Complete Implementation Guide: Adding Embedding Support for Decoder-Only Models in CTranslate2

Bottom line: Adding embedding extraction for decoder-only models like Qwen3 requires modifications across **12-15 files** spanning Python converters, C++ runtime, and Python bindings. The critical change is intercepting hidden states **after final layer norm but before lm_head projection** in the decoder forward pass. Two implementation paths exist: extending the Generator class or creating a dedicated DecoderEmbedder class.

The interception point for hidden states

The decoder's forward pass computes hidden states that are then projected to vocabulary logits. Your embedding extraction must capture the tensor **after** `_output_norm` **but before** `_proj`:

```cpp
// Inside TransformerDecoder::operator() - the key modification point
_output_norm(hidden, hidden);  // ← HIDDEN STATES AVAILABLE HERE (shape: [batch, seq, hidden_dim
_proj(hidden, *logits);        // ← This projects to vocab logits (shape: [batch, seq, vocab_size])
```

The hidden states at this point represent the model's final contextualized representations—exactly what embedding models extract.

File modifications organized by component

1. Python Conversion Layer (Delta from existing Qwen3Loader)

File: `python/ctranslate2/converters/transformers.py`

Add a new `Qwen3EmbeddingLoader` class that skips lm_head projection:

python

```python
class Qwen3EmbeddingLoader(ModelLoader):
    """Loader for Qwen3 embedding models (without lm_head projection)."""

    @property
    def architecture_name(self):
        return "Qwen3ForEmbedding"  # Custom architecture for embedding variant

    def get_model_spec(self, model):
        config = self._config
        # Use TransformerDecoderModelSpec but mark as embedding model
        spec = transformer_spec.TransformerDecoderModelSpec.from_config(
            num_layers=config.num_hidden_layers,
            num_heads=config.num_attention_heads,
            num_heads_kv=getattr(config, 'num_key_value_heads', config.num_attention_heads),
            pre_norm=True,
            activation=common_spec.Activation.SWISH,  # SiLU/SWISH for Qwen3
            rms_norm=True,
            rotary_dim=0,  # Full rotary (applied to all dimensions)
            rotary_interleave=False,
            rotary_base=getattr(config, 'rope_theta', 10000),
            ffn_glu=True,
            # qk_norm=True,  # Qwen3-specific - requires C++ support
        )
        # Mark as embedding model in config
        spec._config.add_attribute("model_purpose", "embedding")
        spec._config.add_attribute("pooling_method", "last_token")  # Qwen3 Embedding uses last token
        return spec

    def set_decoder(self, spec, module):
        """Set decoder weights WITHOUT lm_head projection."""
        # Embeddings
        self.set_embeddings(spec.embeddings, module.model.embed_tokens)

        # Transformer layers
        for layer_spec, layer in zip(spec.layer, module.model.layers):
            self._set_decoder_layer(layer_spec, layer)

        # Final layer norm
```

```
        self.set_layer_norm(spec.layer_norm, module.model.norm)

        # CRITICAL: Skip lm_head for embedding models
        # spec.projection is left unset (OPTIONAL weight)

    def _set_decoder_layer(self, spec, layer):
        # Self-attention with Q/K/V projections
        split_layers = [common_spec.LinearSpec() for _ in range(3)]
        self.set_linear(split_layers[0], layer.self_attn.q_proj)
        self.set_linear(split_layers[1], layer.self_attn.k_proj)
        self.set_linear(split_layers[2], layer.self_attn.v_proj)
        utils.fuse_linear(spec.self_attention.linear[0], split_layers)
        self.set_linear(spec.self_attention.linear[1], layer.self_attn.o_proj)

        # Qwen3 QK-norm (if C++ supports it)
        # self.set_layer_norm(spec.self_attention.q_norm, layer.self_attn.q_norm)
        # self.set_layer_norm(spec.self_attention.k_norm, layer.self_attn.k_norm)

        # FFN with SwiGLU
        self.set_linear(spec.ffn.linear_0, layer.mlp.gate_proj)
        self.set_linear(spec.ffn.linear_0_noact, layer.mlp.up_proj)
        self.set_linear(spec.ffn.linear_1, layer.mlp.down_proj)

        # Layer norms
        self.set_layer_norm(spec.self_attention.layer_norm, layer.input_layernorm)
        self.set_layer_norm(spec.ffn.layer_norm, layer.post_attention_layernorm)
```

Detection function to distinguish embedding vs generation models:

```python
def detect_model_purpose(model, config):
    """Detect if HuggingFace model is for embeddings or generation."""
    model_class = type(model).__name__

    # Check class name patterns
    embedding_indicators = ['ForSequenceEmbedding', 'Model', 'Encoder']
    if any(ind in model_class for ind in embedding_indicators):
        return "embedding"

    # Check if lm_head exists and has weights
    if not hasattr(model, 'lm_head') or model.lm_head is None:
        return "embedding"

    # Check architecture field in config
    if hasattr(config, 'architectures'):
        for arch in config.architectures:
            if 'Embed' in arch or 'Sentence' in arch:
                return "embedding"

    return "generation"
```

---

2. Model Specification Changes

File: `python/ctranslate2/specs/transformer_spec.py`

Extend `TransformerDecoderModelSpec` to support optional projection:

```python
class TransformerDecoderModelSpec(model_spec.LanguageModelSpec):
  """Describes a Transformer decoder model (e.g. GPT-2, Llama, Qwen3)."""

  @classmethod
  def from_config(cls, num_layers, num_heads, ...,
                  skip_output_projection=False,  # NEW PARAMETER
                  pooling_method="none",         # NEW: none, mean, last_token, max
                  **kwargs):
    spec = cls(
      TransformerDecoderSpec(
        num_layers, num_heads,
        with_output_projection=not skip_output_projection,  # Control projection
        **kwargs
      )
    )
    spec._config.add_attribute("pooling_method", pooling_method)
    return spec
```

File: **python/ctranslate2/specs/attention_spec.py**

Add QK-norm support for Qwen3:

```python
class MultiHeadAttentionSpec(model_spec.LayerSpec):
  def __init__(self, self_attention=False, relative_attention_bias=False,
               relative_position_keys=False, relative_position_values=False,
               rotary_dim=None, rotary_interleave=True,
               qk_norm=False):  # NEW PARAMETER
    self.linear = [common_spec.LinearSpec() for _ in range(2)]
    self.layer_norm = common_spec.LayerNormSpec()

    # NEW: QK-norm for Qwen3
    if qk_norm:
      self.q_norm = common_spec.LayerNormSpec()
      self.k_norm = common_spec.LayerNormSpec()
```

---

3. C++ Layer Changes (Core Runtime)

File: `include/ctranslate2/layers/decoder.h`

Add parameter to return hidden states:

```cpp
namespace ctranslate2 {
namespace layers {

// New output struct for embedding extraction
struct DecoderForwardOutput {
    StorageView logits;
    StorageView hidden_states;  // NEW: pre-projection hidden states
};

class Decoder : public Layer {
public:
    // Extended operator with hidden states output
    virtual void operator()(dim_t step,
                            const StorageView& ids,
                            const StorageView* lengths,
                            DecoderState& state,
                            StorageView* logits = nullptr,
                            StorageView* attention = nullptr,
                            StorageView* hidden_states = nullptr) = 0;  // NEW PARAMETER

    // Flag to skip output projection (for embedding models)
    bool has_output_projection() const { return _has_projection; }

protected:
    bool _has_projection = true;
};

}}
```

File: `src/layers/decoder.cc`

Modify `TransformerDecoder::operator()` to optionally return hidden states:

cpp

```cpp
void TransformerDecoder::operator()(dim_t step,
                                    const StorageView& ids,
                                    const StorageView* lengths,
                                    DecoderState& state,
                                    StorageView* logits,
                                    StorageView* attention,
                                    StorageView* hidden_states) {
  // 1. Embedding lookup
  StorageView hidden;
  _embeddings(ids, hidden);

  // 2. Position encoding
  if (_position_encoder)
    (*_position_encoder)(hidden, step);

  // 3. Pass through transformer layers
  for (const auto& layer : _layers) {
    (*layer)(step, hidden, state, attention);
  }

  // 4. Final layer normalization
  if (_output_norm)
    (*_output_norm)(hidden, hidden);

  // NEW: Capture hidden states BEFORE projection
  if (hidden_states) {
    *hidden_states = hidden.copy();  // Copy pre-projection hidden states
  }

  // 5. Project to vocabulary (skip for embedding models)
  if (logits && _proj && has_output_projection()) {
    (*_proj)(hidden, *logits);
  }
}
```

4. C++ Model Layer Changes

File: `include/ctranslate2/models/language_model.h`

Add embedding extraction method:

cpp

```cpp
namespace ctranslate2 {
namespace models {

class LanguageModel : public Model {
public:
  // Existing methods...
  std::vector<GenerationResult> generate(...);
  StorageView forward(const StorageView& ids, ...);

  // NEW: Embedding extraction methods
  struct EmbeddingResult {
    StorageView embeddings;      // Shape: [batch, hidden_dim] after pooling
    StorageView hidden_states;   // Shape: [batch, seq_len, hidden_dim] raw
  };

  EmbeddingResult embed(const std::vector<std::vector<size_t>>& tokens,
                        const std::string& pooling = "last_token");

  EmbeddingResult embed_batch(const std::vector<std::vector<size_t>>& tokens,
                              const std::string& pooling = "last_token",
                              bool normalize = true);

private:
  // NEW: Pooling implementations
  void pool_last_token(const StorageView& hidden_states,
                       const StorageView& lengths,
                       StorageView& pooled);
  void pool_mean(const StorageView& hidden_states,
                 const StorageView& lengths,
                 StorageView& pooled);
  void pool_max(const StorageView& hidden_states,
                StorageView& pooled);
};

}}
```

File: `src/models/language_model.cc`

Implement embedding extraction:

cpp

```cpp
LanguageModel::EmbeddingResult LanguageModel::embed_batch(
    const std::vector<std::vector<size_t>>& tokens,
    const std::string& pooling,
    bool normalize) {

    EmbeddingResult result;

    // Prepare inputs
    StorageView ids = make_sequence_inputs(tokens, _device);
    StorageView lengths = compute_lengths(tokens, _device);

    // Forward pass with hidden states capture
    DecoderState state = _decoder->initial_state();
    StorageView hidden_states(_decoder->output_type(), _device);

    // Call decoder without requesting logits, only hidden states
    (*_decoder)(/*step=*/0, ids, &lengths, state,
                /*logits=*/nullptr,
                /*attention=*/nullptr,
                /*hidden_states=*/&hidden_states);  // NEW: Request hidden states

    result.hidden_states = std::move(hidden_states);

    // Apply pooling
    StorageView pooled(_decoder->output_type(), _device);
    if (pooling == "last_token") {
        pool_last_token(result.hidden_states, lengths, pooled);
    } else if (pooling == "mean") {
        pool_mean(result.hidden_states, lengths, pooled);
    } else if (pooling == "max") {
        pool_max(result.hidden_states, pooled);
    }

    // Optional L2 normalization
    if (normalize) {
        ops::L2Norm()(pooled, pooled);
    }
```

```
    result.embeddings = std::move(pooled);
    return result;
}

void LanguageModel::pool_last_token(const StorageView& hidden_states,
                                    const StorageView& lengths,
                                    StorageView& pooled) {
    // hidden_states: [batch, seq_len, hidden_dim]
    // lengths: [batch] - actual sequence lengths
    // pooled: [batch, hidden_dim]

    const dim_t batch_size = hidden_states.dim(0);
    const dim_t hidden_dim = hidden_states.dim(2);

    pooled.resize({batch_size, hidden_dim});

    for (dim_t b = 0; b < batch_size; ++b) {
        dim_t last_idx = lengths.at<int32_t>(b) - 1;
        // Copy hidden_states[b, last_idx, :] to pooled[b, :]
        ops::Copy()(hidden_states.index({b, last_idx}), pooled.index({b}));
    }
}

void LanguageModel::pool_mean(const StorageView& hidden_states,
                              const StorageView& lengths,
                              StorageView& pooled) {
    // Mean pooling over sequence dimension, respecting actual lengths
    const dim_t batch_size = hidden_states.dim(0);
    const dim_t hidden_dim = hidden_states.dim(2);

    pooled.resize({batch_size, hidden_dim});
    pooled.zero();

    for (dim_t b = 0; b < batch_size; ++b) {
        dim_t seq_len = lengths.at<int32_t>(b);
        for (dim_t s = 0; s < seq_len; ++s) {
            // Accumulate: pooled[b] += hidden_states[b, s]
            ops::Add()(pooled.index({b}), hidden_states.index({b, s}), pooled.index({b}));
        }
        // Divide by length
```

```
        float scale = 1.0f / static_cast<float>(seq_len);
        ops::Mul()(pooled.index({b}), scale, pooled.index({b}));
    }
}
```

---

5. Model Factory Changes

File: `src/models/model_factory.cc`

Register embedding model variant:

```cpp
std::unique_ptr<Model> create_model(const std::string& path, ...) {
    auto config = read_config(path);
    std::string model_spec = config["model_spec"];

    // Check for embedding model variant
    std::string model_purpose = config.value("model_purpose", "generation");

    if (model_spec == "TransformerDecoderModelSpec") {
        if (model_purpose == "embedding") {
            // Load without expecting output projection weights
            return std::make_unique<LanguageModel>(path, device, /*for_embedding=*/true);
        } else {
            return std::make_unique<LanguageModel>(path, device, /*for_embedding=*/false);
        }
    }
    // ... other model types
}
```

---

6. Python Binding Changes

File: `python/cpp/generator.cc`

Add embedding extraction method to Generator:

cpp

```cpp
// Add to GeneratorWrapper class
EmbeddingOutput embed_batch(
    const BatchTokens& tokens,
    const std::string& pooling,
    bool normalize,
    size_t max_batch_size) {

    auto results = execute_in_parallel(
        _pool,
        [&pooling, normalize](Generator& generator,
                              const std::vector<std::vector<size_t>>& batch) {
            return generator.get_model().embed_batch(batch, pooling, normalize);
        },
        tokens,
        max_batch_size);

    return aggregate_embedding_results(results);
}

// Register in pybind11
.def("embed_batch", &GeneratorWrapper::embed_batch,
     py::arg("tokens"),
     py::kw_only(),
     py::arg("pooling") = "last_token",
     py::arg("normalize") = true,
     py::arg("max_batch_size") = 0,
     R"pbdoc(
        Extract embeddings from a batch of token sequences.

        Arguments:
            tokens: Batch of token sequences (list of lists).
            pooling: Pooling strategy - "last_token", "mean", or "max".
            normalize: Whether to L2-normalize the embeddings.
            max_batch_size: Maximum batch size for processing.

        Returns:
            EmbeddingOutput with embeddings and optional hidden_states.
     )pbdoc")
```

File: `python/cpp/embedding_result.cc` (NEW FILE)

Create bindings for embedding results:

```cpp
#include "module.h"
#include <pybind11/pybind11.h>

namespace py = pybind11;

namespace ctranslate2 {
namespace python {

void register_embedding_result(py::module& m) {
  py::class_<EmbeddingResult>(m, "EmbeddingResult",
      "Result of embedding extraction from a language model.")
      .def_property_readonly("embeddings",
          [](const EmbeddingResult& r) { return r.embeddings; },
          "Pooled embeddings with shape [batch_size, hidden_dim].")
      .def_property_readonly("hidden_states",
          [](const EmbeddingResult& r) { return r.hidden_states; },
          "Raw hidden states with shape [batch_size, seq_len, hidden_dim].");
}

}}
```

File: `python/cpp/module.cc`

Register the new embedding result type:

```cpp
// Add to PYBIND11_MODULE
register_embedding_result(m);
```

---

7. Python API Wrapper

File: **python/ctranslate2/__init__.py**

Expose embedding functionality:

python

```python
from ctranslate2._ext import Generator, Encoder, EmbeddingResult

# Optional: Create dedicated Embedder class for cleaner API
class Embedder(Generator):
    """Wrapper for using decoder models as embedding extractors."""

    def encode(self, texts, tokenizer, pooling="last_token",
               normalize=True, instruction=None):
        """
        Extract embeddings from texts.

        Args:
            texts: List of input texts
            tokenizer: HuggingFace tokenizer
            pooling: "last_token", "mean", or "max"
            normalize: Whether to L2-normalize embeddings
            instruction: Optional instruction prefix (for Qwen3 Embedding)

        Returns:
            numpy array of embeddings [batch, hidden_dim]
        """
        # Prepend instruction if provided (Qwen3 Embedding style)
        if instruction:
            texts = [f"Instruct: {instruction}\nQuery: {t}" for t in texts]

        # Tokenize
        tokens = [tokenizer.encode(t) for t in texts]
        token_strs = [tokenizer.convert_ids_to_tokens(t) for t in tokens]

        # Extract embeddings
        result = self.embed_batch(token_strs, pooling=pooling, normalize=normalize)

        return np.array(result.embeddings)
```

---

8. Attention Layer Changes (For QK-Norm Support)

File: `include/ctranslate2/layers/attention.h`

Add QK-norm members for Qwen3:

```cpp
class MultiHeadAttention : public Layer {
private:
    // Existing members...
    const Dense _linear_q;
    const Dense _linear_kv;
    const Dense _linear_output;

    // NEW: QK normalization for Qwen3
    std::unique_ptr<LayerNorm> _q_norm;
    std::unique_ptr<LayerNorm> _k_norm;

public:
    MultiHeadAttention(const Model& model, const std::string& scope,
                       dim_t num_heads, bool self_attention,
                       bool qk_norm = false);  // NEW PARAMETER
};
```

File: `src/layers/attention.cc`

Implement QK-norm in attention forward:

cpp

```cpp
void MultiHeadAttention::operator()(const StorageView& queries, ...) {
    // Project Q, K, V
    StorageView q, k, v;
    _linear_q(queries, q);
    // ... K, V projection

    // NEW: Apply QK normalization (Qwen3)
    if (_q_norm) {
        (*_q_norm)(q, q);
    }
    if (_k_norm) {
        (*_k_norm)(k, k);
    }

    // Continue with scaled dot-product attention...
}
```

Complete file list summary

Category	File Path	Change Type
Python Converter	python/ctranslate2/converters/transformers.py	Add Qwen3EmbeddingLoader
Model Spec	python/ctranslate2/specs/transformer_spec.py	Add skip_output_projection, pooling_method
Model Spec	python/ctranslate2/specs/attention_spec.py	Add qk_norm support
C++ Decoder Header	include/ctranslate2/layers/decoder.h	Add hidden_states parameter
C++ Decoder Impl	src/layers/decoder.cc	Return hidden states before projection
C++ Attention Header	include/ctranslate2/layers/attention.h	Add _q_norm, _k_norm members
C++ Attention Impl	src/layers/attention.cc	Apply QK-norm in forward
C++ Model Header	include/ctranslate2/models/language_model.h	Add embed_batch() method
C++ Model Impl	src/models/language_model.cc	Implement embedding + pooling
C++ Factory	src/models/model_factory.cc	Handle embedding model type
Python Bindings	python/cpp/generator.cc	Add embed_batch binding
Python Bindings	python/cpp/embedding_result.cc	NEW: EmbeddingResult class
Python Bindings	python/cpp/module.cc	Register embedding types
Python API	python/ctranslate2/__init__.py	Expose Embedder class
Build System	CMakeLists.txt	Add new source files

Key implementation decisions

Why extend Generator rather than create DecoderEmbedder? Reusing the Generator infrastructure minimizes code duplication. The decoder forward pass is identical—only the output handling differs. Adding `embed_batch()` to Generator allows the same model to be used for both generation and embedding extraction.

Why last_token pooling for Qwen3? Qwen3 Embedding models are trained with last-token pooling, where the final token's hidden state captures the full sequence representation. Mean pooling is provided as an alternative for compatibility with other embedding approaches.

Why skip lm_head during conversion? Embedding models don't need vocabulary projection weights, saving disk space and memory. The converter explicitly skips `spec.projection` assignment, leaving it as an optional/unset weight.

Instruction-aware prompting is handled at the Python API level by prepending the instruction template. This keeps the C++ runtime simple while supporting Qwen3 Embedding's instruction-based approach.

Testing the implementation

After implementing these changes, verify with:

python

```python
import ctranslate2

# Convert Qwen3 embedding model
# ct2-transformers-converter --model Alibaba-NLP/gte-Qwen2-7B-instruct \
#     --output_dir qwen3-embedding --quantization int8

# Load and extract embeddings
embedder = ctranslate2.Generator("qwen3-embedding", device="cuda")

tokens = [["▁Hello", "▁world"], ["▁Test", "▁sentence"]]
result = embedder.embed_batch(tokens, pooling="last_token", normalize=True)

print(result.embeddings.shape)  # [2, hidden_dim]
print(result.hidden_states.shape)  # [2, seq_len, hidden_dim]
```

This implementation provides a clean delta from the existing Qwen3Loader for generation, adding the specific infrastructure needed for embedding extraction while maintaining compatibility with CTranslate2's architecture.