```
        self.gguf_writer.add_rope_scaling_orig_ctx_len(orig_max_pos_embds)
        self.gguf_writer.add_embedding_length(n_embd)
        self.gguf_writer.add_feed_forward_length(self.find_hparam(["intermediate_size"]))
        self.gguf_writer.add_block_count(block_count)
        self.gguf_writer.add_head_count(n_head)
        self.gguf_writer.add_head_count_kv(n_head_kv)
        self.gguf_writer.add_layer_norm_rms_eps(rms_eps)
        self.gguf_writer.add_rope_dimension_count(rope_dims)
        self.gguf_writer.add_rope_freq_base(self.find_hparam(["rope_theta"]))
        self.gguf_writer.add_file_type(self.ftype)
        sliding_window = self.hparams.get("sliding_window")
        # use zero value of sliding_window to distinguish Phi-4 from other PHI3 models
        if sliding_window is None:
            sliding_window = 0
        self.gguf_writer.add_sliding_window(sliding_window)

    def generate_extra_tensors(self) -> Iterable[tuple[str, Tensor]]:
        n_embd = self.find_hparam(["hidden_size", "n_embd"])
        n_head = self.find_hparam(["num_attention_heads", "n_head"])
        max_pos_embds = self.find_hparam(["n_positions", "max_position_embeddings"])
        orig_max_pos_embds = self.find_hparam(["original_max_position_embeddings"])
        rot_pct = self.hparams.get("partial_rotary_factor", 1.0)
        rope_dims = int(rot_pct * n_embd) // n_head

        # write rope scaling for long context (128k) model
        rope_scaling = self.find_hparam(['rope_scaling'], True)
        if rope_scaling is None:
            return

        scale = max_pos_embds / orig_max_pos_embds

        rope_scaling_type = rope_scaling.get('type', '').lower()
        if len(rope_scaling_type) == 0:
            raise KeyError('Missing the required key rope_scaling.type')

        if rope_scaling_type == 'su' or rope_scaling_type == 'longrope':
            attn_factor = math.sqrt(1 + math.log(scale) / math.log(orig_max_pos_embds)) if scale > 1.0 else 1.0
        elif rope_scaling_type == 'yarn':
            attn_factor = 0.1 * math.log(scale) + 1.0 if scale > 1.0 else 1.0
        else:
            raise NotImplementedError(f'The rope scaling type {rope_scaling_type} is not supported yet')

        self.gguf_writer.add_rope_scaling_attn_factors(attn_factor)

        long_factors = rope_scaling.get('long_factor', None)
        short_factors = rope_scaling.get('short_factor', None)

        if long_factors is None or short_factors is None:
            raise KeyError('Missing the required key rope_scaling.long_factor or rope_scaling_short_factor')

        if len(long_factors) != len(short_factors) or len(long_factors) != rope_dims / 2:
            raise ValueError(f'The length of rope long and short factors must be {rope_dims / 2}. long_factors
= {len(long_factors)}, short_factors = {len(short_factors)}.')
```

```
            yield (self.format_tensor_name(gguf.MODEL_TENSOR.ROPE_FACTORS_LONG), torch.tensor(long_factors, dtype=torch.float32))
            yield (self.format_tensor_name(gguf.MODEL_TENSOR.ROPE_FACTORS_SHORT), torch.tensor(short_factors, dtype=torch.float32))


@Model.register("PhiMoEForCausalLM")
class PhiMoeModel(Phi3MiniModel):
    model_arch = gguf.MODEL_ARCH.PHIMOE

    _experts: list[dict[str, Tensor]] | None = None

    def set_gguf_parameters(self):
        super().set_gguf_parameters()
        self.gguf_writer.add_expert_used_count(self.hparams["num_experts_per_tok"])
        self.gguf_writer.add_expert_count(self.hparams["num_local_experts"])

    def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str, Tensor]]:
        # process the experts separately
        if name.find("block_sparse_moe.experts") != -1:
            n_experts = self.hparams["num_local_experts"]
            assert bid is not None

            if self._experts is None:
                self._experts = [{} for _ in range(self.block_count)]

            self._experts[bid][name] = data_torch

            if len(self._experts[bid]) >= n_experts * 3:
                tensors: list[tuple[str, Tensor]] = []

                # merge the experts into a single 3d tensor
                for w_name in ["w1", "w2", "w3"]:
                    datas: list[Tensor] = []

                    for xid in range(n_experts):
                        ename = f"model.layers.{bid}.block_sparse_moe.experts.{xid}.{w_name}.weight"
                        datas.append(self._experts[bid][ename])
                        del self._experts[bid][ename]

                    data_torch = torch.stack(datas, dim=0)

                    merged_name = f"model.layers.{bid}.block_sparse_moe.experts.{w_name}.weight"

                    new_name = self.map_tensor_name(merged_name)

                    tensors.append((new_name, data_torch))
                return tensors
            else:
                return []

        return [(self.map_tensor_name(name), data_torch)]

    def prepare_tensors(self):
```

```
        super().prepare_tensors()

        if self._experts is not None:
            # flatten `list[dict[str, Tensor]]` into `list[str]`
            experts = [k for d in self._experts for k in d.keys()]
            if len(experts) > 0:
                raise ValueError(f"Unprocessed experts: {experts}")


@Model.register("PlamoForCausalLM")
class PlamoModel(Model):
    model_arch = gguf.MODEL_ARCH.PLAMO

    def set_vocab(self):
        self._set_vocab_sentencepiece()

    def set_gguf_parameters(self):
        hparams = self.hparams
        block_count = hparams["num_hidden_layers"]

        self.gguf_writer.add_context_length(4096)  # not in config.json
        self.gguf_writer.add_embedding_length(hparams["hidden_size"])
        self.gguf_writer.add_feed_forward_length(hparams["intermediate_size"])
        self.gguf_writer.add_block_count(block_count)
        self.gguf_writer.add_head_count(hparams["num_attention_heads"])
        self.gguf_writer.add_head_count_kv(5)  # hparams["num_key_value_heads"]) is wrong
        self.gguf_writer.add_layer_norm_rms_eps(hparams["rms_norm_eps"])
        self.gguf_writer.add_file_type(self.ftype)

    def shuffle_attn_q_weight(self, data_torch):
        assert data_torch.size() == (5120, 5120)
        data_torch = data_torch.reshape(8, 5, 128, 5120)
        data_torch = torch.permute(data_torch, (1, 0, 2, 3))
        data_torch = torch.reshape(data_torch, (5120, 5120))
        return data_torch

    def shuffle_attn_output_weight(self, data_torch):
        assert data_torch.size() == (5120, 5120)
        data_torch = data_torch.reshape(5120, 8, 5, 128)
        data_torch = torch.permute(data_torch, (0, 2, 1, 3))
        data_torch = torch.reshape(data_torch, (5120, 5120))
        return data_torch

    def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str, Tensor]]:
        del bid  # unused

        new_name = self.map_tensor_name(name)

        # shuffle for broadcasting of gqa in ggml_mul_mat
        if new_name.endswith("attn_q.weight"):
            data_torch = self.shuffle_attn_q_weight(data_torch)
        elif new_name.endswith("attn_output.weight"):
            data_torch = self.shuffle_attn_output_weight(data_torch)
```

```
        return [(new_name, data_torch)]


@Model.register("CodeShellForCausalLM")
class CodeShellModel(Model):
    model_arch = gguf.MODEL_ARCH.CODESHELL

    def set_gguf_parameters(self):
        block_count = self.hparams["n_layer"]

        self.gguf_writer.add_context_length(self.hparams["n_positions"])
        self.gguf_writer.add_embedding_length(self.hparams["n_embd"])
        self.gguf_writer.add_feed_forward_length(4 * self.hparams["n_embd"])
        self.gguf_writer.add_block_count(block_count)
        self.gguf_writer.add_head_count(self.hparams["n_head"])
        self.gguf_writer.add_head_count_kv(self.hparams["num_query_groups"])
        self.gguf_writer.add_layer_norm_eps(self.hparams["layer_norm_epsilon"])
        self.gguf_writer.add_file_type(self.ftype)
        self.gguf_writer.add_rope_freq_base(10000.0)
        self.gguf_writer.add_rope_scaling_type(gguf.RopeScalingType.LINEAR)
        self.gguf_writer.add_rope_scaling_factor(1.0)

    _has_tok_embd = False

    def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str, Tensor]]:
        del bid  # unused

        output_name = self.format_tensor_name(gguf.MODEL_TENSOR.OUTPUT)
        tok_embd_name = self.format_tensor_name(gguf.MODEL_TENSOR.TOKEN_EMBD)

        new_name = self.map_tensor_name(name)

        # assuming token_embd.weight is seen before output.weight
        if not self._has_tok_embd and new_name == self.format_tensor_name(gguf.MODEL_TENSOR.OUTPUT):
            # even though the tensor file(s) does not contain the word embeddings they are still in the weight
map
            if self.tensor_names and "transformer.wte.weight" in self.tensor_names:
                logger.debug(f"{tok_embd_name} not found before {output_name}, assuming they are tied")
                self.tensor_names.remove("transformer.wte.weight")
        elif new_name == tok_embd_name:
            self._has_tok_embd = True

        return [(new_name, data_torch)]


@Model.register("InternLM2ForCausalLM")
class InternLM2Model(Model):
    model_arch = gguf.MODEL_ARCH.INTERNLM2

    def set_vocab(self):
        # (TODO): Is there a better way?
        # Copy from _set_vocab_sentencepiece, The only difference is that we will treat the character
        # \x00 specially and convert it into an emoji character to prevent it from being mistakenly
        # recognized as an empty string in C++.
```

```
from sentencepiece import SentencePieceProcessor
from sentencepiece import sentencepiece_model_pb2 as model

tokenizer_path = self.dir_model / 'tokenizer.model'

tokens: list[bytes] = []
scores: list[float] = []
toktypes: list[int] = []

if not tokenizer_path.is_file():
    logger.error(f'Error: Missing {tokenizer_path}')
    sys.exit(1)

sentencepiece_model = model.ModelProto()  # pyright: ignore[reportAttributeAccessIssue]
sentencepiece_model.ParseFromString(open(tokenizer_path, "rb").read())
add_prefix = sentencepiece_model.normalizer_spec.add_dummy_prefix

tokenizer = SentencePieceProcessor()
tokenizer.LoadFromFile(str(tokenizer_path))

vocab_size = self.hparams.get('vocab_size', tokenizer.vocab_size())

for token_id in range(vocab_size):
    piece = tokenizer.IdToPiece(token_id)
    text = piece.encode("utf-8")
    score = tokenizer.GetScore(token_id)
    if text == b"\x00":
        # (TODO): fixme
        # Hack here and replace the \x00 characters.
        logger.warning(f"InternLM2 convert token '{text}' to '?'!")
        text = "?".encode("utf-8")

    toktype = SentencePieceTokenTypes.NORMAL
    if tokenizer.IsUnknown(token_id):
        toktype = SentencePieceTokenTypes.UNKNOWN
    elif tokenizer.IsControl(token_id):
        toktype = SentencePieceTokenTypes.CONTROL
    elif tokenizer.IsUnused(token_id):
        toktype = SentencePieceTokenTypes.UNUSED
    elif tokenizer.IsByte(token_id):
        toktype = SentencePieceTokenTypes.BYTE
    # take care of ununsed raw token
    if piece.startswith('[UNUSED'):
        toktype = SentencePieceTokenTypes.UNUSED

    tokens.append(text)
    scores.append(score)
    toktypes.append(toktype)

added_tokens_file = self.dir_model / 'added_tokens.json'
if added_tokens_file.is_file():
    with open(added_tokens_file, "r", encoding="utf-8") as f:
        added_tokens_json = json.load(f)
```

```
                for key in added_tokens_json:
                    tokens.append(key.encode("utf-8"))
                    scores.append(-1000.0)
                    toktypes.append(SentencePieceTokenTypes.USER_DEFINED)

        chat_eos_token = '<|im_end|>'
        chat_eos_token_id = None

        tokenizer_config_file = self.dir_model / 'tokenizer_config.json'
        if tokenizer_config_file.is_file():
            with open(tokenizer_config_file, "r", encoding="utf-8") as f:
                tokenizer_config_json = json.load(f)
                added_tokens_decoder = tokenizer_config_json.get("added_tokens_decoder", {})
                for token_id, foken_data in added_tokens_decoder.items():
                    token_id = int(token_id)
                    token = foken_data["content"]
                    if token == chat_eos_token:
                        chat_eos_token_id = token_id
                    token = token.encode("utf-8")
                    if toktypes[token_id] != SentencePieceTokenTypes.UNUSED:
                        if tokens[token_id] != token:
                              logger.warning(f'replacing token {token_id}: {tokens[token_id].decode("utf-8")!r}
-> {token.decode("utf-8")!r}')
                    tokens[token_id] = token
                    scores[token_id] = -1000.0
                    toktypes[token_id] = SentencePieceTokenTypes.USER_DEFINED
                    if foken_data.get("special"):
                        toktypes[token_id] = SentencePieceTokenTypes.CONTROL

        tokenizer_file = self.dir_model / 'tokenizer.json'
        if tokenizer_file.is_file():
            with open(tokenizer_file, "r", encoding="utf-8") as f:
                tokenizer_json = json.load(f)
                added_tokens = tokenizer_json.get("added_tokens", [])
                for foken_data in added_tokens:
                    token_id = int(foken_data["id"])
                    token = foken_data["content"]
                    if token == chat_eos_token:
                        chat_eos_token_id = token_id
                    token = token.encode("utf-8")
                    if toktypes[token_id] != SentencePieceTokenTypes.UNUSED:
                        if tokens[token_id] != token:
                              logger.warning(f'replacing token {token_id}: {tokens[token_id].decode("utf-8")!r}
-> {token.decode("utf-8")!r}')
                    tokens[token_id] = token
                    scores[token_id] = -1000.0
                    toktypes[token_id] = SentencePieceTokenTypes.USER_DEFINED
                    if foken_data.get("special"):
                        toktypes[token_id] = SentencePieceTokenTypes.CONTROL

        self.gguf_writer.add_tokenizer_model("llama")
        self.gguf_writer.add_tokenizer_pre("default")
        self.gguf_writer.add_token_list(tokens)
        self.gguf_writer.add_token_scores(scores)
```

```
    self.gguf_writer.add_token_types(toktypes)
    self.gguf_writer.add_add_space_prefix(add_prefix)

    special_vocab = gguf.SpecialVocab(self.dir_model, n_vocab=len(tokens))
    old_eos = special_vocab.special_token_ids["eos"]
    if chat_eos_token_id is not None:
        # For the chat model, we replace the eos with '<|im_end|>'.
        # TODO: this is a hack, should be fixed
        #       https://github.com/ggml-org/llama.cpp/pull/6745#issuecomment-2067687048
        special_vocab.special_token_ids["eos"] = chat_eos_token_id
        logger.warning(f"Replace eos:{old_eos} with a special token:{chat_eos_token_id}"
                       " in chat mode so that the conversation can end normally.")

    special_vocab.add_to_gguf(self.gguf_writer)

def set_gguf_parameters(self):
    self.gguf_writer.add_context_length(self.hparams["max_position_embeddings"])
    self.gguf_writer.add_block_count(self.hparams["num_hidden_layers"])
    self.gguf_writer.add_embedding_length(self.hparams["hidden_size"])
    self.gguf_writer.add_feed_forward_length(self.hparams["intermediate_size"])
    self.gguf_writer.add_rope_freq_base(self.hparams["rope_theta"])
    self.gguf_writer.add_head_count(self.hparams["num_attention_heads"])
    self.gguf_writer.add_layer_norm_rms_eps(self.hparams["rms_norm_eps"])
    self.gguf_writer.add_head_count_kv(self.hparams["num_key_value_heads"])
    self.gguf_writer.add_file_type(self.ftype)
    if self.hparams.get("rope_scaling") is not None and "factor" in self.hparams["rope_scaling"]:
        if self.hparams["rope_scaling"].get("type") == "linear":
            self.gguf_writer.add_rope_scaling_type(gguf.RopeScalingType.LINEAR)
            self.gguf_writer.add_rope_scaling_factor(self.hparams["rope_scaling"]["factor"])

def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str, Tensor]]:
    num_heads = self.hparams["num_attention_heads"]
    num_kv_heads = self.hparams["num_key_value_heads"]
    n_embd = self.hparams["hidden_size"]
    q_per_kv = num_heads // num_kv_heads
    head_dim = n_embd // num_heads
    num_groups = num_heads // q_per_kv

    if bid is not None and f"model.layers.{bid}.attention.wqkv" in name:
        qkv = data_torch

        qkv = qkv.reshape((num_groups, q_per_kv + 2, head_dim, n_embd))
        q, k, v = qkv[:, : q_per_kv], qkv[:, -2], qkv[:, -1]

        # The model weights of q and k equire additional reshape.
        q = LlamaModel.permute(q.reshape((-1, q.shape[-1])), num_heads, num_heads)
        k = LlamaModel.permute(k.reshape((-1, k.shape[-1])), num_heads, num_kv_heads)
        v = v.reshape((-1, v.shape[-1]))

        return [
            (self.format_tensor_name(gguf.MODEL_TENSOR.ATTN_Q, bid), q),
            (self.format_tensor_name(gguf.MODEL_TENSOR.ATTN_K, bid), k),
            (self.format_tensor_name(gguf.MODEL_TENSOR.ATTN_V, bid), v),
        ]
```

```
        else:
            return [(self.map_tensor_name(name), data_torch)]


@Model.register("InternLM3ForCausalLM")
class InternLM3Model(Model):
    model_arch = gguf.MODEL_ARCH.LLAMA

    def set_vocab(self):
        tokens, scores, toktypes = self._create_vocab_sentencepiece()

        self.gguf_writer.add_tokenizer_model("llama")
        self.gguf_writer.add_tokenizer_pre("default")
        self.gguf_writer.add_token_list(tokens)
        self.gguf_writer.add_token_scores(scores)
        self.gguf_writer.add_token_types(toktypes)

        special_vocab = gguf.SpecialVocab(self.dir_model, n_vocab=len(tokens))

        tokenizer_config_file = self.dir_model / 'tokenizer_config.json'
        if tokenizer_config_file.is_file():
            with open(tokenizer_config_file, "r", encoding="utf-8") as f:
                tokenizer_config_json = json.load(f)
                if "add_prefix_space" in tokenizer_config_json:
                    self.gguf_writer.add_add_space_prefix(tokenizer_config_json["add_prefix_space"])

                if "added_tokens_decoder" in tokenizer_config_json:
                    for token_id, token_data in tokenizer_config_json["added_tokens_decoder"].items():
                        if token_data.get("special"):
                            token_id = int(token_id)
                            token = token_data["content"]
                            special_vocab._set_special_token(token, token_id)
                            # update eos token
                            if token == '<|im_end|>' and "eos" in special_vocab.special_token_ids:
                                special_vocab.special_token_ids["eos"] = token_id

        special_vocab.add_to_gguf(self.gguf_writer)

    def set_gguf_parameters(self):
        super().set_gguf_parameters()
        hparams = self.hparams
        self.gguf_writer.add_vocab_size(hparams["vocab_size"])

        if "head_dim" in hparams:
            rope_dim = hparams["head_dim"]
        else:
            rope_dim = hparams["hidden_size"] // hparams["num_attention_heads"]
        self.gguf_writer.add_rope_dimension_count(rope_dim)

        if self.hparams.get("rope_scaling") is not None and "factor" in self.hparams["rope_scaling"]:
                                            if    self.hparams["rope_scaling"].get("type")    ==    "linear"    or
self.hparams["rope_scaling"].get("rope_type") == "linear":
                self.gguf_writer.add_rope_scaling_type(gguf.RopeScalingType.LINEAR)
                self.gguf_writer.add_rope_scaling_factor(self.hparams["rope_scaling"]["factor"])
```

```
    def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str, Tensor]]:
        n_head = self.hparams["num_attention_heads"]
        n_kv_head = self.hparams.get("num_key_value_heads")
        if name.endswith(("q_proj.weight", "q_proj.bias")):
            data_torch = LlamaModel.permute(data_torch, n_head, n_head)
        if name.endswith(("k_proj.weight", "k_proj.bias")):
            data_torch = LlamaModel.permute(data_torch, n_head, n_kv_head)
        return [(self.map_tensor_name(name), data_torch)]


@Model.register("BertModel", "BertForMaskedLM", "CamembertModel")
class BertModel(Model):
    model_arch = gguf.MODEL_ARCH.BERT

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.vocab_size = None

    def set_gguf_parameters(self):
        super().set_gguf_parameters()
        self.gguf_writer.add_causal_attention(False)

        # get pooling path
        pooling_path = None
        module_path = self.dir_model / "modules.json"
        if module_path.is_file():
            with open(module_path, encoding="utf-8") as f:
                modules = json.load(f)
            for mod in modules:
                if mod["type"] == "sentence_transformers.models.Pooling":
                    pooling_path = mod["path"]
                    break

        # get pooling type
        if pooling_path is not None:
            with open(self.dir_model / pooling_path / "config.json", encoding="utf-8") as f:
                pooling = json.load(f)
            if pooling["pooling_mode_mean_tokens"]:
                pooling_type = gguf.PoolingType.MEAN
            elif pooling["pooling_mode_cls_token"]:
                pooling_type = gguf.PoolingType.CLS
            else:
                raise NotImplementedError("Only MEAN and CLS pooling types supported")
            self.gguf_writer.add_pooling_type(pooling_type)

    def set_vocab(self):
        tokens, toktypes, tokpre = self.get_vocab_base()
        self.vocab_size = len(tokens)

        # we need this to validate the size of the token_type embeddings
        # though currently we are passing all zeros to the token_type embeddings
        # "Sequence A" or "Sequence B"
        self.gguf_writer.add_token_type_count(self.hparams.get("type_vocab_size", 1))
```

```
        # convert to phantom space vocab
        def phantom(tok):
            if tok.startswith("[") and tok.endswith("]"):
                return tok
            if tok.startswith("##"):
                return tok[2:]
            return "\u2581" + tok
        tokens = list(map(phantom, tokens))

        # add vocab to gguf
        self.gguf_writer.add_tokenizer_model("bert")
        self.gguf_writer.add_tokenizer_pre(tokpre)
        self.gguf_writer.add_token_list(tokens)
        self.gguf_writer.add_token_types(toktypes)

        # handle special tokens
        special_vocab = gguf.SpecialVocab(self.dir_model, n_vocab=len(tokens))
        special_vocab.add_to_gguf(self.gguf_writer)

    def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str, Tensor]]:
        del bid  # unused

        if name.startswith("bert."):
            name = name[5:]

        if name.endswith(".gamma"):
            name = name[:-6] + ".weight"

        if name.endswith(".beta"):
            name = name[:-5] + ".bias"

        # we are only using BERT for embeddings so we don't need the pooling layer
        if name in ("embeddings.position_ids", "pooler.dense.weight", "pooler.dense.bias"):
            return [] # we don't need these

        if name.startswith("cls.predictions"):
            return []

        if name.startswith("cls.seq_relationship"):
            return []

        return [(self.map_tensor_name(name), data_torch)]


@Model.register("RobertaModel")
class RobertaModel(BertModel):
    model_arch = gguf.MODEL_ARCH.BERT

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        # we need the pad_token_id to know how to chop down position_embd matrix
        if (pad_token_id := self.hparams.get("pad_token_id")) is not None:
```

```
            self._position_offset = 1 + pad_token_id
            if "max_position_embeddings" in self.hparams:
                self.hparams["max_position_embeddings"] -= self._position_offset
        else:
            self._position_offset = None

    def set_vocab(self):
        """Support BPE tokenizers for roberta models"""
        bpe_tok_path = self.dir_model / "tokenizer.json"
        if bpe_tok_path.exists():
            self._set_vocab_gpt2()
            self.gguf_writer.add_add_bos_token(True)
            self.gguf_writer.add_add_eos_token(True)

            # we need this to validate the size of the token_type embeddings
            # though currently we are passing all zeros to the token_type embeddings
            # "Sequence A" or "Sequence B"
            self.gguf_writer.add_token_type_count(self.hparams.get("type_vocab_size", 1))

        else:
            return super().set_vocab()

    def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str, Tensor]]:
        # if name starts with "roberta.", remove the prefix
        # e.g. https://huggingface.co/BAAI/bge-reranker-v2-m3/tree/main
        if name.startswith("roberta."):
            name = name[8:]

        # position embeddings start at pad_token_id + 1, so just chop down the weight tensor
        if name == "embeddings.position_embeddings.weight":
            if self._position_offset is not None:
                data_torch = data_torch[self._position_offset:,:]

        return super().modify_tensors(data_torch, name, bid)


@Model.register("NomicBertModel")
class NomicBertModel(BertModel):
    model_arch = gguf.MODEL_ARCH.NOMIC_BERT

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        # the HF config claims n_ctx=8192, but it uses RoPE scaling
        self.hparams["n_ctx"] = 2048

        # SwigLU activation
        assert self.hparams["activation_function"] == "swiglu"
        # this doesn't do anything in the HF version
        assert self.hparams["causal"] is False
        # no bias tensors
        assert self.hparams["qkv_proj_bias"] is False
        assert self.hparams["mlp_fc1_bias"] is False
        assert self.hparams["mlp_fc2_bias"] is False
```

```
        # norm at end of layer
        assert self.hparams["prenorm"] is False
        # standard RoPE
        assert self.hparams["rotary_emb_fraction"] == 1.0
        assert self.hparams["rotary_emb_interleaved"] is False
        assert self.hparams["rotary_emb_scale_base"] is None

    def set_gguf_parameters(self):
        super().set_gguf_parameters()
        self.gguf_writer.add_rope_freq_base(self.hparams["rotary_emb_base"])


@Model.register("XLMRobertaModel", "XLMRobertaForSequenceClassification")
class XLMRobertaModel(BertModel):
    model_arch = gguf.MODEL_ARCH.BERT

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        # we need the pad_token_id to know how to chop down position_embd matrix
        if (pad_token_id := self.hparams.get("pad_token_id")) is not None:
            self._position_offset = 1 + pad_token_id
            if "max_position_embeddings" in self.hparams:
                self.hparams["max_position_embeddings"] -= self._position_offset
        else:
            self._position_offset = None

    def set_vocab(self):
        # to avoid TypeError: Descriptors cannot be created directly
        # exception when importing sentencepiece_model_pb2
        os.environ["PROTOCOL_BUFFERS_PYTHON_IMPLEMENTATION"] = "python"
        from sentencepiece import SentencePieceProcessor
        from sentencepiece import sentencepiece_model_pb2 as model

        tokenizer_path = self.dir_model / 'sentencepiece.bpe.model'
        if not tokenizer_path.is_file():
            raise FileNotFoundError(f"File not found: {tokenizer_path}")

        sentencepiece_model = model.ModelProto()  # pyright: ignore[reportAttributeAccessIssue]
        sentencepiece_model.ParseFromString(open(tokenizer_path, "rb").read())
        assert sentencepiece_model.trainer_spec.model_type == 1  # UNIGRAM

        add_prefix = sentencepiece_model.normalizer_spec.add_dummy_prefix
        remove_whitespaces = sentencepiece_model.normalizer_spec.remove_extra_whitespaces
        precompiled_charsmap = sentencepiece_model.normalizer_spec.precompiled_charsmap

        tokenizer = SentencePieceProcessor()
        tokenizer.LoadFromFile(str(tokenizer_path))

        vocab_size = self.hparams.get('vocab_size', tokenizer.vocab_size())

        tokens: list[bytes] = [f"[PAD{i}]".encode("utf-8") for i in range(vocab_size)]
        scores: list[float] = [-10000.0] * vocab_size
        toktypes: list[int] = [SentencePieceTokenTypes.UNUSED] * vocab_size
```

```
for token_id in range(tokenizer.vocab_size()):
    piece = tokenizer.IdToPiece(token_id)
    text = piece.encode("utf-8")
    score = tokenizer.GetScore(token_id)

    toktype = SentencePieceTokenTypes.NORMAL
    if tokenizer.IsUnknown(token_id):
        toktype = SentencePieceTokenTypes.UNKNOWN
    elif tokenizer.IsControl(token_id):
        toktype = SentencePieceTokenTypes.CONTROL
    elif tokenizer.IsUnused(token_id):
        toktype = SentencePieceTokenTypes.UNUSED
    elif tokenizer.IsByte(token_id):
        toktype = SentencePieceTokenTypes.BYTE

    tokens[token_id] = text
    scores[token_id] = score
    toktypes[token_id] = toktype

if vocab_size > len(tokens):
    pad_count = vocab_size - len(tokens)
    logger.debug(f"Padding vocab with {pad_count} token(s) - [PAD1] through [PAD{pad_count}]")
    for i in range(1, pad_count + 1):
        tokens.append(bytes(f"[PAD{i}]", encoding="utf-8"))
        scores.append(-1000.0)
        toktypes.append(SentencePieceTokenTypes.UNUSED)

# realign tokens (see HF tokenizer code)
tokens = [b'<s>', b'<pad>', b'</s>', b'<unk>'] + tokens[3:-1]
scores = [0.0, 0.0, 0.0, 0.0] + scores[3:-1]
toktypes = [
    SentencePieceTokenTypes.CONTROL,
    SentencePieceTokenTypes.CONTROL,
    SentencePieceTokenTypes.CONTROL,
    SentencePieceTokenTypes.UNKNOWN,
] + toktypes[3:-1]

self.gguf_writer.add_tokenizer_model("t5")
self.gguf_writer.add_tokenizer_pre("default")
self.gguf_writer.add_token_list(tokens)
self.gguf_writer.add_token_scores(scores)
self.gguf_writer.add_token_types(toktypes)
self.gguf_writer.add_add_space_prefix(add_prefix)
self.gguf_writer.add_token_type_count(self.hparams.get("type_vocab_size", 1))
self.gguf_writer.add_remove_extra_whitespaces(remove_whitespaces)
if precompiled_charsmap:
    self.gguf_writer.add_precompiled_charsmap(precompiled_charsmap)

special_vocab = gguf.SpecialVocab(self.dir_model, n_vocab=len(tokens))
special_vocab.add_to_gguf(self.gguf_writer)

self.gguf_writer.add_add_bos_token(True)
self.gguf_writer.add_add_eos_token(True)
```

```
    def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str, Tensor]]:
        # if name starts with "roberta.", remove the prefix
        # e.g. https://huggingface.co/BAAI/bge-reranker-v2-m3/tree/main
        if name.startswith("roberta."):
            name = name[8:]

        # position embeddings start at pad_token_id + 1, so just chop down the weight tensor
        if name == "embeddings.position_embeddings.weight":
            if self._position_offset is not None:
                data_torch = data_torch[self._position_offset:,:]

        return super().modify_tensors(data_torch, name, bid)


@Model.register("GemmaForCausalLM")
class GemmaModel(Model):
    model_arch = gguf.MODEL_ARCH.GEMMA

    def set_vocab(self):
        self._set_vocab_sentencepiece()

        # TODO: these special tokens should be exported only for the CodeGemma family
        special_vocab = gguf.SpecialVocab(self.dir_model, load_merges=False,
                                          special_token_types = ['prefix', 'suffix', 'middle', 'fsep', 'eot'])
        special_vocab._set_special_token("prefix", 67)
        special_vocab._set_special_token("suffix", 69)
        special_vocab._set_special_token("middle", 68)
        special_vocab._set_special_token("fsep",   70)
        special_vocab._set_special_token("eot",    107)
        special_vocab.chat_template = None  # do not add it twice
        special_vocab.add_to_gguf(self.gguf_writer)

        self.gguf_writer.add_add_space_prefix(False)

    def set_gguf_parameters(self):
        hparams = self.hparams
        block_count = hparams["num_hidden_layers"]

        self.gguf_writer.add_context_length(hparams["max_position_embeddings"])
        self.gguf_writer.add_embedding_length(hparams["hidden_size"])
        self.gguf_writer.add_block_count(block_count)
        self.gguf_writer.add_feed_forward_length(hparams["intermediate_size"])
        self.gguf_writer.add_head_count(hparams["num_attention_heads"])
            self.gguf_writer.add_head_count_kv(self.hparams["num_key_value_heads"] if "num_key_value_heads" in
hparams else hparams["num_attention_heads"])
        self.gguf_writer.add_layer_norm_rms_eps(self.hparams["rms_norm_eps"])
        self.gguf_writer.add_key_length(hparams["head_dim"])
        self.gguf_writer.add_value_length(hparams["head_dim"])
        self.gguf_writer.add_file_type(self.ftype)

    def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str, Tensor]]:
        del bid  # unused
```

```
        # lm_head is not used in llama.cpp, while autoawq will include this tensor in model
        # To prevent errors, skip loading lm_head.weight.
        if name == "lm_head.weight":
            logger.debug(f"Skipping get tensor {name!r} in safetensors so that convert can end normally.")
            return []

                                                                                #           ref:
https://github.com/huggingface/transformers/blob/fc37f38915372c15992b540dfcbbe00a916d4fc6/src/transformers/mode
ls/gemma/modeling_gemma.py#L89
        if name.endswith("norm.weight"):
            data_torch = data_torch + 1

        return [(self.map_tensor_name(name), data_torch)]


@Model.register("Gemma2ForCausalLM")
class Gemma2Model(Model):
    model_arch = gguf.MODEL_ARCH.GEMMA2

    def set_vocab(self):
        self._set_vocab_sentencepiece()

        self.gguf_writer.add_add_space_prefix(False)

    def set_gguf_parameters(self):
        hparams = self.hparams
        block_count = hparams["num_hidden_layers"]

        self.gguf_writer.add_context_length(hparams["max_position_embeddings"])
        self.gguf_writer.add_embedding_length(hparams["hidden_size"])
        self.gguf_writer.add_block_count(block_count)
        self.gguf_writer.add_feed_forward_length(hparams["intermediate_size"])
        self.gguf_writer.add_head_count(hparams["num_attention_heads"])
            self.gguf_writer.add_head_count_kv(self.hparams["num_key_value_heads"] if "num_key_value_heads" in
hparams else hparams["num_attention_heads"])
        self.gguf_writer.add_layer_norm_rms_eps(self.hparams["rms_norm_eps"])
        self.gguf_writer.add_key_length(hparams["head_dim"])
        self.gguf_writer.add_value_length(hparams["head_dim"])
        self.gguf_writer.add_file_type(self.ftype)
        self.gguf_writer.add_attn_logit_softcapping(
            self.hparams["attn_logit_softcapping"]
        )
        self.gguf_writer.add_final_logit_softcapping(
            self.hparams["final_logit_softcapping"]
        )
        self.gguf_writer.add_sliding_window(self.hparams["sliding_window"])

    def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str, Tensor]]:
        del bid  # unused

        # lm_head is not used in llama.cpp, while autoawq will include this tensor in model
        # To prevent errors, skip loading lm_head.weight.
        if name == "lm_head.weight":
            logger.debug(f"Skipping get tensor {name!r} in safetensors so that convert can end normally.")
```

```
            return []

                                                                                    #            ref:
https://github.com/huggingface/transformers/blob/fc37f38915372c15992b540dfcbbe00a916d4fc6/src/transformers/mode
ls/gemma/modeling_gemma.py#L89
        if name.endswith("norm.weight"):
            data_torch = data_torch + 1

        return [(self.map_tensor_name(name), data_torch)]


@Model.register("Gemma3ForCausalLM", "Gemma3ForConditionalGeneration")
class Gemma3Model(Model):
    model_arch = gguf.MODEL_ARCH.GEMMA3
    has_vision: bool = False

    # we need to merge the text_config into the root level of hparams
    def __init__(self, *args, **kwargs):
        hparams = kwargs["hparams"] if "hparams" in kwargs else Model.load_hparams(args[0])
        if "text_config" in hparams:
            hparams = {**hparams, **hparams["text_config"]}
            kwargs["hparams"] = hparams
        super().__init__(*args, **kwargs)
        if "vision_config" in hparams:
            logger.info("Has vision encoder, but it will be ignored")
            self.has_vision = True

    def write(self):
        super().write()
        if self.has_vision:
            logger.info("NOTE: this script only convert the language model to GGUF")
            logger.info("      for the vision model, please use gemma3_convert_encoder_to_gguf.py")

    def set_vocab(self):
        self._set_vocab_sentencepiece()

        self.gguf_writer.add_add_space_prefix(False)

    def set_gguf_parameters(self):
        hparams = self.hparams
        block_count = hparams["num_hidden_layers"]

        # some default values are not specified in the hparams
        self.gguf_writer.add_context_length(hparams.get("max_position_embeddings", 131072))
        self.gguf_writer.add_embedding_length(hparams["hidden_size"])
        self.gguf_writer.add_block_count(block_count)
        self.gguf_writer.add_feed_forward_length(hparams["intermediate_size"])
        self.gguf_writer.add_head_count(hparams.get("num_attention_heads", 8))
        self.gguf_writer.add_layer_norm_rms_eps(self.hparams.get("rms_norm_eps", 1e-6))
        self.gguf_writer.add_key_length(hparams.get("head_dim", 256))
        self.gguf_writer.add_value_length(hparams.get("head_dim", 256))
        self.gguf_writer.add_file_type(self.ftype)
        self.gguf_writer.add_rope_freq_base(hparams.get("rope_theta", 1_000_000.0)) # for global layers
        # both attn_logit_softcapping and final_logit_softcapping are removed in Gemma3
```

```
        assert hparams.get("attn_logit_softcapping") is None
        assert hparams.get("final_logit_softcapping") is None
        self.gguf_writer.add_sliding_window(hparams["sliding_window"])
        self.gguf_writer.add_head_count_kv(hparams.get("num_key_value_heads", 4))
        if hparams.get("rope_scaling") is not None:
            assert hparams["rope_scaling"]["rope_type"] == "linear"
            # important: this rope_scaling is only applied for global layers, and not used by 1B model
            self.gguf_writer.add_rope_scaling_type(gguf.RopeScalingType.LINEAR)
            self.gguf_writer.add_rope_scaling_factor(hparams["rope_scaling"]["factor"])

    def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str, Tensor]]:
        del bid  # unused

        if name.startswith("language_model."):
            name = name.replace("language_model.", "")
        elif name.startswith("multi_modal_projector.") or name.startswith("vision_tower.") \
                or name.startswith("multimodal_projector.") or name.startswith("vision_model."): # this is for
old HF model, should be removed later
            # ignore vision tensors
            return []

        # remove OOV (out-of-vocabulary) rows in token_embd
        if "embed_tokens.weight" in name:
            vocab = self._create_vocab_sentencepiece()
            tokens = vocab[0]
            data_torch = data_torch[:len(tokens)]

        # ref code in Gemma3RMSNorm
        # output = output * (1.0 + self.weight.float())
        if name.endswith("norm.weight"):
            data_torch = data_torch + 1

        return [(self.map_tensor_name(name), data_torch)]


@Model.register("Starcoder2ForCausalLM")
class StarCoder2Model(Model):
    model_arch = gguf.MODEL_ARCH.STARCODER2


@Model.register("Rwkv6ForCausalLM")
class Rwkv6Model(Model):
    model_arch = gguf.MODEL_ARCH.RWKV6

    def set_vocab(self):
        self._set_vocab_rwkv_world()

    def set_gguf_parameters(self):
        block_count = self.hparams["num_hidden_layers"]
        head_size = self.hparams["head_size"]
        hidden_size = self.hparams["hidden_size"]
        layer_norm_eps = self.hparams["layer_norm_epsilon"]
        rescale_every_n_layers = self.hparams["rescale_every"]
        intermediate_size = self.hparams["intermediate_size"] if self.hparams["intermediate_size"] is not None
```

```
else int((hidden_size * 3.5) // 32 * 32)
        time_mix_extra_dim = 64 if hidden_size == 4096 else 32
        time_decay_extra_dim = 128 if hidden_size == 4096 else 64

        # RWKV isn't context limited
        self.gguf_writer.add_context_length(1048576)
        self.gguf_writer.add_embedding_length(hidden_size)
        self.gguf_writer.add_block_count(block_count)
        self.gguf_writer.add_layer_norm_eps(layer_norm_eps)
        self.gguf_writer.add_rescale_every_n_layers(rescale_every_n_layers)
        self.gguf_writer.add_wkv_head_size(head_size)
        self.gguf_writer.add_time_mix_extra_dim(time_mix_extra_dim)
        self.gguf_writer.add_time_decay_extra_dim(time_decay_extra_dim)
        self.gguf_writer.add_feed_forward_length(intermediate_size)
        self.gguf_writer.add_file_type(self.ftype)

        # required by llama.cpp, unused
        self.gguf_writer.add_head_count(0)

    lerp_weights: dict[int, dict[str, Tensor]] = {}

    def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str, Tensor]]:
        new_name = self.map_tensor_name(name)

        if not (new_name.endswith(".weight") or new_name.endswith(".bias")):
            new_name += ".weight"

              if  new_name.endswith("time_mix_w1.weight")  or  new_name.endswith("time_mix_decay_w1.weight")  or
new_name.endswith("time_mix_decay_w2.weight"):
            data_torch = data_torch.transpose(0, 1)

        if new_name.endswith("time_mix_w2.weight"):
            data_torch = data_torch.permute(0, 2, 1)

        if new_name.endswith("time_mix_decay.weight") or "lerp" in new_name:
            data_torch = data_torch.squeeze()

        try:
            rescale_every_n_layers = self.hparams["rescale_every"]
            if rescale_every_n_layers > 0:
                                                        if    new_name.endswith("time_mix_output.weight")    or
new_name.endswith("channel_mix_value.weight"):
                    data_torch = data_torch.div_(2 ** int(bid // rescale_every_n_layers))
        except KeyError:
            pass

        # concat time_mix_lerp weights to reduce some cpu overhead
        # also reduces the number of tensors in the model
        if bid is not None and "time_mix_lerp" in new_name and "time_mix_lerp_x" not in new_name:
            try:
                self.lerp_weights[bid][new_name] = data_torch
            except KeyError:
                self.lerp_weights[bid] = {new_name: data_torch}
             if all(f"blk.{bid}.time_mix_lerp_{i}.weight" in self.lerp_weights[bid].keys() for i in ["w", "k",
```

```
"v", "r", "g"]):
                new_name = f"blk.{bid}.time_mix_lerp_fused.weight"
                 data = torch.stack([self.lerp_weights[bid][f"blk.{bid}.time_mix_lerp_{i}.weight"].unsqueeze(0)
for i in ["w", "k", "v", "r", "g"]], dim=0).unsqueeze(1)
                yield (new_name, data)
            return

        yield (new_name, data_torch)


@Model.register("RWKV6Qwen2ForCausalLM")
class RWKV6Qwen2Model(Rwkv6Model):
    model_arch = gguf.MODEL_ARCH.RWKV6QWEN2

    def set_vocab(self):
        try:
            self._set_vocab_sentencepiece()
        except FileNotFoundError:
            self._set_vocab_gpt2()

    def set_gguf_parameters(self):
        block_count = self.hparams["num_hidden_layers"]
        num_attention_heads = self.hparams["num_attention_heads"]
        num_key_value_heads = self.hparams["num_key_value_heads"]
        hidden_size = self.hparams["hidden_size"]
        head_size = hidden_size // num_attention_heads
        rms_norm_eps = self.hparams["rms_norm_eps"]
        intermediate_size = self.hparams["intermediate_size"]
        time_mix_extra_dim = self.hparams.get("lora_rank_tokenshift", 64 if hidden_size >= 4096 else 32)
        time_decay_extra_dim = self.hparams.get("lora_rank_decay", 128 if hidden_size >= 4096 else 64)

        # RWKV isn't context limited
        self.gguf_writer.add_context_length(1048576)
        self.gguf_writer.add_embedding_length(hidden_size)
        self.gguf_writer.add_block_count(block_count)
        self.gguf_writer.add_wkv_head_size(head_size)
        self.gguf_writer.add_time_mix_extra_dim(time_mix_extra_dim)
        self.gguf_writer.add_time_decay_extra_dim(time_decay_extra_dim)
        self.gguf_writer.add_feed_forward_length(intermediate_size)
        self.gguf_writer.add_file_type(self.ftype)

        # special parameters for time_mixing in RWKV6QWEN2
        self.gguf_writer.add_layer_norm_rms_eps(rms_norm_eps)
        self.gguf_writer.add_token_shift_count(1)
        # RWKV6QWEN2 use grouped key/value like GQA
        self.gguf_writer.add_head_count_kv(num_key_value_heads)

        # required by llama.cpp, unused
        self.gguf_writer.add_head_count(0)

    def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str, Tensor]]:
        for new_name, data in super().modify_tensors(data_torch, name, bid):
            if "time_mix_w1" in new_name or "time_mix_w2" in new_name:
                data = data.view(5, -1, data.shape[-1])
```

```
                # rwkv6qwen2 has a different order of rkvwg instead of the original wkvrg
                # permute them here to avoid code changes
                        data = torch.stack([data[3], data[1], data[2], data[0], data[4]], dim=0).view(-1,
data.shape[-1])
                if "w2" in new_name:
                    data = data.view(5, -1, data.shape[-1])
                yield (new_name, data)
                continue
            yield (new_name, data)


@Model.register("Rwkv7ForCausalLM", "RWKV7ForCausalLM")
class Rwkv7Model(Model):
    model_arch = gguf.MODEL_ARCH.RWKV7

    def set_vocab(self):
        self._set_vocab_rwkv_world()

    def calc_lora_rank(self, hidden_size, exponent, multiplier):
        return max(1, round(hidden_size ** exponent * multiplier / 32)) * 32

    def set_gguf_parameters(self):
        block_count = self.hparams["num_hidden_layers"]
        try:
            head_size = self.hparams["head_size"]
            layer_norm_eps = self.hparams["layer_norm_epsilon"]
        except KeyError:
            head_size = self.hparams["head_dim"]
            layer_norm_eps = self.hparams["norm_eps"]
        hidden_size = self.hparams["hidden_size"]
         intermediate_size = self.hparams["intermediate_size"] if self.hparams["intermediate_size"] is not None
else (hidden_size * 4)

        # ICLR: In-Context-Learning-Rate
        try:
              lora_rank_decay = self.hparams["lora_rank_decay"] if self.hparams["lora_rank_decay"] is not None
else self.calc_lora_rank(hidden_size, 0.5, 1.8)
             lora_rank_iclr = self.hparams["lora_rank_iclr"] if self.hparams["lora_rank_iclr"] is not None else
self.calc_lora_rank(hidden_size, 0.5, 1.8)
                            lora_rank_value_residual_mix = self.hparams["lora_rank_value_residual_mix"] if
self.hparams["lora_rank_value_residual_mix"] is not None else self.calc_lora_rank(hidden_size, 0.5, 1.3)
             lora_rank_gate = self.hparams["lora_rank_gate"] if self.hparams["lora_rank_gate"] is not None else
self.calc_lora_rank(hidden_size, 0.8, 0.6)
        except KeyError:
              lora_rank_decay = self.hparams["decay_low_rank_dim"] if self.hparams["decay_low_rank_dim"] is not
None else self.calc_lora_rank(hidden_size, 0.5, 1.8)
             lora_rank_iclr = self.hparams["a_low_rank_dim"] if self.hparams["a_low_rank_dim"] is not None else
self.calc_lora_rank(hidden_size, 0.5, 1.8)
             lora_rank_value_residual_mix = self.hparams["v_low_rank_dim"] if self.hparams["v_low_rank_dim"] is
not None else self.calc_lora_rank(hidden_size, 0.5, 1.3)
            lora_rank_gate = self.hparams["gate_low_rank_dim"] if self.hparams["gate_low_rank_dim"] is not None
else self.calc_lora_rank(hidden_size, 0.8, 0.6)

        # RWKV isn't context limited
```

```
        self.gguf_writer.add_context_length(1048576)
        self.gguf_writer.add_embedding_length(hidden_size)
        self.gguf_writer.add_block_count(block_count)
        self.gguf_writer.add_layer_norm_eps(layer_norm_eps)
        self.gguf_writer.add_wkv_head_size(head_size)
        self.gguf_writer.add_decay_lora_rank(lora_rank_decay)
        self.gguf_writer.add_iclr_lora_rank(lora_rank_iclr)
        self.gguf_writer.add_value_residual_mix_lora_rank(lora_rank_value_residual_mix)
        self.gguf_writer.add_gate_lora_rank(lora_rank_gate)
        self.gguf_writer.add_feed_forward_length(intermediate_size)
        self.gguf_writer.add_file_type(self.ftype)

        # required by llama.cpp, unused
        self.gguf_writer.add_head_count(0)

    lerp_weights: dict[int, dict[str, Tensor]] = {}
    lora_needs_transpose: bool = True

    def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str, Tensor]]:
        # unify tensor names here to make life easier
        name = name.replace("blocks", "layers").replace("ffn", "feed_forward")
        name = name.replace("self_attn", "attention").replace("attn", "attention")
        name = name.replace("time_mixer.", "")
        # lora layer names in fla-hub's impl
        if "_lora.lora" in name:
            self.lora_needs_transpose = False
        name = name.replace("_lora.lora.0.weight", "1.weight")
        name = name.replace("_lora.lora.2.weight", "2.weight")
        name = name.replace("_lora.lora.2.bias", "0.weight")

        name = name.replace("feed_forward_norm", "ln2")
        name = name.replace("g_norm", "ln_x")

        if "attention.v" in name and "value" not in self.map_tensor_name(name) and bid == 0:
            # some models have dummy v0/v1/v2 on first layer while others don't
            # ignore them all since they are not used
            return

        wkv_has_gate = self.hparams.get("wkv_has_gate", True)
        lerp_list = ["r", "w", "k", "v", "a", "g"] if wkv_has_gate else ["r", "w", "k", "v", "a"]

        if bid is not None and "attention.x_" in name:
            if "attention.x_x" in name:
                # already concatenated
                new_name = f"blk.{bid}.time_mix_lerp_fused.weight"
                data = data_torch.reshape(len(lerp_list), 1, 1, -1)
                yield (new_name, data)
            else:
                try:
                    self.lerp_weights[bid][name] = data_torch
                except KeyError:
                    self.lerp_weights[bid] = {name: data_torch}
                if all(f"model.layers.{bid}.attention.x_{i}" in self.lerp_weights[bid].keys() for i in
lerp_list):
```

```
                    new_name = f"blk.{bid}.time_mix_lerp_fused.weight"
                     data = torch.stack([self.lerp_weights[bid][f"model.layers.{bid}.attention.x_{i}"] for i in
lerp_list], dim=0)
                    yield (new_name, data)
            return
        else:
            data_torch = data_torch.squeeze()
            new_name = self.map_tensor_name(name)

            if not (new_name.endswith(".weight") or new_name.endswith(".bias")):
                new_name += ".weight"

            if self.lora_needs_transpose and any(
                new_name.endswith(t) for t in [
                    "time_mix_w1.weight", "time_mix_w2.weight",
                    "time_mix_a1.weight", "time_mix_a2.weight",
                    "time_mix_v1.weight", "time_mix_v2.weight",
                    "time_mix_g1.weight", "time_mix_g2.weight",
                ]
            ):
                data_torch = data_torch.transpose(0, 1)

            if 'r_k' in new_name:
                data_torch = data_torch.flatten()

            if bid == 0 and "time_mix_a" in new_name:
                # dummy v0/v1/v2 on first layer
                # easist way to make llama happy
                yield (new_name.replace("time_mix_a", "time_mix_v"), data_torch)

            yield (new_name, data_torch)


@Model.register("RwkvHybridForCausalLM")
class ARwkv7Model(Rwkv7Model):
    model_arch = gguf.MODEL_ARCH.ARWKV7

    def set_vocab(self):
        try:
            self._set_vocab_sentencepiece()
        except FileNotFoundError:
            self._set_vocab_gpt2()

    def set_gguf_parameters(self):
        block_count = self.hparams["num_hidden_layers"]
        hidden_size = self.hparams["hidden_size"]
        head_size = self.hparams["head_size"]
        rms_norm_eps = self.hparams["rms_norm_eps"]
        intermediate_size = self.hparams["intermediate_size"]
        wkv_has_gate = self.hparams["wkv_has_gate"]
        assert self.hparams["wkv_version"] == 7

        # ICLR: In-Context-Learning-Rate
        lora_rank_decay = 64
```

```
        lora_rank_iclr = 64
        lora_rank_value_residual_mix = 32
        lora_rank_gate = 128 if wkv_has_gate else 0

        # RWKV isn't context limited
        self.gguf_writer.add_context_length(1048576)
        self.gguf_writer.add_embedding_length(hidden_size)
        self.gguf_writer.add_block_count(block_count)
        self.gguf_writer.add_layer_norm_rms_eps(rms_norm_eps)
        self.gguf_writer.add_wkv_head_size(head_size)
        self.gguf_writer.add_decay_lora_rank(lora_rank_decay)
        self.gguf_writer.add_iclr_lora_rank(lora_rank_iclr)
        self.gguf_writer.add_value_residual_mix_lora_rank(lora_rank_value_residual_mix)
        self.gguf_writer.add_gate_lora_rank(lora_rank_gate)
        self.gguf_writer.add_feed_forward_length(intermediate_size)
        self.gguf_writer.add_file_type(self.ftype)
        self.gguf_writer.add_token_shift_count(1)

        # required by llama.cpp, unused
        self.gguf_writer.add_head_count(0)


@Model.register("MambaForCausalLM", "MambaLMHeadModel", "FalconMambaForCausalLM")
class MambaModel(Model):
    model_arch = gguf.MODEL_ARCH.MAMBA

    def set_vocab(self):
        vocab_size = self.hparams["vocab_size"]
        # Round vocab size to next multiple of 8
        pad_vocab = self.hparams.get("pad_vocab_size_multiple", 8)
        # pad using ceiling division
        # ref: https://stackoverflow.com/a/17511341/22827863
        vocab_size = -(vocab_size // -pad_vocab) * pad_vocab
        self.hparams["vocab_size"] = vocab_size

        if (self.dir_model / "tokenizer.json").is_file():
            self._set_vocab_gpt2()
        elif (self.dir_model / "tokenizer.model").is_file():
            self._set_vocab_sentencepiece()
        else:
            # Use the GPT-NeoX tokenizer when no tokenizer files are present
            self._set_vocab_builtin("gpt-neox", vocab_size)

    def set_gguf_parameters(self):
        d_model = self.find_hparam(["hidden_size",       "d_model"])
        d_conv  = self.find_hparam(["conv_kernel",       "d_conv"],  optional=True) or 4
        d_inner = self.find_hparam(["intermediate_size", "d_inner"], optional=True) or 2 * d_model
        d_state = self.find_hparam(["state_size",        "d_state"], optional=True) or 16
        # ceiling division
        # ref: https://stackoverflow.com/a/17511341/22827863
                                                                                    #            ref:
https://github.com/state-spaces/mamba/blob/ce59daea3a090d011d6476c6e5b97f6d58ddad8b/mamba_ssm/modules/mamba_sim
ple.py#L58
        dt_rank      = self.find_hparam(["time_step_rank",     "dt_rank"],      optional=True) or -(d_model //
```

```
-16)
        rms_norm_eps = self.find_hparam(["layer_norm_epsilon", "rms_norm_eps"], optional=True) or 1e-5
        use_dt_b_c_norm = False
        # For falconmamba we do apply RMS norm on B / DT and C layers
        if self.find_hparam(["model_type"], optional=True) in ("falcon_mamba",):
            use_dt_b_c_norm = True
        # Fail early for models which don't have a block expansion factor of 2
        assert d_inner == 2 * d_model

        self.gguf_writer.add_context_length(2**20) # arbitrary value; for those who use the default
        self.gguf_writer.add_embedding_length(d_model)
        self.gguf_writer.add_feed_forward_length(0) # unused, but seemingly required when loading
        self.gguf_writer.add_head_count(0) # unused, but seemingly required when loading
        self.gguf_writer.add_block_count(self.block_count)
        self.gguf_writer.add_ssm_conv_kernel(d_conv)
        self.gguf_writer.add_ssm_inner_size(d_inner)
        self.gguf_writer.add_ssm_state_size(d_state)
        self.gguf_writer.add_ssm_time_step_rank(dt_rank)
        self.gguf_writer.add_layer_norm_rms_eps(rms_norm_eps)
        self.gguf_writer.add_ssm_dt_b_c_rms(use_dt_b_c_norm) # For classic Mamba we don't apply rms norm on B /
DT layers
        self.gguf_writer.add_file_type(self.ftype)

    _tok_embd = None

    def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str, Tensor]]:
        output_name = self.format_tensor_name(gguf.MODEL_TENSOR.OUTPUT)
        tok_embd_name = self.format_tensor_name(gguf.MODEL_TENSOR.TOKEN_EMBD)

        new_name = self.map_tensor_name(name)

        if name.endswith(".A_log"):
            logger.debug("A_log --> A ==> " + new_name)
            data_torch = -torch.exp(data_torch)

        # [4 1 8192 1] -> [4 8192 1 1]
        if self.match_model_tensor_name(new_name, gguf.MODEL_TENSOR.SSM_CONV1D, bid):
            data_torch = data_torch.squeeze()

        # assuming token_embd.weight is seen before output.weight
        if self._tok_embd is not None and new_name == output_name:
            if torch.equal(self._tok_embd, data_torch):
                logger.debug(f"{output_name} is equivalent to {tok_embd_name}, omitting")
                return []
        elif new_name == tok_embd_name:
            self._tok_embd = data_torch

        return [(new_name, data_torch)]


@Model.register("CohereForCausalLM")
class CommandR2Model(Model):
    model_arch = gguf.MODEL_ARCH.COMMAND_R
```

```
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        # max_position_embeddings = 8192 in config.json but model was actually
        # trained on 128k context length
        # aya-23 models don't have model_max_length specified
                            self.hparams["max_position_embeddings"]    =    self.find_hparam(["model_max_length",
"max_position_embeddings"])

    def set_gguf_parameters(self):
        super().set_gguf_parameters()
        self.gguf_writer.add_logit_scale(self.hparams["logit_scale"])
        self.gguf_writer.add_rope_scaling_type(gguf.RopeScalingType.NONE)


@Model.register("Cohere2ForCausalLM")
class Cohere2Model(Model):
    model_arch = gguf.MODEL_ARCH.COHERE2

    def set_gguf_parameters(self):
        super().set_gguf_parameters()

        self.gguf_writer.add_logit_scale(self.hparams["logit_scale"])
        self.gguf_writer.add_sliding_window(self.hparams["sliding_window"])
        self.gguf_writer.add_vocab_size(self.hparams["vocab_size"])

        rotary_pct = self.hparams["rotary_pct"]
        hidden_size = self.hparams["hidden_size"]
        num_attention_heads = self.hparams["num_attention_heads"]
        self.gguf_writer.add_rope_dimension_count(int(rotary_pct * (hidden_size // num_attention_heads)))
        self.gguf_writer.add_rope_scaling_type(gguf.RopeScalingType.NONE)


@Model.register("OlmoForCausalLM")
@Model.register("OLMoForCausalLM")
class OlmoModel(Model):
    model_arch = gguf.MODEL_ARCH.OLMO

    def set_gguf_parameters(self):
        super().set_gguf_parameters()
        self.gguf_writer.add_layer_norm_eps(1e-5)
        clip_qkv = self.hparams.get("clip_qkv")
        if clip_qkv is not None:
            self.gguf_writer.add_clamp_kqv(clip_qkv)

    # Same as super class, but permuting q_proj, k_proj
    # Copied from: LlamaModel
    def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str, Tensor]]:
        del bid  # unused

        n_head = self.hparams["num_attention_heads"]
        n_kv_head = self.hparams.get("num_key_value_heads")

        if name.endswith("q_proj.weight"):
```

```
            data_torch = LlamaModel.permute(data_torch, n_head, n_head)
        if name.endswith("k_proj.weight"):
            data_torch = LlamaModel.permute(data_torch, n_head, n_kv_head)

        return [(self.map_tensor_name(name), data_torch)]


@Model.register("Olmo2ForCausalLM")
class Olmo2Model(Model):
    model_arch = gguf.MODEL_ARCH.OLMO2


@Model.register("OlmoeForCausalLM")
class OlmoeModel(Model):
    model_arch = gguf.MODEL_ARCH.OLMOE

    def set_gguf_parameters(self):
        super().set_gguf_parameters()
        self.gguf_writer.add_layer_norm_rms_eps(1e-5)
        if (n_experts := self.hparams.get("num_experts")) is not None:
            self.gguf_writer.add_expert_count(n_experts)

    _experts: list[dict[str, Tensor]] | None = None

    # Copied from: Qwen2MoeModel
    def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str, Tensor]]:
        # process the experts separately
        if name.find("experts") != -1:
            n_experts = self.hparams["num_experts"]
            assert bid is not None

            if self._experts is None:
                self._experts = [{} for _ in range(self.block_count)]

            self._experts[bid][name] = data_torch

            if len(self._experts[bid]) >= n_experts * 3:
                tensors: list[tuple[str, Tensor]] = []

                # merge the experts into a single 3d tensor
                for w_name in ["down_proj", "gate_proj", "up_proj"]:
                    datas: list[Tensor] = []

                    for xid in range(n_experts):
                        ename = f"model.layers.{bid}.mlp.experts.{xid}.{w_name}.weight"
                        datas.append(self._experts[bid][ename])
                        del self._experts[bid][ename]

                    data_torch = torch.stack(datas, dim=0)

                    merged_name = f"model.layers.{bid}.mlp.experts.{w_name}.weight"

                    new_name = self.map_tensor_name(merged_name)
```

```
                    tensors.append((new_name, data_torch))
                return tensors
            else:
                return []

        return [(self.map_tensor_name(name), data_torch)]

    # Copied from: Qwen2MoeModel
    def prepare_tensors(self):
        super().prepare_tensors()

        if self._experts is not None:
            # flatten `list[dict[str, Tensor]]` into `list[str]`
            experts = [k for d in self._experts for k in d.keys()]
            if len(experts) > 0:
                raise ValueError(f"Unprocessed experts: {experts}")


@Model.register("JinaBertModel", "JinaBertForMaskedLM")
class JinaBertV2Model(BertModel):
    model_arch = gguf.MODEL_ARCH.JINA_BERT_V2

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.intermediate_size = self.hparams["intermediate_size"]

    def get_tensors(self):
        for name, data in super().get_tensors():
            if 'gated_layer' in name:
                d1 = data[:self.intermediate_size, :]
                name1 = name.replace('gated_layers', 'gated_layers_w')
                name1 = name1.replace('up_gated_layer', 'gated_layers_v')
                d2 = data[self.intermediate_size:, :]
                name2 = name.replace('gated_layers', 'gated_layers_v')
                name2 = name2.replace('up_gated_layer', 'gated_layers_w')
                yield name1, d1
                yield name2, d2
                continue

            yield name, data

    def set_vocab(self):
        tokenizer_class = 'BertTokenizer'
        with open(self.dir_model / "tokenizer_config.json", "r", encoding="utf-8") as f:
            tokenizer_class = json.load(f)['tokenizer_class']

        if tokenizer_class == 'BertTokenizer':
            super().set_vocab()
        elif tokenizer_class == 'RobertaTokenizer':
            self._set_vocab_gpt2()
            self.gguf_writer.add_token_type_count(2)
        else:
            raise NotImplementedError(f'Tokenizer {tokenizer_class} is not supported for JinaBertModel')
        self.gguf_writer.add_add_bos_token(True)
```

```
        self.gguf_writer.add_add_eos_token(True)

    def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str, Tensor]]:
        # if name starts with "bert.", remove the prefix
        # e.g. https://huggingface.co/jinaai/jina-reranker-v1-tiny-en
        if name.startswith("bert."):
            name = name[5:]

        return super().modify_tensors(data_torch, name, bid)


@Model.register("OpenELMForCausalLM")
class OpenELMModel(Model):
    model_arch = gguf.MODEL_ARCH.OPENELM

    @staticmethod
    def _make_divisible(v: float | int, divisor: int) -> int:
                                                                                    #            ref:
https://huggingface.co/apple/OpenELM-270M-Instruct/blob/eb111ff2e6724348e5b905984063d4064d4bc579/configuration_
openelm.py#L34-L38
        new_v = max(divisor, int(v + divisor / 2) // divisor * divisor)
        # Make sure that round down does not go down by more than 10%.
        if new_v < 0.9 * v:
            new_v += divisor
        return new_v

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        ffn_multipliers: list[float] = self.hparams["ffn_multipliers"]
        ffn_dim_divisor: int = self.hparams["ffn_dim_divisor"]
        self._n_embd: int = self.hparams["model_dim"]
        self._num_kv_heads: list[int] = self.hparams["num_kv_heads"]
        self._num_query_heads: list[int] = self.hparams["num_query_heads"]
        self._ffn_dims: list[int] = [
            OpenELMModel._make_divisible(multiplier * self._n_embd, ffn_dim_divisor)
            for multiplier in ffn_multipliers
        ]
        assert isinstance(self._num_kv_heads, list) and isinstance(self._num_kv_heads[0], int)
        assert isinstance(self._num_query_heads, list) and isinstance(self._num_query_heads[0], int)

    # Uses the tokenizer from meta-llama/Llama-2-7b-hf
    def set_vocab(self):
        try:
            self._set_vocab_sentencepiece()
        except FileNotFoundError:
            self._set_vocab_builtin("llama-spm", self.hparams["vocab_size"])

    def set_gguf_parameters(self):
        n_embd = self._n_embd
        head_dim = self.hparams["head_dim"]
        rot_pct = 1.0
        assert self.block_count == len(self._num_kv_heads)
        assert self.block_count == len(self._num_query_heads)
```

```
        assert self.block_count == len(self._ffn_dims)

        self.gguf_writer.add_block_count(self.block_count)
        self.gguf_writer.add_context_length(self.hparams["max_context_length"])
        self.gguf_writer.add_embedding_length(n_embd)
        self.gguf_writer.add_feed_forward_length(self._ffn_dims)
        self.gguf_writer.add_head_count(self._num_query_heads)
        self.gguf_writer.add_head_count_kv(self._num_kv_heads)
        self.gguf_writer.add_rope_freq_base(self.hparams["rope_freq_constant"])
        # https://huggingface.co/apple/OpenELM-270M-Instruct/blob/c401df2/modeling_openelm.py#L30
        self.gguf_writer.add_layer_norm_rms_eps(1e-6)
        self.gguf_writer.add_rope_dimension_count(int(rot_pct * head_dim))
        self.gguf_writer.add_key_length(head_dim)
        self.gguf_writer.add_value_length(head_dim)
        self.gguf_writer.add_file_type(self.ftype)

    def find_hparam(self, keys: Iterable[str], optional: bool = False) -> Any:
        if "n_layers" in keys:
            return self.hparams["num_transformer_layers"]

        return super().find_hparam(keys, optional)

    def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str, Tensor]]:

        # split ff
        if bid is not None and name == f"transformer.layers.{bid}.ffn.proj_1.weight":
            ff_dim = self._ffn_dims[bid]
            yield (self.format_tensor_name(gguf.MODEL_TENSOR.FFN_GATE, bid), data_torch[:ff_dim])
            yield (self.format_tensor_name(gguf.MODEL_TENSOR.FFN_UP, bid), data_torch[ff_dim:])
            return

        yield (self.map_tensor_name(name), data_torch)


@Model.register("ArcticForCausalLM")
class ArcticModel(Model):
    model_arch = gguf.MODEL_ARCH.ARCTIC

    def set_vocab(self):
        # The reason for using a custom implementation here is that the
        # snowflake-arctic-instruct model redefined tokens 31998 and 31999 from
        # tokenizer.model and used them as BOS and EOS instead of adding new tokens.
        from sentencepiece import SentencePieceProcessor

        tokenizer_path = self.dir_model / 'tokenizer.model'

        if not tokenizer_path.is_file():
            logger.error(f'Error: Missing {tokenizer_path}')
            sys.exit(1)

        # Read the whole vocabulary from the tokenizer.model file
        tokenizer = SentencePieceProcessor()
        tokenizer.LoadFromFile(str(tokenizer_path))
```

```
vocab_size = self.hparams.get('vocab_size', tokenizer.vocab_size())

tokens: list[bytes] = [f"[PAD{i}]".encode("utf-8") for i in range(vocab_size)]
scores: list[float] = [-10000.0] * vocab_size
toktypes: list[int] = [SentencePieceTokenTypes.UNUSED] * vocab_size

for token_id in range(tokenizer.vocab_size()):

    piece = tokenizer.IdToPiece(token_id)
    text = piece.encode("utf-8")
    score = tokenizer.GetScore(token_id)

    toktype = SentencePieceTokenTypes.NORMAL
    if tokenizer.IsUnknown(token_id):
        toktype = SentencePieceTokenTypes.UNKNOWN
    elif tokenizer.IsControl(token_id):
        toktype = SentencePieceTokenTypes.CONTROL
    elif tokenizer.IsUnused(token_id):
        toktype = SentencePieceTokenTypes.UNUSED
    elif tokenizer.IsByte(token_id):
        toktype = SentencePieceTokenTypes.BYTE

    tokens[token_id] = text
    scores[token_id] = score
    toktypes[token_id] = toktype

# Use the added_tokens_decoder field from tokeniser_config.json as the source
# of information about added/redefined tokens and modify them accordingly.
tokenizer_config_file = self.dir_model / 'tokenizer_config.json'
if tokenizer_config_file.is_file():
    with open(tokenizer_config_file, "r", encoding="utf-8") as f:
        tokenizer_config_json = json.load(f)

        if "added_tokens_decoder" in tokenizer_config_json:
            added_tokens_decoder = tokenizer_config_json["added_tokens_decoder"]
            for token_id, token_json in added_tokens_decoder.items():
                token_id = int(token_id)
                if token_id >= vocab_size:
                    logger.debug(f'ignore token {token_id}: id is out of range, max={vocab_size - 1}')
                    continue

                token_content = token_json["content"]
                token_type = SentencePieceTokenTypes.USER_DEFINED
                token_score = -10000.0

                # Map unk_token to UNKNOWN, other special tokens to CONTROL
                # Set the score to 0.0 as in the original tokenizer.model
                if ("special" in token_json) and token_json["special"]:
                    if token_content == tokenizer_config_json["unk_token"]:
                        token_type = SentencePieceTokenTypes.UNKNOWN
                    else:
                        token_type = SentencePieceTokenTypes.CONTROL
                    token_score = 0.0
```

```
                        logger.info(f"Setting added token {token_id} to '{token_content}' (type: {token_type},
score: {token_score:.2f})")
                    tokens[token_id] = token_content.encode("utf-8")
                    toktypes[token_id] = token_type
                    scores[token_id] = token_score

        self.gguf_writer.add_tokenizer_model("llama")
        self.gguf_writer.add_tokenizer_pre("default")
        self.gguf_writer.add_token_list(tokens)
        self.gguf_writer.add_token_scores(scores)
        self.gguf_writer.add_token_types(toktypes)

        special_vocab = gguf.SpecialVocab(self.dir_model, n_vocab=len(tokens))
        special_vocab.add_to_gguf(self.gguf_writer)

    def set_gguf_parameters(self):
        super().set_gguf_parameters()
        hparams = self.hparams
        self.gguf_writer.add_vocab_size(hparams["vocab_size"])
        self.gguf_writer.add_rope_dimension_count(hparams["hidden_size"] // hparams["num_attention_heads"])

    _experts: list[dict[str, Tensor]] | None = None

    def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str, Tensor]]:
        n_head = self.hparams["num_attention_heads"]
        n_kv_head = self.hparams.get("num_key_value_heads")

        if name.endswith("q_proj.weight"):
            data_torch = LlamaModel.permute(data_torch, n_head, n_head)
        if name.endswith("k_proj.weight"):
            data_torch = LlamaModel.permute(data_torch, n_head, n_kv_head)

        # process the experts separately
        if name.find("block_sparse_moe.experts") != -1:
            n_experts = self.hparams["num_local_experts"]

            assert bid is not None

            if self._experts is None:
                self._experts = [{} for _ in range(self.block_count)]

            self._experts[bid][name] = data_torch

            if len(self._experts[bid]) >= n_experts * 3:
                tensors: list[tuple[str, Tensor]] = []

                # merge the experts into a single 3d tensor
                for wid in ["w1", "w2", "w3"]:
                    datas: list[Tensor] = []

                    for xid in range(n_experts):
                        ename = f"model.layers.{bid}.block_sparse_moe.experts.{xid}.{wid}.weight"
                        datas.append(self._experts[bid][ename])
                        del self._experts[bid][ename]
```

```
                    data_torch = torch.stack(datas, dim=0)

                    merged_name = f"layers.{bid}.feed_forward.experts.{wid}.weight"

                    new_name = self.map_tensor_name(merged_name)

                    tensors.append((new_name, data_torch))
                return tensors
            else:
                return []

        return [(self.map_tensor_name(name), data_torch)]

    def prepare_tensors(self):
        super().prepare_tensors()

        if self._experts is not None:
            # flatten `list[dict[str, Tensor]]` into `list[str]`
            experts = [k for d in self._experts for k in d.keys()]
            if len(experts) > 0:
                raise ValueError(f"Unprocessed experts: {experts}")


@Model.register("DeepseekForCausalLM")
class DeepseekModel(Model):
    model_arch = gguf.MODEL_ARCH.DEEPSEEK

    def set_vocab(self):
        try:
            self._set_vocab_sentencepiece()
        except FileNotFoundError:
            self._set_vocab_gpt2()

    def set_gguf_parameters(self):
        super().set_gguf_parameters()
        hparams = self.hparams
        if "head_dim" in hparams:
            rope_dim = hparams["head_dim"]
        else:
            rope_dim = hparams["hidden_size"] // hparams["num_attention_heads"]

        self.gguf_writer.add_rope_dimension_count(rope_dim)
        self.gguf_writer.add_rope_scaling_type(gguf.RopeScalingType.NONE)
        self.gguf_writer.add_leading_dense_block_count(hparams["first_k_dense_replace"])
        self.gguf_writer.add_vocab_size(hparams["vocab_size"])
        self.gguf_writer.add_expert_feed_forward_length(hparams["moe_intermediate_size"])
        self.gguf_writer.add_expert_weights_scale(1.0)
        self.gguf_writer.add_expert_count(hparams["n_routed_experts"])
        self.gguf_writer.add_expert_shared_count(hparams["n_shared_experts"])

    _experts: list[dict[str, Tensor]] | None = None

    @staticmethod
```

```
def permute(weights: Tensor, n_head: int, n_head_kv: int | None):
    if n_head_kv is not None and n_head != n_head_kv:
        n_head = n_head_kv
    return (weights.reshape(n_head, 2, weights.shape[0] // n_head // 2, *weights.shape[1:])
            .swapaxes(1, 2)
            .reshape(weights.shape))

def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str, Tensor]]:
    n_head = self.hparams["num_attention_heads"]
    n_kv_head = self.hparams.get("num_key_value_heads")

    if name.endswith(("q_proj.weight", "q_proj.bias")):
        data_torch = DeepseekModel.permute(data_torch, n_head, n_head)
    if name.endswith(("k_proj.weight", "k_proj.bias")):
        data_torch = DeepseekModel.permute(data_torch, n_head, n_kv_head)

    # process the experts separately
    if name.find("mlp.experts") != -1:
        n_experts = self.hparams["n_routed_experts"]
        assert bid is not None

        if self._experts is None:
            self._experts = [{} for _ in range(self.block_count)]

        self._experts[bid][name] = data_torch

        if len(self._experts[bid]) >= n_experts * 3:
            tensors: list[tuple[str, Tensor]] = []

            # merge the experts into a single 3d tensor
            for w_name in ["down_proj", "gate_proj", "up_proj"]:
                datas: list[Tensor] = []

                for xid in range(n_experts):
                    ename = f"model.layers.{bid}.mlp.experts.{xid}.{w_name}.weight"
                    datas.append(self._experts[bid][ename])
                    del self._experts[bid][ename]

                data_torch = torch.stack(datas, dim=0)

                merged_name = f"model.layers.{bid}.mlp.experts.{w_name}.weight"

                new_name = self.map_tensor_name(merged_name)

                tensors.append((new_name, data_torch))
            return tensors
        else:
            return []

    return [(self.map_tensor_name(name), data_torch)]

def prepare_tensors(self):
    super().prepare_tensors()
```

```
        if self._experts is not None:
            # flatten `list[dict[str, Tensor]]` into `list[str]`
            experts = [k for d in self._experts for k in d.keys()]
            if len(experts) > 0:
                raise ValueError(f"Unprocessed experts: {experts}")


@Model.register("DeepseekV2ForCausalLM")
@Model.register("DeepseekV3ForCausalLM")
class DeepseekV2Model(Model):
    model_arch = gguf.MODEL_ARCH.DEEPSEEK2

    def set_vocab(self):
        self._set_vocab_gpt2()

    def set_gguf_parameters(self):
        super().set_gguf_parameters()
        hparams = self.hparams

        self.gguf_writer.add_leading_dense_block_count(hparams["first_k_dense_replace"])
        self.gguf_writer.add_vocab_size(hparams["vocab_size"])
        if "q_lora_rank" in hparams and hparams["q_lora_rank"] is not None:
            self.gguf_writer.add_q_lora_rank(hparams["q_lora_rank"])
        self.gguf_writer.add_kv_lora_rank(hparams["kv_lora_rank"])
        self.gguf_writer.add_key_length(hparams["qk_nope_head_dim"] + hparams["qk_rope_head_dim"])
        self.gguf_writer.add_value_length(hparams["v_head_dim"])
        self.gguf_writer.add_expert_feed_forward_length(hparams["moe_intermediate_size"])
        self.gguf_writer.add_expert_count(hparams["n_routed_experts"])
        self.gguf_writer.add_expert_shared_count(hparams["n_shared_experts"])
        self.gguf_writer.add_expert_weights_scale(hparams["routed_scaling_factor"])
        self.gguf_writer.add_expert_weights_norm(hparams["norm_topk_prob"])

        if hparams["scoring_func"] == "sigmoid":
            self.gguf_writer.add_expert_gating_func(gguf.ExpertGatingFuncType.SIGMOID)
        elif hparams["scoring_func"] == "softmax":
            self.gguf_writer.add_expert_gating_func(gguf.ExpertGatingFuncType.SOFTMAX)
        else:
            raise ValueError(f"Unsupported scoring_func value: {hparams['scoring_func']}")

        self.gguf_writer.add_rope_dimension_count(hparams["qk_rope_head_dim"])

        if self.hparams.get("rope_scaling") is not None and "factor" in self.hparams["rope_scaling"]:
            if self.hparams["rope_scaling"].get("type") == "yarn":
                self.gguf_writer.add_rope_scaling_type(gguf.RopeScalingType.YARN)
                self.gguf_writer.add_rope_scaling_factor(self.hparams["rope_scaling"]["factor"])

self.gguf_writer.add_rope_scaling_orig_ctx_len(self.hparams["rope_scaling"]["original_max_position_embeddings"]
)
                self.gguf_writer.add_rope_scaling_yarn_log_mul(0.1 * hparams["rope_scaling"]["mscale_all_dim"])

    _experts: list[dict[str, Tensor]] | None = None

    def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str, Tensor]]:
        # rename e_score_correction_bias tensors
```

```
    if name.endswith("e_score_correction_bias"):
        name = name.replace("e_score_correction_bias", "e_score_correction.bias")

    # skip Multi-Token Prediction (MTP) layers
    block_count = self.hparams["num_hidden_layers"]
    match = re.match(r"model.layers.(\d+)", name)
    if match and int(match.group(1)) >= block_count:
        return []

    # process the experts separately
    if name.find("mlp.experts") != -1:
        n_experts = self.hparams["n_routed_experts"]
        assert bid is not None

        if self._experts is None:
            self._experts = [{} for _ in range(self.block_count)]

        self._experts[bid][name] = data_torch

        if len(self._experts[bid]) >= n_experts * 3:
            tensors: list[tuple[str, Tensor]] = []

            # merge the experts into a single 3d tensor
            for w_name in ["down_proj", "gate_proj", "up_proj"]:
                datas: list[Tensor] = []

                for xid in range(n_experts):
                    ename = f"model.layers.{bid}.mlp.experts.{xid}.{w_name}.weight"
                    datas.append(self._experts[bid][ename])
                    del self._experts[bid][ename]

                data_torch = torch.stack(datas, dim=0)

                merged_name = f"model.layers.{bid}.mlp.experts.{w_name}.weight"

                new_name = self.map_tensor_name(merged_name)

                tensors.append((new_name, data_torch))
            return tensors
        else:
            return []

    return [(self.map_tensor_name(name), data_torch)]

def prepare_tensors(self):
    super().prepare_tensors()

    if self._experts is not None:
        # flatten `list[dict[str, Tensor]]` into `list[str]`
        experts = [k for d in self._experts for k in d.keys()]
        if len(experts) > 0:
            raise ValueError(f"Unprocessed experts: {experts}")
```

```
@Model.register("PLMForCausalLM")
class PLMModel(Model):
    model_arch = gguf.MODEL_ARCH.PLM

    def set_vocab(self):
        self._set_vocab_gpt2()

    def set_gguf_parameters(self):
        super().set_gguf_parameters()
        hparams = self.hparams
        self.gguf_writer.add_vocab_size(hparams["vocab_size"])
        self.gguf_writer.add_kv_lora_rank(hparams["kv_lora_rank"])
        self.gguf_writer.add_key_length(hparams["qk_nope_head_dim"] + hparams["qk_rope_head_dim"])
        self.gguf_writer.add_value_length(hparams["v_head_dim"])
        self.gguf_writer.add_rope_dimension_count(hparams["qk_rope_head_dim"])

    def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str, Tensor]]:
        return [(self.map_tensor_name(name), data_torch)]

    def prepare_tensors(self):
        super().prepare_tensors()


@Model.register("T5WithLMHeadModel")
@Model.register("T5ForConditionalGeneration")
@Model.register("MT5ForConditionalGeneration")
@Model.register("UMT5ForConditionalGeneration")
class T5Model(Model):
    model_arch = gguf.MODEL_ARCH.T5

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.shared_token_embeddings_found = False

    def set_vocab(self):
        # to avoid TypeError: Descriptors cannot be created directly
        # exception when importing sentencepiece_model_pb2
        os.environ["PROTOCOL_BUFFERS_PYTHON_IMPLEMENTATION"] = "python"
        from sentencepiece import SentencePieceProcessor
        from sentencepiece import sentencepiece_model_pb2 as model

        tokenizer_path = self.dir_model / 'tokenizer.model'

        # many older models use spiece.model tokenizer model filename
        if not tokenizer_path.is_file():
            tokenizer_path = self.dir_model / 'spiece.model'

        if not tokenizer_path.is_file():
            raise FileNotFoundError(f"File not found: {tokenizer_path}")

        sentencepiece_model = model.ModelProto()  # pyright: ignore[reportAttributeAccessIssue]
        sentencepiece_model.ParseFromString(open(tokenizer_path, "rb").read())

        # some models like Pile-T5 family use BPE tokenizer instead of Unigram
```

```
if sentencepiece_model.trainer_spec.model_type == 2:  # BPE
    # assure the tokenizer model file name is correct
    assert tokenizer_path.name == 'tokenizer.model'
    return self._set_vocab_sentencepiece()
else:
    assert sentencepiece_model.trainer_spec.model_type == 1  # UNIGRAM

add_prefix = sentencepiece_model.normalizer_spec.add_dummy_prefix
remove_whitespaces = sentencepiece_model.normalizer_spec.remove_extra_whitespaces
precompiled_charsmap = sentencepiece_model.normalizer_spec.precompiled_charsmap

tokenizer = SentencePieceProcessor()
tokenizer.LoadFromFile(str(tokenizer_path))

vocab_size = self.hparams.get('vocab_size', tokenizer.vocab_size())

tokens: list[bytes] = [f"[PAD{i}]".encode("utf-8") for i in range(vocab_size)]
scores: list[float] = [-10000.0] * vocab_size
toktypes: list[int] = [SentencePieceTokenTypes.UNUSED] * vocab_size

for token_id in range(tokenizer.vocab_size()):
    piece = tokenizer.IdToPiece(token_id)
    text = piece.encode("utf-8")
    score = tokenizer.GetScore(token_id)

    toktype = SentencePieceTokenTypes.NORMAL
    if tokenizer.IsUnknown(token_id):
        toktype = SentencePieceTokenTypes.UNKNOWN
    elif tokenizer.IsControl(token_id):
        toktype = SentencePieceTokenTypes.CONTROL
    elif tokenizer.IsUnused(token_id):
        toktype = SentencePieceTokenTypes.UNUSED
    elif tokenizer.IsByte(token_id):
        toktype = SentencePieceTokenTypes.BYTE

    tokens[token_id] = text
    scores[token_id] = score
    toktypes[token_id] = toktype

added_tokens_file = self.dir_model / 'added_tokens.json'
if added_tokens_file.is_file():
    with open(added_tokens_file, "r", encoding="utf-8") as f:
        added_tokens_json = json.load(f)
        for key in added_tokens_json:
            token_id = added_tokens_json[key]
            if token_id >= vocab_size:
                logger.warning(f'ignore token {token_id}: id is out of range, max={vocab_size - 1}')
                continue

            tokens[token_id] = key.encode("utf-8")
            scores[token_id] = -1000.0
            toktypes[token_id] = SentencePieceTokenTypes.USER_DEFINED

if vocab_size > len(tokens):
```

```
            pad_count = vocab_size - len(tokens)
            logger.debug(f"Padding vocab with {pad_count} token(s) - [PAD1] through [PAD{pad_count}]")
            for i in range(1, pad_count + 1):
                tokens.append(bytes(f"[PAD{i}]", encoding="utf-8"))
                scores.append(-1000.0)
                toktypes.append(SentencePieceTokenTypes.UNUSED)

        self.gguf_writer.add_tokenizer_model("t5")
        self.gguf_writer.add_tokenizer_pre("default")
        self.gguf_writer.add_token_list(tokens)
        self.gguf_writer.add_token_scores(scores)
        self.gguf_writer.add_token_types(toktypes)
        self.gguf_writer.add_add_space_prefix(add_prefix)
        self.gguf_writer.add_remove_extra_whitespaces(remove_whitespaces)
        if precompiled_charsmap:
            self.gguf_writer.add_precompiled_charsmap(precompiled_charsmap)

        special_vocab = gguf.SpecialVocab(self.dir_model, n_vocab=len(tokens))
        special_vocab.add_to_gguf(self.gguf_writer)

        self.gguf_writer.add_add_bos_token(False)
        self.gguf_writer.add_add_eos_token(True)

    def set_gguf_parameters(self):
        if (n_ctx := self.find_hparam(["n_positions"], optional=True)) is None:
            logger.warning("Couldn't find context length in config.json, assuming default value of 512")
            n_ctx = 512
        self.gguf_writer.add_context_length(n_ctx)
        self.gguf_writer.add_embedding_length(self.hparams["d_model"])
        self.gguf_writer.add_feed_forward_length(self.hparams["d_ff"])
        self.gguf_writer.add_block_count(self.hparams["num_layers"])
        self.gguf_writer.add_head_count(self.hparams["num_heads"])
        self.gguf_writer.add_key_length(self.hparams["d_kv"])
        self.gguf_writer.add_value_length(self.hparams["d_kv"])
        self.gguf_writer.add_layer_norm_eps(self.hparams["layer_norm_epsilon"])
        self.gguf_writer.add_relative_attn_buckets_count(self.hparams["relative_attention_num_buckets"])
        self.gguf_writer.add_layer_norm_rms_eps(self.hparams["layer_norm_epsilon"])
        self.gguf_writer.add_decoder_start_token_id(self.hparams["decoder_start_token_id"])
        self.gguf_writer.add_file_type(self.ftype)

    def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str, Tensor]]:
        del bid  # unused

                # T5 based models contain shared token embeddings tensors saved randomly as either "encoder.embed_tokens.weight",
         # "decoder.embed_tokens.weight" or "shared.weight" tensor. In some models there are even multiple of them stored
         # in the safetensors files. We use the first tensor from these three as the token embeddings for both encoder
        # and decoder and ignore the remaining ones.
        if name in ["decoder.embed_tokens.weight", "encoder.embed_tokens.weight", "shared.weight"]:
            if not self.shared_token_embeddings_found:
                name = "shared.weight"
                self.shared_token_embeddings_found = True
```

```
            else:
                        logger.debug(f"Skipping shared tensor {name!r} in safetensors so that convert can end
normally.")
                return []

        return [(self.map_tensor_name(name), data_torch)]


@Model.register("T5EncoderModel")
class T5EncoderModel(Model):
    model_arch = gguf.MODEL_ARCH.T5ENCODER

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.shared_token_embeddings_found = False

    def set_vocab(self):
        # to avoid TypeError: Descriptors cannot be created directly
        # exception when importing sentencepiece_model_pb2
        os.environ["PROTOCOL_BUFFERS_PYTHON_IMPLEMENTATION"] = "python"
        from sentencepiece import SentencePieceProcessor
        from sentencepiece import sentencepiece_model_pb2 as model

        tokenizer_path = self.dir_model / 'tokenizer.model'

        # many older models use spiece.model tokenizer model filename
        if not tokenizer_path.is_file():
            tokenizer_path = self.dir_model / 'spiece.model'

        if not tokenizer_path.is_file():
            raise FileNotFoundError(f"File not found: {tokenizer_path}")

        sentencepiece_model = model.ModelProto()  # pyright: ignore[reportAttributeAccessIssue]
        sentencepiece_model.ParseFromString(open(tokenizer_path, "rb").read())

        # some models like Pile-T5 family use BPE tokenizer instead of Unigram
        if sentencepiece_model.trainer_spec.model_type == 2:  # BPE
            # assure the tokenizer model file name is correct
            assert tokenizer_path.name == 'tokenizer.model'
            return self._set_vocab_sentencepiece()
        else:
            assert sentencepiece_model.trainer_spec.model_type == 1  # UNIGRAM

        add_prefix = sentencepiece_model.normalizer_spec.add_dummy_prefix
        remove_whitespaces = sentencepiece_model.normalizer_spec.remove_extra_whitespaces
        precompiled_charsmap = sentencepiece_model.normalizer_spec.precompiled_charsmap

        tokenizer = SentencePieceProcessor()
        tokenizer.LoadFromFile(str(tokenizer_path))

        vocab_size = self.hparams.get('vocab_size', tokenizer.vocab_size())

        tokens: list[bytes] = [f"[PAD{i}]".encode("utf-8") for i in range(vocab_size)]
        scores: list[float] = [-10000.0] * vocab_size
```

```
toktypes: list[int] = [SentencePieceTokenTypes.UNUSED] * vocab_size

for token_id in range(tokenizer.vocab_size()):
    piece = tokenizer.IdToPiece(token_id)
    text = piece.encode("utf-8")
    score = tokenizer.GetScore(token_id)

    toktype = SentencePieceTokenTypes.NORMAL
    if tokenizer.IsUnknown(token_id):
        toktype = SentencePieceTokenTypes.UNKNOWN
    elif tokenizer.IsControl(token_id):
        toktype = SentencePieceTokenTypes.CONTROL
    elif tokenizer.IsUnused(token_id):
        toktype = SentencePieceTokenTypes.UNUSED
    elif tokenizer.IsByte(token_id):
        toktype = SentencePieceTokenTypes.BYTE

    tokens[token_id] = text
    scores[token_id] = score
    toktypes[token_id] = toktype

added_tokens_file = self.dir_model / 'added_tokens.json'
if added_tokens_file.is_file():
    with open(added_tokens_file, "r", encoding="utf-8") as f:
        added_tokens_json = json.load(f)
        for key in added_tokens_json:
            token_id = added_tokens_json[key]
            if token_id >= vocab_size:
                logger.warning(f'ignore token {token_id}: id is out of range, max={vocab_size - 1}')
                continue

            tokens[token_id] = key.encode("utf-8")
            scores[token_id] = -1000.0
            toktypes[token_id] = SentencePieceTokenTypes.USER_DEFINED

if vocab_size > len(tokens):
    pad_count = vocab_size - len(tokens)
    logger.debug(f"Padding vocab with {pad_count} token(s) - [PAD1] through [PAD{pad_count}]")
    for i in range(1, pad_count + 1):
        tokens.append(bytes(f"[PAD{i}]", encoding="utf-8"))
        scores.append(-1000.0)
        toktypes.append(SentencePieceTokenTypes.UNUSED)

self.gguf_writer.add_tokenizer_model("t5")
self.gguf_writer.add_tokenizer_pre("default")
self.gguf_writer.add_token_list(tokens)
self.gguf_writer.add_token_scores(scores)
self.gguf_writer.add_token_types(toktypes)
self.gguf_writer.add_add_space_prefix(add_prefix)
self.gguf_writer.add_remove_extra_whitespaces(remove_whitespaces)
if precompiled_charsmap:
    self.gguf_writer.add_precompiled_charsmap(precompiled_charsmap)

special_vocab = gguf.SpecialVocab(self.dir_model, n_vocab=len(tokens))
```

```
        special_vocab.add_to_gguf(self.gguf_writer)

        self.gguf_writer.add_add_bos_token(False)
        self.gguf_writer.add_add_eos_token(True)

    def set_gguf_parameters(self):
        if (n_ctx := self.find_hparam(["n_positions"], optional=True)) is None:
            logger.warning("Couldn't find context length in config.json, assuming default value of 512")
            n_ctx = 512
        self.gguf_writer.add_context_length(n_ctx)
        self.gguf_writer.add_embedding_length(self.hparams["d_model"])
        self.gguf_writer.add_feed_forward_length(self.hparams["d_ff"])
        self.gguf_writer.add_block_count(self.hparams["num_layers"])
        self.gguf_writer.add_head_count(self.hparams["num_heads"])
        self.gguf_writer.add_key_length(self.hparams["d_kv"])
        self.gguf_writer.add_value_length(self.hparams["d_kv"])
        self.gguf_writer.add_layer_norm_eps(self.hparams["layer_norm_epsilon"])
        self.gguf_writer.add_relative_attn_buckets_count(self.hparams["relative_attention_num_buckets"])
        self.gguf_writer.add_layer_norm_rms_eps(self.hparams["layer_norm_epsilon"])
        self.gguf_writer.add_file_type(self.ftype)

    def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str, Tensor]]:
        del bid  # unused

                # T5 based models contain shared token embeddings tensors saved randomly as either "encoder.embed_tokens.weight",
         # "decoder.embed_tokens.weight" or "shared.weight" tensor. In some models there are even multiple of them stored
         # in the safetensors files. We use the first tensor from these three as the token embeddings for both encoder
        # and decoder and ignore the remaining ones.
        if name in ["decoder.embed_tokens.weight", "encoder.embed_tokens.weight", "shared.weight"]:
            if not self.shared_token_embeddings_found:
                name = "shared.weight"
                self.shared_token_embeddings_found = True
            else:
                        logger.debug(f"Skipping shared tensor {name!r} in safetensors so that convert can end normally.")
                return []

        return [(self.map_tensor_name(name), data_torch)]


@Model.register("JAISLMHeadModel")
class JaisModel(Model):
    model_arch = gguf.MODEL_ARCH.JAIS

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        # SwigLU activation
        assert self.hparams["activation_function"] == "swiglu"
        # ALiBi position embedding
        assert self.hparams["position_embedding_type"] == "alibi"
```

```
        # Embeddings scale
        self.embeddings_scale = 1.0
        if 'mup_embeddings_scale' in self.hparams:
            self.embeddings_scale = self.hparams['mup_embeddings_scale']
        elif 'embeddings_scale' in self.hparams:
            self.embeddings_scale = self.hparams['embeddings_scale']
        else:
            assert False

        self.width_scale = 1.0
        if 'mup_output_alpha' in self.hparams:
            assert 'mup_width_scale' in self.hparams
            self.width_scale = self.hparams['mup_output_alpha'] * self.hparams['mup_width_scale']
        elif 'width_scale' in self.hparams:
            self.width_scale = self.hparams['width_scale']
        else:
            assert False

        self.max_alibi_bias = 8.0

    def set_vocab(self):
        self._set_vocab_gpt2()

    def set_gguf_parameters(self):
        self.gguf_writer.add_block_count(self.hparams["n_layer"])
        self.gguf_writer.add_context_length(self.hparams["n_positions"])
        self.gguf_writer.add_embedding_length(self.hparams["n_embd"])
        self.gguf_writer.add_feed_forward_length(self.hparams["n_inner"])
        self.gguf_writer.add_head_count(self.hparams["n_head"])
        self.gguf_writer.add_layer_norm_eps(self.hparams["layer_norm_epsilon"])
        self.gguf_writer.add_file_type(self.ftype)

    def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str, Tensor]]:
        del bid  # unused

        tensors: list[tuple[str, Tensor]] = []

        # we don't need these
        if name.endswith((".attn.bias")):
            return tensors

        if name.endswith(("relative_pe.slopes")):
            # Calculate max ALiBi bias (this is the inverse of the ALiBi calculation)
            # Some other models has max_alibi_bias spelled out explicitly in the hyperparams,
            # but Jais's PyTorch model simply precalculates the slope values and places them
            # in relative_pes.slopes
            n_head_closest_log2 = 2 ** math.floor(math.log2(self.hparams["n_head"]))
            first_val = float(data_torch[0].item())
            self.max_alibi_bias = -round(math.log2(first_val) * n_head_closest_log2)

            return tensors

        if name.endswith((".c_attn.weight", ".c_proj.weight", ".c_fc.weight", ".c_fc2.weight")):
```

```
            data_torch = data_torch.transpose(1, 0)

        new_name = self.map_tensor_name(name)

        if new_name == self.format_tensor_name(gguf.MODEL_TENSOR.TOKEN_EMBD):
            tensors.append((new_name, data_torch * self.embeddings_scale))
        elif new_name == self.format_tensor_name(gguf.MODEL_TENSOR.OUTPUT):
            tensors.append((new_name, data_torch * self.width_scale))
        else:
            tensors.append((new_name, data_torch))

        return tensors

    def prepare_tensors(self):
        super().prepare_tensors()
        self.gguf_writer.add_max_alibi_bias(self.max_alibi_bias)


@Model.register("Glm4ForCausalLM")
class Glm4Model(Model):
    model_arch = gguf.MODEL_ARCH.GLM4

    def set_vocab(self):
        self._set_vocab_gpt2()

    def set_gguf_parameters(self):
        super().set_gguf_parameters()
        if self.hparams.get("rope_scaling") is not None and "factor" in self.hparams["rope_scaling"]:
            if self.hparams["rope_scaling"].get("type") == "yarn":
                self.gguf_writer.add_rope_scaling_type(gguf.RopeScalingType.YARN)
                self.gguf_writer.add_rope_scaling_factor(self.hparams["rope_scaling"]["factor"])

self.gguf_writer.add_rope_scaling_orig_ctx_len(self.hparams["rope_scaling"]["original_max_position_embeddings"]
)


@Model.register("GlmForCausalLM", "ChatGLMModel", "ChatGLMForConditionalGeneration")
class ChatGLMModel(Model):
    model_arch = gguf.MODEL_ARCH.CHATGLM

    def set_vocab_chatglm3(self):
        dir_model = self.dir_model
        hparams = self.hparams
        tokens: list[bytes] = []
        toktypes: list[int] = []
        scores: list[float] = []

        from transformers import AutoTokenizer
        tokenizer = AutoTokenizer.from_pretrained(dir_model, trust_remote_code=True)
        vocab_size = hparams.get("padded_vocab_size", len(tokenizer.get_vocab()))
        assert max(tokenizer.get_vocab().values()) < vocab_size
        role_special_tokens = ["<|system|>", "<|user|>", "<|assistant|>", "<|observation|>"]
        special_tokens = ["[MASK]", "[gMASK]", "[sMASK]", "sop", "eop"] + role_special_tokens
        for token_id in range(vocab_size):
```

```
            piece = tokenizer._convert_id_to_token(token_id)
            if token_id == 0:
                piece = "<unk>"
            elif token_id == 1:
                piece = "<bos>"
            elif token_id == 2:
                piece = "<eos>"

            text = piece.encode("utf-8")
            score = 0.0
                                                                # Referencing the tokenizer Python
implementation(https://huggingface.co/THUDM/chatglm3-6b/blob/main/tokenization_chatglm.py),
            # it is only valid if it is less than tokenizer.tokenizer.sp_model.vocab_size()
            if len(piece) != 0 and token_id < tokenizer.tokenizer.sp_model.vocab_size():
                score = tokenizer.tokenizer.sp_model.get_score(token_id)

            if token_id >= tokenizer.tokenizer.sp_model.vocab_size():
                if piece in special_tokens:
                    toktype = SentencePieceTokenTypes.CONTROL
                elif len(piece) == 0:
                    text = f"[PAD{token_id}]".encode("utf-8")
                    toktype = SentencePieceTokenTypes.UNUSED
                else:
                    toktype = SentencePieceTokenTypes.USER_DEFINED
                tokens.append(text)
                scores.append(score)
                toktypes.append(toktype)
                continue

            toktype = SentencePieceTokenTypes.NORMAL
            if tokenizer.tokenizer.sp_model.is_unknown(token_id):
                toktype = SentencePieceTokenTypes.UNKNOWN
            elif tokenizer.tokenizer.sp_model.is_control(token_id):
                toktype = SentencePieceTokenTypes.CONTROL
            elif tokenizer.tokenizer.sp_model.is_unused(token_id):
                toktype = SentencePieceTokenTypes.UNUSED
            elif tokenizer.tokenizer.sp_model.is_byte(token_id):
                toktype = SentencePieceTokenTypes.BYTE

            tokens.append(text)
            scores.append(score)
            toktypes.append(toktype)

        self.gguf_writer.add_tokenizer_model("llama")
        # glm3 needs prefix and suffix formatted as:
        # prompt = "[gMASK]sop<|user|>\n" + prompt + "<|assistant|>"
        self.gguf_writer.add_tokenizer_pre("chatglm-spm")
        self.gguf_writer.add_token_list(tokens)
        self.gguf_writer.add_token_scores(scores)
        self.gguf_writer.add_token_types(toktypes)

        special_vocab = gguf.SpecialVocab(self.dir_model, n_vocab=len(tokens))
        special_vocab.add_to_gguf(self.gguf_writer)
```

```
@staticmethod
def token_bytes_to_string(b):
    from transformers.models.gpt2.tokenization_gpt2 import bytes_to_unicode
    byte_encoder = bytes_to_unicode()
    return ''.join([byte_encoder[ord(char)] for char in b.decode('latin-1')])

@staticmethod
def bpe(mergeable_ranks: dict[bytes, int], token: bytes, max_rank: int | None = None) -> list[bytes]:
    parts = [bytes([b]) for b in token]
    while True:
        min_idx = None
        min_rank = None
        for i, pair in enumerate(zip(parts[:-1], parts[1:])):
            rank = mergeable_ranks.get(pair[0] + pair[1])
            if rank is not None and (min_rank is None or rank < min_rank):
                min_idx = i
                min_rank = rank
        if min_rank is None or (max_rank is not None and min_rank >= max_rank):
            break
        assert min_idx is not None
        parts = parts[:min_idx] + [parts[min_idx] + parts[min_idx + 1]] + parts[min_idx + 2:]
    return parts

def set_vocab(self):
    if "THUDM/chatglm3-6b" in self.hparams.get("_name_or_path", ""):
        self.set_vocab_chatglm3()
        return

    dir_model = self.dir_model
    hparams = self.hparams
    tokens: list[str] = []
    toktypes: list[int] = []

    from transformers import AutoTokenizer
    tokenizer = AutoTokenizer.from_pretrained(dir_model, trust_remote_code=True)
    vocab_size = hparams.get("padded_vocab_size",hparams["vocab_size"])
    assert max(tokenizer.get_vocab().values()) < vocab_size

    tokens, toktypes, tokpre = self.get_vocab_base()
    self.gguf_writer.add_tokenizer_model("gpt2")
    self.gguf_writer.add_tokenizer_pre(tokpre)
    self.gguf_writer.add_token_list(tokens)
    self.gguf_writer.add_token_types(toktypes)
    special_vocab = gguf.SpecialVocab(self.dir_model, load_merges=True)
    # only add special tokens when they were not already loaded from config.json
    special_vocab._set_special_token("eos", tokenizer.get_added_vocab()["<|endoftext|>"])
    special_vocab._set_special_token("eot", tokenizer.get_added_vocab()["<|user|>"])
    # this one is usually not in config.json anyway
    special_vocab._set_special_token("unk", tokenizer.get_added_vocab()["<|endoftext|>"])
    special_vocab.add_to_gguf(self.gguf_writer)

def set_gguf_parameters(self):
    n_embed = self.hparams.get("hidden_size", self.hparams.get("n_embed"))
    n_head = self.hparams.get("n_head", self.hparams.get("num_attention_heads"))
```

```
        n_head_kv = self.hparams.get("multi_query_group_num", self.hparams.get("num_key_value_heads", n_head))
        self.gguf_writer.add_context_length(self.hparams.get("seq_length", n_embed))
        self.gguf_writer.add_embedding_length(n_embed)
                                    self.gguf_writer.add_feed_forward_length(self.hparams.get("ffn_hidden_size",
self.hparams.get("intermediate_size", 4 * n_embed)))
        self.gguf_writer.add_block_count(self.hparams.get("num_layers", self.hparams["num_hidden_layers"]))
        self.gguf_writer.add_head_count(n_head)
        self.gguf_writer.add_head_count_kv(n_head_kv)
        self.gguf_writer.add_layer_norm_rms_eps(self.hparams.get("layernorm_epsilon",1e-5))
        self.gguf_writer.add_file_type(self.ftype)
        if "attention_dim" in self.hparams:
            rope_dim = self.hparams["attention_dim"]
        else:
            rope_dim = self.hparams["hidden_size"] // self.hparams["num_attention_heads"]
            self.gguf_writer.add_rope_dimension_count(int(rope_dim  *  self.hparams.get("partial_rotary_factor",
0.5)))
        self.gguf_writer.add_add_bos_token(False)
        rope_freq = 10000
        if "rope_ratio" in self.hparams:
            rope_freq = rope_freq * self.hparams["rope_ratio"]
        self.gguf_writer.add_rope_freq_base(rope_freq)

    def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str, Tensor]]:
        del bid  # unused

        if name.endswith(".rotary_pos_emb.inv_freq") or name.startswith("model.vision."):
            return []

        name = name.removeprefix("transformer.")
        return [(self.map_tensor_name(name), data_torch)]


@Model.register("NemotronForCausalLM")
class NemotronModel(Model):
    model_arch = gguf.MODEL_ARCH.NEMOTRON

    def set_vocab(self):
        self._set_vocab_sentencepiece()
        self.gguf_writer.add_pad_token_id(0)
        self.gguf_writer.add_unk_token_id(1)

    def set_gguf_parameters(self):
        super().set_gguf_parameters()
        hparams = self.hparams
        self.gguf_writer.add_vocab_size(hparams["vocab_size"])

        f_norm_eps = self.find_hparam(["layer_norm_eps", "layer_norm_epsilon", "norm_epsilon", "norm_eps"])
        self.gguf_writer.add_layer_norm_eps(f_norm_eps)

        # * Partial RoPE
        rot_pct = self.find_hparam(["partial_rotary_factor", "rope_pct", "rope_percent"])
        n_embd = self.find_hparam(["hidden_size", "n_embd"])
        n_head = self.find_hparam(["num_attention_heads", "n_head"])
        self.gguf_writer.add_rope_dimension_count(int(rot_pct * n_embd) // n_head)
```

```
        # * RopeScaling for Nemotron
        if "rope_scaling" not in self.hparams or self.hparams["rope_scaling"] is None:
            self.gguf_writer.add_rope_scaling_type(gguf.RopeScalingType.NONE)
        else:
            self.gguf_writer.add_rope_scaling_type(gguf.RopeScalingType.LINEAR)
            self.gguf_writer.add_rope_scaling_factor(self.hparams["factor"])

    def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str, Tensor]]:
         # * Adding +1 to LayerNorm's weights here to implement layernorm1p w/o changing anything on the GGML
engine side
        #   model.layers.{l}.input_layernorm.weight
        #   model.layers.{l}.post_attention_layernorm.weight
        #   model.norm.weight
        if name.endswith("norm.weight"):
            data_torch = data_torch + 1

        return [(self.map_tensor_name(name), data_torch)]


@Model.register("ExaoneForCausalLM")
class ExaoneModel(Model):
    model_arch = gguf.MODEL_ARCH.EXAONE

    def set_gguf_parameters(self):
        hparams = self.hparams

        assert (hparams["activation_function"] == "silu")

        max_position_embeddings = hparams["max_position_embeddings"]
        embed_dim = hparams["hidden_size"]
        num_heads = hparams["num_attention_heads"]
        num_kv_heads = hparams.get("num_key_value_heads", num_heads)
        layer_norm_eps = hparams["layer_norm_epsilon"]
        intermediate_size = hparams["intermediate_size"] if "intermediate_size" in hparams else 4 * embed_dim
        num_layers = hparams["num_layers"]
        # ignore for now as EXAONE-3.0-7.8B-Instruct attentino_dropout is 0.0
        # attention_dropout_rate = hparams["attention_dropout"]
        # ignore for now as EXAONE-3.0-7.8B-Instruct embed_dropout is 0.0
        # embed_dropout_rate = hparams["embed_dropout"]
        self.gguf_writer.add_embedding_length(embed_dim)
        self.gguf_writer.add_head_count(num_heads)
        self.gguf_writer.add_head_count_kv(num_kv_heads)
        self.gguf_writer.add_context_length(max_position_embeddings)
        self.gguf_writer.add_layer_norm_rms_eps(layer_norm_eps)
        self.gguf_writer.add_feed_forward_length(intermediate_size)
        self.gguf_writer.add_block_count(num_layers)
        self.gguf_writer.add_file_type(self.ftype)

        if (rope_theta := self.hparams.get("rope_theta")) is not None:
            self.gguf_writer.add_rope_freq_base(rope_theta)
        rotary_factor = self.find_hparam(["partial_rotary_factor", "rope_pct"], optional=True)
        rotary_factor = rotary_factor if rotary_factor is not None else 1.0
                    self.gguf_writer.add_rope_dimension_count(int(rotary_factor  *  (hparams["hidden_size"]  //
```

```
hparams["num_attention_heads"])))
        if hparams.get("rope_scaling") is not None and "factor" in hparams["rope_scaling"]:
            if hparams["rope_scaling"].get("type") == "linear":
                self.gguf_writer.add_rope_scaling_type(gguf.RopeScalingType.LINEAR)
                self.gguf_writer.add_rope_scaling_factor(hparams["rope_scaling"]["factor"])

    def generate_extra_tensors(self) -> Iterable[tuple[str, Tensor]]:
        if rope_scaling := self.find_hparam(["rope_scaling"], optional=True):
            if rope_scaling.get("rope_type", '').lower() == "llama3":
                base = self.hparams.get("rope_theta", 10000.0)
                                    dim = self.hparams.get("head_dim", self.hparams["hidden_size"] //
self.hparams["num_attention_heads"])
                freqs = 1.0 / (base ** (torch.arange(0, dim, 2, dtype=torch.float32) / dim))

                factor = rope_scaling.get("factor", 8.0)
                low_freq_factor = rope_scaling.get("low_freq_factor", 1.0)
                high_freq_factor = rope_scaling.get("high_freq_factor", 4.0)
                old_context_len = self.hparams.get("original_max_position_embeddings", 8192)

                low_freq_wavelen = old_context_len / low_freq_factor
                high_freq_wavelen = old_context_len / high_freq_factor
                assert low_freq_wavelen != high_freq_wavelen

                rope_factors = []
                for freq in freqs:
                    wavelen = 2 * math.pi / freq
                    if wavelen < high_freq_wavelen:
                        rope_factors.append(1)
                    elif wavelen > low_freq_wavelen:
                        rope_factors.append(factor)
                    else:
                                smooth = (old_context_len / wavelen - low_freq_factor) / (high_freq_factor -
low_freq_factor)
                        rope_factors.append(1 / ((1 - smooth) / factor + smooth))

                  yield (self.format_tensor_name(gguf.MODEL_TENSOR.ROPE_FREQS), torch.tensor(rope_factors,
dtype=torch.float32))


@Model.register("GraniteForCausalLM")
class GraniteModel(LlamaModel):
    """Conversion for IBM's GraniteForCausalLM"""
    model_arch = gguf.MODEL_ARCH.GRANITE

    def set_gguf_parameters(self):
        """Granite uses standard llama parameters with the following differences:

        - No head_dim support
        - New multiplier params:
            - attention_scale
            - embedding_scale
            - residual_scale
        - logits_scaling
        """
```

```
        if head_dim := self.hparams.pop("head_dim", None):
            logger.warning("Ignoring head_dim (%s) from config for Granite", head_dim)
        super().set_gguf_parameters()
        # NOTE: Convert _multiplier params to _scale params for naming
        #   consistency
        if attention_scale := self.hparams.get("attention_multiplier"):
            self.gguf_writer.add_attention_scale(attention_scale)
            logger.info("gguf: (granite) attention_scale = %s", attention_scale)
        if embedding_scale := self.hparams.get("embedding_multiplier"):
            self.gguf_writer.add_embedding_scale(embedding_scale)
            logger.info("gguf: (granite) embedding_scale = %s", embedding_scale)
        if residual_scale := self.hparams.get("residual_multiplier"):
            self.gguf_writer.add_residual_scale(residual_scale)
            logger.info("gguf: (granite) residual_scale = %s", residual_scale)
        if logits_scale := self.hparams.get("logits_scaling"):
            self.gguf_writer.add_logit_scale(logits_scale)
            logger.info("gguf: (granite) logits_scale = %s", logits_scale)


@Model.register("GraniteMoeForCausalLM")
class GraniteMoeModel(GraniteModel):
    """Conversion for IBM's GraniteMoeForCausalLM"""
    model_arch = gguf.MODEL_ARCH.GRANITE_MOE

    def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str, Tensor]]:
        """In modeling_granitemoe, the JetMoe implementation of parallel experts
        is used. This essentially merges w1 and w3 into a single tensor with 2x
        the hidden size that is then split during forward. To keep compatibility
        with existing mixtral support, we pull them apart here.
        """

        if name.endswith("block_sparse_moe.input_linear.weight"):
            ffn_dim = self.hparams["intermediate_size"]
            assert data_torch.shape[-2] == 2 * ffn_dim, "Merged FFN tensor size must be 2 * intermediate_size"
            gate, up = data_torch[..., :ffn_dim, :], data_torch[..., ffn_dim:, :]
            return [
                (self.format_tensor_name(gguf.MODEL_TENSOR.FFN_GATE_EXP, bid), gate),
                (self.format_tensor_name(gguf.MODEL_TENSOR.FFN_UP_EXP, bid), up),
            ]

        return super().modify_tensors(data_torch, name, bid)


@Model.register("BailingMoeForCausalLM")
class BailingMoeModel(Model):
    model_arch = gguf.MODEL_ARCH.BAILINGMOE

    def set_vocab(self):
        self._set_vocab_gpt2()

    def set_gguf_parameters(self):
        super().set_gguf_parameters()
        hparams = self.hparams
        rope_dim = hparams.get("head_dim") or hparams["hidden_size"] // hparams["num_attention_heads"]
```