```
                                                                for     name,     remote_tensor     in
gguf.utility.SafetensorRemote.get_list_tensors_hf_model(remote_hf_model_id).items():
                    yield (name, LazyTorchTensor.from_remote_tensor(remote_tensor))

            self.get_tensors = get_remote_tensors
        else:
            self.part_names = Model.get_model_part_names(self.dir_model, "model", ".safetensors")
            self.is_safetensors = len(self.part_names) > 0
            if not self.is_safetensors:
                self.part_names = Model.get_model_part_names(self.dir_model, "pytorch_model", ".bin")
        self.hparams = Model.load_hparams(self.dir_model) if hparams is None else hparams
        self.block_count = self.find_hparam(["n_layers", "num_hidden_layers", "n_layer", "num_layers"])
        self.tensor_map = gguf.get_tensor_name_map(self.model_arch, self.block_count)
        self.tensor_names = None
        self.metadata_override = metadata_override
        self.model_name = model_name
        self.dir_model_card = dir_model  # overridden in convert_lora_to_gguf.py

        # Apply heuristics to figure out typical tensor encoding based on first layer tensor encoding type
        if self.ftype == gguf.LlamaFileType.GUESSED:
            # NOTE: can't use field "torch_dtype" in config.json, because some finetunes lie.
            _, first_tensor = next(self.get_tensors())
            if first_tensor.dtype == torch.float16:
                logger.info(f"choosing --outtype f16 from first tensor type ({first_tensor.dtype})")
                self.ftype = gguf.LlamaFileType.MOSTLY_F16
            else:
                logger.info(f"choosing --outtype bf16 from first tensor type ({first_tensor.dtype})")
                self.ftype = gguf.LlamaFileType.MOSTLY_BF16

        # Configure GGUF Writer
                 self.gguf_writer  =  gguf.GGUFWriter(path=None,  arch=gguf.MODEL_ARCH_NAMES[self.model_arch],
endianess=self.endianess, use_temp_file=self.use_temp_file,
                                             split_max_tensors=split_max_tensors, split_max_size=split_max_size,
dry_run=dry_run, small_first_shard=small_first_shard)

    @classmethod
    def __init_subclass__(cls):
        # can't use an abstract property, because overriding it without type errors
        # would require using decorated functions instead of simply defining the property
        if "model_arch" not in cls.__dict__:
            raise TypeError(f"Missing property 'model_arch' for {cls.__name__!r}")

    def find_hparam(self, keys: Iterable[str], optional: bool = False) -> Any:
        key = next((k for k in keys if k in self.hparams), None)
        if key is not None:
            return self.hparams[key]
        if optional:
            return None
        raise KeyError(f"could not find any of: {keys}")

    def set_vocab(self):
        self._set_vocab_gpt2()

    def get_tensors(self) -> Iterator[tuple[str, Tensor]]:
```

```
        tensor_names_from_parts: set[str] = set()

        index_name = "model.safetensors" if self.is_safetensors else "pytorch_model.bin"
        index_name += ".index.json"
        index_file = self.dir_model / index_name

        if index_file.is_file():
            self.tensor_names = set()
            logger.info(f"gguf: loading model weight map from '{index_name}'")
            with open(index_file, "r", encoding="utf-8") as f:
                index: dict[str, Any] = json.load(f)
                weight_map = index.get("weight_map")
                if weight_map is None or not isinstance(weight_map, dict):
                    raise ValueError(f"Can't load 'weight_map' from {index_name!r}")
                self.tensor_names.update(weight_map.keys())
        else:
            self.tensor_names = tensor_names_from_parts
            weight_map = {}

        for part_name in self.part_names:
            logger.info(f"gguf: loading model part '{part_name}'")
            ctx: ContextManager[Any]
            if self.is_safetensors:
                from safetensors import safe_open
                       ctx = cast(ContextManager[Any], safe_open(self.dir_model / part_name, framework="pt",
device="cpu"))
            else:
                  ctx = contextlib.nullcontext(torch.load(str(self.dir_model / part_name), map_location="cpu",
mmap=True, weights_only=True))

            with ctx as model_part:
                tensor_names_from_parts.update(model_part.keys())

                for name in model_part.keys():
                    if self.is_safetensors:
                        if self.lazy:
                            data = model_part.get_slice(name)
                            data = LazyTorchTensor.from_safetensors_slice(data)
                        else:
                            data = model_part.get_tensor(name)
                    else:
                        data = model_part[name]
                        if self.lazy:
                            data = LazyTorchTensor.from_eager(data)
                    yield name, data

        # verify tensor name presence and identify potentially missing files
        if len(tensor_names_from_parts.symmetric_difference(self.tensor_names)) > 0:
            missing = sorted(self.tensor_names.difference(tensor_names_from_parts))
            extra = sorted(tensor_names_from_parts.difference(self.tensor_names))
            missing_files = sorted(set(weight_map[n] for n in missing if n in weight_map))
            if len(extra) == 0 and len(missing_files) > 0:
                raise ValueError(f"Missing or incomplete model files: {missing_files}\n"
                                 f"Missing tensors: {missing}")
```

```
            else:
                raise ValueError("Mismatch between weight map and model parts for tensor names:\n"
                                 f"Missing tensors: {missing}\n"
                                 f"Extra tensors: {extra}")

    def format_tensor_name(self, key: gguf.MODEL_TENSOR, bid: int | None = None, suffix: str = ".weight") ->
str:
        if key not in gguf.MODEL_TENSORS[self.model_arch]:
            raise ValueError(f"Missing {key!r} for MODEL_TENSORS of {self.model_arch!r}")
        name: str = gguf.TENSOR_NAMES[key]
        if "{bid}" in name:
            assert bid is not None
            name = name.format(bid=bid)
        return name + suffix

     def match_model_tensor_name(self, name: str, key: gguf.MODEL_TENSOR, bid: int | None, suffix: str =
".weight") -> bool:
        if key not in gguf.MODEL_TENSORS[self.model_arch]:
            return False
        key_name: str = gguf.TENSOR_NAMES[key]
        if "{bid}" in key_name:
            if bid is None:
                return False
            key_name = key_name.format(bid=bid)
        else:
            if bid is not None:
                return False
        return name == (key_name + suffix)

    def map_tensor_name(self, name: str, try_suffixes: Sequence[str] = (".weight", ".bias")) -> str:
        new_name = self.tensor_map.get_name(key=name, try_suffixes=try_suffixes)
        if new_name is None:
            raise ValueError(f"Can not map tensor {name!r}")
        return new_name

    def set_gguf_parameters(self):
        self.gguf_writer.add_block_count(self.block_count)

        if (n_ctx := self.find_hparam(["max_position_embeddings", "n_ctx"], optional=True)) is not None:
            self.gguf_writer.add_context_length(n_ctx)
            logger.info(f"gguf: context length = {n_ctx}")

        if (n_embd := self.find_hparam(["hidden_size", "n_embd"], optional=True)) is not None:
            self.gguf_writer.add_embedding_length(n_embd)
            logger.info(f"gguf: embedding length = {n_embd}")

        if (n_ff := self.find_hparam(["intermediate_size", "n_inner"], optional=True)) is not None:
            self.gguf_writer.add_feed_forward_length(n_ff)
            logger.info(f"gguf: feed forward length = {n_ff}")

        if (n_head := self.find_hparam(["num_attention_heads", "n_head"], optional=True)) is not None:
            self.gguf_writer.add_head_count(n_head)
            logger.info(f"gguf: head count = {n_head}")
```

```
        if (n_head_kv := self.hparams.get("num_key_value_heads")) is not None:
            self.gguf_writer.add_head_count_kv(n_head_kv)
            logger.info(f"gguf: key-value head count = {n_head_kv}")

        if (rope_theta := self.hparams.get("rope_theta")) is not None:
            self.gguf_writer.add_rope_freq_base(rope_theta)
            logger.info(f"gguf: rope theta = {rope_theta}")
        if (f_rms_eps := self.hparams.get("rms_norm_eps")) is not None:
            self.gguf_writer.add_layer_norm_rms_eps(f_rms_eps)
            logger.info(f"gguf: rms norm epsilon = {f_rms_eps}")
               if  (f_norm_eps  :=  self.find_hparam(["layer_norm_eps",  "layer_norm_epsilon",  "norm_epsilon"],
optional=True)) is not None:
            self.gguf_writer.add_layer_norm_eps(f_norm_eps)
            logger.info(f"gguf: layer norm epsilon = {f_norm_eps}")
        if (n_experts := self.hparams.get("num_local_experts")) is not None:
            self.gguf_writer.add_expert_count(n_experts)
            logger.info(f"gguf: expert count = {n_experts}")
        if (n_experts_used := self.hparams.get("num_experts_per_tok")) is not None:
            self.gguf_writer.add_expert_used_count(n_experts_used)
            logger.info(f"gguf: experts used count = {n_experts_used}")

        if (head_dim := self.hparams.get("head_dim")) is not None:
            self.gguf_writer.add_key_length(head_dim)
            self.gguf_writer.add_value_length(head_dim)

        self.gguf_writer.add_file_type(self.ftype)
        logger.info(f"gguf: file type = {self.ftype}")

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

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

         def  tensor_force_quant(self,  name:  str,  new_name:  str,  bid:  int  |  None,  n_dims:  int)  ->
gguf.GGMLQuantizationType | bool:
        del name, new_name, bid, n_dims  # unused

        return False

    # some models need extra generated tensors (like rope_freqs)
    def generate_extra_tensors(self) -> Iterable[tuple[str, Tensor]]:
        return ()

    def prepare_tensors(self):
        max_name_len = max(len(s) for _, s in self.tensor_map.mapping.values()) + len(".weight,")

        for name, data_torch in chain(self.generate_extra_tensors(), self.get_tensors()):
            # we don't need these
            if name.endswith((".attention.masked_bias", ".attention.bias", ".rotary_emb.inv_freq")):
                continue

            old_dtype = data_torch.dtype

            # convert any unsupported data types to float32
```

```
            if data_torch.dtype not in (torch.float16, torch.float32):
                data_torch = data_torch.to(torch.float32)

            # use the first number-like part of the tensor name as the block id
            bid = None
            for part in name.split("."):
                if part.isdecimal():
                    bid = int(part)
                    break

            for new_name, data_torch in (self.modify_tensors(data_torch, name, bid)):
                # TODO: why do we squeeze here?
                # data = data_torch.squeeze().numpy()
                data = data_torch.numpy()

                # if data ends up empty, it means data_torch was a scalar tensor -> restore
                if len(data.shape) == 0:
                    data = data_torch.numpy()

                n_dims = len(data.shape)
                   data_qtype: gguf.GGMLQuantizationType | bool = self.tensor_force_quant(name, new_name, bid,
n_dims)

                # Most of the codebase that takes in 1D tensors or norms only handles F32 tensors
                if n_dims <= 1 or new_name.endswith("_norm.weight"):
                    data_qtype = gguf.GGMLQuantizationType.F32

                # Conditions should closely match those in llama_model_quantize_internal in llama.cpp
                # Some tensor types are always in float32
                if data_qtype is False and (
                    any(
                        self.match_model_tensor_name(new_name, key, bid)
                        for key in (
                            gguf.MODEL_TENSOR.FFN_GATE_INP,
                            gguf.MODEL_TENSOR.POS_EMBD,
                            gguf.MODEL_TENSOR.TOKEN_TYPES,
                            gguf.MODEL_TENSOR.SSM_CONV1D,
                            gguf.MODEL_TENSOR.TIME_MIX_FIRST,
                            gguf.MODEL_TENSOR.TIME_MIX_W1,
                            gguf.MODEL_TENSOR.TIME_MIX_W2,
                            gguf.MODEL_TENSOR.TIME_MIX_DECAY_W1,
                            gguf.MODEL_TENSOR.TIME_MIX_DECAY_W2,
                            gguf.MODEL_TENSOR.TIME_MIX_LERP_FUSED,
                            gguf.MODEL_TENSOR.POSNET_NORM1,
                            gguf.MODEL_TENSOR.POSNET_NORM2,
                        )
                    )
                    or not new_name.endswith(".weight")
                ):
                    data_qtype = gguf.GGMLQuantizationType.F32

                if data_qtype is False and any(
                    self.match_model_tensor_name(new_name, key, bid)
                    for key in (
```

```
                        gguf.MODEL_TENSOR.TOKEN_EMBD,
                        gguf.MODEL_TENSOR.OUTPUT,
                    )
                ):
                    if self.ftype in (
                        gguf.LlamaFileType.MOSTLY_TQ1_0,
                        gguf.LlamaFileType.MOSTLY_TQ2_0,
                    ):
                        # TODO: use Q4_K and Q6_K
                        data_qtype = gguf.GGMLQuantizationType.F16

                # No override (data_qtype is False), or wants to be quantized (data_qtype is True)
                if isinstance(data_qtype, bool):
                    if self.ftype == gguf.LlamaFileType.ALL_F32:
                        data_qtype = gguf.GGMLQuantizationType.F32
                    elif self.ftype == gguf.LlamaFileType.MOSTLY_F16:
                        data_qtype = gguf.GGMLQuantizationType.F16
                    elif self.ftype == gguf.LlamaFileType.MOSTLY_BF16:
                        data_qtype = gguf.GGMLQuantizationType.BF16
                    elif self.ftype == gguf.LlamaFileType.MOSTLY_Q8_0:
                        data_qtype = gguf.GGMLQuantizationType.Q8_0
                    elif self.ftype == gguf.LlamaFileType.MOSTLY_TQ1_0:
                        data_qtype = gguf.GGMLQuantizationType.TQ1_0
                    elif self.ftype == gguf.LlamaFileType.MOSTLY_TQ2_0:
                        data_qtype = gguf.GGMLQuantizationType.TQ2_0
                    else:
                        raise ValueError(f"Unknown file type: {self.ftype.name}")

                try:
                    data = gguf.quants.quantize(data, data_qtype)
                except gguf.QuantError as e:
                    logger.warning("%s, %s", e, "falling back to F16")
                    data_qtype = gguf.GGMLQuantizationType.F16
                    data = gguf.quants.quantize(data, data_qtype)

                shape = gguf.quant_shape_from_byte_shape(data.shape, data_qtype) if data.dtype == np.uint8 else
data.shape

                # reverse shape to make it similar to the internal ggml dimension order
                shape_str = f"{{{', '.join(str(n) for n in reversed(shape))}}}"

                # n_dims is implicit in the shape
                 logger.info(f"{f'%-{max_name_len}s' % f'{new_name},'} {old_dtype} --> {data_qtype.name}, shape
= {shape_str}")

                self.gguf_writer.add_tensor(new_name, data, raw_dtype=data_qtype)

    def set_type(self):
        self.gguf_writer.add_type(gguf.GGUFType.MODEL)

    def prepare_metadata(self, vocab_only: bool):

        total_params, shared_params, expert_params, expert_count = self.gguf_writer.get_total_parameter_count()
```

```
            self.metadata = gguf.Metadata.load(self.metadata_override, self.dir_model_card, self.model_name, total_params)

        # If we are using HF model id, set the metadata name to the model id
        if self.remote_hf_model_id:
            self.metadata.name = self.remote_hf_model_id

        # Fallback to model directory name if metadata name is still missing
        if self.metadata.name is None:
            self.metadata.name = self.dir_model.name

        # Generate parameter weight class (useful for leader boards) if not yet determined
        if self.metadata.size_label is None and total_params > 0:
                    self.metadata.size_label = gguf.size_label(total_params, shared_params, expert_params, expert_count)

        # Extract the encoding scheme from the file type name. e.g. 'gguf.LlamaFileType.MOSTLY_Q8_0' --> 'Q8_0'
        output_type: str = self.ftype.name.partition("_")[2]

        # Filename Output
        if self.fname_out.is_dir():
            # Generate default filename based on model specification and available metadata
            if not vocab_only:
                        fname_default: str = gguf.naming_convention(self.metadata.name, self.metadata.basename, self.metadata.finetune, self.metadata.version, self.metadata.size_label, output_type, model_type="LoRA" if total_params < 0 else None)
            else:
                        fname_default: str = gguf.naming_convention(self.metadata.name, self.metadata.basename, self.metadata.finetune, self.metadata.version, size_label=None, output_type=None, model_type="vocab")

            # Use the default filename
            self.fname_out = self.fname_out / f"{fname_default}.gguf"
        else:
            # Output path is a custom defined templated filename
            # Note: `not is_dir()` is used because `.is_file()` will not detect
            #       file template strings as it doesn't actually exist as a file

            # Process templated file name with the output ftype, useful with the "auto" ftype
                   self.fname_out = self.fname_out.parent / gguf.fill_templated_filename(self.fname_out.name, output_type)

        self.set_type()

        logger.info("Set meta model")
        self.metadata.set_gguf_meta_model(self.gguf_writer)

        logger.info("Set model parameters")
        self.set_gguf_parameters()

        logger.info("Set model tokenizer")
        self.set_vocab()

        logger.info("Set model quantization version")
        self.gguf_writer.add_quantization_version(gguf.GGML_QUANT_VERSION)
```

```
def write(self):
    self.prepare_tensors()
    self.prepare_metadata(vocab_only=False)
    self.gguf_writer.write_header_to_file(path=self.fname_out)
    self.gguf_writer.write_kv_data_to_file()
    self.gguf_writer.write_tensors_to_file(progress=True)
    self.gguf_writer.close()

def write_vocab(self):
    if len(self.gguf_writer.tensors) != 1:
        raise ValueError('Splitting the vocabulary is not supported')

    self.prepare_metadata(vocab_only=True)
    self.gguf_writer.write_header_to_file(path=self.fname_out)
    self.gguf_writer.write_kv_data_to_file()
    self.gguf_writer.close()

@staticmethod
def get_model_part_names(dir_model: Path, prefix: str, suffix: str) -> list[str]:
    part_names: list[str] = []
    for filename in os.listdir(dir_model):
        if filename.startswith(prefix) and filename.endswith(suffix):
            part_names.append(filename)

    part_names.sort()

    return part_names

@staticmethod
def load_hparams(dir_model: Path):
    with open(dir_model / "config.json", "r", encoding="utf-8") as f:
        return json.load(f)

@classmethod
def register(cls, *names: str) -> Callable[[AnyModel], AnyModel]:
    assert names

    def func(modelcls: AnyModel) -> AnyModel:
        for name in names:
            cls._model_classes[name] = modelcls
        return modelcls
    return func

@classmethod
def print_registered_models(cls):
    for name in sorted(cls._model_classes.keys()):
        logger.error(f"- {name}")

@classmethod
def from_model_architecture(cls, arch: str) -> type[Model]:
    try:
        return cls._model_classes[arch]
    except KeyError:
```

```
            raise NotImplementedError(f'Architecture {arch!r} not supported!') from None

    def does_token_look_special(self, token: str | bytes) -> bool:
        if isinstance(token, (bytes, bytearray)):
            token_text = token.decode(encoding="utf-8")
        elif isinstance(token, memoryview):
            token_text = token.tobytes().decode(encoding="utf-8")
        else:
            token_text = token

        # Some models mark some added tokens which ought to be control tokens as not special.
        # (e.g. command-r, command-r-plus, deepseek-coder, gemma{,-2})
        seems_special = token_text in (
            "<pad>",  # deepseek-coder
            "<mask>", "<2mass>", "[@BOS@]",  # gemma{,-2}
        )

        seems_special = seems_special or (token_text.startswith("<|") and token_text.endswith("|>"))
            seems_special = seems_special or (token_text.startswith("<?") and token_text.endswith("?>"))  # deepseek-coder

        # TODO: should these be marked as UNUSED instead? (maybe not)
          seems_special = seems_special or (token_text.startswith("<unused") and token_text.endswith(">"))  # gemma{,-2}

        return seems_special

    # used for GPT-2 BPE and WordPiece vocabs
    def get_vocab_base(self) -> tuple[list[str], list[int], str]:
        tokens: list[str] = []
        toktypes: list[int] = []

        from transformers import AutoTokenizer
        tokenizer = AutoTokenizer.from_pretrained(self.dir_model)
        vocab_size = self.hparams.get("vocab_size", len(tokenizer.vocab))
        assert max(tokenizer.vocab.values()) < vocab_size

        tokpre = self.get_vocab_base_pre(tokenizer)

        reverse_vocab = {id_: encoded_tok for encoded_tok, id_ in tokenizer.vocab.items()}
        added_vocab = tokenizer.get_added_vocab()

        added_tokens_decoder = tokenizer.added_tokens_decoder

        for i in range(vocab_size):
            if i not in reverse_vocab:
                tokens.append(f"[PAD{i}]")
                toktypes.append(gguf.TokenType.UNUSED)
            else:
                token: str = reverse_vocab[i]
                if token in added_vocab:
                            # The tokenizer in llama.cpp assumes the CONTROL and USER_DEFINED tokens are pre-normalized.
                    # To avoid unexpected issues - we make sure to normalize non-normalized tokens
```

```
                    if not added_tokens_decoder[i].normalized:
                        previous_token = token
                        token = tokenizer.decode(tokenizer.encode(token, add_special_tokens=False))
                        if previous_token != token:
                            logger.info(f"{repr(previous_token)} is encoded and decoded back to {repr(token)}
using AutoTokenizer")

                    if added_tokens_decoder[i].special or self.does_token_look_special(token):
                        toktypes.append(gguf.TokenType.CONTROL)
                    else:
                        # NOTE: this was added for Gemma.
                        # Encoding and decoding the tokens above isn't sufficient for this case.
                        token = token.replace(b"\xe2\x96\x81".decode("utf-8"), " ")  # pre-normalize
user-defined spaces
                        toktypes.append(gguf.TokenType.USER_DEFINED)
                else:
                    toktypes.append(gguf.TokenType.NORMAL)
                tokens.append(token)

        return tokens, toktypes, tokpre

    # NOTE: this function is generated by convert_hf_to_gguf_update.py
    #       do not modify it manually!
    # ref:  https://github.com/ggml-org/llama.cpp/pull/6920
    # Marker: Start get_vocab_base_pre
    def get_vocab_base_pre(self, tokenizer) -> str:
        # encoding this string and hashing the resulting tokens would (hopefully) give us a unique identifier
that
        # is specific for the BPE pre-tokenizer used by the model
        # we will use this unique identifier to write a "tokenizer.ggml.pre" entry in the GGUF file which we
can
        # use in llama.cpp to implement the same pre-tokenizer

        chktxt = '\n \n\n \n\n\n \t \t\t \t\n  \n   \n    \n     \nLAUNCH (normal) ?\u200d?? (multiple emojis
concatenated)  [OK]  ??  3  33  333  3333  33333  333333  3333333  33333333  3.3  3..3  3...3  ??????????????
????apple??13314151??  ------=======  ????  ??  ?????????  \'\'\'\'\'\`````````"""".....!!!!!!??????  I\'ve  been
\'told he\'s there, \'RE you sure? \'M not sure I\'ll make it, \'D you like some tea? We\'Ve a\'lL'

        chktok = tokenizer.encode(chktxt)
        chkhsh = sha256(str(chktok).encode()).hexdigest()

        logger.debug(f"chktok: {chktok}")
        logger.debug(f"chkhsh: {chkhsh}")

        res = None

        # NOTE: if you get an error here, you need to update the convert_hf_to_gguf_update.py script
        #       or pull the latest version of the model from Huggingface
        #       don't edit the hashes manually!
        if chkhsh == "0ef9807a4087ebef797fc749390439009c3b9eda9ad1a097abbe738f486c01e5":
            # ref: https://huggingface.co/meta-llama/Meta-Llama-3-8B
            res = "llama-bpe"
        if chkhsh == "049ecf7629871e3041641907f3de7c733e4dbfdc736f57d882ba0b0845599754":
            # ref: https://huggingface.co/deepseek-ai/deepseek-llm-7b-base
```

```
        res = "deepseek-llm"
    if chkhsh == "347715f544604f9118bb75ed199f68779f423cabb20db6de6f31b908d04d7821":
        # ref: https://huggingface.co/deepseek-ai/deepseek-coder-6.7b-base
        res = "deepseek-coder"
    if chkhsh == "8aeee3860c56296a157a1fe2fad249ec40aa59b1bb5709f4ade11c4e6fe652ed":
        # ref: https://huggingface.co/tiiuae/falcon-7b
        res = "falcon"
    if chkhsh == "9d032fcbd5501f4a38150912590928bfb36091efb5df11b8e2124b0390e3fb1e":
        # ref: https://huggingface.co/tiiuae/Falcon3-7B-Base
        res = "falcon3"
    if chkhsh == "0876d13b50744004aa9aeae05e7b0647eac9d801b5ba4668afc01e709c15e19f":
        # ref: https://huggingface.co/BAAI/bge-small-en-v1.5
        res = "bert-bge"
    if chkhsh == "8e62295832751ca1e8f92f2226f403dea30dc5165e448b5bfa05af5340c64ec7":
        # ref: https://huggingface.co/BAAI/bge-large-zh-v1.5
        res = "bert-bge-large"
    if chkhsh == "b6dc8df998e1cfbdc4eac8243701a65afe638679230920b50d6f17d81c098166":
        # ref: https://huggingface.co/mosaicml/mpt-7b
        res = "mpt"
    if chkhsh == "35d91631860c815f952d711435f48d356ebac988362536bed955d43bfa436e34":
        # ref: https://huggingface.co/bigcode/starcoder2-3b
        res = "starcoder"
    if chkhsh == "3ce83efda5659b07b1ad37ca97ca5797ea4285d9b9ab0dc679e4a720c9da7454":
        # ref: https://huggingface.co/openai-community/gpt2
        res = "gpt-2"
    if chkhsh == "32d85c31273f8019248f2559fed492d929ea28b17e51d81d3bb36fff23ca72b3":
        # ref: https://huggingface.co/stabilityai/stablelm-2-zephyr-1_6b
        res = "stablelm2"
    if chkhsh == "6221ad2852e85ce96f791f476e0b390cf9b474c9e3d1362f53a24a06dc8220ff":
        # ref: https://huggingface.co/smallcloudai/Refact-1_6-base
        res = "refact"
    if chkhsh == "9c2227e4dd922002fb81bde4fc02b0483ca4f12911410dee2255e4987644e3f8":
        # ref: https://huggingface.co/CohereForAI/c4ai-command-r-v01
        res = "command-r"
    if chkhsh == "e636dc30a262dcc0d8c323492e32ae2b70728f4df7dfe9737d9f920a282b8aea":
        # ref: https://huggingface.co/Qwen/Qwen1.5-7B
        res = "qwen2"
    if chkhsh == "b6dc8df998e1cfbdc4eac8243701a65afe638679230920b50d6f17d81c098166":
        # ref: https://huggingface.co/allenai/OLMo-1.7-7B-hf
        res = "olmo"
    if chkhsh == "a8594e3edff7c29c003940395316294b2c623e09894deebbc65f33f1515df79e":
        # ref: https://huggingface.co/databricks/dbrx-base
        res = "dbrx"
    if chkhsh == "c7699093ba4255a91e702aa38a596aa81669f3525dae06c2953267dde580f448":
        # ref: https://huggingface.co/jinaai/jina-reranker-v1-tiny-en
        res = "jina-v1-en"
    if chkhsh == "0876d13b50744004aa9aeae05e7b0647eac9d801b5ba4668afc01e709c15e19f":
        # ref: https://huggingface.co/jinaai/jina-embeddings-v2-base-en
        res = "jina-v2-en"
    if chkhsh == "171aeeedd6fb548d418a7461d053f11b6f1f1fc9b387bd66640d28a4b9f5c643":
        # ref: https://huggingface.co/jinaai/jina-embeddings-v2-base-es
        res = "jina-v2-es"
    if chkhsh == "27949a2493fc4a9f53f5b9b029c82689cfbe5d3a1929bb25e043089e28466de6":
        # ref: https://huggingface.co/jinaai/jina-embeddings-v2-base-de
```

```
            res = "jina-v2-de"
        if chkhsh == "c136ed14d01c2745d4f60a9596ae66800e2b61fa45643e72436041855ad4089d":
            # ref: https://huggingface.co/abacusai/Smaug-Llama-3-70B-Instruct
            res = "smaug-bpe"
        if chkhsh == "c7ea5862a53e4272c035c8238367063e2b270d51faa48c0f09e9d5b54746c360":
            # ref: https://huggingface.co/LumiOpen/Poro-34B-chat
            res = "poro-chat"
        if chkhsh == "7967bfa498ade6b757b064f31e964dddbb80f8f9a4d68d4ba7998fcf281c531a":
            # ref: https://huggingface.co/jinaai/jina-embeddings-v2-base-code
            res = "jina-v2-code"
        if chkhsh == "b6e8e1518dc4305be2fe39c313ed643381c4da5db34a98f6a04c093f8afbe99b" or chkhsh == "81d72c7348a9f0ebe86f23298d37debe0a5e71149e29bd283904c02262b27516":
            # ref: https://huggingface.co/THUDM/glm-4-9b-chat
            res = "chatglm-bpe"
        if chkhsh == "7fc505bd3104ca1083b150b17d088b59534ede9bde81f0dd2090967d7fe52cee":
            # ref: https://huggingface.co/LumiOpen/Viking-7B
            res = "viking"
        if chkhsh == "b53802fb28e26d645c3a310b34bfe07da813026ec7c7716883404d5e0f8b1901":
            # ref: https://huggingface.co/core42/jais-13b
            res = "jais"
        if chkhsh == "7b3e7548e4308f52a76e8229e4e6cc831195d0d1df43aed21ac6c93da05fec5f":
            # ref: https://huggingface.co/WisdomShell/CodeShell-7B
            res = "codeshell"
        if chkhsh == "63b97e4253352e6f357cc59ea5b583e3a680eaeaf2632188c2b952de2588485e":
            # ref: https://huggingface.co/mistralai/Mistral-Nemo-Base-2407
            res = "tekken"
        if chkhsh == "855059429035d75a914d1eda9f10a876752e281a054a7a3d421ef0533e5b6249":
            # ref: https://huggingface.co/HuggingFaceTB/SmolLM-135M
            res = "smollm"
        if chkhsh == "3c30d3ad1d6b64202cd222813e7736c2db6e1bd6d67197090fc1211fbc612ae7":
            # ref: https://huggingface.co/bigscience/bloom
            res = "bloom"
        if chkhsh == "bc01ce58980e1db43859146dc51b1758b3b88729b217a74792e9f8d43e479d21":
            # ref: https://huggingface.co/TurkuNLP/gpt3-finnish-small
            res = "gpt3-finnish"
        if chkhsh == "4e2b24cc4770243d65a2c9ec19770a72f08cffc161adbb73fcbb6b7dd45a0aae":
            # ref: https://huggingface.co/LGAI-EXAONE/EXAONE-3.0-7.8B-Instruct
            res = "exaone"
        if chkhsh == "fcace8b9cac38ce847670c970cd5892031a753a1ef381abd1d9af00f713da085":
            # ref: https://huggingface.co/microsoft/phi-2
            res = "phi-2"
        if chkhsh == "60824e3c0d9401f89943cbb2fff727f0e2d4c545ba4df2d6e4f09a6db0f5b450":
            # ref: https://huggingface.co/facebook/chameleon-7b
            res = "chameleon"
        if chkhsh == "1431a23e583c97432bc230bff598d103ddb5a1f89960c8f1d1051aaa944d0b35":
            # ref: https://huggingface.co/sapienzanlp/Minerva-7B-base-v1.0
            res = "minerva-7b"
        if chkhsh == "8b5a93ed704057481f240da0be7e7dca721d7f8f4755263b6807227a2cbeae65":
            # ref: https://huggingface.co/sentence-transformers/stsb-roberta-base
            res = "roberta-bpe"
        if chkhsh == "ad851be1dba641f2e3711822f816db2c265f788b37c63b4e1aeacb9ee92de8eb":
            # ref: https://huggingface.co/ai-sage/GigaChat-20B-A3B-instruct
            res = "gigachat"
        if chkhsh == "d4c8f286ea6b520b3d495c4455483cfa2302c0cfcd4be05d781b6a8a0a7cdaf1":
```

```
            # ref: https://huggingface.co/Infinigence/Megrez-3B-Instruct
            res = "megrez"
        if chkhsh == "877081d19cf6996e2c4ff0e1236341e9b7bde288f5311a56a937f0afbbb3aeb5":
            # ref: https://huggingface.co/deepseek-ai/DeepSeek-V3
            res = "deepseek-v3"
        if chkhsh == "b3f499bb4255f8ca19fccd664443283318f2fd2414d5e0b040fbdd0cc195d6c5":
            # ref: https://huggingface.co/deepseek-ai/DeepSeek-R1-Distill-Qwen-1.5B
            res = "deepseek-r1-qwen"
        if chkhsh == "ccc2ef013c104be7bae2965776d611e1d7a8a2a9c547dd93a682c9a9fc80352e":
            # ref: https://huggingface.co/Xenova/gpt-4o
            res = "gpt-4o"
        if chkhsh == "7dec86086fcc38b66b7bc1575a160ae21cf705be7718b9d5598190d7c12db76f":
            # ref: https://huggingface.co/UW/OLMo2-8B-SuperBPE-t180k
            res = "superbpe"
        if chkhsh == "1994ffd01900cfb37395608534236ecd63f2bd5995d6cb1004dda1af50240f15":
            # ref: https://huggingface.co/trillionlabs/Trillion-7B-preview
            res = "trillion"
        if chkhsh == "96a5f08be6259352137b512d4157e333e21df7edd3fcd152990608735a65b224":
            # ref: https://huggingface.co/inclusionAI/Ling-lite
            res = "bailingmoe"
        if chkhsh == "d353350c764d8c3b39c763113960e4fb4919bea5fbf208a0e3b22e8469dc7406":
            # ref: https://huggingface.co/meta-llama/Llama-4-Scout-17B-16E-Instruct
            res = "llama4"
        if chkhsh == "a1336059768a55c99a734006ffb02203cd450fed003e9a71886c88acf24fdbc2":
            # ref: https://huggingface.co/THUDM/glm-4-9b-hf
            res = "glm4"

        if res is None:
            logger.warning("\n")

logger.warning("**********************************************************************************")
            logger.warning("** WARNING: The BPE pre-tokenizer was not recognized!")
            logger.warning("**          There are 2 possible reasons for this:")
            logger.warning("**          - the model has not been added to convert_hf_to_gguf_update.py yet")
            logger.warning("**          - the pre-tokenization config has changed upstream")
            logger.warning("**          Check your model files and convert_hf_to_gguf_update.py and update them
accordingly.")
            logger.warning("** ref:     https://github.com/ggml-org/llama.cpp/pull/6920")
            logger.warning("**")
            logger.warning(f"** chkhsh:  {chkhsh}")

logger.warning("**********************************************************************************")
            logger.warning("\n")
            raise NotImplementedError("BPE pre-tokenizer was not recognized - update get_vocab_base_pre()")

        logger.debug(f"tokenizer.ggml.pre: {repr(res)}")
        logger.debug(f"chkhsh: {chkhsh}")

        return res
        # Marker: End get_vocab_base_pre

    def _set_vocab_none(self) -> None:
        self.gguf_writer.add_tokenizer_model("none")
```

```
def _set_vocab_gpt2(self) -> None:
    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)
    special_vocab.add_to_gguf(self.gguf_writer)

def _set_vocab_qwen(self):
    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["vocab_size"]
    assert max(tokenizer.get_vocab().values()) < vocab_size

    tokpre = self.get_vocab_base_pre(tokenizer)

    merges = []
    vocab = {}
    mergeable_ranks = tokenizer.mergeable_ranks
    for token, rank in mergeable_ranks.items():
        vocab[QwenModel.token_bytes_to_string(token)] = rank
        if len(token) == 1:
            continue
        merged = QwenModel.bpe(mergeable_ranks, token, max_rank=rank)
        assert len(merged) == 2
        merges.append(' '.join(map(QwenModel.token_bytes_to_string, merged)))

    # for this kind of tokenizer, added_vocab is not a subset of vocab, so they need to be combined
    added_vocab = tokenizer.special_tokens
    reverse_vocab = {id_ : encoded_tok for encoded_tok, id_ in {**vocab, **added_vocab}.items()}

    for i in range(vocab_size):
        if i not in reverse_vocab:
            tokens.append(f"[PAD{i}]")
            toktypes.append(gguf.TokenType.UNUSED)
        elif reverse_vocab[i] in added_vocab:
            tokens.append(reverse_vocab[i])
            toktypes.append(gguf.TokenType.CONTROL)
        else:
            tokens.append(reverse_vocab[i])
            toktypes.append(gguf.TokenType.NORMAL)

    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(dir_model, load_merges=False)
    special_vocab.merges = merges
    # only add special tokens when they were not already loaded from config.json
    if len(special_vocab.special_token_ids) == 0:
        special_vocab._set_special_token("bos", tokenizer.special_tokens["<|endoftext|>"])
        special_vocab._set_special_token("eos", tokenizer.special_tokens["<|endoftext|>"])
    # this one is usually not in config.json anyway
    special_vocab._set_special_token("unk", tokenizer.special_tokens["<|endoftext|>"])
    special_vocab.add_to_gguf(self.gguf_writer)

def _set_vocab_sentencepiece(self, add_to_gguf=True):
    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))
    special_vocab.add_to_gguf(self.gguf_writer)

def _create_vocab_sentencepiece(self):
    from sentencepiece import SentencePieceProcessor

    tokenizer_path = self.dir_model / 'tokenizer.model'

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

    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

        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, token_data in added_tokens_decoder.items():
                    token_id = int(token_id)
                    token: str = token_data["content"]
                    if token_id >= vocab_size:
                        logger.warning(f'ignore token {token_id}: id is out of range, max={vocab_size - 1}')
                        continue
                    if toktypes[token_id] != SentencePieceTokenTypes.UNUSED:
                        if tokens[token_id] != token.encode("utf-8"):
                              logger.warning(f'replacing token {token_id}: {tokens[token_id].decode("utf-8")!r}
-> {token!r}')
                    if token_data.get("special") or self.does_token_look_special(token):
                        toktypes[token_id] = SentencePieceTokenTypes.CONTROL
                    else:
                                token = token.replace(b"\xe2\x96\x81".decode("utf-8"), " ")  # pre-normalize
user-defined spaces
                        toktypes[token_id] = SentencePieceTokenTypes.USER_DEFINED

                    scores[token_id] = -1000.0
                    tokens[token_id] = token.encode("utf-8")

        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)

        return tokens, scores, toktypes

    def _set_vocab_llama_hf(self):
```

```
    vocab = gguf.LlamaHfVocab(self.dir_model)
    tokens = []
    scores = []
    toktypes = []

    for text, score, toktype in vocab.all_tokens():
        tokens.append(text)
        scores.append(score)
        toktypes.append(toktype)

    assert len(tokens) == vocab.vocab_size

    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_vocab_rwkv_world(self):
    assert (self.dir_model / "rwkv_vocab_v20230424.txt").is_file()
    vocab_size = self.hparams.get("vocab_size", 65536)

    tokens: list[bytes] = ['<s>'.encode("utf-8")]
    toktypes: list[int] = [gguf.TokenType.CONTROL]

    with open(self.dir_model / "rwkv_vocab_v20230424.txt", "r", encoding="utf-8") as f:
        lines = f.readlines()
        for line in lines:
            parts = line.split(' ')
            assert len(parts) >= 3
            token, token_len = ast.literal_eval(' '.join(parts[1:-1])), int(parts[-1])
            token = token.encode("utf-8") if isinstance(token, str) else token
            assert isinstance(token, bytes)
            assert len(token) == token_len
            token_text: str = repr(token)[2:-1]  # "b'\xff'" -> "\xff"
            tokens.append(token_text.encode("utf-8"))
            toktypes.append(gguf.TokenType.NORMAL)
    remainder = vocab_size - len(tokens)
    assert remainder >= 0
    for i in range(len(tokens), vocab_size):
        tokens.append(f"[PAD{i}]".encode("utf-8"))
        toktypes.append(gguf.TokenType.UNUSED)

    self.gguf_writer.add_tokenizer_model("rwkv")
    self.gguf_writer.add_token_list(tokens)
    self.gguf_writer.add_token_types(toktypes)
    special_vocab = gguf.SpecialVocab(self.dir_model, load_merges=False)
    special_vocab.chat_template = "rwkv-world"
    # hack: Add '\n\n' as the EOT token to make it chat normally
    special_vocab._set_special_token("eot", 261)
    special_vocab.add_to_gguf(self.gguf_writer)
```

```
    def _set_vocab_builtin(self, model_name: Literal["gpt-neox", "llama-spm"], vocab_size: int):
        tokenizer_path = Path(sys.path[0]) / "models" / f"ggml-vocab-{model_name}.gguf"
        logger.warning(f"Using tokenizer from '{os.path.relpath(tokenizer_path, os.getcwd())}'")
        vocab_reader = gguf.GGUFReader(tokenizer_path, "r")

        default_pre = "mpt" if model_name == "gpt-neox" else "default"

        field = vocab_reader.get_field(gguf.Keys.Tokenizer.MODEL)
        assert field  # tokenizer model
        self.gguf_writer.add_tokenizer_model(bytes(field.parts[-1]).decode("utf-8"))

        field = vocab_reader.get_field(gguf.Keys.Tokenizer.PRE)
        self.gguf_writer.add_tokenizer_pre(bytes(field.parts[-1]).decode("utf-8") if field else default_pre)

        field = vocab_reader.get_field(gguf.Keys.Tokenizer.LIST)
        assert field  # token list
        self.gguf_writer.add_token_list([bytes(field.parts[i]) for i in field.data][:vocab_size])

        if model_name == "llama-spm":
            field = vocab_reader.get_field(gguf.Keys.Tokenizer.SCORES)
            assert field  # token scores
            self.gguf_writer.add_token_scores([field.parts[i].tolist()[0] for i in field.data][:vocab_size])

        field = vocab_reader.get_field(gguf.Keys.Tokenizer.TOKEN_TYPE)
        assert field  # token types
        self.gguf_writer.add_token_types([field.parts[i].tolist()[0] for i in field.data][:vocab_size])

        if model_name != "llama-spm":
            field = vocab_reader.get_field(gguf.Keys.Tokenizer.MERGES)
            assert field  # token merges
            self.gguf_writer.add_token_merges([bytes(field.parts[i]) for i in field.data])

        if (field := vocab_reader.get_field(gguf.Keys.Tokenizer.BOS_ID)) is not None:
            self.gguf_writer.add_bos_token_id(field.parts[-1].tolist()[0])
        if (field := vocab_reader.get_field(gguf.Keys.Tokenizer.EOS_ID)) is not None:
            self.gguf_writer.add_eos_token_id(field.parts[-1].tolist()[0])
        if (field := vocab_reader.get_field(gguf.Keys.Tokenizer.UNK_ID)) is not None:
            self.gguf_writer.add_unk_token_id(field.parts[-1].tolist()[0])
        if (field := vocab_reader.get_field(gguf.Keys.Tokenizer.PAD_ID)) is not None:
            self.gguf_writer.add_pad_token_id(field.parts[-1].tolist()[0])
        if (field := vocab_reader.get_field(gguf.Keys.Tokenizer.ADD_BOS)) is not None:
            self.gguf_writer.add_add_bos_token(field.parts[-1].tolist()[0])
        if (field := vocab_reader.get_field(gguf.Keys.Tokenizer.ADD_EOS)) is not None:
            self.gguf_writer.add_add_eos_token(field.parts[-1].tolist()[0])


@Model.register("GPTNeoXForCausalLM")
class GPTNeoXModel(Model):
    model_arch = gguf.MODEL_ARCH.GPTNEOX

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

```
        self.gguf_writer.add_context_length(self.hparams["max_position_embeddings"])
        self.gguf_writer.add_embedding_length(self.hparams["hidden_size"])
        self.gguf_writer.add_block_count(block_count)
        self.gguf_writer.add_feed_forward_length(self.hparams["intermediate_size"])
        self.gguf_writer.add_rope_dimension_count(
                                        int(self.hparams["rotary_pct"]   *   (self.hparams["hidden_size"]   //
self.hparams["num_attention_heads"])),
        )
        self.gguf_writer.add_head_count(self.hparams["num_attention_heads"])
        self.gguf_writer.add_parallel_residual(self.hparams.get("use_parallel_residual", True))
        self.gguf_writer.add_layer_norm_eps(self.hparams["layer_norm_eps"])

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

        n_head = self.hparams.get("n_head", self.hparams.get("num_attention_heads"))
        n_embed = self.hparams.get("hidden_size", self.hparams.get("n_embed"))

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

        if re.match(r"gpt_neox\.layers\.\d+\.attention\.query_key_value\.weight", name):
            # Map bloom-style qkv_linear to gpt-style qkv_linear
                                                                                #        bloom:
https://github.com/huggingface/transformers/blob/main/src/transformers/models/bloom/modeling_bloom.py#L238-L252
 # noqa
                                                                                #        gpt-2:
https://github.com/huggingface/transformers/blob/main/src/transformers/models/gpt2/modeling_gpt2.py#L312      #
noqa
            qkv_weights = data_torch.reshape((n_head, 3, n_embed // n_head, n_embed))
            data_torch = torch.cat(
                (
                    qkv_weights[:, 0, :, :].reshape((-1, n_embed)),
                    qkv_weights[:, 1, :, :].reshape((-1, n_embed)),
                    qkv_weights[:, 2, :, :].reshape((-1, n_embed)),
                ),
                dim=0,
            )
            logger.info("re-format attention.linear_qkv.weight")
        elif re.match(r"gpt_neox\.layers\.\d+\.attention\.query_key_value\.bias", name):
            qkv_bias = data_torch.reshape((n_head, 3, n_embed // n_head))
            data_torch = torch.cat(
                (
                    qkv_bias[:, 0, :].reshape((n_embed,)),
                    qkv_bias[:, 1, :].reshape((n_embed,)),
                    qkv_bias[:, 2, :].reshape((n_embed,)),
                ),
                dim=0,
            )
            logger.info("re-format attention.linear_qkv.bias")

        tensors.append((self.map_tensor_name(name), data_torch))

        return tensors
```

```
@Model.register("BloomForCausalLM", "BloomModel")
class BloomModel(Model):
    model_arch = gguf.MODEL_ARCH.BLOOM

    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"))
        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(4 * n_embed)
        self.gguf_writer.add_block_count(self.hparams["n_layer"])
        self.gguf_writer.add_head_count(n_head)
        self.gguf_writer.add_head_count_kv(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

        n_head = self.hparams.get("n_head", self.hparams.get("num_attention_heads"))
        n_embed = self.hparams.get("hidden_size", self.hparams.get("n_embed"))

        name = re.sub(r'transformer\.', '', name)

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

        if re.match(r"h\.\d+\.self_attention\.query_key_value\.weight", name):
            # Map bloom-style qkv_linear to gpt-style qkv_linear
                                                                                #        bloom:
https://github.com/huggingface/transformers/blob/main/src/transformers/models/bloom/modeling_bloom.py#L238-L252
 # noqa
                                                                                #        gpt-2:
https://github.com/huggingface/transformers/blob/main/src/transformers/models/gpt2/modeling_gpt2.py#L312      #
noqa
            qkv_weights = data_torch.reshape((n_head, 3, n_embed // n_head, n_embed))
            data_torch = torch.cat(
                (
                    qkv_weights[:, 0, :, :].reshape((-1, n_embed)),
                    qkv_weights[:, 1, :, :].reshape((-1, n_embed)),
                    qkv_weights[:, 2, :, :].reshape((-1, n_embed)),
                ),
                dim=0,
            )
            logger.info("re-format attention.linear_qkv.weight")
        elif re.match(r"h\.\d+\.self_attention\.query_key_value\.bias", name):
            qkv_bias = data_torch.reshape((n_head, 3, n_embed // n_head))
            data_torch = torch.cat(
                (
                    qkv_bias[:, 0, :].reshape((n_embed,)),
                    qkv_bias[:, 1, :].reshape((n_embed,)),
                    qkv_bias[:, 2, :].reshape((n_embed,)),
                ),
                dim=0,
```

```
            )
            logger.info("re-format attention.linear_qkv.bias")

        tensors.append((self.map_tensor_name(name), data_torch))

        return tensors


@Model.register("MPTForCausalLM")
class MPTModel(Model):
    model_arch = gguf.MODEL_ARCH.MPT

    def set_vocab(self):
        try:
            self._set_vocab_gpt2()
        except Exception:
            # Fallback for SEA-LION model
            self._set_vocab_sentencepiece()
            self.gguf_writer.add_add_bos_token(False)
            self.gguf_writer.add_pad_token_id(3)
            self.gguf_writer.add_eos_token_id(1)
            self.gguf_writer.add_unk_token_id(0)

    def set_gguf_parameters(self):
        block_count = self.hparams["n_layers"]
        self.gguf_writer.add_context_length(self.hparams["max_seq_len"])
        self.gguf_writer.add_embedding_length(self.hparams["d_model"])
        self.gguf_writer.add_block_count(block_count)
        self.gguf_writer.add_feed_forward_length(4 * self.hparams["d_model"])
        self.gguf_writer.add_head_count(self.hparams["n_heads"])
        if kv_n_heads := self.hparams["attn_config"].get("kv_n_heads"):
            self.gguf_writer.add_head_count_kv(kv_n_heads)
        self.gguf_writer.add_layer_norm_eps(1e-5)
        if self.hparams["attn_config"]["clip_qkv"] is not None:
            self.gguf_writer.add_clamp_kqv(self.hparams["attn_config"]["clip_qkv"])
        if self.hparams["attn_config"]["alibi"]:
            self.gguf_writer.add_max_alibi_bias(self.hparams["attn_config"]["alibi_bias_max"])
        else:
            self.gguf_writer.add_max_alibi_bias(0.0)

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

        if "scales" in name:
            new_name = self.map_tensor_name(name, try_suffixes=(".weight", ".bias", ".scales"))
            new_name = new_name.replace("scales", "act.scales")
        else:
            new_name = self.map_tensor_name(name, try_suffixes=(".weight", ".bias"))

        return [(new_name, data_torch)]


@Model.register("OrionForCausalLM")
class OrionModel(Model):
```

```
    model_arch = gguf.MODEL_ARCH.ORION

    def set_vocab(self):
        self._set_vocab_sentencepiece()

    def set_gguf_parameters(self):
        block_count = self.hparams["num_hidden_layers"]
        head_count = self.hparams["num_attention_heads"]
        head_count_kv = self.hparams.get("num_key_value_heads", head_count)

        ctx_length = 0
        if "max_sequence_length" in self.hparams:
            ctx_length = self.hparams["max_sequence_length"]
        elif "max_position_embeddings" in self.hparams:
            ctx_length = self.hparams["max_position_embeddings"]
        elif "model_max_length" in self.hparams:
            ctx_length = self.hparams["model_max_length"]
        else:
            raise ValueError("gguf: can not find ctx length parameter.")

        self.gguf_writer.add_file_type(self.ftype)
        self.gguf_writer.add_tensor_data_layout("Meta AI original pth")
        self.gguf_writer.add_context_length(ctx_length)
        self.gguf_writer.add_embedding_length(self.hparams["hidden_size"])
        self.gguf_writer.add_block_count(block_count)
        self.gguf_writer.add_feed_forward_length(self.hparams["intermediate_size"])
        self.gguf_writer.add_head_count(head_count)
        self.gguf_writer.add_head_count_kv(head_count_kv)
        # note: config provides rms norm but it is actually layer norm
                                                                                #           ref:
https://huggingface.co/OrionStarAI/Orion-14B-Chat/blob/276a17221ce42beb45f66fac657a41540e71f4f5/modeling_orion.
py#L570-L571
        self.gguf_writer.add_layer_norm_eps(self.hparams["rms_norm_eps"])


@Model.register("BaichuanForCausalLM", "BaiChuanForCausalLM")
class BaichuanModel(Model):
    model_arch = gguf.MODEL_ARCH.BAICHUAN

    def set_vocab(self):
        self._set_vocab_sentencepiece()

    def set_gguf_parameters(self):
        block_count = self.hparams["num_hidden_layers"]
        head_count = self.hparams["num_attention_heads"]
        head_count_kv = self.hparams.get("num_key_value_heads", head_count)

        ctx_length = 0
        if "max_sequence_length" in self.hparams:
            ctx_length = self.hparams["max_sequence_length"]
        elif "max_position_embeddings" in self.hparams:
            ctx_length = self.hparams["max_position_embeddings"]
        elif "model_max_length" in self.hparams:
            ctx_length = self.hparams["model_max_length"]
```

```
        else:
            raise ValueError("gguf: can not find ctx length parameter.")

        self.gguf_writer.add_tensor_data_layout("Meta AI original pth")
        self.gguf_writer.add_context_length(ctx_length)
        self.gguf_writer.add_embedding_length(self.hparams["hidden_size"])
        self.gguf_writer.add_block_count(block_count)
        self.gguf_writer.add_feed_forward_length(self.hparams["intermediate_size"])
                                    self.gguf_writer.add_rope_dimension_count(self.hparams["hidden_size"]    //
self.hparams["num_attention_heads"])
        self.gguf_writer.add_head_count(head_count)
        self.gguf_writer.add_head_count_kv(head_count_kv)
        self.gguf_writer.add_layer_norm_rms_eps(self.hparams["rms_norm_eps"])
        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]]:
        head_count = self.hparams["num_attention_heads"]
        head_count_kv = self.hparams.get("num_key_value_heads", head_count)

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

        if bid is not None and name == f"model.layers.{bid}.self_attn.W_pack.weight":
            logger.info(f"Unpacking and permuting layer {bid}")
            tensors = [
                (self.format_tensor_name(gguf.MODEL_TENSOR.ATTN_Q, bid),
                    self._reverse_hf_permute_part(data_torch, 0, head_count, head_count)),
                (self.format_tensor_name(gguf.MODEL_TENSOR.ATTN_K, bid),
                    self._reverse_hf_permute_part(data_torch, 1, head_count, head_count_kv)),
                (self.format_tensor_name(gguf.MODEL_TENSOR.ATTN_V, bid),
                    self._reverse_hf_part(data_torch, 2)),
            ]
        else:
            tensors = [(self.map_tensor_name(name), data_torch)]

        return tensors

    def _reverse_hf_permute(self, weights: Tensor, n_head: int, n_kv_head: int | None = None) -> Tensor:
        if n_kv_head is not None and n_head != n_kv_head:
            n_head //= n_kv_head

        return (
            weights.reshape(n_head, 2, weights.shape[0] // n_head // 2, *weights.shape[1:])
            .swapaxes(1, 2)
            .reshape(weights.shape)
        )

    def _reverse_hf_permute_part(
        self, weights: Tensor, n_part: int, n_head: int, n_head_kv: int | None = None,
    ) -> Tensor:
```

```
        r = weights.shape[0] // 3
        return self._reverse_hf_permute(weights[r * n_part:r * n_part + r, ...], n_head, n_head_kv)

    def _reverse_hf_part(self, weights: Tensor, n_part: int) -> Tensor:
        r = weights.shape[0] // 3
        return weights[r * n_part:r * n_part + r, ...]


@Model.register("XverseForCausalLM")
class XverseModel(Model):
    model_arch = gguf.MODEL_ARCH.XVERSE

    def set_vocab(self):
        assert (self.dir_model / "tokenizer.json").is_file()
        dir_model = self.dir_model
        hparams = self.hparams

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

        from transformers import AutoTokenizer
        tokenizer = AutoTokenizer.from_pretrained(dir_model)
        vocab_size = hparams.get("vocab_size", len(tokenizer.vocab))
        # Since we are checking the maximum index, we need to ensure it's strictly less than vocab_size,
        # because vocab_size is the count of items, and indexes start at 0.
        max_vocab_index = max(tokenizer.get_vocab().values())
        if max_vocab_index >= vocab_size:
            raise ValueError("Vocabulary size exceeds expected maximum size.")

        reverse_vocab: dict[int, str] = {id_: encoded_tok for encoded_tok, id_ in tokenizer.vocab.items()}
        added_vocab = tokenizer.get_added_vocab()

        for token_id in range(vocab_size):
            token_text = reverse_vocab[token_id].encode('utf-8')
            # replace "\x00" to string with length > 0
            if token_text == b"\x00":
                toktype = gguf.TokenType.BYTE  # special
                token_text = f"<{token_text}>".encode('utf-8')
            elif re.fullmatch(br"<0x[0-9A-Fa-f]{2}>", token_text):
                toktype = gguf.TokenType.BYTE  # special
            elif reverse_vocab[token_id] in added_vocab:
                if tokenizer.added_tokens_decoder[token_id].special:
                    toktype = gguf.TokenType.CONTROL
                else:
                    toktype = gguf.TokenType.USER_DEFINED
            else:
                toktype = gguf.TokenType.NORMAL

            tokens.append(token_text)
            toktypes.append(toktype)

        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_types(toktypes)

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

    def set_gguf_parameters(self):
        block_count = self.hparams["num_hidden_layers"]
        head_count = self.hparams["num_attention_heads"]
        head_count_kv = self.hparams.get("num_key_value_heads", head_count)

        ctx_length = 0
        if "max_sequence_length" in self.hparams:
            ctx_length = self.hparams["max_sequence_length"]
        elif "max_position_embeddings" in self.hparams:
            ctx_length = self.hparams["max_position_embeddings"]
        elif "model_max_length" in self.hparams:
            ctx_length = self.hparams["model_max_length"]
        else:
            raise ValueError("gguf: can not find ctx length parameter.")

        self.gguf_writer.add_tensor_data_layout("Meta AI original pth")
        self.gguf_writer.add_context_length(ctx_length)
        self.gguf_writer.add_embedding_length(self.hparams["hidden_size"])
        self.gguf_writer.add_block_count(block_count)
        self.gguf_writer.add_feed_forward_length(self.hparams["intermediate_size"])
                                    self.gguf_writer.add_rope_dimension_count(self.hparams["hidden_size"]    //
self.hparams["num_attention_heads"])
        self.gguf_writer.add_head_count(head_count)
        self.gguf_writer.add_head_count_kv(head_count_kv)
        self.gguf_writer.add_layer_norm_rms_eps(self.hparams["rms_norm_eps"])
        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]]:
        del bid  # unused

        head_count = self.hparams["num_attention_heads"]
        head_count_kv = self.hparams.get("num_key_value_heads", head_count)

        # HF models permute some of the tensors, so we need to undo that
        if name.endswith("q_proj.weight"):
            data_torch = self._reverse_hf_permute(data_torch, head_count, head_count)
        if name.endswith("k_proj.weight"):
            data_torch = self._reverse_hf_permute(data_torch, head_count, head_count_kv)

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

    def _reverse_hf_permute(self, weights: Tensor, n_head: int, n_kv_head: int | None = None) -> Tensor:
        if n_kv_head is not None and n_head != n_kv_head:
            n_head //= n_kv_head
```

```
        return (
            weights.reshape(n_head, 2, weights.shape[0] // n_head // 2, *weights.shape[1:])
            .swapaxes(1, 2)
            .reshape(weights.shape)
        )


@Model.register("FalconForCausalLM", "RWForCausalLM")
class FalconModel(Model):
    model_arch = gguf.MODEL_ARCH.FALCON

    def set_gguf_parameters(self):
        block_count = self.hparams.get("num_hidden_layers")
        if block_count is None:
            block_count = self.hparams["n_layer"]  # old name

        n_head = self.hparams.get("num_attention_heads")
        if n_head is None:
            n_head = self.hparams["n_head"]  # old name

        n_head_kv = self.hparams.get("num_kv_heads")
        if n_head_kv is None:
            n_head_kv = self.hparams.get("n_head_kv", 1)  # old name

        self.gguf_writer.add_context_length(2048)  # not in config.json
        self.gguf_writer.add_tensor_data_layout("jploski")  # qkv tensor transform
        self.gguf_writer.add_embedding_length(self.hparams["hidden_size"])
        self.gguf_writer.add_feed_forward_length(4 * self.hparams["hidden_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_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

        # QKV tensor transform
        # The original query_key_value tensor contains n_head_kv "kv groups",
        # each consisting of n_head/n_head_kv query weights followed by one key
        # and one value weight (shared by all query heads in the kv group).
        # This layout makes it a big pain to work with in GGML.
        # So we rearrange them here,, so that we have n_head query weights
        # followed by n_head_kv key weights followed by n_head_kv value weights,
        # in contiguous fashion.
        # ref: https://github.com/jploski/ggml/blob/falcon40b/examples/falcon/convert-hf-to-ggml.py

        if "query_key_value" in name:
            n_head = self.find_hparam(["num_attention_heads", "n_head"])
            n_head_kv = self.find_hparam(["num_kv_heads", "n_head_kv"], optional=True) or 1
            head_dim = self.hparams["hidden_size"] // n_head

            qkv = data_torch.view(n_head_kv, n_head // n_head_kv + 2, head_dim, head_dim * n_head)
```

```
            q = qkv[:, :-2].reshape(n_head * head_dim, head_dim * n_head)
            k = qkv[:, [-2]].reshape(n_head_kv * head_dim, head_dim * n_head)
            v = qkv[:, [-1]].reshape(n_head_kv * head_dim, head_dim * n_head)
            data_torch = torch.cat((q, k, v)).reshape_as(data_torch)

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


@Model.register("GPTBigCodeForCausalLM")
class StarCoderModel(Model):
    model_arch = gguf.MODEL_ARCH.STARCODER

    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(1)
        self.gguf_writer.add_layer_norm_eps(self.hparams["layer_norm_epsilon"])
        self.gguf_writer.add_file_type(self.ftype)


@Model.register("GPTRefactForCausalLM")
class RefactModel(Model):
    model_arch = gguf.MODEL_ARCH.REFACT

    def set_vocab(self):
        super().set_vocab()

        # TODO: how to determine special FIM tokens automatically?
        special_vocab = gguf.SpecialVocab(self.dir_model, load_merges=False,
                                          special_token_types = ['prefix', 'suffix', 'middle', 'eot'])
        special_vocab._set_special_token("prefix", 1)
        special_vocab._set_special_token("suffix", 3)
        special_vocab._set_special_token("middle", 2)
        special_vocab.chat_template = None  # do not add it twice
        special_vocab.add_to_gguf(self.gguf_writer)

    def set_gguf_parameters(self):
        hidden_dim = self.hparams["n_embd"]
        inner_dim = 4 * hidden_dim
        hidden_dim = int(2 * inner_dim / 3)
        multiple_of = 256
        ff_dim = multiple_of * ((hidden_dim + multiple_of - 1) // multiple_of)

        block_count = self.hparams["n_layer"]

        # refact uses Alibi. So this is from config.json which might be used by training.
        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(ff_dim)
        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(1)
        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]]:
        hidden_dim = self.hparams["n_embd"]
        inner_dim = 4 * hidden_dim
        hidden_dim = int(2 * inner_dim / 3)
        multiple_of = 256
        ff_dim = multiple_of * ((hidden_dim + multiple_of - 1) // multiple_of)
        n_head = self.hparams["n_head"]
        n_head_kv = 1
        head_dim = self.hparams["n_embd"] // n_head

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

        if bid is not None:
            if name == f"transformer.h.{bid}.attn.kv.weight":
                tensors.append((self.format_tensor_name(gguf.MODEL_TENSOR.ATTN_K, bid), data_torch[:n_head_kv *
head_dim]))
                 tensors.append((self.format_tensor_name(gguf.MODEL_TENSOR.ATTN_V, bid), data_torch[n_head_kv *
head_dim:]))
            elif name == f"transformer.h.{bid}.attn.q.weight":
                tensors.append((self.format_tensor_name(gguf.MODEL_TENSOR.ATTN_Q, bid), data_torch))
            elif name == f"transformer.h.{bid}.mlp.gate_up_proj.weight":
                tensors.append((self.format_tensor_name(gguf.MODEL_TENSOR.FFN_GATE, bid), data_torch[:ff_dim]))
                tensors.append((self.format_tensor_name(gguf.MODEL_TENSOR.FFN_UP, bid), data_torch[ff_dim:]))

        if len(tensors) == 0:
            tensors.append((self.map_tensor_name(name), data_torch))

        return tensors


@Model.register("StableLmForCausalLM", "StableLMEpochForCausalLM", "LlavaStableLMEpochForCausalLM")
class StableLMModel(Model):
    model_arch = gguf.MODEL_ARCH.STABLELM

    def set_vocab(self):
        if (self.dir_model / "tokenizer.json").is_file():
            self._set_vocab_gpt2()
        else:
            # StableLM 2 1.6B used to have a vocab in a similar format to Qwen's vocab
            self._set_vocab_qwen()

    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"])
        rotary_factor = self.find_hparam(["partial_rotary_factor", "rope_pct"])
                  self.gguf_writer.add_rope_dimension_count(int(rotary_factor   *   (hparams["hidden_size"]   //
hparams["num_attention_heads"])))
        self.gguf_writer.add_head_count(hparams["num_attention_heads"])
        self.gguf_writer.add_head_count_kv(hparams["num_key_value_heads"])
          self.gguf_writer.add_parallel_residual(hparams["use_parallel_residual"] if "use_parallel_residual" in
hparams else True)
        self.gguf_writer.add_layer_norm_eps(self.find_hparam(["layer_norm_eps", "norm_eps"]))
        self.gguf_writer.add_file_type(self.ftype)

    _q_norms: list[dict[str, Tensor]] | None = None
    _k_norms: 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["num_key_value_heads"]

        if name.find("q_layernorm.norms") != -1:
            assert bid is not None

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

            self._q_norms[bid][name] = data_torch

            if len(self._q_norms[bid]) >= n_head:
                return self._stack_qk_norm(bid, n_head, self._q_norms[bid], "q_layernorm")
            else:
                return []

        if name.find("k_layernorm.norms") != -1:
            assert bid is not None

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

            self._k_norms[bid][name] = data_torch

            if len(self._k_norms[bid]) >= n_kv_head:
                return self._stack_qk_norm(bid, n_kv_head, self._k_norms[bid], "k_layernorm")
            else:
                return []

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

    def _stack_qk_norm(self, bid: int, n_head: int, norms: dict[str, Tensor], layer_name: str = "q_layernorm"):
        datas: list[Tensor] = []
        # extract the norms in order
        for xid in range(n_head):
            ename = f"model.layers.{bid}.self_attn.{layer_name}.norms.{xid}.weight"
            datas.append(norms[ename])
            del norms[ename]
```

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

        merged_name = f"model.layers.{bid}.self_attn.{layer_name}.weight"
        new_name = self.map_tensor_name(merged_name)

        return [(new_name, data_torch)]

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

        if self._q_norms is not None or self._k_norms is not None:
            # flatten two `list[dict[str, Tensor]]` into a single `list[str]`
            norms = (
                [k for d in self._q_norms for k in d.keys()] if self._q_norms is not None else []
            ) + (
                [k for d in self._k_norms for k in d.keys()] if self._k_norms is not None else []
            )
            if len(norms) > 0:
                raise ValueError(f"Unprocessed norms: {norms}")


@Model.register("LLaMAForCausalLM", "LlamaForCausalLM", "MistralForCausalLM", "MixtralForCausalLM")
class LlamaModel(Model):
    model_arch = gguf.MODEL_ARCH.LLAMA
    undo_permute = True

    def set_vocab(self):
        try:
            self._set_vocab_sentencepiece()
        except FileNotFoundError:
            try:
                self._set_vocab_llama_hf()
            except (FileNotFoundError, TypeError):
                # Llama 3
                self._set_vocab_gpt2()

        # Apply to CodeLlama only (and ignore for Llama 3 with a vocab size of 128256)
        if self.hparams.get("vocab_size", 32000) == 32016:
            special_vocab = gguf.SpecialVocab(
                self.dir_model, load_merges=False,
                special_token_types = ['prefix', 'suffix', 'middle', 'eot']
            )
            special_vocab._set_special_token("prefix", 32007)
            special_vocab._set_special_token("suffix", 32008)
            special_vocab._set_special_token("middle", 32009)
            special_vocab._set_special_token("eot",    32010)
            special_vocab.add_to_gguf(self.gguf_writer)

        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"])
```

```
    # Apply to granite small models only
    if self.hparams.get("vocab_size", 32000) == 49152:
        self.gguf_writer.add_add_bos_token(False)

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":
            self.gguf_writer.add_rope_scaling_type(gguf.RopeScalingType.LINEAR)
            self.gguf_writer.add_rope_scaling_factor(self.hparams["rope_scaling"]["factor"])

@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))

_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 self.undo_permute:
        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)

    # 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 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 # Errors for Llama4

                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))
```

```
    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("Llama4ForConditionalGeneration")
class Llama4Model(LlamaModel):
    model_arch = gguf.MODEL_ARCH.LLAMA4
    has_vision: bool = False
    undo_permute = False

    # TODO @ngxson : avoid duplicate this code everywhere by at least support "text_config"
    # same with llama, but 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
        # IMPORTANT: the normal "intermediate_size" is renamed to "intermediate_size_mlp", we need to undo this
        self.hparams["intermediate_size_moe"] = self.hparams["intermediate_size"]
        self.hparams["intermediate_size"] = self.hparams["intermediate_size_mlp"]

    def set_vocab(self):
        self._set_vocab_gpt2()
        self.gguf_writer.add_add_bos_token(True)

    def set_gguf_parameters(self):
        super().set_gguf_parameters()
        self.gguf_writer.add_interleave_moe_layer_step(self.hparams["interleave_moe_layer_step"])
        self.gguf_writer.add_expert_feed_forward_length(self.hparams["intermediate_size_moe"])

    def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None):
        # split the gate_up into gate and up
        if "gate_up_proj" in name:
            name_up = name.replace("gate_up_proj", "up_proj.weight")
            name_gate = name.replace("gate_up_proj", "gate_proj.weight")
            dim_half = data_torch.shape[-1] // 2
            gate_proj_weight, up_proj_weight = data_torch.transpose(-1, -2).split(dim_half, dim=-2)
            return [
                (self.map_tensor_name(name_gate), gate_proj_weight),
                (self.map_tensor_name(name_up), up_proj_weight)
            ]

        if name.endswith("down_proj"):
```

```
            name += ".weight"
            data_torch = data_torch.transpose(-1, -2)

        if "multi_modal_projector" in name or "vision_model" in name:
            return []
        return super().modify_tensors(data_torch, name, bid)


@Model.register("Mistral3ForConditionalGeneration")
class Mistral3Model(LlamaModel):
    model_arch = gguf.MODEL_ARCH.LLAMA

    # 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)

    def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None):
        name = name.replace("language_model.", "")
        if "multi_modal_projector" in name or "vision_tower" in name:
            return []
        return super().modify_tensors(data_torch, name, bid)


@Model.register("DeciLMForCausalLM")
class DeciModel(Model):
    model_arch = gguf.MODEL_ARCH.DECI

    @staticmethod
    def _ffn_mult_to_intermediate_size(ffn_mult: float, n_embd: int) -> int:
        # DeciLM-specific code
        intermediate_size = int(2 * ffn_mult * n_embd / 3)
        return DeciModel._find_multiple(intermediate_size, 256)

    @staticmethod
    def _find_multiple(n: int, k: int) -> int:
        # DeciLM-specific code
        if n % k == 0:
            return n
        return n + k - (n % k)

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

        if "block_configs" in self.hparams: # Llama-3_1-Nemotron-51B
            _block_configs: list[dict[str,Any]] = self.hparams["block_configs"]
            assert self.block_count == len(_block_configs)
            self._num_kv_heads = list()
            self._num_heads = list()
            _ffn_multipliers = list()
            # ***linear attention layer***
```

```
            # if n_heads_in_group is None and replace_with_linear is True
            # then _num_kv_heads[il] is 0 and _num_heads[il] is num_attention_heads
            # ***attention-free layer***
            # if n_heads_in_group is None and replace_with_linear is False
            # then _num_kv_heads[il] is 0 and _num_heads[il] is 0
            # ***normal attention-layer***
            # if n_heads_in_group is not None, then
            # _num_kv_heads[il] is num_attention_head // n_heads_in_group and
            # _num_heads[il] is num_attention_head
            for il in range(len(_block_configs)):
                if _block_configs[il]["attention"]["n_heads_in_group"] is None:
                    if _block_configs[il]["attention"]["replace_with_linear"] is True:
                        self._num_kv_heads.append(0)
                        self._num_heads.append(self.hparams["num_attention_heads"])
                    else:
                        self._num_kv_heads.append(0)
                        self._num_heads.append(0)
                else:
                                                self._num_kv_heads.append(self.hparams["num_attention_heads"]  //
_block_configs[il]["attention"]["n_heads_in_group"])
                    self._num_heads.append(self.hparams["num_attention_heads"])
                _ffn_multipliers.append(_block_configs[il]["ffn"]["ffn_mult"])
            assert self.block_count == len(self._num_kv_heads)
            assert self.block_count == len(self._num_heads)
            assert self.block_count == len(_ffn_multipliers)
            assert isinstance(self._num_kv_heads, list) and isinstance(self._num_kv_heads[0], int)
            assert isinstance(self._num_heads, list) and isinstance(self._num_heads[0], int)
            assert isinstance(_ffn_multipliers, list) and isinstance(_ffn_multipliers[0], float)
            self._ffn_dims: list[int] = [
                DeciModel._ffn_mult_to_intermediate_size(multiplier, self.hparams["hidden_size"])
                for multiplier in _ffn_multipliers
            ]

    def set_vocab(self):
        # Please change tokenizer_config.json of Llama-3_1-Nemotron-51B's
        # eos_token from '|eot_id|' to '|end_of_text|'
        if self.hparams.get("vocab_size", 128256) == 128256:
            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)
            special_vocab.add_to_gguf(self.gguf_writer)
        else:
            # DeciLM-7B
            self._set_vocab_llama_hf()

    def set_gguf_parameters(self):
        if "block_configs" in self.hparams: # Llama-3_1-Nemotron-51B
            assert self.block_count == len(self._num_kv_heads)
            assert self.block_count == len(self._num_heads)
            assert self.block_count == len(self._ffn_dims)
```

```
            if (rope_theta := self.hparams.get("rope_theta")) is not None:
                self.gguf_writer.add_rope_freq_base(rope_theta)
            self.gguf_writer.add_head_count_kv(self._num_kv_heads)
            self.gguf_writer.add_head_count(self._num_heads)
            self.gguf_writer.add_feed_forward_length(self._ffn_dims)
            self.gguf_writer.add_block_count(self.block_count)
            self.gguf_writer.add_context_length(self.hparams["max_position_embeddings"])
            self.gguf_writer.add_embedding_length(self.hparams["hidden_size"])
            self.gguf_writer.add_layer_norm_rms_eps(self.hparams["rms_norm_eps"])
            self.gguf_writer.add_key_length(self.hparams["hidden_size"] // self.hparams["num_attention_heads"])
                                            self.gguf_writer.add_value_length(self.hparams["hidden_size"]    //
self.hparams["num_attention_heads"])
            self.gguf_writer.add_file_type(self.ftype)
        else: # DeciLM-7B
            super().set_gguf_parameters()
            if "num_key_value_heads_per_layer" in self.hparams: # DeciLM-7B
                self._num_kv_heads: list[int] = self.hparams["num_key_value_heads_per_layer"]
                assert self.block_count == len(self._num_kv_heads)
                self.gguf_writer.add_head_count_kv(self._num_kv_heads)
        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":
                self.gguf_writer.add_rope_scaling_type(gguf.RopeScalingType.LINEAR)
                self.gguf_writer.add_rope_scaling_factor(self.hparams["rope_scaling"]["factor"])

    @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"]
        if bid is not None:
            if "num_key_value_heads_per_layer" in self.hparams:
                n_kv_head = self.hparams["num_key_value_heads_per_layer"][bid]
            elif "block_configs" in self.hparams:
                n_kv_head = self._num_kv_heads[bid]
                n_head = self._num_heads[bid]
            else:
                n_kv_head = self.hparams.get("num_key_value_heads")
        else:
            n_kv_head = self.hparams.get("num_key_value_heads")
```

```
        if name.endswith(("q_proj.weight", "q_proj.bias")):
            data_torch = DeciModel.permute(data_torch, n_head, n_head)
        if name.endswith(("k_proj.weight", "k_proj.bias")):
            data_torch = DeciModel.permute(data_torch, n_head, n_kv_head)
        return [(self.map_tensor_name(name), data_torch)]

    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))

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


@Model.register("BitnetForCausalLM")
class BitnetModel(Model):
    model_arch = gguf.MODEL_ARCH.BITNET

    def set_vocab(self):
        self._set_vocab_sentencepiece()

    def set_gguf_parameters(self):
        super().set_gguf_parameters()
        self.gguf_writer.add_rope_scaling_type(gguf.RopeScalingType.LINEAR)
        self.gguf_writer.add_rope_scaling_factor(1.0)
```

```
    def weight_quant(self, weight: Tensor) -> Tensor:
        dtype = weight.dtype
        weight = weight.float()
        scale = weight.abs().mean().clamp(min=1e-5)
        iscale = 1 / scale
        # TODO: multiply by the scale directly instead of inverting it twice
        # (this is also unnecessarily doubly inverted upstream)
                                                                                #            ref:
https://huggingface.co/1bitLLM/bitnet_b1_58-3B/blob/af89e318d78a70802061246bf037199d2fb97020/utils_quant.py#L10
        result = (weight * iscale).round().clamp(-1, 1) / iscale
        return result.type(dtype)

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

        if any(self.match_model_tensor_name(new_name, key, bid) for key in [
            gguf.MODEL_TENSOR.ATTN_Q,
            gguf.MODEL_TENSOR.ATTN_K,
            gguf.MODEL_TENSOR.ATTN_V,
            gguf.MODEL_TENSOR.ATTN_OUT,
            gguf.MODEL_TENSOR.FFN_UP,
            gguf.MODEL_TENSOR.FFN_DOWN,
            gguf.MODEL_TENSOR.FFN_GATE,
        ]):
            # transform weight into 1/0/-1 (in fp32)
            data_torch = self.weight_quant(data_torch)

        yield (new_name, data_torch)


@Model.register("GrokForCausalLM")
class GrokModel(Model):
    model_arch = gguf.MODEL_ARCH.GROK

    def set_vocab(self):
        self._set_vocab_sentencepiece()

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

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

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

    def modify_tensors(self, data_torch: Tensor, name: str, bid: int | None) -> Iterable[tuple[str, Tensor]]:
        # process the experts separately
        if name.find(".moe.") != -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 ["linear", "linear_1", "linear_v"]:
                    datas: list[Tensor] = []

                    for xid in range(n_experts):
                        ename = f"transformer.decoder_layer.{bid}.moe.{xid}.{wid}.weight"
                        datas.append(self._experts[bid][ename])
                        del self._experts[bid][ename]

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

                    merged_name = f"transformer.decoder_layer.{bid}.moe.{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)]


@Model.register("DbrxForCausalLM")
class DbrxModel(Model):
    model_arch = gguf.MODEL_ARCH.DBRX

    def set_gguf_parameters(self):
        ffn_config = self.hparams["ffn_config"]
        attn_config = self.hparams["attn_config"]
        self.gguf_writer.add_block_count(self.hparams["n_layers"])

        self.gguf_writer.add_context_length(self.hparams["max_seq_len"])
        self.gguf_writer.add_embedding_length(self.hparams["d_model"])
        self.gguf_writer.add_feed_forward_length(ffn_config["ffn_hidden_size"])

        self.gguf_writer.add_head_count(self.hparams["n_heads"])
        self.gguf_writer.add_head_count_kv(attn_config["kv_n_heads"])

        self.gguf_writer.add_rope_freq_base(attn_config["rope_theta"])

        self.gguf_writer.add_clamp_kqv(attn_config["clip_qkv"])

        self.gguf_writer.add_expert_count(ffn_config["moe_num_experts"])
        self.gguf_writer.add_expert_used_count(ffn_config["moe_top_k"])

        self.gguf_writer.add_layer_norm_eps(1e-5)
```

```
        self.gguf_writer.add_file_type(self.ftype)
        logger.info(f"gguf: file type = {self.ftype}")

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

        n_expert = self.hparams["ffn_config"]["moe_num_experts"]
        n_ff = self.hparams["ffn_config"]["ffn_hidden_size"]
        n_embd = self.hparams["d_model"]

        # Specific behavior for experts tensors: suffix .weight, view as 3D and transpose
        # original implementation expects (n_expert, n_ff, n_embd) for all experts weights
        # But llama.cpp moe graph works differently
        # AND the dimensions in ggml are typically in the reverse order of the pytorch dimensions
        # so (n_expert, n_ff, n_embd) in pytorch is {n_embd, n_ff, n_expert} in ggml_tensor
                  exp_tensor_names  =  {"ffn.experts.mlp.w1":  None,               #  LLM_TENSOR_FFN_GATE_EXPS
ggml_tensor->ne{n_embd, n_ff,   n_expert}
                             "ffn.experts.mlp.w2": (0, 2, 1),  # LLM_TENSOR_FFN_DOWN_EXPS ggml_tensor->ne{n_ff,
 n_embd, n_expert}
                                                "ffn.experts.mlp.v1":  None}           #  LLM_TENSOR_FFN_UP_EXPS
ggml_tensor->ne{n_embd, n_ff,   n_expert}
        experts = False

        for exp_tensor_name in exp_tensor_names.keys():
            if name.find(exp_tensor_name) != -1 and name.find(".weight") == -1:
                experts = True
                data_torch = data_torch.view(n_expert, n_ff, n_embd)
                if (permute_tensor := exp_tensor_names[exp_tensor_name]) is not None:
                    data_torch = data_torch.permute(*permute_tensor)
                break

        # map tensor names
        # In MoE models the ffn tensors are typically most of the model weights,
        # and need to be quantizable. Quantize expects tensor names to be suffixed by .weight.
        # Every other model has the weight names ending in .weight,
        # let's assume that is the convention which is not the case for dbrx:
        # https://huggingface.co/databricks/dbrx-instruct/blob/main/model.safetensors.index.json#L15
        new_name = self.map_tensor_name(name if not experts else name + ".weight", try_suffixes=(".weight",))

        return [(new_name, data_torch)]

         def  tensor_force_quant(self,  name:  str,  new_name:  str,  bid:  int  |  None,  n_dims:  int)  ->
gguf.GGMLQuantizationType | bool:
        del name, new_name, bid  # unused

        return n_dims > 1


@Model.register("MiniCPMForCausalLM")
class MiniCPMModel(Model):
    model_arch = gguf.MODEL_ARCH.MINICPM

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

```
        embedding_scale = float(self.hparams["scale_emb"])
        self.gguf_writer.add_embedding_scale(embedding_scale)
        logger.info(f"gguf: (minicpm) embedding_scale = {embedding_scale}")
        residual_scale = self.hparams["scale_depth"] / self.hparams["num_hidden_layers"] ** 0.5
        self.gguf_writer.add_residual_scale(residual_scale)
        logger.info(f"gguf: (minicpm) residual_scale = {residual_scale}")
        logit_scale = self.hparams["hidden_size"] / self.hparams["dim_model_base"]
        self.gguf_writer.add_logit_scale(logit_scale)
        logger.info(f"gguf: (minicpm) logit_scale = {logit_scale}")
        if self.hparams.get("rope_scaling") is not None:
            if self.hparams["rope_scaling"].get("type") == "longrope":
                self.gguf_writer.add_rope_scaling_type(gguf.RopeScalingType.LONGROPE)
                logger.info(f"gguf: (minicpm) rope_scaling_type = {gguf.RopeScalingType.LONGROPE}")

    def generate_extra_tensors(self) -> Iterable[tuple[str, Tensor]]:
        rope_dims = self.hparams["hidden_size"] // self.hparams["num_attention_heads"]

        rope_scaling = self.find_hparam(['rope_scaling'], True)
        if rope_scaling is not None:
            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}')

               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))

    def set_vocab(self):
        self._set_vocab_sentencepiece()

    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")

        # HF models permute some of the tensors, so we need to undo that
        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("MiniCPM3ForCausalLM")
class MiniCPM3Model(Model):
```

```
    model_arch = gguf.MODEL_ARCH.MINICPM3

    def set_gguf_parameters(self):
        hparams = self.hparams

        self.gguf_writer.add_file_type(self.ftype)
        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(self.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(hparams["num_key_value_heads"])
        self.gguf_writer.add_layer_norm_rms_eps(hparams["rms_norm_eps"])
        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_rope_dimension_count(hparams["qk_rope_head_dim"])

    def generate_extra_tensors(self) -> Iterable[tuple[str, Tensor]]:
        rope_scaling = self.find_hparam(['rope_scaling'], True)
        if rope_scaling is not None:
            rope_dims = self.hparams["qk_rope_head_dim"]

            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}')

                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))

    def set_vocab(self):
        self._set_vocab_sentencepiece()

    def _reverse_hf_permute(self, weights: Tensor, n_head: int, n_kv_head: int | None = None) -> Tensor:
        if n_kv_head is not None and n_head != n_kv_head:
            n_head //= n_kv_head

        return (
            weights.reshape(n_head, 2, weights.shape[0] // n_head // 2, *weights.shape[1:])
            .swapaxes(1, 2)
            .reshape(weights.shape)
        )
```

```
@Model.register("QWenLMHeadModel")
class QwenModel(Model):
    model_arch = gguf.MODEL_ARCH.QWEN

    @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):
        self._set_vocab_qwen()

    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["rotary_emb_base"])
                                    self.gguf_writer.add_rope_dimension_count(self.hparams["hidden_size"]     //
self.hparams["num_attention_heads"])
        self.gguf_writer.add_head_count(self.hparams["num_attention_heads"])
        self.gguf_writer.add_layer_norm_rms_eps(self.hparams["layer_norm_epsilon"])
        self.gguf_writer.add_file_type(self.ftype)


@Model.register("Qwen2ForCausalLM")
class Qwen2Model(Model):
    model_arch = gguf.MODEL_ARCH.QWEN2

    def set_vocab(self):
        try:
            self._set_vocab_sentencepiece()
        except FileNotFoundError:
            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("Qwen2VLForConditionalGeneration", "Qwen2_5_VLForConditionalGeneration")
class Qwen2VLModel(Model):
    model_arch = gguf.MODEL_ARCH.QWEN2VL

    def set_gguf_parameters(self):
        super().set_gguf_parameters()
        mrope_section = self.hparams["rope_scaling"]["mrope_section"]
        mrope_section += [0] * max(0, 4 - len(mrope_section))
        self.gguf_writer.add_rope_dimension_sections(mrope_section)

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

    def get_tensors(self) -> Iterator[tuple[str, Tensor]]:
        for name, data in super().get_tensors():
            if name.startswith("visual."):
                continue
            yield name, data


@Model.register("WavTokenizerDec")
class WavTokenizerDecModel(Model):
    model_arch = gguf.MODEL_ARCH.WAVTOKENIZER_DEC

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

        if \
                name.endswith("codebook.cluster_size") or \
                name.endswith("codebook.embed_avg") or \
                name.endswith("codebook.inited"):
            logger.debug(f"Skipping {name!r}")
            return []

        logger.info(f"{self.map_tensor_name(name)} -> {data_torch.shape}")

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

    def set_vocab(self):
        self._set_vocab_none()
```

```
    def set_gguf_parameters(self):
        super().set_gguf_parameters()
        self.gguf_writer.add_vocab_size         (self.hparams["vocab_size"])
        self.gguf_writer.add_features_length    (self.hparams["n_embd_features"])
        self.gguf_writer.add_feed_forward_length(self.hparams["n_ff"])
        self.gguf_writer.add_group_norm_eps     (self.hparams["group_norm_epsilon"])
        self.gguf_writer.add_group_norm_groups  (self.hparams["group_norm_groups"])

        self.gguf_writer.add_posnet_embedding_length(self.hparams["posnet"]["n_embd"])
        self.gguf_writer.add_posnet_block_count     (self.hparams["posnet"]["n_layer"])

        self.gguf_writer.add_convnext_embedding_length(self.hparams["convnext"]["n_embd"])
        self.gguf_writer.add_convnext_block_count     (self.hparams["convnext"]["n_layer"])

        self.gguf_writer.add_causal_attention(False)


@Model.register("Qwen2MoeForCausalLM")
class Qwen2MoeModel(Model):
    model_arch = gguf.MODEL_ARCH.QWEN2MOE

    def set_gguf_parameters(self):
        super().set_gguf_parameters()
        if (n_experts := self.hparams.get("num_experts")) is not None:
            self.gguf_writer.add_expert_count(n_experts)
        if (moe_intermediate_size := self.hparams.get("moe_intermediate_size")) is not None:
            self.gguf_writer.add_expert_feed_forward_length(moe_intermediate_size)
            logger.info(f"gguf: expert feed forward length = {moe_intermediate_size}")
           if (shared_expert_intermediate_size := self.hparams.get('shared_expert_intermediate_size')) is not
None:
            self.gguf_writer.add_expert_shared_feed_forward_length(shared_expert_intermediate_size)
            logger.info(f"gguf: expert shared feed forward length = {shared_expert_intermediate_size}")

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

    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)]

    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("Qwen3ForCausalLM")
class Qwen3Model(Qwen2Model):
    model_arch = gguf.MODEL_ARCH.QWEN3


@Model.register("Qwen3MoeForCausalLM")
class Qwen3MoeModel(Qwen2MoeModel):
    model_arch = gguf.MODEL_ARCH.QWEN3MOE


@Model.register("GPT2LMHeadModel")
class GPT2Model(Model):
    model_arch = gguf.MODEL_ARCH.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_ctx"])
        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_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", ".attn.masked_bias")):
            return tensors

        if name.endswith((".c_attn.weight", ".c_proj.weight", ".c_fc.weight", ".c_proj.weight")):
            data_torch = data_torch.transpose(1, 0)

        new_name = self.map_tensor_name(name)

        tensors.append((new_name, data_torch))

        return tensors


@Model.register("PhiForCausalLM")
class Phi2Model(Model):
    model_arch = gguf.MODEL_ARCH.PHI2

    def set_gguf_parameters(self):
        block_count = self.find_hparam(["num_hidden_layers", "n_layer"])

        rot_pct = self.find_hparam(["partial_rotary_factor"])
        n_embd = self.find_hparam(["hidden_size", "n_embd"])
        n_head = self.find_hparam(["num_attention_heads", "n_head"])

        self.gguf_writer.add_context_length(self.find_hparam(["n_positions", "max_position_embeddings"]))

        self.gguf_writer.add_embedding_length(n_embd)
        self.gguf_writer.add_feed_forward_length(4 * n_embd)
        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)
        self.gguf_writer.add_layer_norm_eps(self.find_hparam(["layer_norm_epsilon", "layer_norm_eps"]))
        self.gguf_writer.add_rope_dimension_count(int(rot_pct * n_embd) // n_head)
        self.gguf_writer.add_file_type(self.ftype)
        self.gguf_writer.add_add_bos_token(False)


@Model.register("Phi3ForCausalLM")
class Phi3MiniModel(Model):
    model_arch = gguf.MODEL_ARCH.PHI3

    def set_vocab(self):
        # Phi-4 model uses GPT2Tokenizer
        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)
                tokenizer_class = tokenizer_config_json['tokenizer_class']
                if tokenizer_class == 'GPT2Tokenizer':
                    return self._set_vocab_gpt2()

        from sentencepiece import SentencePieceProcessor
```

```
tokenizer_path = self.dir_model / 'tokenizer.model'

if not tokenizer_path.is_file():
    raise ValueError(f'Error: Missing {tokenizer_path}')

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.debug(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

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"].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"].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)

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

    def set_gguf_parameters(self):
        block_count = self.find_hparam(["num_hidden_layers", "n_layer"])

        n_embd = self.find_hparam(["hidden_size", "n_embd"])
        n_head = self.find_hparam(["num_attention_heads", "n_head"])
        n_head_kv = self.find_hparam(["num_key_value_heads", "n_head_kv"])
        rms_eps = self.find_hparam(["rms_norm_eps"])
        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

        self.gguf_writer.add_context_length(max_pos_embds)
```