Relevant Code

Code Repository

Repository: https://github.com/google-research/bert

Status: Official Implementation

Language: Python (TensorFlow)

Last Updated: November 2018 (initial release)

Stars/Popularity: 37k+ GitHub stars

Architecture Overview

The BERT repository contains a comprehensive TensorFlow implementation of the Bidirectional Encoder Representations from Transformers model. The codebase is structured to support both pre-training and fine-tuning workflows, with clear separation between model architecture, training utilities, and task-specific applications.

The repository follows a modular design where the core Transformer architecture is implemented in `modeling.py`, while task-specific training scripts like `run_classifier.py` for sentence classification and `run_squad.py` for question answering handle the fine-tuning process. The tokenization system in `tokenization.py` implements WordPiece tokenization with support for both cased and uncased models.

Directory Structure

```
bert/
├── modeling.py           - Core BERT model architecture and Transformer layers
├── run_classifier.py     - Fine-tuning script for sentence/sentence-pair classification
├── run_squad.py          - Fine-tuning script for SQuAD question answering
├── run_pretraining.py    - Pre-training script for masked LM and next sentence prediction
├── create_pretraining_data.py - Data preprocessing for pre-training
├── extract_features.py   - Feature extraction without fine-tuning
├── tokenization.py       - WordPiece tokenization and input processing
├── optimization.py       - Learning rate schedule and optimization utilities
```

```
├── sample_text.txt      - Sample text for pre-training demonstration
└── README.md           - Comprehensive documentation and usage examples
```

Key Implementation

BERT Model Architecture

The core implementation in `modeling.py` defines the BERT model structure based on the Transformer encoder:

```
class BertModel(object):
  """BERT model ("Bidirectional Encoder Representations from Transformers").

  Example usage:
  ```python
  # Already been converted into WordPiece token ids
  input_ids = tf.constant([[31, 51, 99], [15, 5, 0]])
  input_mask = tf.constant([[1, 1, 1], [1, 1, 0]])
  token_type_ids = tf.constant([[0, 0, 1], [0, 2, 0]])

  config = modeling.BertConfig(vocab_size=32000, hidden_size=512,
    num_hidden_layers=8, num_attention_heads=6, intermediate_size=1024)

  model = modeling.BertModel(config=config, is_training=True,
    input_ids=input_ids, input_mask=input_mask, token_type_ids=token_type_ids)

  label_embeddings = tf.get_variable(...)
  pooled_output = model.get_pooled_output()
  logits = tf.matmul(pooled_output, label_embeddings)
  ```
  """

  def __init__(self,
               config,
               is_training,
               input_ids,
               input_mask=None,
               token_type_ids=None,
               use_one_hot_embeddings=False,
               scope=None):
```

```
    """Constructor for BertModel.

    Args:
      config: `BertConfig` instance.
      is_training: bool. true for training model, false for eval model. Controls
        whether dropout will be applied.
      input_ids: int32 Tensor of shape [batch_size, seq_length].
      input_mask: (optional) int32 Tensor of shape [batch_size, seq_length].
      token_type_ids: (optional) int32 Tensor of shape [batch_size, seq_length].
      use_one_hot_embeddings: (optional) bool. Whether to use one-hot word
        embeddings or tf.embedding_lookup() for the word embeddings.
      scope: (optional) variable scope. Defaults to "bert".

    Raises:
      ValueError: The config is invalid or one of the input tensor shapes
        is invalid.
    """
    config = copy.deepcopy(config)
    if not is_training:
      config.hidden_dropout_prob = 0.0
      config.attention_probs_dropout_prob = 0.0

    input_shape = get_shape_list(input_ids, expected_rank=2)
    batch_size = input_shape[0]
    seq_length = input_shape[1]

    if input_mask is None:
      input_mask = tf.ones(shape=[batch_size, seq_length], dtype=tf.int32)

    if token_type_ids is None:
      token_type_ids = tf.zeros(shape=[batch_size, seq_length], dtype=tf.int32)

    with tf.variable_scope(scope, default_name="bert"):
      with tf.variable_scope("embeddings"):
        # Perform embedding lookup on the word ids.
        (self.embedding_output, self.embedding_table) = embedding_lookup(
            input_ids=input_ids,
            vocab_size=config.vocab_size,
            embedding_size=config.hidden_size,
            initializer_range=config.initializer_range,
            word_embedding_name="word_embeddings",
```

```
        use_one_hot_embeddings=use_one_hot_embeddings)

    # Add positional embeddings and token type embeddings, then layer
    # normalize and perform dropout.
    self.embedding_output = embedding_postprocessor(
        input_tensor=self.embedding_output,
        use_token_type=True,
        token_type_ids=token_type_ids,
        token_type_vocab_size=config.type_vocab_size,
        token_type_embedding_name="token_type_embeddings",
        use_position_embeddings=True,
        position_embedding_name="position_embeddings",
        initializer_range=config.initializer_range,
        max_position_embeddings=config.max_position_embeddings,
        dropout_prob=config.hidden_dropout_prob)

  # This converts a 2D mask of shape [batch_size, seq_length] to a 3D
  # mask of shape [batch_size, seq_length, seq_length] which is used
  # for the attention scores.
  attention_mask = create_attention_mask_from_input_mask(
      input_ids, input_mask)

  # Run the stacked transformer.
  # `sequence_output` shape = [batch_size, seq_length, hidden_size].
  self.all_encoder_layers = transformer_model(
      input_tensor=self.embedding_output,
      attention_mask=attention_mask,
      hidden_size=config.hidden_size,
      num_hidden_layers=config.num_hidden_layers,
      num_attention_heads=config.num_attention_heads,
      intermediate_size=config.intermediate_size,
      intermediate_act=config.hidden_act,
      hidden_dropout_prob=config.hidden_dropout_prob,
      attention_probs_dropout_prob=config.attention_probs_dropout_prob,
      initializer_range=config.initializer_range,
      do_return_all_layers=True)

  self.sequence_output = self.all_encoder_layers[-1]
  # The "pooler" converts the encoded sequence tensor of shape
  # [batch_size, seq_length, hidden_size] to a tensor of shape
  # [batch_size, hidden_size]. This is necessary for segment-level
```

```
      # (or segment-pair-level) classification tasks in which we need a fixed
      # dimensional representation of the segment.
      with tf.variable_scope("pooler"):
        # We "pool" the model by simply taking the hidden state corresponding
        # to the first token. We assume that this has been pre-trained
        first_token_tensor = tf.squeeze(self.sequence_output[:, 0:1, :], axis=1)
        self.pooled_output = tf.layers.dense(
            first_token_tensor,
            config.hidden_size,
            activation=tf.tanh,
            kernel_initializer=create_initializer(config.initializer_range))
```

Key aspects:
- Implements bidirectional Transformer encoder with masked self-attention
- Supports both pre-training and fine-tuning modes through `is_training` flag
- Provides `pooled_output` for classification tasks and `sequence_output` for token-level tasks
- Includes dropout and layer normalization as specified in the original paper

Transformer Encoder Layer

The core Transformer implementation shows the multi-head attention mechanism:

```
def attention_layer(from_tensor,
                    to_tensor,
                    attention_mask=None,
                    num_attention_heads=1,
                    size_per_head=512,
                    query_act=None,
                    key_act=None,
                    value_act=None,
                    attention_probs_dropout_prob=0.0,
                    initializer_range=0.02,
                    do_return_2d_tensor=False,
                    batch_size=None,
                    from_seq_length=None,
                    to_seq_length=None):
  """Performs multi-headed attention from `from_tensor` to `to_tensor`.

  This is an implementation of multi-headed attention based on "Attention
  is all you Need". If `from_tensor` and `to_tensor` are the same, then
```

```
this is self-attention. Each timestep in `from_tensor` attends to the
corresponding sequence in `to_tensor`, and returns a fixed-with vector.

This function first projects `from_tensor` into a "query" tensor and
`to_tensor` into "key" and "value" tensors. These are (effectively) a list
of tensors of length `num_attention_heads`, where each tensor is of shape
[batch_size, seq_length, size_per_head].

Then, the query and key tensors are dot-producted and scaled. These are
softmaxed to obtain attention probabilities. The value tensors are then
interpolated by these probabilities, then concatenated back to a single
tensor and returned.

In practice, the multi-headed attention can be implemented using
transposition and reshaping rather than actual independent tensors.
"""

def transpose_for_scores(input_tensor, batch_size, seq_length, num_attention_heads, size_per_head
  """Transpose the 3D tensor to prepare for matrix multiplication."""
  output_tensor = tf.reshape(
      input_tensor, [batch_size, seq_length, num_attention_heads, size_per_head])
  output_tensor = tf.transpose(output_tensor, [0, 2, 1, 3])
  return output_tensor

# Scalar dimensions referenced here:
#   B = batch size (number of sequences)
#   F = `from_tensor` sequence length
#   T = `to_tensor` sequence length
#   N = number of attention heads
#   H = size per head

# `query_layer = [B*F, N*H]`
query_layer = tf.layers.dense(
    from_tensor,
    num_attention_heads * size_per_head,
    activation=query_act,
    name="query",
    kernel_initializer=create_initializer(initializer_range))

# `key_layer = [B*T, N*H]`
key_layer = tf.layers.dense(
```

```
    to_tensor, num_attention_heads * size_per_head, activation=key_act,
    name="key", kernel_initializer=create_initializer(initializer_range))

# `value_layer = [B*T, N*H]`
value_layer = tf.layers.dense(
    to_tensor, num_attention_heads * size_per_head, activation=value_act,
    name="value", kernel_initializer=create_initializer(initializer_range))

# `query_layer` = [B, N, F, H]
query_layer = transpose_for_scores(query_layer, batch_size, from_seq_length,
                                   num_attention_heads, size_per_head)

# `key_layer` = [B, N, T, H]
key_layer = transpose_for_scores(key_layer, batch_size, to_seq_length,
                                 num_attention_heads, size_per_head)

# Take the dot product between "query" and "key" to get the raw
# attention scores.
# `attention_scores = [B, N, F, T]`
attention_scores = tf.matmul(query_layer, key_layer, transpose_b=True)
attention_scores = tf.multiply(attention_scores,
                               1.0 / math.sqrt(float(size_per_head)))

if attention_mask is not None:
  # `attention_mask` = [B, 1, F, T]
  attention_mask = tf.expand_dims(attention_mask, axis=[1])

  # Since attention_mask is 1.0 for positions we want to attend and 0.0 for
  # masked positions, this operation will create a tensor which is 0.0 for
  # positions we want to attend and -10000.0 for masked positions.
  adder = (1.0 - tf.cast(attention_mask, tf.float32)) * -10000.0

  # Since we are adding it to the raw scores before the softmax, this is
  # effectively the same as removing these entirely.
  attention_scores += adder

# Normalize the attention scores to probabilities.
# `attention_probs = [B, N, F, T]`
attention_probs = tf.nn.softmax(attention_scores)

# This is actually dropping out entire tokens to attend to, which might
```

```
  # seem a bit unusual, but is taken from the original Transformer paper.
  attention_probs = dropout(attention_probs, attention_probs_dropout_prob)

  # `value_layer = [B, T, N, H]`
  value_layer = tf.reshape(
      value_layer,
      [batch_size, to_seq_length, num_attention_heads, size_per_head])

  # `value_layer = [B, N, T, H]`
  value_layer = tf.transpose(value_layer, [0, 2, 1, 3])

  # `context_layer = [B, N, F, H]`
  context_layer = tf.matmul(attention_probs, value_layer)

  # `context_layer = [B, F, N, H]`
  context_layer = tf.transpose(context_layer, [0, 2, 1, 3])

  if do_return_2d_tensor:
    # `context_layer = [B*F, N*H]`
    context_layer = tf.reshape(
        context_layer,
        [batch_size * from_seq_length, num_attention_heads * size_per_head])
  else:
    # `context_layer = [B, F, N*H]`
    context_layer = tf.reshape(
        context_layer,
        [batch_size, from_seq_length, num_attention_heads * size_per_head])

  return context_layer
```

Fine-tuning for Classification Tasks

The `run_classifier.py` script demonstrates how BERT is fine-tuned for sentence-level tasks:

```
def create_model(bert_config, is_training, input_ids, input_mask, segment_ids,
                 labels, num_labels, use_one_hot_embeddings):
  """Creates a classification model."""
  model = modeling.BertModel(
      config=bert_config,
```

```
      is_training=is_training,
      input_ids=input_ids,
      input_mask=input_mask,
      token_type_ids=segment_ids,
      use_one_hot_embeddings=use_one_hot_embeddings)

  # In the demo, we are doing a simple classification task on the final
  # hidden state of the [CLS] token of the sequence.
  output_layer = model.get_pooled_output()

  hidden_size = output_layer.shape[-1].value

  output_weights = tf.get_variable(
      "output_weights", [num_labels, hidden_size],
      initializer=tf.truncated_normal_initializer(stddev=0.02))

  output_bias = tf.get_variable(
      "output_bias", [num_labels], initializer=tf.zeros_initializer())

  with tf.variable_scope("loss"):
    if is_training:
      # I.e., 0.1 dropout
      output_layer = tf.nn.dropout(output_layer, keep_prob=0.9)

    logits = tf.matmul(output_layer, output_weights, transpose_b=True)
    logits = tf.nn.bias_add(logits, output_bias)
    probabilities = tf.nn.softmax(logits, axis=-1)
    log_probs = tf.nn.log_softmax(logits, axis=-1)

    one_hot_labels = tf.one_hot(labels, depth=num_labels, dtype=tf.float32)

    per_example_loss = -tf.reduce_sum(one_hot_labels * log_probs, axis=-1)
    loss = tf.reduce_mean(per_example_loss)

    return (loss, per_example_loss, logits, probabilities)
```

Key aspects:
- Uses the [CLS] token's pooled output for classification
- Adds a simple classification layer on top of pre-trained BERT
- Implements cross-entropy loss for multi-class classification
- Includes dropout during training for regularization

SQuAD Question Answering Implementation

The `run_squad.py` shows how BERT handles span prediction for question answering:

```
def create_model(bert_config, is_training, input_ids, input_mask, segment_ids,
                 start_positions, end_positions, use_one_hot_embeddings):
  """Creates a Q&A model."""
  model = modeling.BertModel(
      config=bert_config,
      is_training=is_training,
      input_ids=input_ids,
      input_mask=input_mask,
      token_type_ids=segment_ids,
      use_one_hot_embeddings=use_one_hot_embeddings)

  # Get the sequence output for the Q&A task
  sequence_output = model.get_sequence_output()

  # Create start and end vectors for span prediction
  hidden_size = sequence_output.shape[-1].value

  output_weights = tf.get_variable(
      "cls/squad/output_weights", [2, hidden_size],
      initializer=tf.truncated_normal_initializer(stddev=0.02))

  output_bias = tf.get_variable(
      "cls/squad/output_bias", [2],
      initializer=tf.zeros_initializer())

  final_hidden = tf.layers.dropout(
      sequence_output, rate=0.1, training=is_training)

  logits = tf.matmul(final_hidden, output_weights, transpose_b=True)
  logits = tf.nn.bias_add(logits, output_bias)

  # Split into start and end logits
  start_logits, end_logits = tf.split(logits, axis=1, num_or_size_splits=2)
  start_logits = tf.squeeze(start_logits, axis=-1)
  end_logits = tf.squeeze(end_logits, axis=-1)
```

```
  return (start_logits, end_logits)
```

Key aspects:
- Uses sequence output rather than pooled output for token-level predictions
- Implements separate start and end position classifiers for span prediction
- Uses the same BERT parameters for different tasks with minimal modifications

Pre-training Data Creation

The `create_pretraining_data.py` implements the masked language model and next sentence prediction:

```
def create_instances_from_document(
    all_documents, document_index, max_seq_length, short_seq_prob,
    masked_lm_prob, max_predictions_per_seq, vocab_words, rng):
  """Create `TrainingInstance`s from a `document`."""

  document = all_documents[document_index]

  # Account for [CLS], [SEP], [SEP]
  max_num_tokens = max_seq_length - 3

  # We *usually* want to fill up the entire sequence since we are padding
  # to `max_seq_length` anyways, so short sequences are generally wasted
  # computation. However, we *sometimes*
  # (i.e., short_seq_prob == 0.1 == 10% of the time) want to use shorter
  # sequences to minimize the mismatch between pre-training and fine-tuning.
  # The `target_seq_length` is just a rough target however, whereas
  # `max_seq_length` is a hard limit.
  target_seq_length = max_num_tokens
  if rng.random() < short_seq_prob:
    target_seq_length = rng.randint(2, max_num_tokens)

  # We DON'T just concatenate all of the tokens from a document into a long
  # sequence and choose an arbitrary split point because this would make the
  # next sentence prediction task too easy. Instead, we split the input into
  # segments "A" and "B" based on the actual "Next Sentence" boundaries.
  current_chunk = []
  current_length = 0
  i = 0
```

```
while i < len(document):
  segment = document[i]
  current_chunk.append(segment)
  current_length += len(segment)
  if i == len(document) - 1 or current_length >= target_seq_length:
    if current_chunk:
      # `a_end` is how many tokens from the current chunk will go
      # into the `A` (first) sentence.
      a_end = 1
      if len(current_chunk) >= 2:
        a_end = rng.randint(1, len(current_chunk) - 1)

      tokens_a = []
      for j in range(a_end):
        tokens_a.extend(current_chunk[j])

      tokens_b = []
      # Random next
      is_random_next = False
      if len(current_chunk) == 1 or rng.random() < 0.5:
        is_random_next = True
        target_b_length = target_seq_length - len(tokens_a)

        # This should rarely go for more than one iteration for large
        # corpora. However, just to be careful, we try to make sure that
        # the random document is not the same as the document
        # we're processing.
        for _ in range(10):
          random_document_index = rng.randint(0, len(all_documents) - 1)
          if random_document_index != document_index:
            break

        random_document = all_documents[random_document_index]
        random_start = rng.randint(0, len(random_document) - 1)
        for j in range(random_start, len(random_document)):
          tokens_b.extend(random_document[j])
          if len(tokens_b) >= target_b_length:
            break
        # We didn't actually use these segments, so we "put them back" so
        # we don't have to go to the next one to get the next segment.
```

```
      # However, if `random_document_index` was the last document,
      # we need to increment `document_index` by one.
      if random_document_index == document_index:
        document_index += 1
    # Actual next
    else:
      is_random_next = False
      for j in range(a_end, len(current_chunk)):
        tokens_b.extend(current_chunk[j])
      document_index += 1

    truncate_seq_pair(tokens_a, tokens_b, max_num_tokens, rng)

    tokens = []
    segment_ids = []
    tokens.append("[CLS]")
    segment_ids.append(0)
    for token in tokens_a:
      tokens.append(token)
      segment_ids.append(0)

    tokens.append("[SEP]")
    segment_ids.append(0)

    for token in tokens_b:
      tokens.append(token)
      segment_ids.append(1)

    tokens.append("[SEP]")
    segment_ids.append(1)

    # Masked Language Model
    (tokens, masked_lm_positions, masked_lm_labels) = create_masked_lm_predictions(
        tokens, masked_lm_prob, max_predictions_per_seq, vocab_words, rng)

    instance = TrainingInstance(
        tokens=tokens,
        segment_ids=segment_ids,
        is_random_next=is_random_next,
        masked_lm_positions=masked_lm_positions,
        masked_lm_labels=masked_lm_labels)
```

```
        instances.append(instance)
      current_chunk = []
      current_length = 0
      i += 1
    else:
      i += 1

  return instances

def create_masked_lm_predictions(tokens, masked_lm_prob, max_predictions_per_seq, vocab_words, rng)
  """Creates the predictions for the masked LM objective."""

  cand_indexes = []
  for (i, token) in enumerate(tokens):
    if token == "[CLS]" or token == "[SEP]":
      continue
    # Whole Word Masking means that if we mask all of the wordpieces
    # corresponding to an original word. When a word has been split into
    # WordPieces, the first token does not have any marker and any subsequence
    # tokens are prefixed with ##. So whenever we see the ## token, we
    # append it to the previous set of word indexes.
    #
    # Note that Whole Word Masking does *not* change the training code
    # at all -- we still predict each WordPiece idiomatically, and simply
    # mask the labels of all but the first subword. The code in create_pretraining_data.py
    # is identical.
    if len(cand_indexes) >= 1 and token.startswith("##"):
      cand_indexes[-1].append(i)
    else:
      cand_indexes.append([i])

  rng.shuffle(cand_indexes)

  output_tokens = list(tokens)

  num_to_predict = min(max_predictions_per_seq,
                       max(1, int(round(len(tokens) * masked_lm_prob))))

  masked_lms = []
  covered_indexes = set()
  for index_set in cand_indexes:
```

```
  if len(masked_lms) >= num_to_predict:
    break
  # If adding a whole-word mask would exceed the number of predictions,
  # just skip this candidate.
  if len(masked_lms) + len(index_set) > num_to_predict:
    continue
  is_any_index_covered = False
  for index in index_set:
    if index in covered_indexes:
      is_any_index_covered = True
      break
  if is_any_index_covered:
    continue
  for index in index_set:
    covered_indexes.add(index)

    masked_token = None
    # 80% of the time, replace with [MASK]
    if rng.random() < 0.8:
      masked_token = "[MASK]"
    else:
      # 10% of the time, keep original
      if rng.random() < 0.5:
        masked_token = tokens[index]
      # 10% of the time, replace with random word
      else:
        masked_token = vocab_words[rng.randint(0, len(vocab_words) - 1)]

    output_tokens[index] = masked_token
    masked_lms.append((index, tokens[index]))

masked_lms = sorted(masked_lms, key=lambda x: x[0])
masked_lm_positions = []
masked_lm_labels = []
for (index, token) in masked_lms:
  masked_lm_positions.append(index)
  masked_lm_labels.append(token)

return (output_tokens, masked_lm_positions, masked_lm_labels)
```

Key aspects:
- Implements the 15% masking strategy (80% [MASK], 10% random, 10% original)
- Creates positive (IsNext) and negative (NotNext) sentence pairs for NSP task
- Supports whole word masking for better pre-training quality
- Maintains document structure while creating training instances

Relation to Paper

Paper Section	Code Component	Notes
Section 3: BERT	`modeling.py:BertModel`	Complete implementation of bidirectional Transformer encoder
Section 3.1: Masked LM	`create_pretraining_data.py:create_masked_lm_predictions`	15% masking with 80/10/10 strategy
Section 3.1: Next Sentence Prediction	`create_pretraining_data.py:create_instances_from_document`	50% IsNext/ NotNext sentence pairs
Figure 1: Architecture	`modeling.py` visual implementation	Matches the paper's pre-training/fine-tuning diagram
Section 3.2: Fine-tuning	`run_classifier.py`, `run_squad.py`	Task-specific fine-tuning implementations
Equation (1): MLM Loss	Pre-training loss computation	Standard cross-entropy for masked token prediction
Table 1: Results	GLUE and SQuAD fine-tuning scripts	Reproduces paper's

Paper Section	Code Component	Notes
		benchmark results

Key Differences from Paper

- **Whole Word Masking**: The implementation later added whole word masking which wasn't in the original paper but improves performance
- **Optimization**: The code uses Adam with linear warmup as described, but includes additional hyperparameter tuning options
- **Multi-lingual Support**: Extended support for multiple languages beyond the original English-only models
- **Model Variants**: Includes smaller BERT variants (Tiny, Mini, Small, Medium) not in the original paper