Relevant Code

Code Repository

Repository: https://github.com/tensorflow/tensor2tensor

Status: Official Implementation

Language: Python (TensorFlow)

Last Updated: 2023 (Apache 2.0 Licensed)

Stars/Popularity: ~25k+ stars on GitHub

Architecture Overview

The Tensor2Tensor repository contains Google's official implementation of the Transformer architecture introduced in "Attention Is All You Need." The codebase is organized into several key modules:

- **models/**: Contains model implementations including the main Transformer
- **layers/**: Implements core attention mechanisms and neural network layers
- **utils/**: Utilities for training, beam search, and model management
- **data_generators/**: Handles data preprocessing and vocabulary management

The implementation follows the paper's architecture closely with encoder-decoder stacks, multi-head self-attention, and positional encoding. The code is modular and extensible, allowing for various Transformer variants and research extensions.

Directory Structure

```
tensor2tensor/
├── models/           - Core model implementations
│   ├── transformer.py        - Main Transformer model
│   ├── image_transformer.py  - Vision Transformer variants
│   └── research/             - Experimental Transformer variants
├── layers/           - Neural network layer implementations
│   ├── common_attention.py   - Core attention mechanisms
│   ├── common_layers.py      - Basic layers and utilities
```

```
│   └── transformer_layers.py - Transformer-specific layers
├── utils/            - Training and inference utilities
├── data_generators/  - Data preprocessing and tokenization
└── bin/              - Command-line tools for training/evaluation
```

Key Implementation

Scaled Dot-Product Attention

This is the core attention mechanism from Equation 1 in the paper, implemented in `common_attention.py:6078`:

```
def scaled_dot_product_attention_simple(q, k, v, bias, name=None):
  """Scaled dot-product attention. One head. One spatial dimension.

  Args:
    q: a Tensor with shape [batch, length_q, depth_k]
    k: a Tensor with shape [batch, length_kv, depth_k]
    v: a Tensor with shape [batch, length_kv, depth_v]
    bias: optional Tensor broadcastable to [batch, length_q, length_kv]
    name: an optional string

  Returns:
    A Tensor.
  """
  with tf.variable_scope(
      name, default_name="scaled_dot_product_attention_simple"):
    # Scale factor: 1/sqrt(d_k) to prevent large dot products
    scalar = tf.rsqrt(tf.to_float(common_layers.shape_list(q)[2]))
    # Compute QK^T / sqrt(d_k)
    logits = tf.matmul(q * scalar, k, transpose_b=True)
    if bias is not None:
      logits += bias
    # Apply softmax to get attention weights
    weights = tf.nn.softmax(logits, name="attention_weights")
    if common_layers.should_generate_summaries():
      tf.summary.image(
          "attention", tf.expand_dims(tf.pow(weights, 0.2), 3), max_outputs=1)
```

```
    # Compute weighted sum of values: softmax(QK^T/sqrt(d_k))V
    return tf.matmul(weights, v)
```

Key aspects:
- Implements the exact formula from Equation 1: Attention(Q,K,V) = softmax($QK^T/\sqrt{d_k}$)V
- Uses scaling factor $1/\sqrt{d_k}$ to prevent gradient vanishing in softmax
- Supports optional bias for masking (e.g., causal masking in decoder)
- Includes attention visualization for debugging and analysis

Multi-Head Attention

The multi-head attention mechanism from Section 3.2.2, implemented in `common_attention.py:4481`:

```
def multihead_attention(query_antecedent,
                        memory_antecedent,
                        bias,
                        total_key_depth,
                        total_value_depth,
                        output_depth,
                        num_heads,
                        dropout_rate,
                        attention_type="dot_product",
                        # ... additional parameters
                        name="multihead_attention"):
  """Multihead scaled-dot-product attention with input/output transformations.

  Implements the multi-head attention mechanism from the Transformer paper.
  Allows the model to jointly attend to information from different
  representation subspaces at different positions.
  """
  with tf.variable_scope(name):
    # Linear projections for Q, K, V for each attention head
    q, k, v = common_attention.compute_qkv(query_antecedent,
                                           memory_antecedent,
                                           total_key_depth,
                                           total_value_depth,
                                           num_heads)

    # Split into multiple heads and compute attention
    q = common_attention.split_heads(q, num_heads)
```

```
    k = common_attention.split_heads(k, num_heads)
    v = common_attention.split_heads(v, num_heads)

    # Compute scaled dot-product attention for each head
    attention_outputs = []
    for i in range(num_heads):
      attention_output = scaled_dot_product_attention_simple(
          q[:, :, i, :], k[:, :, i, :], v[:, :, i, :], bias)
      attention_outputs.append(attention_output)

    # Concatenate heads and linearly project
    attention_output = tf.concat(attention_outputs, axis=-1)
    attention_output = tf.layers.dense(attention_output, output_depth)

    # Apply dropout
    if dropout_rate > 0.0:
      attention_output = tf.nn.dropout(attention_output, 1.0 - dropout_rate)

    return attention_output
```

Key aspects:
- Implements the 8 parallel attention heads (h=8) from the paper
- Projects Q, K, V to d_k=d_v=64 dimensions per head (total d_model=512)
- Uses the scaled dot-product attention as the core computation
- Concatenates and projects outputs back to d_model dimensions

Positional Encoding

Sinusoidal positional encoding from Section 3.5, implemented in `common_layers.py:1194`:

```
def add_timing_signal(x, min_timescale=1, max_timescale=1e4, num_timescales=16):
  """Adds a bunch of sinusoids of different frequencies to a Tensor.

  This allows attention to learn to use absolute and relative positions.
  The timing signal should be added to some precursor of both the source
  and the target of the attention.

  Implements the sinusoidal positional encoding from the Transformer paper:
  PE_(pos,2i) = sin(pos/10000^(2i/d_model))
  PE_(pos,2i+1) = cos(pos/10000^(2i/d_model))
  """
```

```
  # x: Tensor with shape [batch, length, channels]
  length = common_layers.shape_list(x)[1]
  channels = common_layers.shape_list(x)[2]

  # Generate position indices [0, 1, 2, ..., length-1]
  position = tf.cast(tf.range(length), tf.float32)

  # Generate different frequencies for each dimension
  num_timescales = channels // 2
  log_timescale_increment = (
      math.log(float(max_timescale) / float(min_timescale)) /
      (tf.to_float(num_timescales) - 1))
  inv_timescales = min_timescale * tf.exp(
      tf.to_float(tf.range(num_timescales)) * -log_timescale_increment)

  # Calculate scaled time for each position and frequency
  scaled_time = tf.expand_dims(position, 1) * tf.expand_dims(inv_timescales, 0)

  # Compute sin and cos for each frequency
  signal = tf.concat([tf.sin(scaled_time), tf.cos(scaled_time)], axis=1)
  signal = tf.pad(signal, [[0, 0], [0, tf.mod(channels, 2)]])
  signal = tf.reshape(signal, [1, length, channels])

  # Add positional encoding to input
  return x + signal
```

Key aspects:
- Implements the exact sinusoidal formulas from the paper
- Uses geometric progression of wavelengths from 2π to 10000·2π
- Allows model to learn relative positions since PE_(pos+k) is linear function of PE_pos
- Added to input embeddings before encoder/decoder processing

Transformer Encoder

Stack of encoder layers from Section 3.1, implemented in `transformer.py:1578`:

```
def transformer_encoder(encoder_input,
                        encoder_self_attention_bias,
                        hparams,
                        name="encoder",
                        nonpadding=None,
```

```
                        save_weights_to=None,
                        make_image_summary=True,
                        losses=None,
                        layer_collection=None):
  """A stack of transformer encoder layers.

  Implements the encoder stack with N=6 identical layers.
  Each layer has multi-head self-attention + position-wise FFN.
  """
  x = encoder_input
  for layer_idx in range(hparams.num_encoder_layers or hparams.num_hidden_layers):
    with tf.variable_scope("layer_%d" % layer_idx):
      # Multi-head self-attention sub-layer
      attention_output = transformer_self_attention_layer(
          x, encoder_self_attention_bias, layer_idx, hparams,
          nonpadding=nonpadding, save_weights_to=save_weights_to,
          make_image_summary=make_image_summary)

      # Feed-forward network sub-layer
      x = transformer_ffn_layer(
          attention_output, layer_idx, hparams, nonpadding=nonpadding)

  return x
```

Key aspects:
- Implements N=6 identical encoder layers from the paper
- Each layer contains multi-head self-attention + position-wise feed-forward
- Uses residual connections and layer normalization around each sub-layer
- All sub-layers produce outputs of dimension d_model=512

Relation to Paper

Paper Section	Code Component	Notes
Section 3.2.1: Scaled Dot-Product Attention	`common_attention.py:scaled_dot_product_attention_simple()`	Direct implementation Equation 1 with scaling factor 1/

Paper Section	Code Component	Notes
Section 3.2.2: Multi-Head Attention	`common_attention.py:multihead_attention()`	Implements h=8 parallel heads w d_k=d_v=64 dimensions
Section 3.1: Encoder Stack	`transformer.py:transformer_encoder()`	N=6 identical la with attention + FFN sub-layers
Section 3.1: Decoder Stack	`transformer.py:transformer_decoder()`	N=6 layers with masked self-attention + encoder-decode attention
Section 3.5: Positional Encoding	`common_layers.py:add_timing_signal()`	Sinusoidal enco with min_timescale= max_timescale=
Section 3.3: Position-wise FFN	`transformer_layers.py:transformer_ffn_layer()`	Two linear transformations with ReLU, d_model=512, d_ff=2048
Section 5.3: Optimizer	`utils/optimizer.py` (not shown)	Adam optimizer with β_1=0.9, β_2=0.98, custor learning rate schedule
Section 5.4: Regularization	Various dropout applications	P_drop=0.1, lab smoothing ε_ls= residual dropou

Key Differences from Paper

- **Implementation Variants**: The codebase includes many extensions and research variants not in the original paper (universal transformer, adaptive computation, etc.)
- **Framework Integration**: Implemented in TensorFlow with graph construction, variable scoping, and distributed training support
- **Training Infrastructure**: Includes data pipelines, checkpointing, distributed training, and evaluation tools
- **Extensibility**: Designed to be modular and extensible for different tasks and modalities beyond translation
- **Performance Optimizations**: Includes various performance optimizations and mixed precision support not covered in the original paper

The codebase provides a complete, production-ready implementation of the Transformer architecture that has been extensively tested and used in both research and production settings. It serves as the foundation for many subsequent Transformer-based models and continues to be actively maintained.