameters from different model
- Curriculum learning: easier part of training first

→ How can a toolkit/framework support this?

- Provide recipes/configs/examples for training on specific datasets
  - Good starting point
- High flexibility

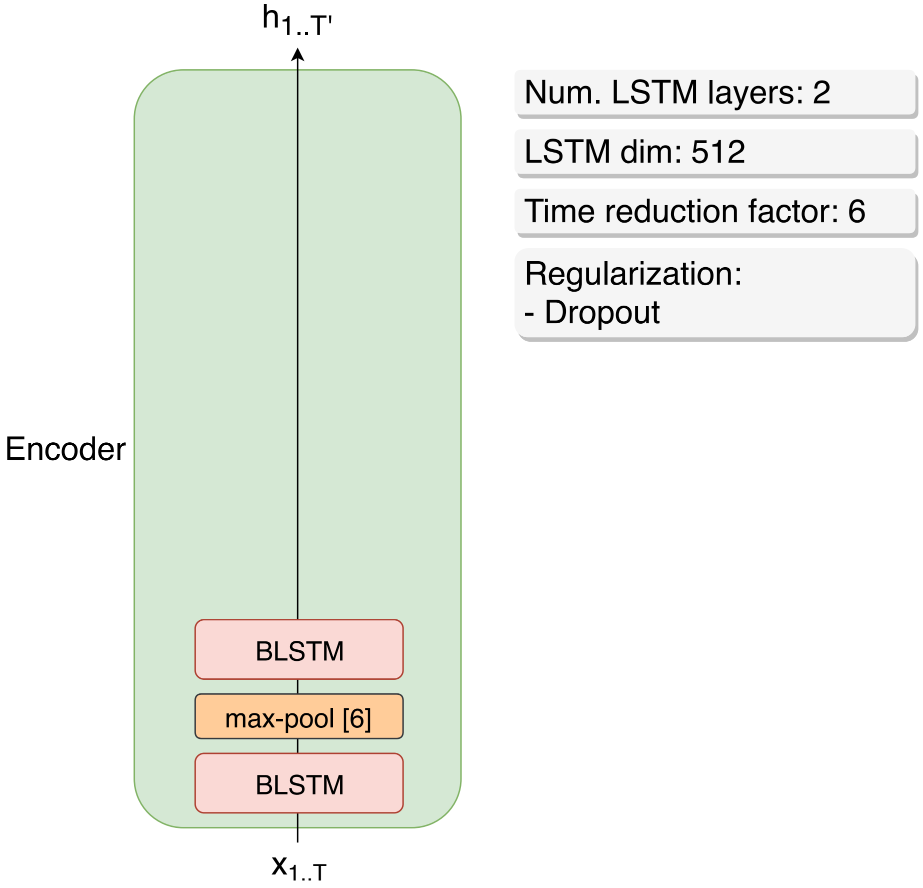
Pretraining scheme

Improved ASR pretraining scheme for the encoder [Zeyer & Merboldt[+] 18]



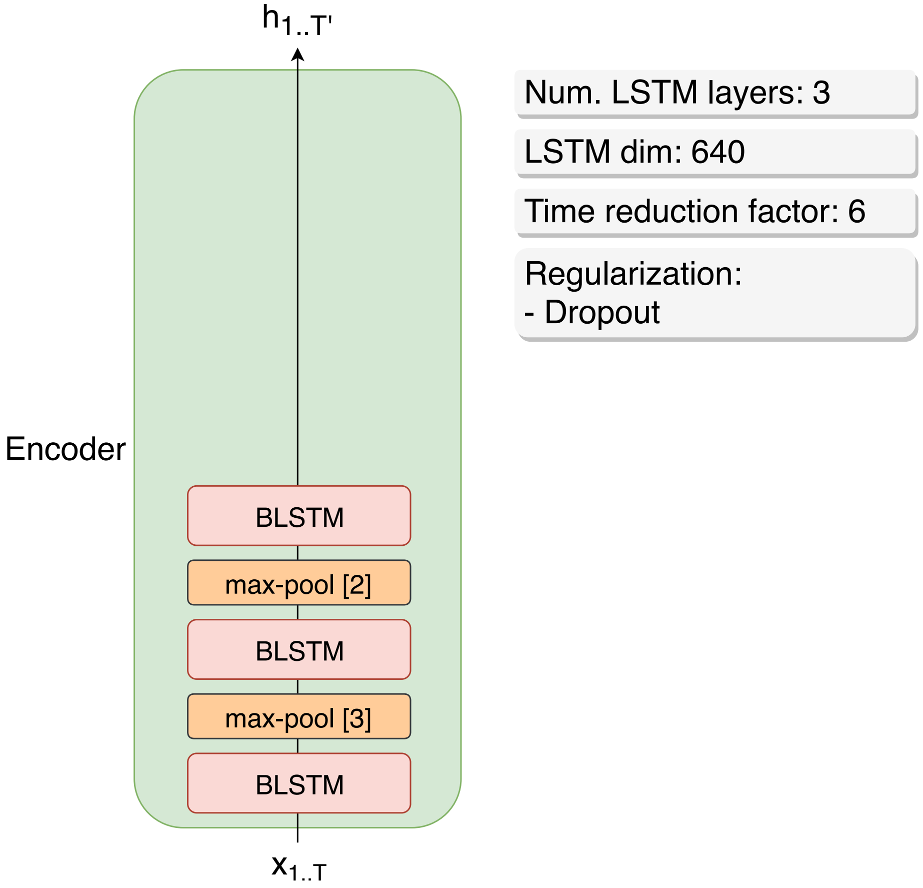
Pretraining scheme

Improved ASR pretraining scheme for the encoder [Zeyer & Merboldt[+] 18]



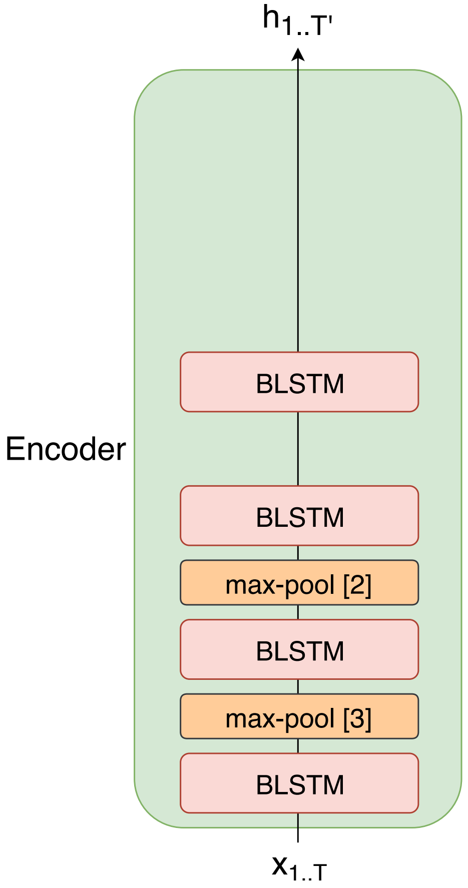
Pretraining scheme

Improved ASR pretraining scheme for the encoder [Zeyer & Merboldt+ 18]



Pretraining scheme

Improved ASR pretraining scheme for the encoder [Zeyer & Merboldt+ 18]



Num. LSTM layers: 4

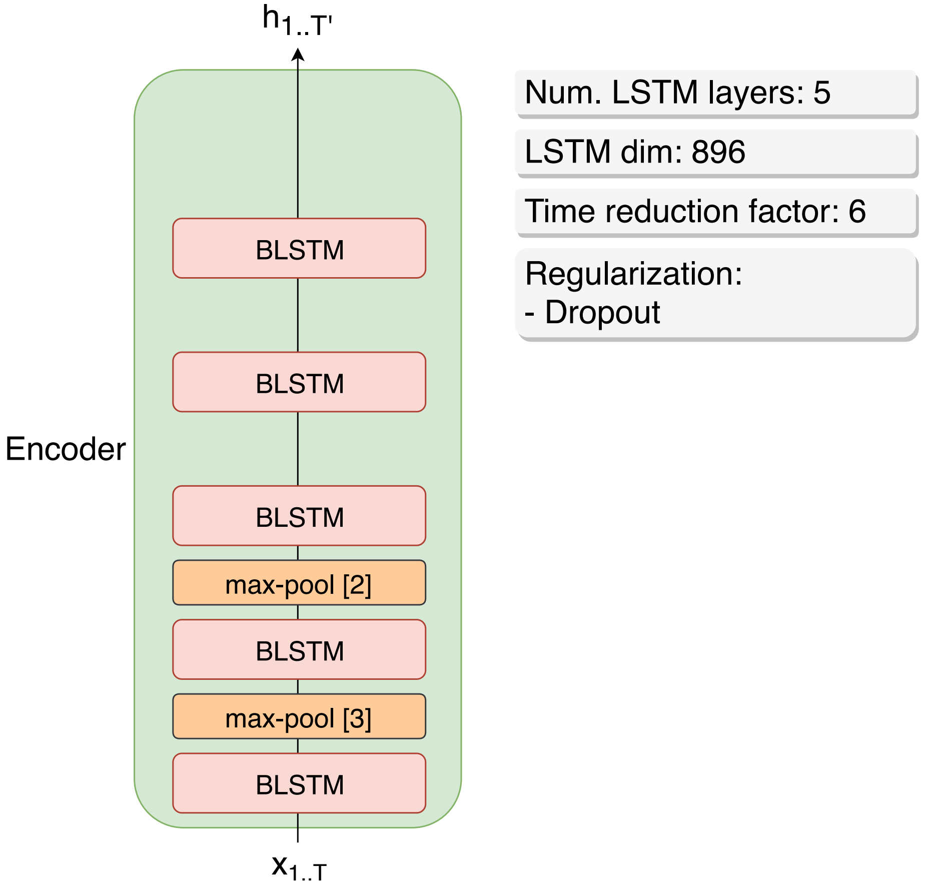
LSTM dim: 768

Time reduction factor: 6

Regularization:
- Dropout

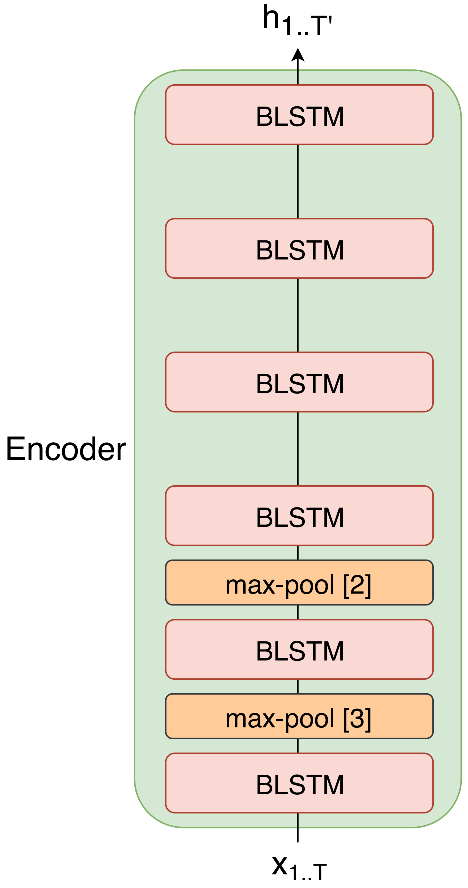
Pretraining scheme

Improved ASR pretraining scheme for the encoder [Zeyer & Merboldt+ 18]



Pretraining scheme

Improved ASR pretraining scheme for the encoder [Zeyer & Merboldt[+] 18]



Num. LSTM layers: 6

LSTM dim: 1024

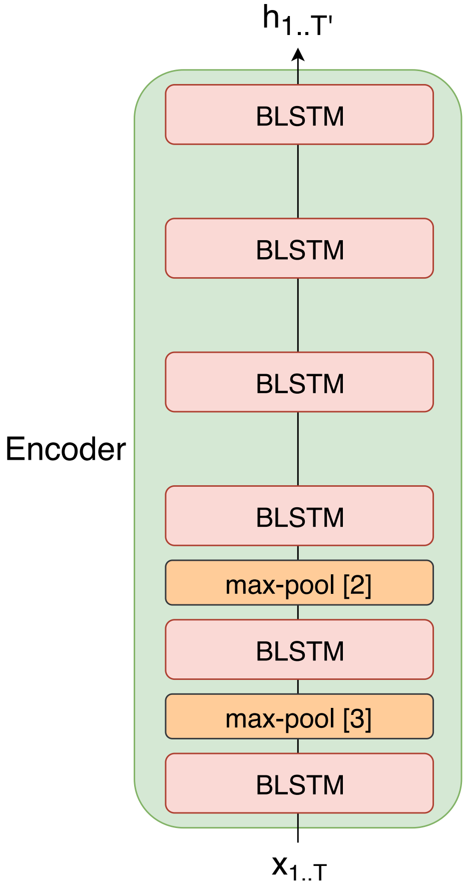
Time reduction factor: 6

Regularization:
- Dropout

ASR: Pretraining

Pretraining scheme

Improved ASR pretraining scheme for the encoder [Zeyer & Merboldt+ 18]



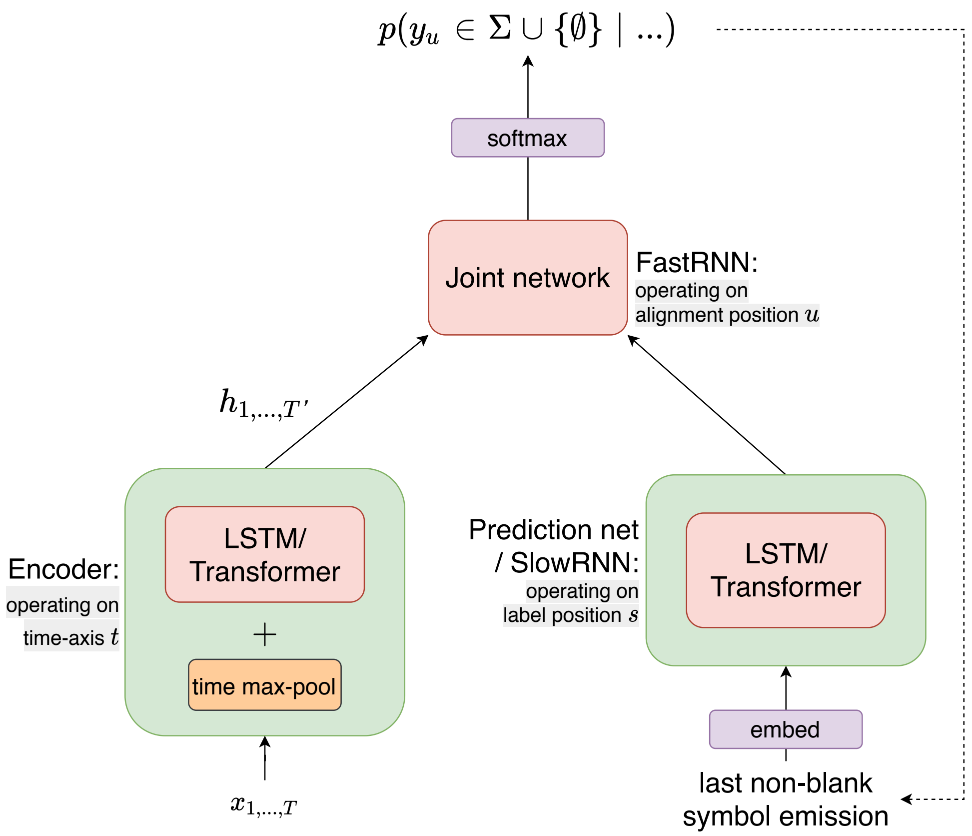
Num. LSTM layers: 6

LSTM dim: 1024

Time reduction factor: 6

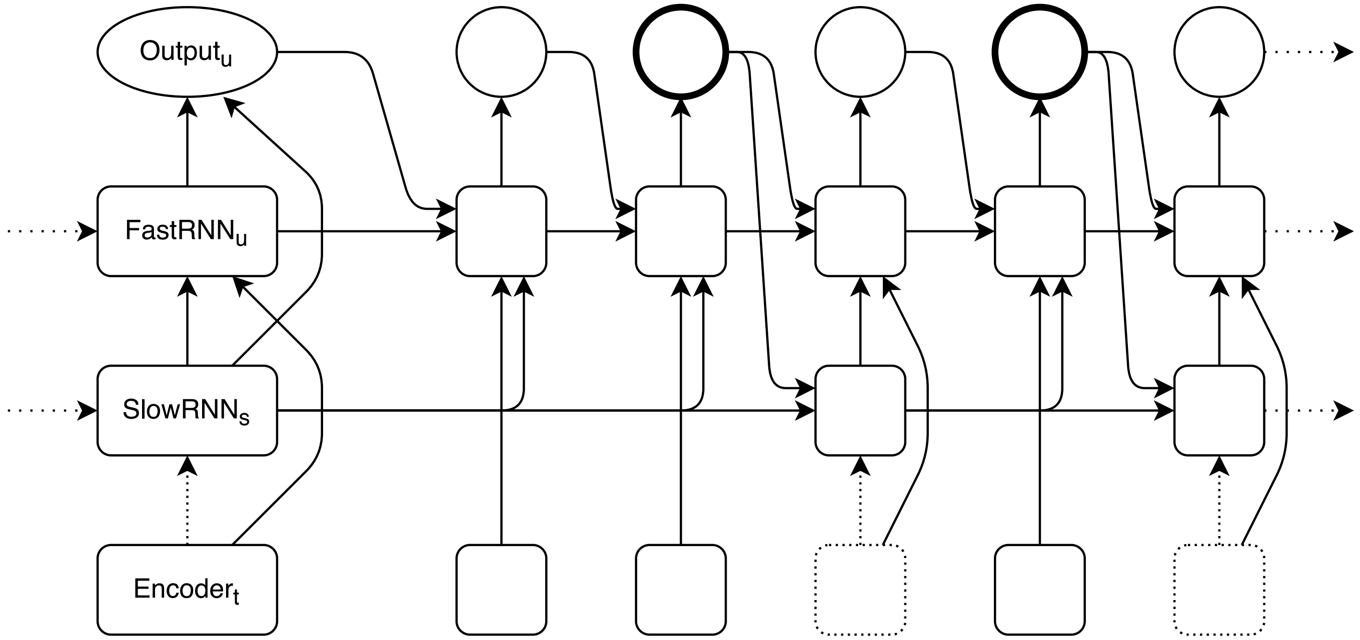
Regularization:
- Dropout
- Label smoothing

ASR: Transducer Models



- RNN transducer (RNN-T) [Graves 12],
  Recurrent neural aligner (RNA) [Sak & Shannon+ 17],
  Generalization & extension [Zeyer & Merboldt+ 20]
- Model: SlowRNN and FastRNN
- Label-topology: RNN-T, RNA, or CTC
- Training criteria:
  - Full-sum (FS) over all possible alignments
    $L := -\log \Sigma_{a_1^U} \Pi_{j=1}^{U} p(a_j | a_1^{j-1}, x_1^T)$
  - Frame-wise cross-entropy (CE) using given alignment
- Recognition:
  - Beam-search over alignment axis (time-sync for CTC/RNA)

Generalized / extended transducer model



- Three different time axes:
  T encoder time axis, downsampled from input
  S label output axis
  U alignment axis, e.g. for RNN-T: $U = T + S$
- FastRNN: runs over alignment axis
- SlowRNN: only updated when a non-blank symbol is emitted
- For RNA/CTC: alignment axis $\equiv$ encoder axis

Slow-/FastRNN does not necessarily need to be a RNN (Transformer, Conv, ...)

Implementation of extended transducer in a framework

Training pipeline: Full-sum training → Forced alignment → Viterbi training

- Full-sum training
  - Existing open-source CUDA implementations for TF/PyTorch (warp-transducer, warp-rnnt, warp-rna)
  - → Support for executing native code in framework
- Forced alignment: modify CUDA implementation or write custom backend code
  - Exchange sum with maximum, then backtrack best path
  - Save alignments to file for train/dev datasets
- Transducer with fixed path is simply cross-entropy (CE) training
  - Use alignments from step 2 to compute CE on the path
  - Train same or bigger model
- Recognition
  - Needs custom beam search
  - → Framework has to be extendable and flexible

→ Requires flexible framework

- RETURNN does not need any modification/extension, all possible via config

Transducer in RETURNN

Model defined once, training and recognition share the same logic (explained in Part 1: Recurrency)

During training:

- All layers are moved outside the loop
- Loss computation:
  - Full-sum: Dynamic programming on 4D tensor `(batch, encoder-len, target-len+1, vocab)`
    - Very memory and fairly computation expensive
  - With fixed path: frame-wise cross-entropy criterion
    - Fast compared to full-sum computation

During recognition/search:

- FastRNN is updated every frame
- SlowRNN is only updated for non-blank symbols
- Blank symbols are masked away for the output

Transducer: SlowRNN Implementation

In training:

- Operations running on different time axes
  - Synchronization using scattering
  - PyTorch: `torch.Tensor.masked_scatter_`, TF: `tf.scatter_nd`

In recognition:

- In each step, make a choice:
  - `mask=False`: copy over last state
  - `mask=True`: update state

Implementation in RETURNN using `MaskedComputationLayer`:

- supports both training and recognition without modifications
- `mask`: $\texttt{prev:output\_emit} = \begin{cases} \text{True}, & \text{if last output} \in \Sigma \text{ (actual symbol)} \\ \text{False}, & \text{if last output} = \varnothing \text{ (blank)} \end{cases}$

```
"slow": {"class": "masked_computation",
  "mask": "prev:output_emit",
  "from": "prev_out_non_blank",  # in decoding
  "unit": { "class": "subnetwork", "from": "data",
  "subnetwork": {
    "input_embed": {"class": "linear", "activation": None, "with_bias": False, "from": "data", "n_out": 621},
    "lstm0": {"class": "rec", "unit": "nativelstm2", "n_out": LstmDim, "from": "input_embed"},
    "output": {"class": "copy", "from": "lstm0"}
  }}},
```

Transducer Model: Variant "explicit blank"

$\mathrm{Readout}(z_u^{\text{fast}}, z_{s_u}^{\text{slow}})$

```
"readout_in": {"class": "linear", "from": ["fast", "slow"],
               "activation": None, "n_out": 1000},
"readout": {"class": "reduce_out", "mode": "max", "num_pieces": 2, "from": "readout_in"}
```

$$p(y_u \mid \ldots) := \mathrm{softmax}_{\Sigma'}(\mathrm{FF}(\mathrm{Readout}(\ldots))),$$
$$y_u \in \Sigma',$$
$$\Sigma' = \Sigma \cup \{\varnothing\}$$

```
"output_log_prob": {"class": "linear", "from": ["readout"],
                    "activation": "log_softmax", "n_out": num_labels+1}
```

Transducer Model: Variant "separate emit model"

$\mathrm{Readout}(z_u^{\text{fast}}, z_{s_u}^{\text{slow}})$

```
"readout_in": {"class": "linear", "from": ["fast", "slow"],
               "activation": None, "n_out": 1000},
"readout": {"class": "reduce_out", "mode": "max", "num_pieces": 2, "from": "readout_in"}
```

$p_u(\Delta s{=}1 \mid \ldots) :=$
$\sigma(\ \ \mathrm{FF}_{\mathrm{emit}}(z_u^{\mathrm{fast}}))$

```
"emit_prob0": {"class": "linear", "from": "fast", "activation": None, "n_out": 1},
"emit_log_prob": {"class": "activation", "from": "emit_prob0", "activation": "log_sigmoid"}
```

$p_u(\Delta s{=}0 \mid \ldots) :=$
$\sigma(-\mathrm{FF}_{\mathrm{emit}}(z_u^{\mathrm{fast}}))$

```
"blank_log_prob": {"class": "eval", "from": "emit_prob0", "eval": "log_sigmoid(−source(0))"}
```

$p_{u,\Sigma}(y_u \mid \ldots) :=$
$\mathrm{softmax}_{\Sigma}(\mathrm{Readout}(\ldots))$

```
"label_log_prob": {"class": "linear", "from": "readout", "activation": "log_softmax",
                   "n_out": num_labels},
```

$p(y_u \mid \ldots) =$
$p_u(\Delta s{=}1|\ldots) \cdot p_{u,\Sigma}(y_u|\ldots)$

```
"label_emit_log_prob": {"class": "combine", "kind": "add", "from": ["label_log_prob", "emit_log_prob"]}
"output_log_prob": {"class": "copy", "from": ["blank_log_prob", "label_log_prob"]}
```

Both variants are equivalent (can be reformulated into each other; but different modelling).

Part 2: Specific Models & Applications

Introduction
Machine translation (RNN/Transformer-based encoder-decoder-attention)
Speech recognition (Hybrid HMM, Attention, other end-to-end approaches)
Language modeling (RNN/LSTM, Transformer)
Features in RETURNN
Language Model Integration
End-to-end speech translation
Text-to-speech

Introduction

Language modeling = estimating probability of sequence of words

- Training only requires texts, unsupervised task no labeling is needed
- Lots of data are available (typically unlimited!)

Scaling-up: a big part in recent advances in language modeling

- Training large scale neural language models
- Using large amount of data
  - How to scale up? Software? Compute? Algorithm?
  - How to make use of the relevant part of available data? How to benefit from large diversity in the data?

Applications

- Crucial component in ASR/MT especially in log-linear framework
  - Limited for neural MT compared to back-translation
  - An important component in ASR, for both conventional and end-to-end systems

Many available frameworks, count-based and neural

LM

What LM requires from a framework:

- Scalability
- Easy architecture setup, e.g. LSTM and Transformer LM
- Proper perplexity evaluation, e.g. correctly normalized scores, per position scores
- Easy integration to ASR experiments, e.g. rescoring, reranking, various fusion techniques
- Support of different units, characters, subwords, words, etc.
- Easy sampling, e.g. for back-translation, corpus filtering
- Easy comparison and interpolation with count-based models
- Speedup methods
- ...

Models Overview

N-gram count based model = relative frequency + smoothing technique
Standard choice: LSTM-RNN model [Sundermeyer & Schlüter+ 12]
Many alternatives to LSTM-LM

- Gated convolution [Dauphin & Fan+ 17]
- WaveNet (dilated convolution) [Kurata & Sethy+ 17]
- Fixed-size ordinarily forgetting encoding (FOFE) [Zhang & Jiang+ 15]
- Recurrent highway network [Zilly & Srivastava+ 17]
- Recurrent memory network [Tran & Bisazza+ 16] and more

But did not clearly replace LSTM
More recently, Transformer model

- Transformer-XL [Dai & Yang+ 19, Irie & Zeyer+ 19]
- Deep (64-layer) Transformer LM [Al-Rfou & Choe+ 19]
- Universal Transformer [Dehghani & Gouws+ 18]

Language Modeling with RETURNN - Feature Overview

RETURNN supports almost all crucial features

- Data
  - Plain line-based text `.txt` / `.txt.gz` files can be directly used (`LmDataset`)
  - For large texts, conversion to HDF format by `tools/hdf_dump.py` can reduce RAM requirement for training jobs (`HDFDataset`)
- Modern language modeling with neural networks on a large scale
  - LSTM-RNN (based on our fast custom kernel, or multiplicative LSTM)
  - Self-attention (Transformer decoder)
- Model units
  - Characters, subwords, words, etc.
  - For large vocabulary (e.g. $>$ 100K), `SamplingBasedLoss` implements, efficiency of softmax:
    - Sampled softmax `tf.nn.sampled_softmax_loss`
    - Noise contrastive estimation (NCE) `tf.nn.nce_loss`
- Applications/combinations in speech, translation, and potentially other tasks
- Other features like training algorithms, batching, etc.

Feature Overview: More

LM checkpoint inspection tool:

- N-gram count models help to obtain a good neural language models
  - Reference perplexity without hyper-parameter tuning
  - Detect the domain signal in the text data
- Easy comparison and interpolation with count-based models
- Evaluate perplexities, with and without context across sentence boundaries
- Token-wise probabilities in the good old SRILM format:

```
p( one | ... )  = [debug2] 0.00478558 [ -2.32007 ]
p( who | ... )  = [debug2] 0.00816807 [ -2.08788 ]
p( writes | ... )  = [debug2] 0.00140808 [ -2.85137 ]
p( of | ... )  = [debug2] 0.0511369 [ -1.29127 ]
p( such | ... )  = [debug2] 0.0143111 [ -1.84433 ]
p( an | ... )  = [debug2] 0.0365669 [ -1.43691 ]
p( era | ... )  = [debug2] 0.000853111 [ -3.06899 ]

...
```

Network Configuration - LSTM Model

- LSTM LM with two hidden layers
- Weight tying, e.g. embedding and softmax weight matrix using `reuse_params`
- Transpose by `use_transposed_weights`

```
network = {
'input': {'class': 'linear','activation': 'identity', 'n_out': 256,'forward_weights_init': 'random_normal_initializer(mean=0.0, stddev=0.1)','from': ['data']},

'lstm0': {'class': 'rec','unit': 'lstm', 'n_out': 2048,'dropout': 0.1,'L2': 0.0,'direction': 1,'from': ['input']},
'lstm1': {'class': 'rec','unit': 'lstm', 'n_out': 2048,'dropout': 0.1,'L2': 0.0,'direction': 1,'from': ['lstm0']},
'proj':{'class': 'linear','activation': 'relu','dropout': 0.1,'n_out': 256,'from': ['lstm1']},

'output':{'class': 'softmax','dropout': 0.1, 'reuse_params': 'input', 'use_transposed_weights': True,
'forward_weights_init': 'random_normal_initializer(mean=0.0, stddev=0.1)',
'bias_init': 'random_normal_initializer(mean=0.0, stddev=0.1)',
'loss': 'ce','target': 'classes','from': ['proj']},
}

calculate_exp_loss = True
```

- By default, `CrossEntropyLoss` does not report exp loss values
- Activate `calculate_exp_loss` to get the perplexity in the default case

LM: Features in RETURNN

Transformer Model

- Give large improvements over LSTM LM
- LM task automatically provides positional information, no need for extra signal [Irie & Zeyer+ 19]
- Network configuration similar to MT decoder
- Using provided building blocks, writing config file
  - Stack L layers each consisting of self-attention and feed-forward modules
  - Apply dropout, residual connections and layer normalization across module
- Use the same dimensionality across all layers reduce hyper-parameters

LM: Language Model Integration

End-to-end (attention based) speech and translation models

- Setting up fusion techniques is a matter of writing a config
  - Shallow fusion (inference) [Gülçehre & Firat+ 17]
  - Deep fusion, cold fusion, so on (training and inference) [Gülçehre & Firat+ 17, Sriram & Jun+ 18, Michel & Schlüter+ 20]
- Loading pretrained LM parameters
- Same output vocabulary, but various model combination

Conventional speech recognition

- Lattice rescoring tool
  - Part of RETURNN tools, TF C++ API
  - Original rescoring code based on our `rwthlm` toolkit
  - Take `HTK` standard lattice format as input
- Lattice rescoring and first pass decoding
  - Part of RASR, TF C++ API.

Shallow fusion in Attention ASR Model

- Shallow fusion = beam search on log-linear combination of softmaxes
- `load_on_init` in `subnetwork` layer and `lm_model_filename`
- `NativeLstm` operating step by step
- `eval` layer for simple implementation of log-linear combination

```
lm_scale = 0.25
lm_model_filename = "lm_dir/net-model/network.020"
fusion_eval_str = "safe_log (source(0)) + %s * safe_log (source (1)) " %str(lm_scale)
network = { ...
"output": {"class": "choice", "target": "bpe", "beam_size": beam_size,
        "initial_output": 0,"input_type": "log_prob", "from":["combo_output_log_prob"], "initial_output": 0},
"combo_output_log_prob": {"class": "eval", "eval": fusion_eval_str, "from": ["am_output_prob",  "lm_output_prob"]},
% ... Skipping E2E models layers ...
"am_output_prob": {"class": "softmax", "from": ["e2e_speech_model"], "dropout": 0.3, "target": "bpe", "loss": "ce"},
"lm_output": {
  "class": "subnetwork", "from": ["prev:output"],
  "load_on_init": lm_model_filename, "n_out": 1030,
  "subnetwork": {"input": {"class": "linear", "n_out": 256, "activation": "identity"},
     "lstm0": {"class": "rec", "unit": "nativelstm2", "n_out": 2048,"unit_opts": {"forget_bias": 0.0}, "from": ["input"]},
     "lstm1": {"class": "rec", "unit": "nativelstm2", "n_out": 2048, "unit_opts": {"forget_bias": 0.0},  "from": ["lstm0"]},
     "output": {"class": "linear", "from": ["lstm1"], "activation": "identity", "n_out": 1030}}},
"lm_output_prob" : {"class": "activation", "activation": "softmax","from": ["lm_output"], "target": "bpe"}}
```

Sampled Softmax and NCE

- Sampled softmax to ease the computation of denominator
- NCE to replace the one-in-vocabulary classification with true-data/noisy-sample binary classification
- Configurable sampler and number of samples
- Easy activation of full softmax for proper perplexity calculation
- Speed-up without performance reduction

```
'output':{
    'class': 'softmax',
    'dropout': 0.1,
    'use_transposed_weights': True,
    'forward_weights_init': 'random_normal_initializer(mean=0.0, stddev=0.1)',
    'bias_init': 'random_normal_initializer(mean=0.0, stddev=0.1)',
    'loss': 'sampling_loss',
    'loss_opts': {
        'use_full_softmax': False,
        'sampler': 'uniform', # uniform, log_uniform, learned_unigram
        'num_sampled': 4096,
        'nce_loss': False,
    },
    'target': 'data',
    'from': ['projection'],
},
```

Part 2: Specific Models & Applications

Introduction
Machine translation (RNN/Transformer-based encoder-decoder-attention)
Speech recognition (Hybrid HMM, Attention, other end-to-end approaches)
Language modeling (RNN/LSTM, Transformer)
End-to-end speech translation
- Direct Model
- Transfer Learning
- Multi-task Learning

Text-to-speech

Introduction

Speech to text translation = translating speech signal to target text

Models

- Cascade system (concatenation of ASR + MT models)
  - Errors propagated from ASR
  - Two-pass decoding, computational complexity, model size
- End-to-end models (trainable in an end-to-end fashion)
  - Direct model, no dependence on the source transcriptions [Berard & Pietquin+ 16, Goldwater & Lopez+ 17]
  - Transfer learning/pretraining [Bansal & Kamper+ 19, Stoian & Bansal+ 20]
  - Multi-task learning, ASR or MT as co-trainer [Weiss & Chorowski+ 17]
  - Two-stage models [Anastasopoulos & Chiang 18, Sung & Liu+ 19, Sperber & Neubig+ 19]

Frameworks (All support cascade, direct modeling):

- ESPNet-ST [Inaguma & Kiyono+ 20]
- Lingvo [Shen & Nguyen+ 19]
- SLT.KIT [Zenkel & Sperber+ 18]
- Fairseq S2T: in addition online/simultaneous ST [Wang & Tang+ 20]
- ...

ST

What ST requires from a framework:

- Flexibility, simplicity, scalability and extensibility as mentioned before
- Easy architecture setup to enable different models
- Easy integration of ASR and MT models for cascading, e.g Hybrid HMM-NN (allows to plug-in other framework), also LM integration
- Extendability, e.g. parameter freezing, transfer learning, multi-task learning
- Speech features: MFCC, log-Mel, Gammatone
- Model units: subwords, characters, etc.
- Data augmentation, e.g. SpecAugment [Park & Chan^{+} 19]
- Generating synthetic data both text and speech, knowledge distillation
- Enable use of presegmenting speech frames, e.g conventional alignments for time boundaries
- Online capability
- ...

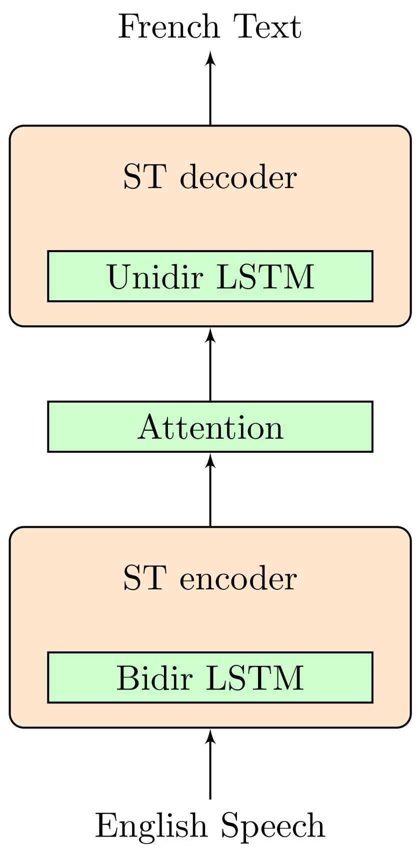
Cascade Model

- Standard choice: write ASR output on disk and read it again as an input to MT
- All models in one config at once and conduct the search, limitation: same vocabulary
  - Load all MT parameters with optional name like "mt_" prefix and ASR parameters with "asr_" prefix
  - Use `preload_from_files` to load parameters from checkpoints
  - Activate `include_eos`

```
preload_from_files = {}
if task == "search":
  preload_from_files = {
  "model0" : {"filename": "PATH_TO_CHECKPOINT", "prefix": "asr_"},
  "model1" : {"filename": "PATH_TO_CHECKPOINT", "prefix": "mt_"}
network = {
# ASR encoder ...
"asr_lstm0_fw" : { "class": "rec", "unit": "nativelstm2", "n_out" : 1024, "direction": 1, "from": ["asr_source"] },
"asr_lstm0_bw" : { "class": "rec", "unit": "nativelstm2", "n_out" : 1024, "direction": -1, "from": ["asr_source"] },
... # so on
"asr_output": {"class": "rec", "from": [], "include_eos": True, "unit": {
  ... # so on
# Passing onehot vectors directly
"one_hot": {"class": "get_beamed_layer", "scores": " asr_output/output_prob_beam_scores", "beams": " asr_output/output_prob_src_beams",
    "beam_size": 12, "mode": "best", "one_hot": True, "from": [" asr_output/output"], "only_on_search": True},
 # MT encoder
" mt_source_embed": {"class": "linear", "activation": None, "with_bias": False, "n_out": 512, "from": ["one_hot"]},
... # so on
 # MT decoder
"output": {"class": "rec", "from": [], "unit": {
  ... # so on
  "mt_s_transformed": {"class": "linear", "activation": None, "with_bias": False, "from": ["mt_s"], "n_out": 1024, "dropout": 0.3},}
```

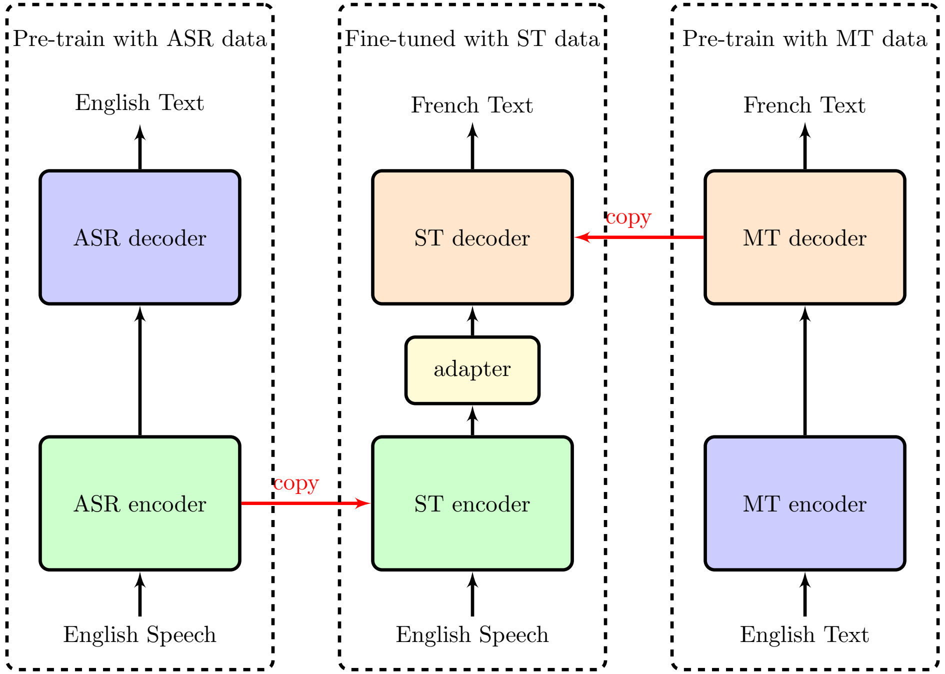
ST: Direct Model

- Allowing joint training of all model components
- Removing the need for explicit intermediate representations
- Challenge
  - Fall short due to data scarcity
- In RETURNN
  - All end-to-end models, RNN attention, Transformer, etc.
  - `MetaDataset` to combine ASR and MT data, if needed



ST: Transfer Learning

- To overcome data scarcity problem
- Transfer knowledge from ASR and/or MT task
- Encoder and decoder networks have been separately pretrained on ASR and MT tasks
- In RETURNN
  - Flexible configuration
  - Avoid pretraining some layers/parameters, add additional layers (e.g adapter layer)
  - Adapter component: to familiarize the pretrained decoder and pretrained encoder
  - Architecture combination, e.g. RNN ASR model and MT Transformer



ST: Transfer Learning

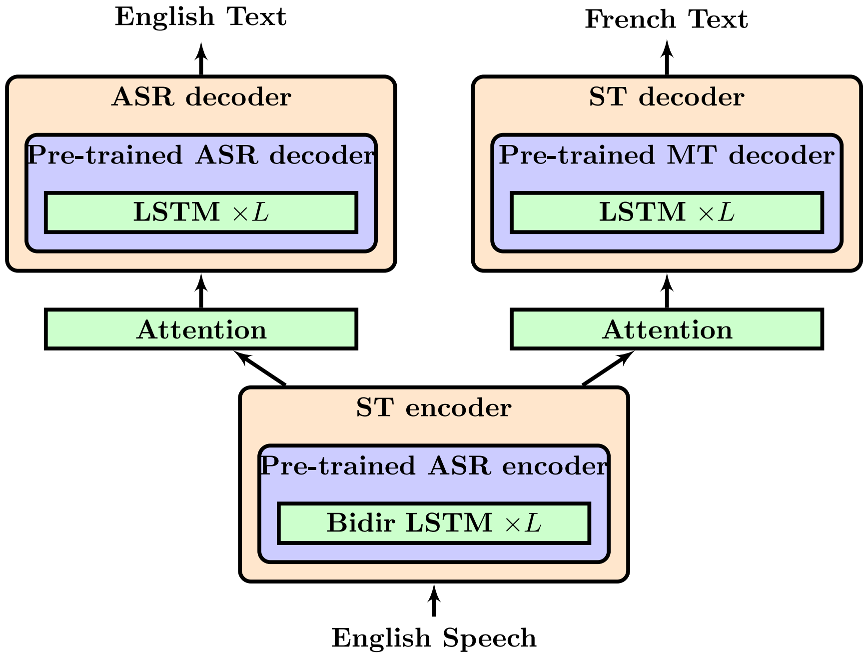
- Load pretrained ASR and MT models using a prefix
- Freezing parameters easily via `"trainable":False`

```
preload_from_files = {}
if not task == "search":
  preload_from_files = {
  "model0" : {"filename": "PATH_TO_CHECKPOINT", "prefix":  "asr_", "init_for_train": True},
  "model1" : {"filename": "PATH_TO_CHECKPOINT", "prefix":  "mt_", "init_for_train": True},}

network = {
 # ASR encoder first layer pretraining
"asr_lstm0_fw" : { "class": "rec", "unit": "nativelstm2", "n_out" : 1024, "direction": 1, "from": ["asr_source"] },
"asr_lstm0_bw" : { "class": "rec", "unit": "nativelstm2", "n_out" : 1024, "direction": −1, "from": ["asr_source"] },
"asr_lstm0_pool": {"class": "pool", "mode": "max", "padding": "same", "pool_size": (3,), "from": ["asr_lstm0_fw", "asr_lstm0_bw"], "trainable": False},
... # and so on
 # MT decoder output layer pretraining
"output": {"class": "rec", "from": [], "unit": {
"output": {"class": "choice", "target": TargetVoc, "beam_size": 12, "from": ["mt_output_prob"], "initial_output": 0},
"end": {"class": "compare", "from": ["output"], "value": 0},
"mt_target_embed": {"class": "linear", "activation": None, "with_bias": False, "from": ["output"], "n_out": 512, "initial_output": 0,  "trainable":False},
"mt_s_transformed": {"class": "linear", "activation": None, "with_bias": False, "from": ["mt_s"], "n_out": 1024, "dropout": 0.3},
... # and so on
}
```

ST: Multi-task Learning

- Auxiliary model is co-trained with speech translation
- Sharing some parameters
- One-to-many
  - ASR model is the auxiliary co-trainer
  - Shared speech encoder
  - Two independent decoders correspond to transcription and translation
  - Independent of model architecture
  - Transfer learning can be applied (blue blocks)



Multi-task Learning - One-to-Many

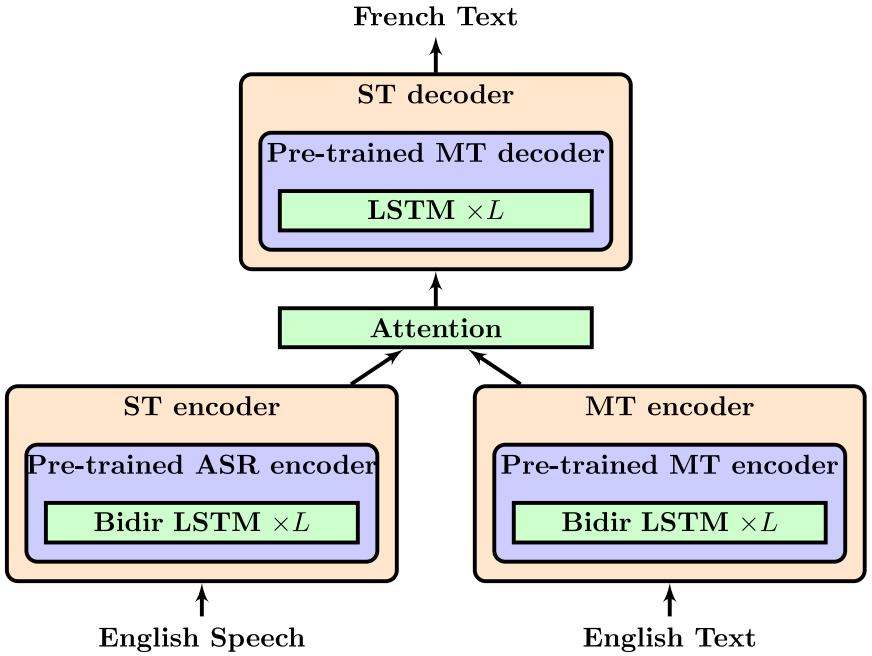
- Construct direct ST network as usual
- Add ASR decoder as an additional network
- Combine the losses by `"scale"`, $L = \lambda \log p_{s2s}(e_1^I | x_1^T) + (1 - \lambda) \log p_{s2s}(f_1^J | x_1^T)$

```
lambda = 0.5
network = {
# ST direct encoder-decdoer network in here
...
"output": {"class": "rec", "from": [], "unit": {
  ...
  "output_prob": {"class": "softmax", "from": ["readout"], "dropout": 0.3,"target": ST_VOC, "loss": "ce", "loss_opts": {"label_smoothing": 0.1, "scale": lambda}}
  }, "target": target, "max_seq_len": "max_len_from('base:encoder')"},
}
# Additional ASR decoder
network_asr_dec = {
"output_asr": {"class": "rec", "from": [], "unit": {
  "output": {"class": "choice", "target": ASR_VOC,"from": ["output_prob"], "initial_output": 0},
  ...
  "output_prob": {"class": "softmax", "from": ["readout"], "dropout": 0.3,"target": ASR_VOC, "loss": "ce", "loss_opts": {"label_smoothing": 0.1, "scale": 1.0-lambda}
  }, "target": target2, "max_seq_len": "max_len_from('base:encoder')"},
}

if task != "search":
    network.update(network_asr_dec)
```

ST: Multi-task Learning

- Many-to-one
  - MT model is auxiliary co-trainer
  - Shared target text decoder
  - Two independent encoders correspond to speech and source text
  - A switch layer to select the encoders
  - Ignore MT encoder in inference
  - Transfer learning can be applied (blue blocks)



Multi-task Learning - Many-to-One

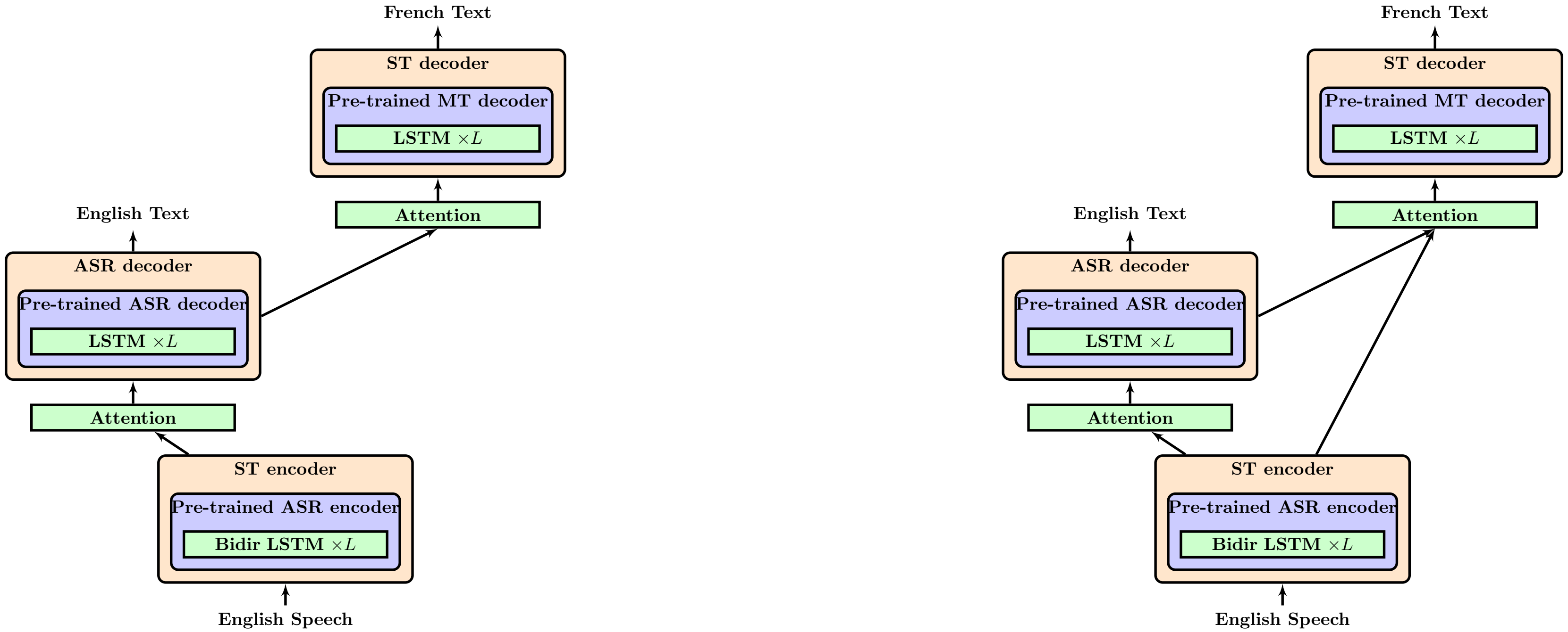
- Construct the direct ST network as usual
- Add the additional MT encoder
- Use `switch` to select each of the encoders based on input data

```
network = {
  # The speech encoder
  "source": {"class": "eval", "eval": "tf.clip_by_value(source(0), −3.0, 3.0)"},
  "lstm0_fw" : { "class": "rec", "unit": "nativelstm2", "n_out" : 1024, "direction": 1, "from": ["source"] },
  "lstm0_bw" : { "class": "rec", "unit": "nativelstm2", "n_out" : 1024, "direction": −1, "from": ["source"] },
  "lstm0_pool": {"class": "pool", "mode": "max", "padding": "same", "pool_size": (2,), "from": ["lstm0_fw", "lstm0_bw"], "trainable": False},
  ...
  # The text encoder
  "source_text": {"class": "linear", "activation": None, "with_bias": False, "n_out": 512, "from": ["data:source_text"]},
  "lstm0_fw_text" : { "class": "rec", "unit": "nativelstm2", "n_out" : 1024, "direction": 1, "from": ["source_text"] },
  "lstm0_bw_text" : { "class": "rec", "unit": "nativelstm2", "n_out" : 1024, "direction": −1, "from": ["source_text"] },
  ...
  "length": {"class": "length", "from": [data:data]},
  "length_is_empty": {"class": "compare", "value": 1, "from": ["length"], "kind": "less_equal"},

  "output": {"class": "rec", "from": [], "unit": {
    ...
    # The context vector in the decoder
    "context": {"class": "switch", "condition": "base:length_is_empty", "true_from": "context_text", "false_from": "context_audio"},
...}
```

End-to-end ST - Two-Stage Model

- Rely on transcripts, but optimized in part through end-to-end training
- Passing internal representation, e.g. attention or decoder state
- Easy setup without touching code



Other Aspects

- Almost all ASR features can be used for ST
- E.g. SpecAugment, CTC loss, layer-wise pretraining, etc.

```
network = {
    % we define the ST model here
}

network_additional_ctc = {
    "ctc": {"class": "softmax", "from": ["encoder"], "loss": "ctc",
        "target": "target_text_sprint", "loss_opts": {"beam_width": 1, "ctc_opts": {"ignore_longer_outputs_than_inputs": True}}}
}

if task != "search":
    network.update(network_additional_ctc)
```

- Interfaces to RASR (Sprint)
  - Hybrid HMM-NN ASR model
  - ASR streaming, thus enabling online ST (not yet public)
  - Segmentation of speech frames using alignments

Part 2: Specific Models & Applications

- Introduction
- Machine translation (RNN/Transformer-based encoder-decoder-attention)
- Speech recognition (Hybrid HMM, Attention, other end-to-end approaches)
- Language modeling (RNN/LSTM, Transformer)
- End-to-end speech translation
- Text-to-speech
  - Tacotron-2
  - Extending TTS models
  - Handling Multispeaker
  - Vocoder Integration

TTS

Neural TTS = neural feature model + neural vocoder model

Prominent neural feature models:

- Tacotron 1/2
- Deep Voice 3 / ClariNET
- FastSpeech
- Transformer TTS

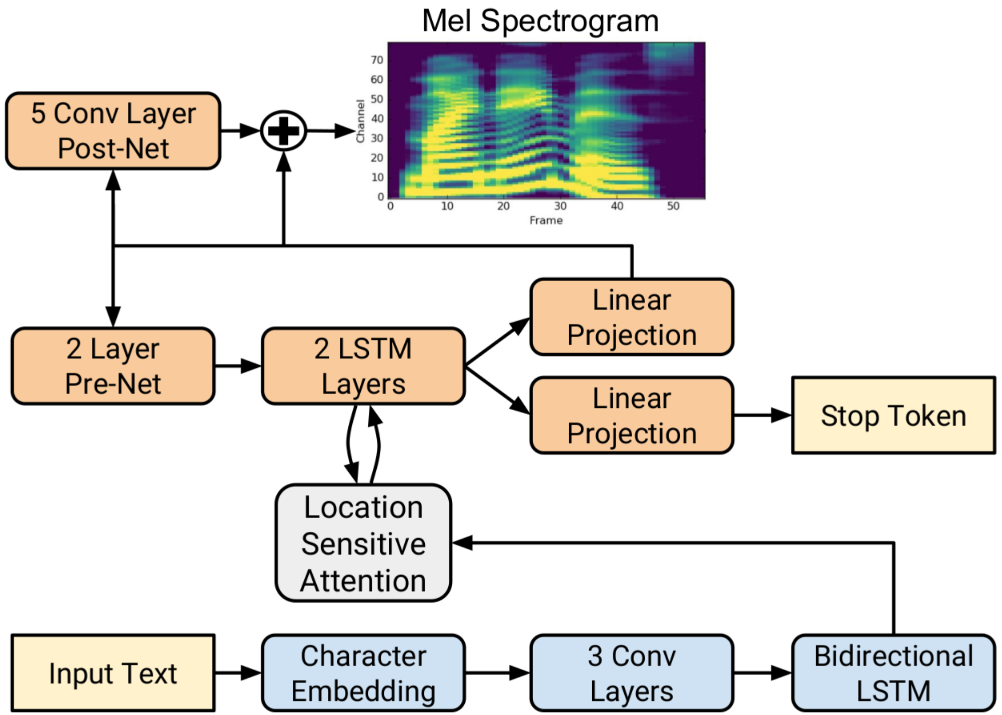
Prominent neural vocoder models:

- WaveNet
- WaveRNN
- WaveGlow
- LPCNet
- MelGAN

- Many non-official repositories, no unified frameworks implementing multiple approaches
- Exception: ESPNet, OpenSeq2Seq and in the future RETURNN

TTS: Tacotron-2

Common architecture: Tacotron-2 [Shen & Pang+ 17]



- Encoder-decoder architecture with LSTMs
- Explicit stop token
- Log-mel feature targets
- Non-autoregressive post-net for spectrum-refinement
- Can run on any text representation

- A complete RETURNN configuration for Tacotron-2 can be found here:
`https://github.com/rwth-i6/returnn-experiments/tree/master/2020-TTS-LJSpeech`

TTS: Tacotron-2

Tacotron-2 uses mostly common modules and features also found in MT and ASR
$\rightarrow$ all needed modules should be part of the framework

Functions not covered by ASR and MT are:

- ZoneoutLSTM
- Location Sensitive Attention
- Explicit stop token instead of stopping via target labels
- The decoder might operate on merged frames

Extensions also include:

- Monotonic attention variants

Example: Native support for windowing

- The Framework should be capable of handling windowing
- Using windowing should be minimal overhead
  → The windowing is simply a layer which generates data

```
'windowed_data': {'class': 'window',
                  'from': ['data'],
                  'window_size': 2,
                  'window_right': 1,
                  'stride': 2},
'windowed_data_target': {'class': 'merge_dims',
                         'from': ['windowed_data'],
                         'axes': 'static',
                         'n_out': 160,
                         'register_as_extern_data': 'windowed_data_target'},
```

- The windowing can be undone at any part of the network

```
'output_split': {'class': 'split_dims', 'from': ['decoder'], 'axis': 'F', 'dims': (2, -1)},
'output_reshape': {'class': 'merge_dims', 'from': ['output_split'], 'axes': ['T', 'static:0'], 'n_out': 80,},
```

→ Slight overhead in reshaping the tensors
→ Good control over the windowing behavior, but missing overlap-and-add

TTS: Extending TTS models

Example: Location Sensitive Attention

- The core component is convolutional feedback:

```
'feedback_pad_left': {'axes': 's:0', 'class': 'pad', 'from': ['prev:accum_att_weights'],
                      'mode': 'constant',  'padding': ((15, 0),), 'value': 1},
'feedback_pad_right': {'axes': 's:0', 'class': 'pad', 'from': ['feedback_pad_left'],
                       'mode': 'constant', 'padding': ((0, 15),), 'value': 0},
'convolved_att': {'L2': 1e-07, 'activation': None, 'class': 'conv', 'filter_size': (31,),
                  'from': ['feedback_pad_right'], 'n_out': 32, 'padding': 'valid'},
```

- The combination of MLP-style attention inputs is only a single command:

```
'att_energy_in': { 'class': 'combine', 'from': ['base:enc_ctx', 's_transformed', 'location_feedback_transformed'],
                   'kind': 'add', 'n_out': 128},
```

→ For some attention mechanisms, implementing and editing is really convenient

A more complicated issue: Dynamic convolution feedback [Battenberg & Skerry-Ryan^{+} 20]

- Convolutional filters are computed by layers
- Filters are independent for each batch position
- `tf.nn.depthwise_conv2d` needed for correct filter computation

→ Custom TF code needed, as depthwise convolution does not exist yet

Comparison to ESPNet

ESPNet offers a well documented class `AttLoc`

- Provides additional functions:
  - Window constraints (forward window, backward window)
  - Softmax scaling
  - Encoder context pre-computation
- Downsides:
  - Redundant code with other attentions
  - Needs code changes to e.g. implement sum feedback
  - Can not be combined with other types of feedback

RETURNN:

- Easy switch from `prev:accum_att_weights` to `prev:att_weights`
- Fast combination of any feedback types (Location-Sensitive, Gaussian, Fertility based etc.)
- Window constraints available in the `softmax_over_spatial` layer
- But: An `eval` layer might be required for very specific constraints

TTS: Handling Multispeaker

Global Style Token

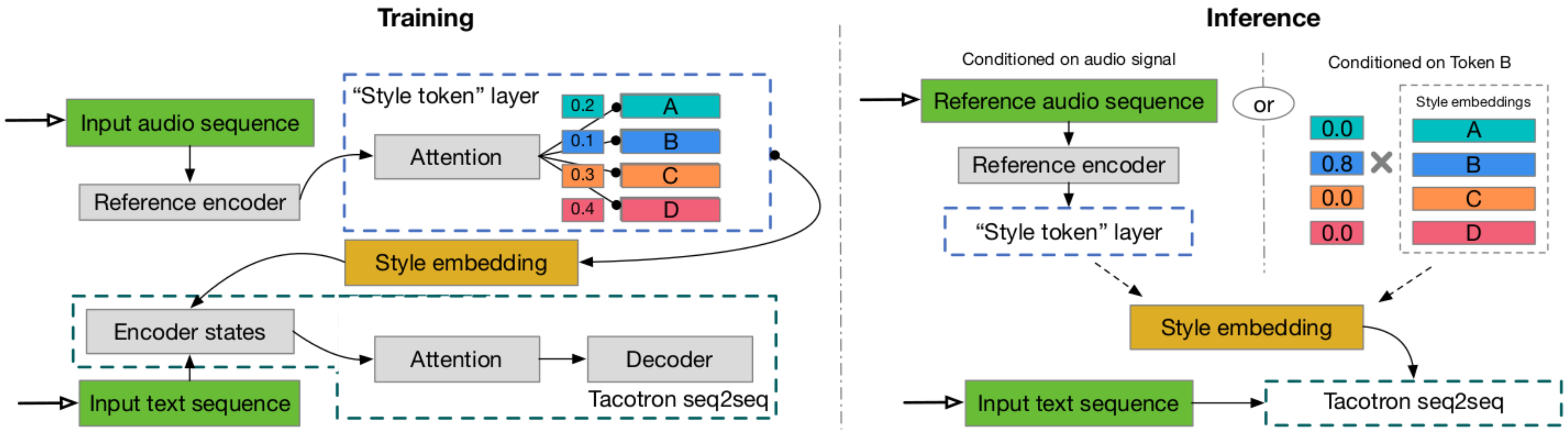


Figure: GST architecture taken from [Wang & Stanton^{+} 18]:

- Reference encoder similar to prosody embedding
- Style or speaker embedding is a combination of fixed weighted tokens
- Weights can be separated by their characteristics
- Allows for manual control over the style or speaker
- Can model other criteria such as recording settings or noise

TTS: Handling Multispeaker

Necessary features for GST:

- Accessing last recurrent output or computing the mean of outputs:

```
'ref_enc': {'class': 'rec', 'direction': 1, 'from': ['reshape_enc'], 'n_out': 128, 'unit': 'nativelstm'},
'ref_enc_reduce': {'class': 'reduce', 'from': ['ref_enc'], 'axis': 'T', 'mode': 'mean', 'n_out': GST_ATT_DIM},
or
'ref_enc_reduce': {'class': 'get_last_hidden_state', 'from': ['ref_enc'], 'combine': 'concat', 'n_out': GST_ATT_DIM},
```

- Providing arbitrary trainable tensors:

```
'style_token_variable': {'class': 'variable',
                         'init': 'glorot_normal',
                         'shape': (NUM_TOKENS, TOKEN_DIM), },
'style_token': {'class': 'reinterpret_data',
                'from': ['style_token_variable'],
                'set_axes': {'T': 's:0'}},
```

- Using pre-computed speaker embeddings or pre-computed GST weights as input:
  → Speaker embeddings can be dumped to HDF files at any time
  → Each layer can be replaced by using HDF inputs

TTS: Vocoder Integration

Example: Parallel-WaveGAN

- `https://github.com/kan-bayashi/ParallelWaveGAN`
- supports ParallelWaveGAN, MelGAN and Multiband-MelGAN

Additional Requirements:

- Ground-Truth-Aligned (GTA) decoding
  → RETURNN eval mode and hdf_dump layer
- Converting RETURNN hdf outputs and audio files into ParallelWaveGAN compatible datasets

Decoding:

- Custom code that executes text-processing, RETURNN and the vocoder
- Text processing can be e.g. Sequitur, num2words ...
- Embedding RETURNN and PyTorch applications into a single software

Summary

- Each application has its own needs with respect to:
  - Data processing
  - Model architectures
  - Decoding methods

- All applications have the same needs with respect to:
  - Flexible network design, e.g adding attention extensions
  - High training/decoding speed (possibly with automatic optimization)
  - Understandable API

Given the right concepts:
$\rightarrow$ A framework can be capable of handling arbitrary tasks and architectures
$\rightarrow$ A framework should not loose convenience over flexibility

What you should take home

Concepts:

- A tensor management concept like `Data` can be useful to increase flexibility and simplicity
- Model concepts with stochastic variables is helpful for implementing a larger variety of architectures (e.g. transducer)

Implementation:

- Consider implementing most used operations in native code as in RETURNN, Lingvo or Flashlight/Wav2Letter++
- Providing generic interfaces to external applications (e.g. RASR/Kaldi)

Overall:

- Many documented tasks examples as in e.g. ESPNet and OpenSeq2Seq reduce the entry barrier
- It is important to include additional tools for any supported task
- Try to target a high flexibility before your code-base gets too (task) specific

Thank you for your attention

Any questions?

References

Reference

- R. Al-Rfou, D. Choe, N. Constant, M. Guo, L. Jones.
  Character-level language modeling with deeper self-attention.
  In Proc. AAAI Conference on Artificial Intelligence, Honolulu, HI, USA, Jan. 2019.

- A. Anastasopoulos, D. Chiang.
  Tied multitask learning for neural speech translation.
  In Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers), pp. 82–91, New Orleans, Louisiana, June 2018. Association for Computational Linguistics.

- P. Bahar, N. Makarov, A. Zeyer, R. Schlüter, H. Ney.
  Exploring a zero-order direct hmm based on latent attention for automatic speech recognition.
  In IEEE International Conference on Acoustics, Speech, and Signal Processing, pp. 7854–7858, Barcelona, Spain, May 2020.

References

P. Bahar, A. Zeyer, R. Schlüter, H. Ney.
On using 2d sequence-to-sequence models for speech recognition.
In IEEE International Conference on Acoustics, Speech, and Signal Processing, Brighton, UK, May 2019.

P. Bahar, A. Zeyer, R. Schlüter, H. Ney.
On using specaugment for end-to-end speech translation.
In International Workshop on Spoken Language Translation, Hong Kong, China, Nov. 2019.

D. Bahdanau, K. Cho, Y. Bengio.
Neural machine translation by jointly learning to align and translate.
In 3rd International Conference on Learning Representations, ICLR, Conference Track Proceedings, San Diego, CA, USA, May 2015.

S. Bansal, H. Kamper, K. Livescu, A. Lopez, S. Goldwater.
Pre-training on high-resource speech recognition improves low-resource speech-to-text translation.
Proceedings of the 2019 Conference of the North, Vol. , 2019.

References

- E. Battenberg, R. Skerry-Ryan, S. Mariooryad, D. Stanton, D. Kao, M. Shannon, T. Bagby.
  Location-relative attention mechanisms for robust long-form speech synthesis.
  In ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 6194–6198, 2020.
- S. Bengio, O. Vinyals, N. Jaitly, N. Shazeer.
  Scheduled sampling for sequence prediction with recurrent neural networks.
  In Advances in Neural Information Processing Systems, pp. 1171–1179, 2015.
- A. Berard, O. Pietquin, C. Servan, L. Besacier.
  Listen and translate: A proof of concept for end-to-end speech-to-text translation, 2016.
- M. X. Chen, et al.
  The best of both worlds: Combining recent advances in neural machine translation.
  In Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers, pp. 76–86, 2018.

References

- M. X. Chen, O. Firat, A. Bapna, M. Johnson, W. Macherey, G. F. Foster, L. Jones, M. Schuster, N. Shazeer, N. Parmar, A. Vaswani, J. Uszkoreit, L. Kaiser, Z. Chen, Y. Wu, M. Hughes.
  The best of both worlds: Combining recent advances in neural machine translation.
  In I. Gurevych, Y. Miyao, editors, Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers, pp. 76–86. Association for Computational Linguistics, 2018.

- C.-C. Chiu*, C. Raffel*.
  Monotonic chunkwise attention.
  In International Conference on Learning Representations, 2018.

- T. Cohn, C. D. V. Hoang, E. Vymolova, K. Yao, C. Dyer, G. Haffari.
  Incorporating structural alignment biases into an attentional neural translation model.
  In NAACL HLT 2016, The 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pp. 876–885, San Diego, California, USA, 2016.

References

- Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. V. Le, R. Salakhutdinov.
  Transformer-XL: Attentive language models beyond a fixed-length context.
  arXiv preprint arXiv:1901.02860, 2019.
- Y. N. Dauphin, A. Fan, M. Auli, D. Grangier.
  Language modeling with gated convolutional networks.
  In Proc. Int. Conf. on Machine Learning (ICML), Sydney, Australia, Aug. 2017.
- M. Dehghani, S. Gouws, O. Vinyals, J. Uszkoreit, Ł. Kaiser.
  Universal transformers.
  arXiv preprint arXiv:1807.03819, 2018.
- P. Doetsch, A. Zeyer, P. Voigtlaender, I. Kulikov, R. Schlüter, H. Ney.
  Returnn: the rwth extensible training framework for universal recurrent neural networks.
  In IEEE International Conference on Acoustics, Speech, and Signal Processing, pp. 5345–5349, New Orleans, LA, USA, March 2017.

References

- T. Domhan, M. Denkowski, D. Vilar, X. Niu, F. Hieber, K. Heafield.
  The sockeye 2 neural machine translation toolkit at AMTA 2020.
  CoRR, Vol. abs/2008.04885, 2020.
- L. Dong, F. Wang, B. Xu.
  Self-attention aligner: A latency-control end-to-end model for asr using self-attention network and chunk-hopping.
  In ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 5656–5660, 2019.
- S. Edunov, M. Ott, M. Auli, D. Grangier, M. Ranzato.
  Classical structured prediction losses for sequence to sequence learning.
  Preprint arXiv:1711.04956, 2017.
- J. Gehring, M. Auli, D. Grangier, D. Yarats, Y. N. Dauphin.
  Convolutional sequence to sequence learning.

References

In Proceedings of the 34th International Conference on Machine Learning, ICML, pp. 1243–1252, Sydney, NSW, Australia, Aug. 2017.

S. Goldwater, A. Lopez, S. Bansal, H. Kamper.
Towards speech-to-text translation without speech recognition.
In Proceedings of the 15th Conference of the European Association for Computational Linguistics (EACL), pp. 474–479, 2017.

A. Graves, S. Fernández, F. Gomez, J. Schmidhuber.
Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks.
In Proceedings of the 23rd international conference on Machine learning, pp. 369–376, 2006.

A. Graves.
Practical variational inference for neural networks.
In Advances in neural information processing systems, pp. 2348–2356, 2011.

A. Graves.

References

Sequence transduction with recurrent neural networks.
Preprint arXiv:1211.3711, 2012.

Ç. Gülçehre, O. Firat, K. Xu, K. Cho, L. Barrault, H.-C. Lin, F. Bougares, H. Schwenk, Y. Bengio.
On using monolingual corpora in neural machine translation.
Computer Speech & Language, Vol. 45, pp. 137–148, Sept. 2017.

A. Hannun, A. Lee, Q. Xu, R. Collobert.
Sequence-to-Sequence Speech Recognition with Time-Depth Separable Convolutions.
In Proc. Interspeech 2019, pp. 3785–3789, 2019.

F. Hieber, T. Domhan, M. Denkowski, D. Vilar, A. Sokolov, A. Clifton, M. Post.
Sockeye: A toolkit for neural machine translation.
CoRR, Vol. abs/1712.05690, 2017.

S. Hochreiter, J. Schmidhuber.
Long short-term memory.

References

Neural computation, Vol. 9, No. 8, pp. 1735–1780, 1997.

J. Hou, S. Zhang, L.-R. Dai.
Gaussian prediction based attention for online end-to-end speech recognition.
In INTERSPEECH, 2017.

G. Huang, Y. Sun, Z. Liu, D. Sedra, K. Q. Weinberger.
Deep networks with stochastic depth.
In European conference on computer vision, pp. 646–661. Springer, 2016.

H. Inaguma, S. Kiyono, K. Duh, S. Karita, N. Yalta, T. Hayashi, S. Watanabe.
ESPnet-ST: All-in-one speech translation toolkit.
In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations, pp. 302–311, Online, July 2020. Association for Computational Linguistics.

K. Irie, A. Zeyer, R. Schlüter, H. Ney.
Language modeling with deep transformers.

References

In Interspeech, pp. 3905–3909, Graz, Austria, Sept. 2019.
ISCA Best Student Paper Award. [slides].

G. Klein, Y. Kim, Y. Deng, J. Senellart, A. M. Rush.
OpenNMT: Open-source toolkit for neural machine translation.
In Proc. ACL, 2017.

G. Kurata, A. Sethy, B. Ramabhadran, G. Saon.
Empirical exploration of novel architectures and objectives for language models.
In Proc. Interspeech, pp. 279–283, Stockholm, Sweden, Aug. 2017.

T. Luong, H. Pham, C. D. Manning.
Effective approaches to attention-based neural machine translation.
In L. Màrquez, C. Callison-Burch, J. Su, D. Pighin, Y. Marton, editors, Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, pp. 1412–1421, Lisbon, Portugal, Sept. 2015.

References

E. Matusov, P. Wilken, P. Bahar, J. Schamper, P. Golik, A. Zeyer, J. A. Silvestre-Cerda, A. Martinez-Villaronga, H. Pesch, J.-T. Peter.
Neural speech translation at apptek.
In Proceedings of the 15th International Workshop on Spoken Language Translation, pp. 104–111, Bruges, Belgium, Oct. 2018.

A. Merboldt, A. Zeyer, R. Schlüter, H. Ney.
An analysis of local monotonic attention variants.
In Interspeech, pp. 1398–1402, Graz, Austria, Sept. 2019.

W. Michel, R. Schlüter, H. Ney.
Early stage lm integration using local and global log-linear combination.
In Interspeech, Oct. 2020.
submitted to.

M. Ott, S. Edunov, A. Baevski, A. Fan, S. Gross, N. Ng, D. Grangier, M. Auli.

References

fairseq: A fast, extensible toolkit for sequence modeling.
In Proceedings of NAACL-HLT 2019: Demonstrations, 2019.

D. S. Park, W. Chan, Y. Zhang, C.-C. Chiu, B. Zoph, E. D. Cubuk, Q. V. Le.
SpecAugment: A Simple Data Augmentation Method for Automatic Speech Recognition.
In Proc. Interspeech 2019, pp. 2613–2617, 2019.

V. T. Pham, H. Xu, Y. Khassanov, Z. Zeng, E. S. Chng, C. Ni, B. Ma, H. Li.
Independent language modeling architecture for end-to-end asr.
In ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 7059–7063, 2020.

D. Povey, H. Hadian, P. Ghahremani, K. Li, S. Khudanpur.
A time-restricted self-attention layer for asr.
In 2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 5874–5878, 2018.

References

- R. Prabhavalkar, T. N. Sainath, Y. Wu, P. Nguyen, Z. Chen, C.-C. Chiu, A. Kannan.
  Minimum word error rate training for attention-based sequence-to-sequence models.
  Preprint arXiv:1712.01818, 2017.
- J. Rosendahl, V. A. K. Tran, W. Wang, H. Ney.
  Analysis of positional encodings for neural machine translation.
  In International Workshop on Spoken Language Translation, Hong Kong, China, Nov. 2019.
- D. Rybach, S. Hahn, P. Lehnen, D. Nolden, M. Sundermeyer, Z. Tüske, S. Wiesler, R. Schlüter, H. Ney.
  Rasr - the rwth aachen university open source speech recognition toolkit.
  In IEEE Automatic Speech Recognition and Understanding Workshop, Waikoloa, HI, USA, Dec. 2011.
- S. Sabour, W. Chan, M. Norouzi.
  Optimal completion distillation for sequence learning.
  Preprint arXiv:1810.01398, 2018.

References

- H. Sak, M. Shannon, K. Rao, F. Beaufays.
  Recurrent neural aligner: An encoder-decoder neural network model for sequence to sequence mapping.
  In Proc. Interspeech, Vol. 2017-Augus, pp. 1298–1302, 2017.
- J. Shen, P. Nguyen, Y. Wu, Z. Chen et al.
  Lingvo: a modular and scalable framework for sequence-to-sequence modeling.
  Preprint arXiv:1902.08295, 2019.
- J. Shen, R. Pang, R. J. Weiss, M. Schuster, N. Jaitly, Z. Yang, Z. Chen, Y. Zhang, Y. Wang, R. J. Skerry-Ryan, R. A. Saurous, Y. Agiomyrgiannakis, Y. Wu.
  Natural TTS synthesis by conditioning wavenet on mel spectrogram predictions.
  CoRR, Vol. abs/1712.05884, 2017.
- M. Sperber, G. Neubig, J. Niehues, A. Waibel.
  Attention-passing models for robust and data-efficient end-to-end speech translation.
  Transactions of the Association for Computational Linguistics, Vol. 7, pp. 313–325, Mar 2019.

References

A. Sriram, H. Jun, S. Satheesh, A. Coates.
Cold fusion: Training seq2seq models together with language models.
In 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings. OpenReview.net, 2018.

M. C. Stoian, S. Bansal, S. Goldwater.
Analyzing asr pretraining for low-resource speech-to-text translation.
ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Vol. , May 2020.

M. Sundermeyer, R. Schlüter, H. Ney.
LSTM neural networks for language modeling.
In Proc. Interspeech, pp. 194–197, Portland, OR, USA, Sept. 2012.

T. Sung, J. Liu, H. Lee, L. Lee.
Towards end-to-end speech-to-text translation with two-pass decoding.

References

In ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 7175–7179, 2019.

K. Tran, A. Bisazza, C. Monz.
Recurrent memory network for language modeling.
In Conf. of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT), pp. 321–331, San Diego, CA, USA, June 2016.

E. Tsunoo, Y. Kashiwagi, T. Kumakura, S. Watanabe.
Transformer asr with contextual block processing.
In 2019 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU), pp. 427–433, 2019.

Z. Tu, Z. Lu, Y. Liu, X. Liu, H. Li.
Modeling coverage for neural machine translation.
In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL, Volume 1: Long Papers, Berlin, Germany, 2016.

References

- A. Vaswani, S. Bengio, E. Brevdo, F. Chollet, A. N. Gomez, S. Gouws, L. Jones, L. Kaiser, N. Kalchbrenner, N. Parmar, R. Sepassi, N. Shazeer, J. Uszkoreit.
  Tensor2tensor for neural machine translation.
  Preprint arXiv:1803.07416, 2018.
- A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin.
  Attention is all you need.
  In Advances in neural information processing systems, pp. 5998–6008, 2017.
- Y. Wang, D. Stanton, Y. Zhang, R. J. Skerry-Ryan, E. Battenberg, J. Shor, Y. Xiao, Y. Jia, F. Ren, R. A. Saurous.
  Style tokens: Unsupervised style modeling, control and transfer in end-to-end speech synthesis.
  In Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, pp. 5167–5176, 10-15 July 2018.
- C. Wang, Y. Tang, X. Ma, A. Wu, D. Okhonko, J. Pino.

References

fairseq s2t: Fast speech-to-text modeling with fairseq.
In Proceedings of the 2020 Conference of the Asian Chapter of the Association for Computational Linguistics (AACL): System Demonstrations, 2020.

S. Watanabe, T. Hori, S. Kim, J. R. Hershey, T. Hayashi.
Hybrid ctc/attention architecture for end-to-end speech recognition.
IEEE Journal of Selected Topics in Signal Processing, Vol. 11, No. 8, pp. 1240–1253, 2017.

R. J. Weiss, J. Chorowski, N. Jaitly, Y. Wu, Z. Chen.
Sequence-to-sequence models can directly translate foreign speech.
In Interspeech 2017, 18th Annual Conference of the International Speech Communication Association, Stockholm, Sweden, Aug. 20-24, pp. 2625–2629, 2017.

T. Zenkel, M. Sperber, J. Niehues, M. Müller, N. Pham, S. Stüker, A. Waibel.
Open source toolkit for speech to text translation.
Prague Bull. Math. Linguistics, Vol. 111, pp. 125–135, 2018.

References

- A. Zeyer, T. Alkhouli, H. Ney.
  RETURNN as a generic flexible neural toolkit with application to translation and speech recognition.
  In Annual Meeting of the Assoc. for Computational Linguistics, Melbourne, Australia, July 2018.
- A. Zeyer, P. Bahar, K. Irie, R. Schlüter, H. Ney.
  A comparison of transformer and lstm encoder decoder models for asr.
  In IEEE Automatic Speech Recognition and Understanding Workshop, pp. 8–15, Sentosa, Singapore, Dec. 2019.
- A. Zeyer, K. Irie, R. Schlüter, H. Ney.
  Improved training of end-to-end attention models for speech recognition.
  In Interspeech, Hyderabad, India, Sept. 2018.
- A. Zeyer, A. Merboldt, R. Schlüter, H. Ney.
  A comprehensive analysis on attention models.

References

In Interpretability and Robustness in Audio, Speech, and Language (IRASL) Workshop, Conference on Neural Information Processing Systems (NeurIPS), Montreal, Canada, Dec. 2018.

A. Zeyer, A. Merboldt, R. Schlüter, H. Ney.
A new training pipeline for an improved neural transducer.
In Interspeech, http://www.interspeech2020.org/, Oct. 2020.

S. Zhang, H. Jiang, M. Xu, J. Hou, L. Dai.
The fixed-size ordinally-forgetting encoding method for neural network language models.
In Proc. of the Joint Conf. of the ACL and the Joint Conf. on Natural Language Processing of the AFNLP, pp. 495–500, Beijing, China, July 2015.

J. G. Zilly, R. K. Srivastava, J. Koutník, J. Schmidhuber.
Recurrent highway networks.
In Proc. Int. Conf. on Machine Learning (ICML), pp. 4189–4198, Sydney, Australia, Aug. 2017.