Qui è contenuta la documentazione e i tutti i codici della libreria **TimesFM**.
I file appartenenti alla directory **src** sono tutti i file che compongono TimesFM.
Alla fine ci sono degli esempi di finetuning e di test.

TimesFM (Time Series Foundation Model) is a pretrained time-series foundation model developed by Google Research for time-series forecasting.

timesfm-1.0-200m is the first open model checkpoint:

- It performs univariate time series forecasting for context lengths up to 512 timepoints and any horizon lengths, with an optional frequency indicator.
- It focuses on point forecasts, and does not support probabilistic forecasts. We experimentally offer quantile heads but they have not been calibrated after pretraining.
- It requires the context to be contiguous (i.e. no "holes"), and the context and the horizon to be of the same frequency.

Usage
Initialize the model and load a checkpoint.
Then the base class can be loaded as,

```
import timesfm

tfm = timesfm.TimesFm(
    context_len=<context>,
    horizon_len=<horizon>,
    input_patch_len=32,
    output_patch_len=128,
    num_layers=20,
    model_dims=1280,
    backend=<backend>, #'cpu' or 'gpu'
)
tfm.load_from_checkpoint(repo_id="google/timesfm-1.0-200m")
```

Note that the four parameters are fixed to load the 200m model

```
input_patch_len=32,
output_patch_len=128,
num_layers=20,
model_dims=1280,
```

The context_len here can be set as the max context length of the model. You can provide a shorter series to the tfm.forecast() function and the model will handle it. Currently, the model handles a max context length of 512, which can be increased in later releases. The input time series can have any context length. Padding / truncation will be handled by the inference code if needed.

The horizon length can be set to anything. We recommend setting it to the largest horizon length you would need in the forecasting tasks for your application. We generally recommend horizon length <= context length but it is not a requirement in the function call.

Perform inference

We provide APIs to forecast from either array inputs or pandas dataframe. Both forecast methods expect (1) the input time series contexts, (2) along with their frequencies. Please look at the documentation of the functions tfm.forecast() and tfm.forecast_on_df() for detailed instructions.

In particular, regarding the frequency, TimesFM expects a categorical indicator valued in {0, 1, 2}:

0 (default): high frequency, long horizon time series. We recommend using this for time series up to daily granularity.
1: medium frequency time series. We recommend using this for weekly and monthly data.
2: low frequency, short horizon time series. We recommend using this for anything beyond monthly, e.g. quarterly or yearly.
This categorical value should be directly provided with the array inputs. For dataframe inputs, we convert the conventional letter coding of frequencies to our expected categories, that

0: T, MIN, H, D, B, U
1: W, M
2: Q, Y
Notice you do NOT have to strictly follow our recommendation here. Although this is our setup during model training and we expect it to offer the best forecast result, you can also view the frequency input as a free parameter and modify it per your specific use case.

Examples:

Array inputs, with the frequencies set to low, medium, and high respectively.

```
import numpy as np
forecast_input = [
    np.sin(np.linspace(0, 20, 100))
    np.sin(np.linspace(0, 20, 200)),
    np.sin(np.linspace(0, 20, 400)),
]
frequency_input = [0, 1, 2]

point_forecast, experimental_quantile_forecast = tfm.forecast(
    forecast_input,
    freq=frequency_input,
)
```

pandas dataframe, with the frequency set to "M" monthly.

```
import pandas as pd

# e.g. input_df is
#       unique_id  ds          y
# 0     T1         1975-12-31  697458.0
# 1     T1         1976-01-31  1187650.0
# 2     T1         1976-02-29  1069690.0
# 3     T1         1976-03-31  1078430.0
# 4     T1         1976-04-30  1059910.0
# ...   ...        ...         ...
# 8175  T99        1986-01-31  602.0
# 8176  T99        1986-02-28  684.0
# 8177  T99        1986-03-31  818.0
# 8178  T99        1986-04-30  836.0
# 8179  T99        1986-05-31  878.0

forecast_df = tfm.forecast_on_df(
    inputs=input_df,
    freq="M",  # monthly
    value_name="y",
    num_jobs=-1,
)
```

1) src/adapter/__init__.py

```
"""adapter init file."""

from .dora_layers import DoraAttentionProjection, DoraCombinedQKVProjection, DoraLinear
from .lora_layers import LoraAttentionProjection, LoraCombinedQKVProjection, LoraLinear
```

2) src/adapter/dora_layers.py

```
# Copyright 2024 The Google Research Authors.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

from jax import numpy as jnp
from praxis import base_layer
from praxis.layers import attentions, linears

WeightInit = base_layer.WeightInit
WeightHParams = base_layer.WeightHParams


class DoraTheta(base_layer.Theta):
    def __init__(self, module):
        self.module = module

    def _dora_initialized(self):
        if (
            self.module.has_variable("params", "lora_a")
            and self.module.has_variable("params", "lora_b")
            and self.module.has_variable("params", "dora_m")
            and "lora_a" in self.module._weight_hparams
            and "lora_b" in self.module._weight_hparams
            and "dora_m" in self.module._weight_hparams
        ):
            return True
        else:
            return False
```

```
  def _dorafy_var(self, w):
    lora_a = super().__getattr__("lora_a")
    lora_b = super().__getattr__("lora_b")
    dora_m = super().__getattr__("dora_m")

    lora_delta = self.module.einsum("...dr,...nr->...dn", lora_a, lora_b)
    lora_delta = jnp.reshape(lora_delta, w.shape)

    w_prime = w + lora_delta

    column_norm = jnp.linalg.norm(w_prime, ord=2, axis=0, keepdims=True)
    norm_adapted = w_prime / column_norm
    w_prime = dora_m * norm_adapted
    return w_prime

  def __getattr__(self, k):
    var = super().__getattr__(k)
    if not self._dora_initialized():
      return var

    if k == "w":
      return self._dorafy_var(var)

    return var

  def __getitem__(self, k):
    var = super().__getattr__(k)
    if not self._dora_initialized():
      return var

    if k == "w":
      return self._dorafy_var(var)

    return var


class DoraThetaDescriptor:
  """Dot syntax accession descriptor."""

  def __get__(self, obj, objtype=None):
    return DoraTheta(obj)


class DoraLinear(linears.Linear):
  rank: int = 0
  lora_init: WeightInit | None = None
  theta = DoraThetaDescriptor()
```

```
  def setup(self) -> None:
    lora_init = self.lora_init if self.lora_init else self.weight_init

    super().setup()
    self.create_variable(
      "lora_a",
      WeightHParams(
        shape=[self.input_dims, self.rank],
        init=lora_init,
        mesh_shape=self.mesh_shape,
        tensor_split_dims_mapping=[None, None],
      ),
    )
    self.create_variable(
      "lora_b",
      WeightHParams(
        shape=[self.output_dims, self.rank],
        init=WeightInit.Constant(scale=0.0),
        mesh_shape=self.mesh_shape,
        tensor_split_dims_mapping=[None, None],
      ),
    )
    self.create_variable(
      "dora_m",
      WeightHParams(
        shape=[1, self.output_dims],
        init=lora_init,
        mesh_shape=self.mesh_shape,
        tensor_split_dims_mapping=[None, None],
      ),
    )


class DoraAttentionProjection(attentions.AttentionProjection):
  rank: int = 0
  lora_init: WeightInit | None = None
  theta = DoraThetaDescriptor()

  def setup(self) -> None:
    super().setup()
    w_weight_params = self._weight_hparams["w"]
    lora_init = self.lora_init if self.lora_init else w_weight_params.init

    self.create_variable(
      "lora_a",
      WeightHParams(
        shape=[self.input_dim, self.rank],
```

```
            init=lora_init,
            mesh_shape=self.mesh_shape,
            tensor_split_dims_mapping=[
                None,
                None,
            ],
        ),
    )
    self.create_variable(
        "lora_b",
        WeightHParams(
            shape=[self.dim_per_head * self.num_heads, self.rank],
            init=WeightInit.Constant(scale=0.0),
            mesh_shape=self.mesh_shape,
            tensor_split_dims_mapping=[
                None,
                None,
            ],
        ),
    )
    self.create_variable(
        "dora_m",
        WeightHParams(
            shape=[1, self.num_heads, self.dim_per_head],
            init=lora_init,
            mesh_shape=self.mesh_shape,
            tensor_split_dims_mapping=[None, None, None],
        ),
    )


class DoraCombinedQKVProjection(attentions.CombinedQKVProjectionLayer):
  rank: int = 0
  lora_init: WeightInit | None = None
  theta = DoraThetaDescriptor()

  def setup(self) -> None:
    super().setup()
    w_weight_params = self._weight_hparams["w"]
    lora_init = self.lora_init if self.lora_init else w_weight_params.init

    self.create_variable(
        "lora_a",
        WeightHParams(
            shape=[3, self.input_dim, self.rank],
            init=lora_init,
            mesh_shape=self.mesh_shape,
            tensor_split_dims_mapping=[None, None, None],
```

```
    ),
)
self.create_variable(
    "lora_b",
    WeightHParams(
        shape=[3, self.dim_per_head * self.num_heads, self.rank],
        init=WeightInit.Constant(scale=0.0),
        mesh_shape=self.mesh_shape,
        tensor_split_dims_mapping=[None, None, None],
    ),
)
self.create_variable(
    "dora_m",
    WeightHParams(
        shape=[3, 1, self.num_heads, self.dim_per_head],
        init=lora_init,
        mesh_shape=self.mesh_shape,
        tensor_split_dims_mapping=[None, None, None, None],
    ),
)
```

3) src/adapter/lora_layers.py

```
# Copyright 2024 The Google Research Authors.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

from jax import numpy as jnp
from praxis import base_layer
from praxis.layers import attentions, linears

WeightInit = base_layer.WeightInit
WeightHParams = base_layer.WeightHParams


class LoraTheta(base_layer.Theta):
  def __init__(self, module):
    self.module = module

  def _lora_initialized(self):
    if (
      self.module.has_variable("params", "lora_a")
      and self.module.has_variable("params", "lora_b")
      and "lora_a" in self.module._weight_hparams
      and "lora_b" in self.module._weight_hparams
    ):
      return True
    else:
      return False

  def _lorafy_var(self, w):
    lora_a = super().__getattr__("lora_a")
    lora_b = super().__getattr__("lora_b")
    lora_delta = self.module.einsum("...dr,...nr->...dn", lora_a, lora_b)
    lora_delta = jnp.reshape(lora_delta, w.shape)
    w_prime = w + lora_delta
    return w_prime
```

```
    def __getattr__(self, k):
        var = super().__getattr__(k)
        if not self._lora_initialized():
            return var

        if k == "w":
            return self._lorafy_var(var)

        return var

    def __getitem__(self, k):
        var = super().__getattr__(k)
        if not self._lora_initialized():
            return var

        if k == "w":
            return self._lorafy_var(var)

        return var


class LoraThetaDescriptor:
    """Dot syntax accession descriptor."""

    def __get__(self, obj, objtype=None):
        return LoraTheta(obj)


class LoraLinear(linears.Linear):
    rank: int = 0
    lora_init: WeightInit | None = None
    theta = LoraThetaDescriptor()

    def setup(self) -> None:
        lora_init = self.lora_init if self.lora_init else self.weight_init

        super().setup()
        self.create_variable(
            "lora_a",
            WeightHParams(
                shape=[self.input_dims, self.rank],
                init=lora_init,
                mesh_shape=self.mesh_shape,
                tensor_split_dims_mapping=[None, None],
            ),
        )
        self.create_variable(
            "lora_b",
```

```
            WeightHParams(
                shape=[self.output_dims, self.rank],
                init=WeightInit.Constant(scale=0.0),
                mesh_shape=self.mesh_shape,
                tensor_split_dims_mapping=[None, None],
            ),
        )


class LoraAttentionProjection(attentions.AttentionProjection):
    rank: int = 0
    lora_init: WeightInit | None = None
    theta = LoraThetaDescriptor()

    def setup(self) -> None:
        super().setup()
        w_weight_params = self._weight_hparams["w"]
        lora_init = self.lora_init if self.lora_init else w_weight_params.init

        self.create_variable(
            "lora_a",
            WeightHParams(
                shape=[self.input_dim, self.rank],
                init=lora_init,
                mesh_shape=self.mesh_shape,
                tensor_split_dims_mapping=[
                    None,
                    None,
                ],
            ),
        )
        self.create_variable(
            "lora_b",
            WeightHParams(
                shape=[self.dim_per_head * self.num_heads, self.rank],
                init=WeightInit.Constant(scale=0.0),
                mesh_shape=self.mesh_shape,
                tensor_split_dims_mapping=[
                    None,
                    None,
                ],
            ),
        )


class LoraCombinedQKVProjection(attentions.CombinedQKVProjectionLayer):
    rank: int = 0
    lora_init: WeightInit | None = None
```

```
theta = LoraThetaDescriptor()

def setup(self) -> None:
    super().setup()
    w_weight_params = self._weight_hparams["w"]
    lora_init = self.lora_init if self.lora_init else w_weight_params.init

    self.create_variable(
        "lora_a",
        WeightHParams(
            shape=[3, self.input_dim, self.rank],
            init=lora_init,
            mesh_shape=self.mesh_shape,
            tensor_split_dims_mapping=[None, None, None],
        ),
    )
    self.create_variable(
        "lora_b",
        WeightHParams(
            shape=[3, self.dim_per_head * self.num_heads, self.rank],
            init=WeightInit.Constant(scale=0.0),
            mesh_shape=self.mesh_shape,
            tensor_split_dims_mapping=[None, None, None],
        ),
    )
```

4) src/adapter/utils.py

```
# Copyright 2024 The Google Research Authors.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""
This file provides functionality for loading and merging adapter weights
in timesfm model, specifically for LoRA and DoRA.
LoRA: https://arxiv.org/abs/2106.09685
DoRA: https://arxiv.org/abs/2402.09353v4
"""

import time

import jax
import jax.numpy as jnp
from paxml import checkpoints, tasks_lib
from paxml.train_states import TrainState
from praxis import pax_fiddle

from adapter.dora_layers import (
    DoraAttentionProjection,
    DoraCombinedQKVProjection,
    DoraLinear,
)
from adapter.lora_layers import (
    LoraAttentionProjection,
    LoraCombinedQKVProjection,
    LoraLinear,
)
from timesfm import TimesFm


def get_adapter_params(
    params: dict, lora_target_modules: str, num_layers: int, use_dora: bool = False
) -> dict:
    """
```

```
Extracts adapter parameters from the given model parameters for saving the checkpoint.

Args:
   params (dict): The full model parameters.
   lora_target_modules (str): Target modules for LoRA/DoRA adaptation.
   num_layers (int): Number of transformer layers.
   use_dora (bool, optional): Whether DoRA was used or not. Defaults to False.

Returns:
   dict: A dictionary containing the extracted adapter parameters.
"""
adapter_params = {}
for i in range(num_layers):
   layer_key = f"x_layers_{i}"
   adapter_params[layer_key] = {}

   if lora_target_modules in ["all", "mlp"]:
      for ff_layer_key in ["ffn_layer1", "ffn_layer2"]:
         linear = params["params"]["core_layer"]["stacked_transformer_layer"][
            layer_key
         ]["ff_layer"][ff_layer_key]["linear"]

         lora_a = linear["lora_a"]
         lora_b = linear["lora_b"]

         adapter_params[layer_key][ff_layer_key] = {
            "lora_a": lora_a,
            "lora_b": lora_b,
         }

         if use_dora:
            adapter_params[layer_key][ff_layer_key]["dora_m"] = linear["dora_m"]

   if lora_target_modules in ["all", "attention"]:
      attention = params["params"]["core_layer"]["stacked_transformer_layer"][
         layer_key
      ]["self_attention"]

      for component in ["key", "query", "value", "post"]:
         lora_a = attention[component]["lora_a"]
         lora_b = attention[component]["lora_b"]

         adapter_params[layer_key][component] = {
            "lora_a": lora_a,
            "lora_b": lora_b,
         }

         if use_dora:
```

```
                adapter_params[layer_key][component]["dora_m"] = attention[
                    component
                ]["dora_m"]
    return adapter_params


def load_adapter_checkpoint(
    model: TimesFm,
    adapter_checkpoint_path: str,
    lora_rank: int,
    lora_target_modules: str,
    use_dora: bool,
) -> None:
    """
    Loads an adapter checkpoint and merges it with the original model weights.

    Args:
        model (TimesFm): The model to update.
        adapter_checkpoint_path (str): Path to the adapter checkpoint.
        lora_rank (int): Rank of the LoRA adaptation.
        lora_target_modules (str): Target modules for adaptation.
        use_dora (bool): Whether DoRA was used or not.

    Returns:
        None
    """

    """
    currently loading and initializing the model with adapter layers first and then merging the
    adapter weights to original weights and replacing the adapter layers back to original layer.
    # NOTE: refactor this. there should be a better way to load the LoRA checkpoint.
    """
    model._logging(f"Restoring adapter checkpoint from {adapter_checkpoint_path}.")
    start_time = time.time()
    original_linear_tpl, original_attn_tpl, original_combined_qkv_tpl = (
        load_adapter_layer(
            mdl_vars=model._train_state.mdl_vars,
            model=model._model,
            lora_rank=lora_rank,
            lora_target_modules=lora_target_modules,
            use_dora=use_dora,
        )
    )

    var_weight_hparams = model._model.abstract_init_with_metadata(
        model._get_sample_inputs(), do_eval=True
    )
```

```
  adapter_weight_hparams = _get_adapter_weight_params(
      var_weight_hparams=var_weight_hparams,
      lora_target_modules=lora_target_modules,
      num_layers=model._model.stacked_transformer_params_tpl.num_layers,
      use_dora=use_dora,
  )

  adapter_state_partition_specs = tasks_lib.create_state_partition_specs(
      adapter_weight_hparams,
      mesh_shape=model.mesh_shape,
      mesh_axis_names=model.mesh_name,
      discard_opt_states=True,
      learners=None,
  )
  adapter_state_local_shapes = tasks_lib.create_state_unpadded_shapes(
      adapter_weight_hparams,
      discard_opt_states=True,
      learners=None,
  )
  adapter_train_state = checkpoints.restore_checkpoint(
      state_global_shapes=adapter_state_local_shapes,
      checkpoint_dir=adapter_checkpoint_path,
      checkpoint_type=checkpoints.CheckpointType.FLAX,
      state_specs=adapter_state_partition_specs,
      step=None,
  )

  # add adapter weights to the original weights
  _merge_adapter_weights(
      model=model,
      adapter_train_state=adapter_train_state,
      lora_target_modules=lora_target_modules,
      num_layers=model._model.stacked_transformer_params_tpl.num_layers,
      use_dora=use_dora,
  )

  # replace back with the original model layer
  if lora_target_modules in ["all", "mlp"]:

model._model.stacked_transformer_params_tpl.transformer_layer_params_tpl.tr_fflayer_tpl.
fflayer_tpl.linear_tpl = (
        original_linear_tpl
    )

  if lora_target_modules in ["all", "attention"]:

model._model.stacked_transformer_params_tpl.transformer_layer_params_tpl.tr_atten_tpl.p
roj_tpl = (
```

```
            original_attn_tpl
        )

model._model.stacked_transformer_params_tpl.transformer_layer_params_tpl.tr_atten_tpl.c
ombined_qkv_proj_tpl = (
            original_combined_qkv_tpl
        )
    model._logging(
        f"Restored adapter checkpoint in {time.time() - start_time:.2f} seconds."
    )

    # jit compile the model
    model.jit_decode()


def _merge_adapter_weights(
    model: TimesFm,
    adapter_train_state: TrainState,
    lora_target_modules: str,
    num_layers: int,
    use_dora: bool,
) -> None:
    """
    Merges adapter weights with the original model weights.

    Args:
        model (TimesFm): The model to update.
        adapter_train_state (TrainState): The adapter's train state.
        lora_target_modules (str): Target modules for adaptation.
        num_layers (int): Number of transformer layers.
        use_dora (bool): Whether DoRA was used or not.
    """
    for i in range(num_layers):
        layer_key = f"x_layers_{i}"

        if lora_target_modules in ["all", "mlp"]:
            for ff_layer_key in ["ffn_layer1", "ffn_layer2"]:
                linear = model._train_state.mdl_vars["params"][
                    "stacked_transformer_layer"
                ][layer_key]["ff_layer"][ff_layer_key]["linear"]

                params = adapter_train_state.mdl_vars[layer_key][ff_layer_key]
                lora_a = params["lora_a"]
                lora_b = params["lora_b"]

                w = linear["w"]

                lora_delta = jnp.einsum("...dr,...nr->...dn", lora_a, lora_b)
```

```
        lora_delta = jnp.reshape(lora_delta, w.shape)
        w_prime = w + lora_delta

        if use_dora:
            dora_m = params["dora_m"]
            column_norm = jnp.linalg.norm(w_prime, ord=2, axis=0, keepdims=True)
            norm_adapted = w_prime / column_norm
            w_prime = dora_m * norm_adapted
            linear["w"] = w_prime
            del linear["dora_m"]

        else:
            linear["w"] = w_prime

        del linear["lora_a"]
        del linear["lora_b"]

if lora_target_modules in ["all", "attention"]:
    attention = model._train_state.mdl_vars["params"][
        "stacked_transformer_layer"
    ][layer_key]["self_attention"]

    for component in ["key", "query", "value", "post"]:
        params = adapter_train_state.mdl_vars[layer_key][component]
        lora_a = params["lora_a"]
        lora_b = params["lora_b"]

        w = attention[component]["w"]

        lora_delta = jnp.einsum("...dr,...nr->...dn", lora_a, lora_b)
        lora_delta = jnp.reshape(lora_delta, w.shape)
        w_prime = w + lora_delta

        if use_dora:
            dora_m = params["dora_m"]
            column_norm = jnp.linalg.norm(w_prime, ord=2, axis=0, keepdims=True)
            norm_adapted = w_prime / column_norm
            w_prime = dora_m * norm_adapted
            attention[component]["w"] = w_prime
            del attention[component]["dora_m"]

        else:
            attention[component]["w"] = w_prime

        del attention[component]["lora_a"]
        del attention[component]["lora_b"]
```

```python
def _get_adapter_weight_params(
    var_weight_hparams: dict, lora_target_modules: str, num_layers: int, use_dora: bool
) -> dict:
    """
    Extracts adapter weight parameters from the given variable weight hyperparameters.

    Args:
        var_weight_hparams (dict): Variable weight hyperparameters.
        lora_target_modules (str): Target modules for adaptation.
        num_layers (int): Number of transformer layers.
        use_dora (bool): Whether DoRA was used or not.

    Returns:
        dict: A dictionary containing the extracted adapter weight parameters.
    """
    adapter_params = {}
    for i in range(num_layers):
        layer = f"x_layers_{i}"
        adapter_params[layer] = {}

        if lora_target_modules in ["all", "mlp"]:
            for ff_layer_key in ["ffn_layer1", "ffn_layer2"]:
                adapter_weight_params = var_weight_hparams["params"][
                    "stacked_transformer_layer"
                ][layer]["ff_layer"][ff_layer_key]["linear"]
                adapter_params[layer][ff_layer_key] = {
                    "lora_a": adapter_weight_params["lora_a"],
                    "lora_b": adapter_weight_params["lora_b"],
                }

                if use_dora:
                    adapter_params[layer][ff_layer_key]["dora_m"] = (
                        adapter_weight_params["dora_m"]
                    )

        if lora_target_modules in ["all", "attention"]:
            for component in ["key", "value", "query", "post"]:
                adapter_weight_params = var_weight_hparams["params"][
                    "stacked_transformer_layer"
                ][layer]["self_attention"][component]
                adapter_params[layer][component] = {
                    "lora_a": adapter_weight_params["lora_a"],
                    "lora_b": adapter_weight_params["lora_b"],
                }

                if use_dora:
                    adapter_params[layer][component]["dora_m"] = adapter_weight_params[
                        "dora_m"
```

```
            ]

    return adapter_params


def load_adapter_layer(
    mdl_vars: dict,
    model: pax_fiddle.Config,
    lora_rank: int,
    lora_target_modules: str,
    use_dora: bool = False,
) -> tuple[pax_fiddle.Config, pax_fiddle.Config]:
    """
    Updates target modules with adapter layers.

    Args:
        mdl_vars (dict): Model variables.
        model (pax_fiddle.Config): Model configuration.
        lora_rank (int): Rank of the LoRA adaptation.
        lora_target_modules (str): Target modules for adaptation.
        use_dora (bool, optional): Whether DoRA was used or not.

    Returns:
        tuple[pax_fiddle.Config, pax_fiddle.Config]: Updated model configurations.
    """
    original_linear_tpl = original_attn_tpl = original_combined_qkv_tpl = None
    if lora_target_modules in ["all", "mlp"]:
        original_linear_tpl = (

model.stacked_transformer_params_tpl.transformer_layer_params_tpl.tr_fflayer_tpl.fflayer_t
pl.linear_tpl
        )
        adapter_linear_tpl = (
            pax_fiddle.Config(
                DoraLinear,
                rank=lora_rank,
            )
            if use_dora
            else pax_fiddle.Config(
                LoraLinear,
                rank=lora_rank,
            )
        )
        adapter_linear_tpl.copy_fields_from(original_linear_tpl)

model.stacked_transformer_params_tpl.transformer_layer_params_tpl.tr_fflayer_tpl.fflayer_t
pl.linear_tpl = (
            adapter_linear_tpl
```

```
    )

  if lora_target_modules in ["all", "attention"]:
    original_attn_tpl = (

model.stacked_transformer_params_tpl.transformer_layer_params_tpl.tr_atten_tpl.proj_tpl
    )

    adapter_attn_tpl = (
      pax_fiddle.Config(DoraAttentionProjection, rank=lora_rank)
      if use_dora
      else pax_fiddle.Config(LoraAttentionProjection, rank=lora_rank)
    )
    adapter_attn_tpl.copy_fields_from(original_attn_tpl)

    original_combined_qkv_tpl = (

model.stacked_transformer_params_tpl.transformer_layer_params_tpl.tr_atten_tpl.combine
d_qkv_proj_tpl
    )

    adapter_combined_qkv_tpl = (
      pax_fiddle.Config(DoraCombinedQKVProjection, rank=lora_rank)
      if use_dora
      else pax_fiddle.Config(LoraCombinedQKVProjection, rank=lora_rank)
    )
    adapter_combined_qkv_tpl.copy_fields_from(original_combined_qkv_tpl)


model.stacked_transformer_params_tpl.transformer_layer_params_tpl.tr_atten_tpl.proj_tpl =
(
      adapter_attn_tpl
    )

model.stacked_transformer_params_tpl.transformer_layer_params_tpl.tr_atten_tpl.combine
d_qkv_proj_tpl = (
      adapter_combined_qkv_tpl
    )

  # initialize and add adapter weights
  _initialize_adapter_params(
    mdl_vars=mdl_vars,
    num_layers=model.stacked_transformer_params_tpl.num_layers,
    lora_rank=lora_rank,
    lora_target_modules=lora_target_modules,
    use_dora=use_dora,
  )
```

```
    return original_linear_tpl, original_attn_tpl, original_combined_qkv_tpl


def _initialize_adapter_params(
    mdl_vars: dict,
    num_layers,
    lora_rank: int,
    lora_target_modules: str,
    use_dora: bool = False,
    seed: int = 1234,
) -> dict:
    """
    Initializes and adds adapter parameters to target modules.

    Args:
        mdl_vars (dict): Model variables.
        num_layers (int): Number of transformer layers.
        lora_rank (int): Rank of the LoRA adaptation.
        lora_target_modules (str): Target modules for adaptation.
        use_dora (bool, optional): Whether DoRA was used or not.
        seed (int, optional): Random seed for initialization. Defaults to 1234.

    Returns:
        dict: Updated model variables with initialized adapter parameters.
    """
    for i in range(num_layers):
        layer_key = f"x_layers_{i}"
        if lora_target_modules in ["all", "mlp"]:
            for ff_layer_key in ["ffn_layer1", "ffn_layer2"]:
                linear = mdl_vars["params"]["stacked_transformer_layer"][layer_key][
                    "ff_layer"
                ][ff_layer_key]["linear"]
                original_w = linear["w"]
                input_dim, output_dim = original_w.shape
                std_dev = 1 / jnp.sqrt(lora_rank)

                normal_initializer = jax.nn.initializers.normal(std_dev)
                lora_a = normal_initializer(
                    jax.random.key(seed), (input_dim, lora_rank), jnp.float32
                )
                lora_b = jnp.zeros((output_dim, lora_rank))

                linear["lora_a"] = lora_a
                linear["lora_b"] = lora_b

                if use_dora:
                    norm = jnp.linalg.norm(original_w, ord=2, axis=0, keepdims=True)
                    linear["dora_m"] = norm
```

```
    if lora_target_modules in ["all", "attention"]:
        attention = mdl_vars["params"]["stacked_transformer_layer"][layer_key][
            "self_attention"
        ]

        for component in ["key", "query", "value", "post"]:
            original_w = attention[component]["w"]
            w_dim = original_w.shape[0]
            std_dev = 1 / jnp.sqrt(lora_rank)

            normal_initializer = jax.nn.initializers.normal(std_dev)
            lora_a = normal_initializer(
                jax.random.key(seed), (w_dim, lora_rank), jnp.float32
            )
            lora_b = jnp.zeros((w_dim, lora_rank))

            attention[component]["lora_a"] = lora_a
            attention[component]["lora_b"] = lora_b

            if use_dora:
                norm = jnp.linalg.norm(
                    original_w, ord=2, axis=0, keepdims=True
                ).astype(jnp.float32)
                attention[component]["dora_m"] = norm
return mdl_vars
```

5) src/timesfm/__init__.py

```
# Copyright 2024 Google LLC
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
"""TimesFM init file."""
print(
    "TimesFM v1.2.0. See
https://github.com/google-research/timesfm/blob/master/README.md for updated APIs."
)
from timesfm.timesfm_base import freq_map, TimesFmCheckpoint, TimesFmHparams,
TimesFmBase
try:
  print("Loaded Jax TimesFM.")
  from timesfm.timesfm_jax import TimesFmJax as TimesFm
  from timesfm import data_loader
except Exception as _:
  print("Loaded PyTorch TimesFM.")
  from timesfm.timesfm_torch import TimesFmTorch as TimesFm
```

6) src/timesfm/data_loader.py

```
# Copyright 2024 The Google Research Authors.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
"""TF dataloaders for general timeseries datasets.

The expected input format is csv file with a datetime index.
"""

from absl import logging
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
import tensorflow as tf
from . import time_features


class TimeSeriesdata(object):
  """Data loader class."""

  def __init__(
      self,
      data_path,
      datetime_col,
      num_cov_cols,
      cat_cov_cols,
      ts_cols,
      train_range,
      val_range,
      test_range,
      hist_len,
      pred_len,
      batch_size,
      freq='H',
      normalize=True,
      epoch_len=None,
      holiday=False,
```

```
    permute=True,
):
  """Initialize objects.

  Args:
    data_path: path to csv file
    datetime_col: column name for datetime col
    num_cov_cols: list of numerical global covariates
    cat_cov_cols: list of categorical global covariates
    ts_cols: columns corresponding to ts
    train_range: tuple of train ranges
    val_range: tuple of validation ranges
    test_range: tuple of test ranges
    hist_len: historical context
    pred_len: prediction length
    batch_size: batch size (number of ts in a batch)
    freq: freq of original data
    normalize: std. normalize data or not
    epoch_len: num iters in an epoch
    holiday: use holiday features or not
    permute: permute ts in train batches or not

  Returns:
    None
  """
  self.data_df = pd.read_csv(open(data_path, 'r'))
  if not num_cov_cols:
    self.data_df['ncol'] = np.zeros(self.data_df.shape[0])
    num_cov_cols = ['ncol']
  if not cat_cov_cols:
    self.data_df['ccol'] = np.zeros(self.data_df.shape[0])
    cat_cov_cols = ['ccol']
  self.data_df.fillna(0, inplace=True)
  self.data_df.set_index(pd.DatetimeIndex(self.data_df[datetime_col]),
                         inplace=True)
  self.num_cov_cols = num_cov_cols
  self.cat_cov_cols = cat_cov_cols
  self.ts_cols = ts_cols
  self.train_range = train_range
  self.val_range = val_range
  self.test_range = test_range
  data_df_idx = self.data_df.index
  date_index = data_df_idx.union(
      pd.date_range(
          data_df_idx[-1] + pd.Timedelta(1, freq=freq),
          periods=pred_len + 1,
          freq=freq,
      ))
```

```
    self.time_df = time_features.TimeCovariates(
        date_index, holiday=holiday).get_covariates()
    self.hist_len = hist_len
    self.pred_len = pred_len
    self.batch_size = batch_size
    self.freq = freq
    self.normalize = normalize
    self.data_mat = self.data_df[self.ts_cols].to_numpy().transpose()
    self.data_mat = self.data_mat[:, 0:self.test_range[1]]
    self.time_mat = self.time_df.to_numpy().transpose()
    self.num_feat_mat = self.data_df[num_cov_cols].to_numpy().transpose()
    self.cat_feat_mat, self.cat_sizes = self._get_cat_cols(cat_cov_cols)
    self.normalize = normalize
    if normalize:
      self._normalize_data()
    logging.info(
        'Data Shapes: %s, %s, %s, %s',
        self.data_mat.shape,
        self.time_mat.shape,
        self.num_feat_mat.shape,
        self.cat_feat_mat.shape,
    )
    self.epoch_len = epoch_len
    self.permute = permute

  def _get_cat_cols(self, cat_cov_cols):
    """Get categorical columns."""
    cat_vars = []
    cat_sizes = []
    for col in cat_cov_cols:
      dct = {x: i for i, x in enumerate(self.data_df[col].unique())}
      cat_sizes.append(len(dct))
      mapped = self.data_df[col].map(lambda x: dct[x]).to_numpy().transpose()  # pylint:
disable=cell-var-from-loop
      cat_vars.append(mapped)
    return np.vstack(cat_vars), cat_sizes

  def _normalize_data(self):
    self.scaler = StandardScaler()
    train_mat = self.data_mat[:, 0:self.train_range[1]]
    self.scaler = self.scaler.fit(train_mat.transpose())
    self.data_mat = self.scaler.transform(self.data_mat.transpose()).transpose()

  def train_gen(self):
    """Generator for training data."""
    num_ts = len(self.ts_cols)
    perm = np.arange(
        self.train_range[0] + self.hist_len,
```

```
      self.train_range[1] - self.pred_len,
    )
    perm = np.random.permutation(perm)
    hist_len = self.hist_len
    logging.info('Hist len: %s', hist_len)
    if not self.epoch_len:
      epoch_len = len(perm)
    else:
      epoch_len = self.epoch_len
    for idx in perm[0:epoch_len]:
      for _ in range(num_ts // self.batch_size + 1):
        if self.permute:
          tsidx = np.random.choice(num_ts, size=self.batch_size, replace=False)
        else:
          tsidx = np.arange(num_ts)
        dtimes = np.arange(idx - hist_len, idx + self.pred_len)
        (
            bts_train,
            bts_pred,
            bfeats_train,
            bfeats_pred,
            bcf_train,
            bcf_pred,
        ) = self._get_features_and_ts(dtimes, tsidx, hist_len)

        all_data = [
            bts_train,
            bfeats_train,
            bcf_train,
            bts_pred,
            bfeats_pred,
            bcf_pred,
            tsidx,
        ]
        yield tuple(all_data)

  def test_val_gen(self, mode='val', shift=1):
    """Generator for validation/test data."""
    if mode == 'val':
      start = self.val_range[0]
      end = self.val_range[1] - self.pred_len + 1
    elif mode == 'test':
      start = self.test_range[0]
      end = self.test_range[1] - self.pred_len + 1
    else:
      raise NotImplementedError('Eval mode not implemented')
    num_ts = len(self.ts_cols)
    hist_len = self.hist_len
```

```
  logging.info('Hist len: %s', hist_len)
  perm = np.arange(start, end)
  if self.epoch_len:
    epoch_len = self.epoch_len
  else:
    epoch_len = len(perm)
  for i in range(0, epoch_len, shift):
    idx = perm[i]
    for batch_idx in range(0, num_ts, self.batch_size):
      tsidx = np.arange(batch_idx, min(batch_idx + self.batch_size, num_ts))
      dtimes = np.arange(idx - hist_len, idx + self.pred_len)
      (
          bts_train,
          bts_pred,
          bfeats_train,
          bfeats_pred,
          bcf_train,
          bcf_pred,
      ) = self._get_features_and_ts(dtimes, tsidx, hist_len)
      all_data = [
          bts_train,
          bfeats_train,
          bcf_train,
          bts_pred,
          bfeats_pred,
          bcf_pred,
          tsidx,
      ]
      yield tuple(all_data)

def _get_features_and_ts(self, dtimes, tsidx, hist_len=None):
  """Get features and ts in specified windows."""
  if hist_len is None:
    hist_len = self.hist_len
  data_times = dtimes[dtimes < self.data_mat.shape[1]]
  bdata = self.data_mat[:, data_times]
  bts = bdata[tsidx, :]
  bnf = self.num_feat_mat[:, data_times]
  bcf = self.cat_feat_mat[:, data_times]
  btf = self.time_mat[:, dtimes]
  if bnf.shape[1] < btf.shape[1]:
    rem_len = btf.shape[1] - bnf.shape[1]
    rem_rep = np.repeat(bnf[:, [-1]], repeats=rem_len)
    rem_rep_cat = np.repeat(bcf[:, [-1]], repeats=rem_len)
    bnf = np.hstack([bnf, rem_rep.reshape(bnf.shape[0], -1)])
    bcf = np.hstack([bcf, rem_rep_cat.reshape(bcf.shape[0], -1)])
  bfeats = np.vstack([btf, bnf])
  bts_train = bts[:, 0:hist_len]
```

```
  bts_pred = bts[:, hist_len:]
  bfeats_train = bfeats[:, 0:hist_len]
  bfeats_pred = bfeats[:, hist_len:]
  bcf_train = bcf[:, 0:hist_len]
  bcf_pred = bcf[:, hist_len:]
  return bts_train, bts_pred, bfeats_train, bfeats_pred, bcf_train, bcf_pred

def tf_dataset(self, mode='train', shift=1):
  """Tensorflow Dataset."""
  if mode == 'train':
    gen_fn = self.train_gen
  else:
    gen_fn = lambda: self.test_val_gen(mode, shift)
  output_types = tuple([tf.float32] * 2 + [tf.int32] + [tf.float32] * 2 +
                       [tf.int32] * 2)
  dataset = tf.data.Dataset.from_generator(gen_fn, output_types)
  dataset = dataset.prefetch(tf.data.experimental.AUTOTUNE)
  return dataset
```

7) src/timesfm/patched_decoder.py

```
# Copyright 2024 Google LLC
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
"""Pax ML model for patched time-series decoder.

The file implements Residual MLPs, Patched Decoder layers and PAX ML models.
"""

import dataclasses
from typing import Optional, Tuple

import einshape as es
from jax import lax
import jax.numpy as jnp
from praxis import base_layer
from praxis import base_model
from praxis import layers
from praxis import pax_fiddle
from praxis import py_utils
from praxis import pytypes
from praxis.layers import activations
from praxis.layers import embedding_softmax
from praxis.layers import linears
from praxis.layers import normalizations
from praxis.layers import stochastics
from praxis.layers import transformers

# PAX shortcuts
NestedMap = py_utils.NestedMap
JTensor = pytypes.JTensor

LayerTpl = pax_fiddle.Config[base_layer.BaseLayer]
template_field = base_layer.template_field

PAD_VAL = 1123581321.0
DEFAULT_QUANTILES = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
```

```
# NestedMap keys
_INPUT_TS = "input_ts"
_TARGET_FUTURE = "actual_ts"
_INPUT_PADDING = "input_padding"
_OUTPUT_TS = "output_ts"
_FREQ = "freq"
_OUTPUT_TOKENS = "output_tokens"
_STATS = "stats"

# Small numerical value.
_TOLERANCE = 1e-7


def _shift_padded_seq(mask: JTensor, seq: JTensor) -> JTensor:
  """Shifts rows of seq based on the first 0 in each row of the mask."""
  num = seq.shape[1]

  # Find the index of the first 0 in each row of the mask
  first_zero_idx = jnp.argmin(mask, axis=1)

  # Create a range array for indexing
  idx_range = jnp.arange(num)

  def shift_row(carry, x):
    seq_row, shift = x
    shifted_idx = (idx_range - shift) % num
    shifted_row = seq_row[shifted_idx]
    return carry, shifted_row

  # Use lax.scan to shift each row of seq based on the corresponding
  # first_zero_idx.
  _, shifted_seq = lax.scan(shift_row, None, (seq, first_zero_idx))

  return shifted_seq


class ResidualBlock(base_layer.BaseLayer):
  """Simple feedforward block with residual connection.

  Attributes:
    input_dims: input dimension.
    hidden_dims: hidden dimension.
    output_dims: output dimension.
    dropout_prob: dropout probability.
    layer_norm: whether to use layer norm or not.
    dropout_tpl: config for dropout.
    ln_tpl: config for layer norm.
```

```
  act_tpl: config for activation in hidden layer.
"""

input_dims: int = 0
hidden_dims: int = 0
output_dims: int = 0
dropout_prob: float = 0.0
layer_norm: bool = False
dropout_tpl: LayerTpl = template_field(stochastics.Dropout)
ln_tpl: LayerTpl = template_field(normalizations.LayerNorm)
act_tpl: LayerTpl = template_field(activations.Swish)

def setup(self):
  lnorm_tpl = self.ln_tpl.clone()
  lnorm_tpl.dim = self.output_dims
  self.create_child("ln_layer", lnorm_tpl)

  dropout_tpl = self.dropout_tpl.clone()
  dropout_tpl.keep_prob = 1.0 - self.dropout_prob
  self.create_child("dropout", dropout_tpl)

  self.create_child(
      "hidden_layer",
      pax_fiddle.Config(
          linears.FeedForward,
          input_dims=self.input_dims,
          output_dims=self.hidden_dims,
          activation_tpl=self.act_tpl.clone(),
      ),
  )

  self.create_child(
      "output_layer",
      pax_fiddle.Config(
          linears.FeedForward,
          input_dims=self.hidden_dims,
          output_dims=self.output_dims,
          activation_tpl=pax_fiddle.Config(activations.Identity),
      ),
  )

  self.create_child(
      "residual_layer",
      pax_fiddle.Config(
          linears.FeedForward,
          input_dims=self.input_dims,
          output_dims=self.output_dims,
          activation_tpl=pax_fiddle.Config(activations.Identity),
```

```
      ),
    )

  def __call__(self, inputs: JTensor) -> JTensor:
    hidden = self.hidden_layer(inputs)
    output = self.output_layer(hidden)
    output = self.dropout(output)
    residual = self.residual_layer(inputs)
    if self.layer_norm:
      return self.ln_layer(output + residual)
    else:
      return output + residual


def _masked_mean_std(inputs: JTensor,
                     padding: JTensor) -> Tuple[JTensor, JTensor]:
  """Calculates mean and standard deviation of arr across axis 1.

  It should exclude values where pad is 1.

  Args:
    inputs: A JAX array of shape [b, n, p].
    padding: A JAX array of shape [b, n, p] with values 0 or 1.

  Returns:
    A tuple containing the mean and standard deviation of arr. We return the
    statistics of the first patch with more than three non-padded values.
  """
  # Selecting the first pad with more than 3 unpadded values.
  pad_sum = jnp.sum(1 - padding, axis=2)

  def _get_patch_index(arr: JTensor):
    indices = jnp.argmax(arr >= 3, axis=1)
    row_sum = (arr >= 3).sum(axis=1)
    return jnp.where(row_sum == 0, arr.shape[1] - 1, indices)

  patch_indices = _get_patch_index(pad_sum)
  bidxs = jnp.arange(inputs.shape[0])

  arr = inputs[bidxs, patch_indices, :]
  pad = padding[bidxs, patch_indices, :]

  # Create a mask where P is 0
  mask = 1 - pad

  # Calculate the number of valid elements
  num_valid_elements = jnp.sum(mask, axis=1)
```

```python
  num_valid_elements = jnp.where(num_valid_elements == 0, 1, num_valid_elements)

  # Calculate the masked sum and squared sum of M
  masked_sum = jnp.sum(arr * mask, axis=1)
  masked_squared_sum = jnp.sum((arr * mask)**2, axis=1)

  # Calculate the masked mean and standard deviation
  masked_mean = masked_sum / num_valid_elements
  masked_var = masked_squared_sum / num_valid_elements - masked_mean**2
  masked_var = jnp.where(masked_var < 0.0, 0.0, masked_var)
  masked_std = jnp.sqrt(masked_var)

  return masked_mean, masked_std


def _create_quantiles() -> list[float]:
  """Returns the quantiles for forecasting."""
  return DEFAULT_QUANTILES


class PatchedTimeSeriesDecoder(base_layer.BaseLayer):
  """Patch decoder layer for time-series foundation model.

  Attributes:
    patch_len: length of input patches.
    horizon_len: length of output patches. Referred to as `output_patch_len`
      during inference.
    model_dims: model dimension of stacked transformer layer.
    hidden_dims: hidden dimensions in fully connected layers.
    quantiles: list of quantiles for non prob model.
    residual_block_tpl: config for residual block.
    stacked_transformer_params_tpl: config for stacked transformer.
    use_freq: whether to use frequency encoding.

  In all of what followed, except specified otherwise, B is batch size, T is
  sequence length of time-series. N is the number of input patches that can be
  obtained from T. P is the input patch length and H is the horizon length. Q is
  number of output logits. D is model dimension.
  """

  patch_len: int = 0
  horizon_len: int = 0
  model_dims: int = 0
  hidden_dims: int = 0
  quantiles: list[float] = dataclasses.field(default_factory=_create_quantiles)
  residual_block_tpl: LayerTpl = template_field(ResidualBlock)
  stacked_transformer_params_tpl: LayerTpl = template_field(
      transformers.StackedTransformer)
```

```
use_freq: bool = True

def setup(self) -> None:
  """Construct the model."""
  num_outputs = len(self.quantiles) + 1

  stl = self.stacked_transformer_params_tpl.clone()
  stl.model_dims = self.model_dims
  stl.hidden_dims = self.hidden_dims
  stl.mask_self_attention = True

  self.create_child("stacked_transformer_layer", stl)

  input_resl = self.residual_block_tpl.clone()
  ff_in_dims = 2 * self.patch_len
  input_resl.input_dims = ff_in_dims
  input_resl.hidden_dims = self.hidden_dims
  input_resl.output_dims = self.model_dims
  self.create_child(
      "input_ff_layer",
      input_resl,
  )

  horizon_resl = self.residual_block_tpl.clone()
  horizon_resl.input_dims = self.model_dims
  horizon_resl.hidden_dims = self.hidden_dims
  horizon_resl.output_dims = self.horizon_len * num_outputs
  self.create_child(
      "horizon_ff_layer",
      horizon_resl,
  )

  self.create_child(
      "position_emb",
      pax_fiddle.Config(layers.PositionalEmbedding,
                        embedding_dims=self.model_dims),
  )

  if self.use_freq:
    self.create_child(
        "freq_emb",
        pax_fiddle.Config(
            embedding_softmax.Embedding,
            num_classes=3,
            input_dims=self.model_dims,
        ),
    )
```

```
def transform_decode_state(
    self, transform_fn: base_layer.DecodeStateTransformFn) -> None:
  """Transforms all decode state variables based on transform_fn."""
  self.stacked_transformer_layer.transform_decode_state(transform_fn)

def _forward_transform(
    self, inputs: JTensor,
    patched_pads: JTensor) -> Tuple[JTensor, Tuple[JTensor, JTensor]]:
  """Input is of shape [B, N, P]."""
  mu, sigma = _masked_mean_std(inputs, patched_pads)
  sigma = jnp.where(sigma < _TOLERANCE, 1.0, sigma)
  # Normalize each patch.
  outputs = (inputs - mu[:, None, None]) / sigma[:, None, None]
  outputs = jnp.where(
      jnp.abs(inputs - PAD_VAL) < _TOLERANCE, PAD_VAL, outputs)
  return outputs, (mu, sigma)

def _reverse_transform(self, outputs: JTensor,
                       stats: Tuple[JTensor, JTensor]) -> JTensor:
  """Output is of shape [B, N, P, Q]."""
  mu, sigma = stats
  return outputs * sigma[:, None, None, None] + mu[:, None, None, None]

def _preprocess_input(
    self,
    input_ts: JTensor,
    input_padding: JTensor,
    pos_emb: Optional[JTensor] = None,
) -> Tuple[JTensor, JTensor, Optional[Tuple[JTensor, JTensor]], JTensor]:
  """Preprocess input for stacked transformer."""
  # Reshape into patches.
  patched_inputs = es.jax_einshape("b(np)->bnp", input_ts, p=self.patch_len)
  patched_pads = es.jax_einshape("b(np)->bnp",
                                 input_padding,
                                 p=self.patch_len)
  patched_inputs = jnp.where(
      jnp.abs(patched_pads - 1.0) < _TOLERANCE, 0.0, patched_inputs)
  patched_pads = jnp.where(
      jnp.abs(patched_inputs - PAD_VAL) < _TOLERANCE, 1, patched_pads)
  patched_inputs, stats = self._forward_transform(patched_inputs,
                                                  patched_pads)

  # B x N x D
  patched_inputs = patched_inputs * (1.0 - patched_pads)
  concat_inputs = jnp.concatenate([patched_inputs, patched_pads], axis=-1)
  model_input = self.input_ff_layer(concat_inputs)
  # A patch should not be padded even if there is at least one zero.
  patched_padding = jnp.min(patched_pads, axis=-1)
```

```
  if pos_emb is None:
    position_emb = self.position_emb(seq_length=model_input.shape[1])
  else:
    position_emb = pos_emb
  if self.do_eval:
    if position_emb.shape[0] != model_input.shape[0]:
      position_emb = jnp.repeat(position_emb, model_input.shape[0], axis=0)
    position_emb = _shift_padded_seq(patched_padding, position_emb)
  model_input += position_emb

  return model_input, patched_padding, stats, patched_inputs

def _postprocess_output(
    self,
    model_output: JTensor,
    num_outputs: int,
    stats: Tuple[JTensor, JTensor],
) -> JTensor:
  """Postprocess output of stacked transformer."""
  # B x N x (H.Q)
  output_ts = self.horizon_ff_layer(model_output)
  output_ts = es.jax_einshape("bn(hq)->bnhq",
                              output_ts,
                              q=num_outputs,
                              h=self.horizon_len)
  return self._reverse_transform(output_ts, stats)

def __call__(self, inputs: NestedMap) -> NestedMap:
  """PatchTST call.

  Args:
    inputs: A NestedMap containing (1) input_ts: input sequence of shape [B,
      T] where T must be multiple of patch_length; (2) input_padding: that
      contains padding map.

  Returns:
    A nested map with two keys:
    (1) 'output_tokens' of shape [B, N, D].
    (2) 'output_ts' of shape [B, N, H, Q]
    (3) 'stats' a Tuple of statistics for renormalization.
  """
  input_ts, input_padding = inputs[_INPUT_TS], inputs[_INPUT_PADDING]
  num_outputs = len(self.quantiles) + 1
  model_input, patched_padding, stats, _ = self._preprocess_input(
      input_ts=input_ts,
      input_padding=input_padding,
  )
```

```
  if self.use_freq:
    freq = inputs[_FREQ].astype(jnp.int32)
    f_emb = self.freq_emb(freq)  # B x 1 x D
    f_emb = jnp.repeat(f_emb, model_input.shape[1], axis=1)
    model_input += f_emb
  model_output = self.stacked_transformer_layer(model_input, patched_padding)

  output_ts = self._postprocess_output(model_output, num_outputs, stats)
  return NestedMap({
      _OUTPUT_TOKENS: model_output,
      _OUTPUT_TS: output_ts,
      _STATS: stats
  })

def decode(
    self,
    inputs: NestedMap,
    horizon_len: int,
    output_patch_len: Optional[int] = None,
    max_len: int = 512,
    return_forecast_on_context: bool = False,
) -> tuple[JTensor, JTensor]:
  """Auto-regressive decoding without caching.

  Args:
    inputs: input time-series and paddings. Time-series shape B x C, padding
      shape shape B x (C + H) where H is the prediction length.
    horizon_len: prediction length.
    output_patch_len: output length to be fetched from one step of
      auto-regressive decoding.
    max_len: maximum training context length.
    return_forecast_on_context: whether to return the model forecast on the
      context except the first input patch.

  Returns:
    Tuple of two forecasting results:
    - Point (mean) output predictions as a tensor with shape B x H'.
    - Full predictions (mean and quantiles) as a tensor with shape
      B x H' x (1 + # quantiles).
    In particular, if return_forecast_on_context is True, H' is H plus
    the forecastable context length, i.e. context_len - (first) patch_len.
  """
  final_out = inputs[_INPUT_TS]
  context_len = final_out.shape[1]
  paddings = inputs[_INPUT_PADDING]
  if self.use_freq:
    freq = inputs[_FREQ].astype(jnp.int32)
  else:
```

```
    freq = jnp.zeros([final_out.shape[0], 1], dtype=jnp.int32)
  full_outputs = []
  if paddings.shape[1] != final_out.shape[1] + horizon_len:
    raise ValueError(
        "Length of paddings must match length of input + horizon_len:"
        f" {paddings.shape[1]} != {final_out.shape[1]} + {horizon_len}")
  if output_patch_len is None:
    output_patch_len = self.horizon_len
  num_decode_patches = (horizon_len + output_patch_len -
                        1) // output_patch_len
  for step_index in range(num_decode_patches):
    current_padding = paddings[:, 0:final_out.shape[1]]
    input_ts = final_out[:, -max_len:]
    input_padding = current_padding[:, -max_len:]
    model_input = NestedMap(
        input_ts=input_ts,
        input_padding=input_padding,
        freq=freq,
    )
    fprop_outputs = self(model_input)[_OUTPUT_TS]
    if return_forecast_on_context and step_index == 0:
      # For the first decodings step, collect the model forecast on the
      # context except the unavailable first input batch forecast.
      new_full_ts = fprop_outputs[:, :-1, :self.patch_len, :]
      new_full_ts = es.jax_einshape("bnph->b(np)h", new_full_ts)

      full_outputs.append(new_full_ts)

    # (full batch, last patch, output_patch_len, index of mean forecast = 0)
    new_ts = fprop_outputs[:, -1, :output_patch_len, 0]
    new_full_ts = fprop_outputs[:, -1, :output_patch_len, :]
    # (full batch, last patch, output_patch_len, all output indices)
    full_outputs.append(new_full_ts)
    final_out = jnp.concatenate([final_out, new_ts], axis=-1)

  if return_forecast_on_context:
    # `full_outputs` indexing starts at after the first input patch.
    full_outputs = jnp.concatenate(full_outputs,
                                   axis=1)[:, :(context_len - self.patch_len +
                                                horizon_len), :]
  else:
    # `full_outputs` indexing starts at the forecast horizon.
    full_outputs = jnp.concatenate(full_outputs, axis=1)[:, 0:horizon_len, :]

  return (full_outputs[:, :, 0], full_outputs)


class PatchedDecoderFinetuneModel(base_model.BaseModel):
```

```
"""Model class for finetuning patched time-series decoder.

Attributes:
  core_layer_tpl: config for core layer.
  freq: freq to finetune on.
"""

core_layer_tpl: LayerTpl = template_field(PatchedTimeSeriesDecoder)
freq: int = 0

def setup(self) -> None:
  self.create_child("core_layer", self.core_layer_tpl)

def compute_predictions(self, input_batch: NestedMap) -> NestedMap:
  input_ts = input_batch[_INPUT_TS]
  input_padding = jnp.zeros_like(input_ts)
  context_len = input_ts.shape[1]
  input_patch_len = self.core_layer_tpl.patch_len
  context_pad = ((context_len + input_patch_len - 1) //
                 input_patch_len) * input_patch_len - context_len

  input_ts = jnp.pad(input_ts, [(0, 0), (context_pad, 0)])
  input_padding = jnp.pad(input_padding, [(0, 0), (context_pad, 0)],
                          constant_values=1)
  freq = jnp.ones([input_ts.shape[0], 1], dtype=jnp.int32) * self.freq
  new_input_batch = NestedMap(
      input_ts=input_ts,
      input_padding=input_padding,
      freq=freq,
  )
  return self.core_layer(new_input_batch)

def _quantile_loss(self, pred: JTensor, actual: JTensor,
                   quantile: float) -> JTensor:
  """Calculates quantile loss.

  Args:
    pred: B x T
    actual: B x T
    quantile: quantile at which loss is computed.

  Returns:
    per coordinate loss.
  """
  dev = actual - pred
  loss_first = dev * quantile
  loss_second = -dev * (1.0 - quantile)
  return 2 * jnp.where(loss_first >= 0, loss_first, loss_second)
```

```
def compute_loss(self, prediction_output: NestedMap,
                 input_batch: NestedMap) -> Tuple[NestedMap, NestedMap]:
  output_ts = prediction_output[_OUTPUT_TS]
  actual_ts = input_batch[_TARGET_FUTURE]
  pred_ts = output_ts[:, -1, 0:actual_ts.shape[1], :]
  loss = jnp.square(pred_ts[:, :, 0] - actual_ts)
  for i, quantile in enumerate(self.core_layer.quantiles):
    loss += self._quantile_loss(pred_ts[:, :, i + 1], actual_ts, quantile)
  loss = loss.mean()
  loss_weight = jnp.array(1.0, dtype=jnp.float32)
  per_example_out = NestedMap()
  return {"avg_qloss": (loss, loss_weight)}, per_example_out
```

8) src/timesfm/pytorch_patched_decoder.py

```
# Copyright 2024 Google LLC
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
"""Pytorch version of patched decoder."""

import dataclasses
import math
from typing import List, Tuple
import torch
from torch import nn
import torch.nn.functional as F


def _create_quantiles() -> list[float]:
  return [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]


@dataclasses.dataclass
class TimesFMConfig:
  """Config for initializing timesfm patched_decoder class."""

  # The number of blocks in the model.
  num_layers: int = 20
  # The number of attention heads used in the attention layers of the model.
  num_heads: int = 16
  # The number of key-value heads for implementing attention.
  num_kv_heads: int = 16
  # The hidden size of the model.
  hidden_size: int = 1280
  # The dimension of the MLP representations.
  intermediate_size: int = 1280
  # The number of head dimensions.
  head_dim: int = 80
  # The epsilon used by the rms normalization layers.
  rms_norm_eps: float = 1e-6
  # Patch length
```

```
  patch_len: int = 32
  # Horizon length
  horizon_len: int = 128
  # quantiles
  quantiles: List[float] = dataclasses.field(default_factory=_create_quantiles)
  # Padding value
  pad_val: float = 1123581321.0
  # Tolerance
  tolerance: float = 1e-6
  # The dtype of the weights.
  dtype: str = "bfloat32"
  # use positional embedding
  use_positional_embedding: bool = True


def _masked_mean_std(
    inputs: torch.Tensor,
    padding: torch.Tensor) -> tuple[torch.Tensor, torch.Tensor]:
  """Calculates mean and standard deviation of `inputs` across axis 1.

  It excludes values where `padding` is 1.

  Args:
    inputs: A PyTorch tensor of shape [b, n, p].
    padding: A PyTorch tensor of shape [b, n, p] with values 0 or 1.

  Returns:
    A tuple containing the mean and standard deviation.
    We return the statistics of the first patch with more than three non-padded
    values.
  """
  # Selecting the first patch with more than 3 unpadded values.
  pad_sum = torch.sum(1 - padding, dim=2)

  def _get_patch_index(arr: torch.Tensor):
    indices = torch.argmax((arr >= 3).to(torch.int32), dim=1)
    row_sum = (arr >= 3).to(torch.int32).sum(dim=1)
    return torch.where(row_sum == 0, arr.shape[1] - 1, indices)

  patch_indices = _get_patch_index(pad_sum)
  bidxs = torch.arange(inputs.shape[0])

  arr = inputs[bidxs, patch_indices, :]
  pad = padding[bidxs, patch_indices, :]

  # Create a mask where padding is 0
  mask = 1 - pad
```

```
  # Calculate the number of valid elements
  num_valid_elements = torch.sum(mask, dim=1)
  num_valid_elements = torch.where(
      num_valid_elements == 0,
      torch.tensor(1,
                   dtype=num_valid_elements.dtype,
                   device=num_valid_elements.device),
      num_valid_elements,
  )

  # Calculate the masked sum and squared sum
  masked_sum = torch.sum(arr * mask, dim=1)
  masked_squared_sum = torch.sum((arr * mask)**2, dim=1)

  # Calculate the masked mean and standard deviation
  masked_mean = masked_sum / num_valid_elements
  masked_var = masked_squared_sum / num_valid_elements - masked_mean**2
  masked_var = torch.where(
      masked_var < 0.0,
      torch.tensor(0.0, dtype=masked_var.dtype, device=masked_var.device),
      masked_var,
  )
  masked_std = torch.sqrt(masked_var)

  return masked_mean, masked_std


def _shift_padded_seq(mask: torch.Tensor, seq: torch.Tensor) -> torch.Tensor:
  """Shifts rows of seq based on the first 0 in each row of the mask.

  Args:
    mask: mask tensor of shape [B, N]
    seq: seq tensor of shape [B, N, P]

  Returns:
    Returns the shifted sequence.
  """
  batch_size, num_seq, feature_dim = seq.shape

  new_mask: torch.BoolTensor = mask == 0

  # Use argmax to find the first True value in each row
  indices = new_mask.to(torch.int32).argmax(dim=1)

  # Handle rows with all zeros
  indices[~new_mask.any(dim=1)] = -1

  # Create index ranges for each sequence in the batch
```

```
  idx_range = (torch.arange(num_seq).to(
      seq.device).unsqueeze(0).unsqueeze(-1).expand(batch_size, -1,
                                              feature_dim))

  # Calculate shifted indices for each element in each sequence
  shifted_idx = (idx_range - indices[:, None, None]) % num_seq

  # Gather values from seq using shifted indices
  shifted_seq = seq.gather(1, shifted_idx)

  return shifted_seq


def get_large_negative_number(dtype: torch.dtype) -> torch.Tensor:
  """Returns a large negative value for the given dtype."""
  if dtype.is_floating_point:
    dtype_max = torch.finfo(dtype).max
  else:
    dtype_max = torch.iinfo(dtype).max
  return torch.tensor(-0.7 * dtype_max, dtype=dtype)


def apply_mask_to_logits(logits: torch.Tensor,
                         mask: torch.Tensor) -> torch.Tensor:
  """Applies a floating-point mask to a set of logits.

  Args:
      logits: A torch.Tensor of logit values.
      mask: A torch.Tensor (float32) of mask values with the encoding described
        in the function documentation.

  Returns:
      Masked logits.
  """

  min_value = get_large_negative_number(logits.dtype)

  return torch.where((mask >= min_value * 0.5), logits, min_value)


def convert_paddings_to_mask(
    paddings: torch.Tensor, dtype: torch.dtype = torch.float32) -> torch.Tensor:
  """Converts binary paddings to a logit mask ready to add to attention matrix.

  Args:
      paddings: binary torch.Tensor of shape [B, T], with 1 denoting padding
        token.
      dtype: data type of the input.
```

```
  Returns:
    A torch.Tensor of shape [B, 1, 1, T] ready to add to attention logits.
  """
  attention_mask = paddings.detach().clone()
  attention_mask = attention_mask[:, None, None, :]  # Equivalent to jnp.newaxis
  attention_mask *= get_large_negative_number(dtype)
  return attention_mask


def causal_mask(input_t: torch.Tensor) -> torch.Tensor:
  """Computes and returns causal mask.

  Args:
    input_t: A torch.Tensor of shape [B, T, D].

  Returns:
    An attention_mask torch.Tensor of shape [1, 1, T, T]. Attention mask has
    already been converted to large negative values.
  """
  assert input_t.dtype.is_floating_point, input_t.dtype
  large_negative_number = get_large_negative_number(input_t.dtype)
  t = input_t.shape[1]
  col_idx = torch.arange(t).unsqueeze(0).repeat(t, 1)
  row_idx = torch.arange(t).unsqueeze(1).repeat(1, t)
  mask = (row_idx < col_idx).to(input_t.dtype) * large_negative_number
  return (mask.unsqueeze(0).unsqueeze(0).to(input_t.device)
      )  # Equivalent to jnp.newaxis


def merge_masks(a: torch.Tensor, b: torch.Tensor) -> torch.Tensor:
  """Merges 2 masks.

  logscale mask is expected but 0/1 mask is also fine.

  Args:
    a: torch.Tensor of shape [1|B, 1, 1|T, S].
    b: torch.Tensor of shape [1|B, 1, 1|T, S].

  Returns:
    torch.Tensor of shape [1|B, 1, 1|T, S].
  """

  def expand_t(key_mask):
    query_mask = key_mask.transpose(-1, -2)  # Equivalent of jnp.transpose
    return torch.minimum(query_mask, key_mask)

  if a.shape[2] != b.shape[2]:
```

```
    if a.shape[2] == 1:
      a = expand_t(a)
    else:
      assert b.shape[2] == 1
      b = expand_t(b)

  assert a.shape[1:] == b.shape[1:], f"a.shape={a.shape}, b.shape={b.shape}."
  return torch.minimum(a, b)  # Element-wise minimum, similar to jnp.minimum


class ResidualBlock(nn.Module):
  """TimesFM residual block."""

  def __init__(
      self,
      input_dims,
      hidden_dims,
      output_dims,
  ):
    super(ResidualBlock, self).__init__()
    self.input_dims = input_dims
    self.hidden_dims = hidden_dims
    self.output_dims = output_dims

    # Hidden Layer
    self.hidden_layer = nn.Sequential(
        nn.Linear(input_dims, hidden_dims),
        nn.SiLU(),
    )

    # Output Layer
    self.output_layer = nn.Linear(hidden_dims, output_dims)
    # Residual Layer
    self.residual_layer = nn.Linear(input_dims, output_dims)

  def forward(self, x):
    hidden = self.hidden_layer(x)
    output = self.output_layer(hidden)
    residual = self.residual_layer(x)
    return output + residual


class RMSNorm(torch.nn.Module):
  """Pax rms norm in pytorch."""

  def __init__(
      self,
      dim: int,
```

```
      eps: float = 1e-6,
      add_unit_offset: bool = False,
  ):
    super().__init__()
    self.eps = eps
    self.add_unit_offset = add_unit_offset
    self.weight = nn.Parameter(torch.zeros(dim))

  def _norm(self, x):
    return x * torch.rsqrt(x.pow(2).mean(-1, keepdim=True) + self.eps)

  def forward(self, x):
    output = self._norm(x.float())
    if self.add_unit_offset:
      output = output * (1 + self.weight.float())
    else:
      output = output * self.weight.float()
    return output.type_as(x)


class TransformerMLP(nn.Module):
  """Pax transformer MLP in pytorch."""

  def __init__(
      self,
      hidden_size: int,
      intermediate_size: int,
  ):
    super().__init__()
    self.gate_proj = nn.Linear(hidden_size, intermediate_size)
    self.down_proj = nn.Linear(intermediate_size, hidden_size)
    self.layer_norm = nn.LayerNorm(normalized_shape=hidden_size, eps=1e-6)

  def forward(self, x, paddings=None):
    gate_inp = self.layer_norm(x)
    gate = self.gate_proj(gate_inp)
    gate = F.relu(gate)
    outputs = self.down_proj(gate)
    if paddings is not None:
      outputs = outputs * (1.0 - paddings[:, :, None])
    return outputs + x


class TimesFMAttention(nn.Module):
  """Implements the attention used in TimesFM."""

  def __init__(
      self,
```

```
    hidden_size: int,
    num_heads: int,
    num_kv_heads: int,
    head_dim: int,
):
  super().__init__()

  self.num_heads = num_heads
  self.num_kv_heads = num_kv_heads

  assert self.num_heads % self.num_kv_heads == 0
  self.num_queries_per_kv = self.num_heads // self.num_kv_heads

  self.hidden_size = hidden_size
  self.head_dim = head_dim

  self.q_size = self.num_heads * self.head_dim
  self.kv_size = self.num_kv_heads * self.head_dim
  self.scaling = nn.Parameter(
     torch.empty((self.head_dim,), dtype=torch.float32),)

  self.qkv_proj = nn.Linear(
     self.hidden_size,
     (self.num_heads + 2 * self.num_kv_heads) * self.head_dim,
  )
  self.o_proj = nn.Linear(self.num_heads * self.head_dim, self.hidden_size)

def _per_dim_scaling(self, query: torch.Tensor) -> torch.Tensor:
  # [batch_size, n_local_heads, input_len, head_dim]
  r_softplus_0 = 1.442695041
  softplus_func = torch.nn.Softplus()
  scale = r_softplus_0 / math.sqrt(self.head_dim)
  scale = scale * softplus_func(self.scaling)
  return query * scale[None, None, None, :]

def forward(
    self,
    hidden_states: torch.Tensor,
    mask: torch.Tensor,
    kv_write_indices: torch.Tensor | None = None,
    kv_cache: Tuple[torch.Tensor, torch.Tensor] | None = None,
) -> torch.Tensor:
  hidden_states_shape = hidden_states.shape
  assert len(hidden_states_shape) == 3

  batch_size, input_len, _ = hidden_states_shape

  qkv = self.qkv_proj(hidden_states)
```

```
    xq, xk, xv = qkv.split([self.q_size, self.kv_size, self.kv_size], dim=-1)

    xq = xq.view(batch_size, -1, self.num_heads, self.head_dim)
    xk = xk.view(batch_size, -1, self.num_kv_heads, self.head_dim)
    xv = xv.view(batch_size, -1, self.num_kv_heads, self.head_dim)
    xq = self._per_dim_scaling(xq)

    # Write new kv cache.
    # [batch_size, input_len, n_local_kv_heads, head_dim]
    if kv_cache is not None and kv_write_indices is not None:
      k_cache, v_cache = kv_cache
      k_cache.index_copy_(1, kv_write_indices, xk)
      v_cache.index_copy_(1, kv_write_indices, xv)

      key = k_cache
      value = v_cache
    else:
      key = xk
      value = xv
    if self.num_kv_heads != self.num_heads:
      # [batch_size, max_seq_len, n_local_heads, head_dim]
      key = torch.repeat_interleave(key, self.num_queries_per_kv, dim=2)
      value = torch.repeat_interleave(value, self.num_queries_per_kv, dim=2)

    # [batch_size, n_local_heads, input_len, head_dim]
    q = xq.transpose(1, 2)
    # [batch_size, n_local_heads, max_seq_len, head_dim]
    k = key.transpose(1, 2)
    v = value.transpose(1, 2)

    # [batch_size, n_local_heads, input_len, max_seq_len]
    scores = torch.matmul(q, k.transpose(2, 3))
    scores = scores + mask
    scores = F.softmax(scores.float(), dim=-1).type_as(q)

    # [batch_size, n_local_heads, input_len, head_dim]
    output = torch.matmul(scores, v)
    # return scores, output.transpose(1, 2).contiguous()

    # [batch_size, input_len, hidden_dim]
    output = output.transpose(1, 2).contiguous().view(batch_size, input_len, -1)
    output = self.o_proj(output)
    return scores, output


class TimesFMDecoderLayer(nn.Module):
  """Transformer layer."""
```

```
  def __init__(
      self,
      hidden_size: int,
      intermediate_size: int,
      num_heads: int,
      num_kv_heads: int,
      head_dim: int,
      rms_norm_eps: float = 1e-6,
  ):
    super().__init__()
    self.self_attn = TimesFMAttention(
        hidden_size=hidden_size,
        num_heads=num_heads,
        num_kv_heads=num_kv_heads,
        head_dim=head_dim,
    )
    self.mlp = TransformerMLP(
        hidden_size=hidden_size,
        intermediate_size=intermediate_size,
    )
    self.input_layernorm = RMSNorm(hidden_size, eps=rms_norm_eps)

  def forward(
      self,
      hidden_states: torch.Tensor,
      mask: torch.Tensor,
      paddings: torch.Tensor,
      kv_write_indices: torch.Tensor | None = None,
      kv_cache: Tuple[torch.Tensor, torch.Tensor] | None = None,
  ) -> torch.Tensor:
    # Self Attention
    residual = hidden_states
    hidden_states = self.input_layernorm(hidden_states)
    scores, hidden_states = self.self_attn(
        hidden_states=hidden_states,
        mask=mask,
        kv_write_indices=kv_write_indices,
        kv_cache=kv_cache,
    )
    hidden_states = residual + hidden_states

    # MLP
    hidden_states = self.mlp(hidden_states, paddings=paddings)

    return scores, hidden_states


class StackedDecoder(nn.Module):
```

```
"""Stacked transformer layer."""

def __init__(
    self,
    hidden_size: int,
    intermediate_size: int,
    num_heads: int,
    num_kv_heads: int,
    head_dim: int,
    num_layers: int,
    rms_norm_eps: float = 1e-6,
):
  super().__init__()

  self.layers = nn.ModuleList()
  for _ in range(num_layers):
    self.layers.append(
        TimesFMDecoderLayer(
            hidden_size=hidden_size,
            intermediate_size=intermediate_size,
            num_heads=num_heads,
            num_kv_heads=num_kv_heads,
            head_dim=head_dim,
            rms_norm_eps=rms_norm_eps,
        ))

def forward(
    self,
    hidden_states: torch.Tensor,
    paddings: torch.Tensor,
    kv_write_indices: torch.Tensor | None = None,
    kv_caches: List[Tuple[torch.Tensor, torch.Tensor]] | None = None,
) -> torch.Tensor:
  padding_mask = convert_paddings_to_mask(paddings, hidden_states.dtype)
  atten_mask = causal_mask(hidden_states)
  mask = merge_masks(padding_mask, atten_mask)
  for i in range(len(self.layers)):
    layer = self.layers[i]
    kv_cache = kv_caches[i] if kv_caches is not None else None
    _, hidden_states = layer(
        hidden_states=hidden_states,
        mask=mask,
        paddings=paddings,
        kv_write_indices=kv_write_indices,
        kv_cache=kv_cache,
    )
  return hidden_states
```

```
class PositionalEmbedding(torch.nn.Module):
  """Generates position embedding for a given 1-d sequence.

  Attributes:
      min_timescale: Start of the geometric index. Determines the periodicity of
        the added signal.
      max_timescale: End of the geometric index. Determines the frequency of the
        added signal.
      embedding_dims: Dimension of the embedding to be generated.
  """

  def __init__(
      self,
      embedding_dims: int,
      min_timescale: int = 1,
      max_timescale: int = 10_000,
  ) -> None:
    super().__init__()
    self.min_timescale = min_timescale
    self.max_timescale = max_timescale
    self.embedding_dims = embedding_dims

  def forward(self, seq_length=None, position=None):
    """Generates a Tensor of sinusoids with different frequencies.

    Args:
        seq_length: an optional Python int defining the output sequence length.
          if the `position` argument is specified.
        position:   [B, seq_length], optional position for each token in the
          sequence, only required when the sequence is packed.

    Returns:
        [B, seqlen, D] if `position` is specified, else [1, seqlen, D]
    """
    if position is None:
      assert seq_length is not None
      # [1, seqlen]
      position = torch.arange(seq_length, dtype=torch.float32).unsqueeze(0)
    else:
      assert position.ndim == 2, position.shape

    num_timescales = self.embedding_dims // 2
    log_timescale_increment = math.log(
        float(self.max_timescale) / float(self.min_timescale)) / max(
            num_timescales - 1, 1)
    inv_timescales = self.min_timescale * torch.exp(
        torch.arange(num_timescales, dtype=torch.float32) *
```

```
        -log_timescale_increment)
    scaled_time = position.unsqueeze(2) * inv_timescales.unsqueeze(0).unsqueeze(
        0)
    signal = torch.cat([torch.sin(scaled_time), torch.cos(scaled_time)], dim=2)
    # Padding to ensure correct embedding dimension
    signal = F.pad(signal, (0, 0, 0, self.embedding_dims % 2))
    return signal


class PatchedTimeSeriesDecoder(nn.Module):
  """Patched time-series decoder."""

  def __init__(self, config: TimesFMConfig):
    super().__init__()
    self.config = config
    self.input_ff_layer = ResidualBlock(
        input_dims=2 * config.patch_len,
        output_dims=config.hidden_size,
        hidden_dims=config.intermediate_size,
    )
    self.freq_emb = nn.Embedding(num_embeddings=3,
                                 embedding_dim=config.hidden_size)
    self.horizon_ff_layer = ResidualBlock(
        input_dims=config.hidden_size,
        output_dims=config.horizon_len * (1 + len(config.quantiles)),
        hidden_dims=config.intermediate_size,
    )
    self.stacked_transformer = StackedDecoder(
        hidden_size=self.config.hidden_size,
        intermediate_size=self.config.intermediate_size,
        num_heads=self.config.num_heads,
        num_kv_heads=self.config.num_kv_heads,
        head_dim=self.config.head_dim,
        num_layers=self.config.num_layers,
        rms_norm_eps=self.config.rms_norm_eps,
    )
    if self.config.use_positional_embedding:
      self.position_emb = PositionalEmbedding(self.config.hidden_size)

  def _forward_transform(
      self, inputs: torch.Tensor, patched_pads: torch.Tensor
  ) -> tuple[torch.Tensor, tuple[torch.Tensor, torch.Tensor]]:
    """Input is of shape [B, N, P]."""
    mu, sigma = _masked_mean_std(inputs, patched_pads)
    sigma = torch.where(
        sigma < self.config.tolerance,
        torch.tensor(1.0, dtype=sigma.dtype, device=sigma.device),
        sigma,
```

```
  )

  # Normalize each patch
  outputs = (inputs - mu[:, None, None]) / sigma[:, None, None]
  outputs = torch.where(
      torch.abs(inputs - self.config.pad_val) < self.config.tolerance,
      torch.tensor(self.config.pad_val,
                   dtype=outputs.dtype,
                   device=outputs.device),
      outputs,
  )
  return outputs, (mu, sigma)

def _reverse_transform(
    self, outputs: torch.Tensor, stats: tuple[torch.Tensor,
                                              torch.Tensor]) -> torch.Tensor:
  """Output is of shape [B, N, P, Q]."""
  mu, sigma = stats
  return outputs * sigma[:, None, None, None] + mu[:, None, None, None]

def _preprocess_input(
    self,
    input_ts: torch.Tensor,
    input_padding: torch.Tensor,
) -> tuple[
    torch.Tensor,
    torch.Tensor,
    tuple[torch.Tensor, torch.Tensor] | None,
    torch.Tensor,
]:
  """Preprocess input for stacked transformer."""

  # Reshape into patches (using view for efficiency)
  bsize = input_ts.shape[0]
  patched_inputs = input_ts.view(bsize, -1, self.config.patch_len)
  patched_pads = input_padding.view(bsize, -1, self.config.patch_len)

  patched_inputs = torch.where(
      torch.abs(patched_pads - 1.0) < self.config.tolerance,
      torch.tensor(0.0,
                   dtype=patched_inputs.dtype,
                   device=patched_inputs.device),
      patched_inputs,
  )
  patched_pads = torch.where(
      torch.abs(patched_inputs - self.config.pad_val) < self.config.tolerance,
      torch.tensor(1.0, dtype=patched_pads.dtype, device=patched_pads.device),
      patched_pads,
```

```
    )
    patched_inputs, stats = self._forward_transform(patched_inputs,
                                                    patched_pads)

    # B x N x D
    patched_inputs = patched_inputs * (1.0 - patched_pads)
    concat_inputs = torch.cat([patched_inputs, patched_pads], dim=-1)
    model_input = self.input_ff_layer(concat_inputs)

    # A patch should not be padded even if there is at least one zero.
    patched_padding = torch.min(patched_pads,
                                dim=-1)[0]  # Get the values from the min result
    if self.config.use_positional_embedding:
      pos_emb = self.position_emb(model_input.shape[1]).to(model_input.device)
      pos_emb = torch.concat([pos_emb] * model_input.shape[0], dim=0)
      pos_emb = _shift_padded_seq(patched_padding, pos_emb)
      model_input += pos_emb

    return model_input, patched_padding, stats, patched_inputs

  def _postprocess_output(
      self,
      model_output: torch.Tensor,
      num_outputs: int,
      stats: tuple[torch.Tensor, torch.Tensor],
  ) -> torch.Tensor:
    """Postprocess output of stacked transformer."""

    # B x N x (H.Q)
    output_ts = self.horizon_ff_layer(model_output)

    # Reshape using view
    b, n, _ = output_ts.shape
    output_ts = output_ts.view(b, n, self.config.horizon_len, num_outputs)

    return self._reverse_transform(output_ts, stats)

  def forward(
      self,
      input_ts: torch.Tensor,
      input_padding: torch.LongTensor,
      freq: torch.Tensor,
  ) -> torch.Tensor:
    num_outputs = len(self.config.quantiles) + 1
    model_input, patched_padding, stats, _ = self._preprocess_input(
        input_ts=input_ts,
        input_padding=input_padding,
    )
```

```
  f_emb = self.freq_emb(freq)  # B x 1 x D
  model_input += f_emb
  model_output = self.stacked_transformer(model_input, patched_padding)

  output_ts = self._postprocess_output(model_output, num_outputs, stats)
  return output_ts

def decode(
    self,
    input_ts: torch.Tensor,
    paddings: torch.Tensor,
    freq: torch.LongTensor,
    horizon_len: int,
    output_patch_len: int | None = None,
    max_len: int = 512,
    return_forecast_on_context: bool = False,
) -> tuple[torch.Tensor, torch.Tensor]:
  """Auto-regressive decoding without caching.

  Args:
    input_ts: input time-series and paddings. Time-series shape B x C.
    paddings: padding shape B x (C + H) where H is the prediction length.
    freq: frequency shape B x 1
    horizon_len: prediction length.
    output_patch_len: output length to be fetched from one step of
      auto-regressive decoding.
    max_len: maximum training context length.
    return_forecast_on_context: whether to return the model forecast on the
      context except the first input patch.

  Returns:
    Tuple of two forecasting results:
    - Point (mean) output predictions as a tensor with shape B x H'.
    - Full predictions (mean and quantiles) as a tensor with shape
      B x H' x (1 + # quantiles).
    In particular, if return_forecast_on_context is True, H' is H plus
    the forecastable context length, i.e. context_len - (first) patch_len.
  """
  final_out = input_ts
  context_len = final_out.shape[1]
  full_outputs = []
  if paddings.shape[1] != final_out.shape[1] + horizon_len:
    raise ValueError(
        "Length of paddings must match length of input + horizon_len:"
        f" {paddings.shape[1]} != {final_out.shape[1]} + {horizon_len}")
  if output_patch_len is None:
    output_patch_len = self.config.horizon_len
  num_decode_patches = (horizon_len + output_patch_len -
```

```
                    1) // output_patch_len
for step_index in range(num_decode_patches):
  current_padding = paddings[:, 0:final_out.shape[1]]
  input_ts = final_out[:, -max_len:]
  input_padding = current_padding[:, -max_len:]
  fprop_outputs = self(input_ts, input_padding, freq)
  if return_forecast_on_context and step_index == 0:
    # For the first decodings step, collect the model forecast on the
    # context except the unavailable first input batch forecast.
    new_full_ts = fprop_outputs[:, :-1, :self.config.patch_len, :]
    new_full_ts = fprop_outputs.view(new_full_ts.size(0), -1,
                                     new_full_ts.size(3))

    full_outputs.append(new_full_ts)

  # (full batch, last patch, output_patch_len, index of mean forecast = 0)
  new_ts = fprop_outputs[:, -1, :output_patch_len, 0]
  new_full_ts = fprop_outputs[:, -1, :output_patch_len, :]
  # (full batch, last patch, output_patch_len, all output indices)
  full_outputs.append(new_full_ts)
  final_out = torch.concatenate([final_out, new_ts], axis=-1)

if return_forecast_on_context:
  # `full_outputs` indexing starts at after the first input patch.
  full_outputs = torch.concatenate(
      full_outputs,
      axis=1)[:, :(context_len - self.config.patch_len + horizon_len), :]
else:
  # `full_outputs` indexing starts at the forecast horizon.
  full_outputs = torch.concatenate(full_outputs, axis=1)[:,
                                                         0:horizon_len, :]

return (full_outputs[:, :, 0], full_outputs)
```

9) src/timesfm/time_features.py

```
# Copyright 2024 The Google Research Authors.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""Directory to extract time covariates.

Extract time covariates from datetime.
"""

import numpy as np
import pandas as pd
from pandas.tseries.holiday import EasterMonday
from pandas.tseries.holiday import GoodFriday
from pandas.tseries.holiday import Holiday
from pandas.tseries.holiday import SU
from pandas.tseries.holiday import TH
from pandas.tseries.holiday import USColumbusDay
from pandas.tseries.holiday import USLaborDay
from pandas.tseries.holiday import USMartinLutherKingJr
from pandas.tseries.holiday import USMemorialDay
from pandas.tseries.holiday import USPresidentsDay
from pandas.tseries.holiday import USThanksgivingDay
from pandas.tseries.offsets import DateOffset
from pandas.tseries.offsets import Day
from pandas.tseries.offsets import Easter
from sklearn.preprocessing import StandardScaler
from tqdm import tqdm


# This is 183 to cover half a year (in both directions), also for leap years
# + 17 as Eastern can be between March, 22 - April, 25
MAX_WINDOW = 183 + 17


def _distance_to_holiday(holiday):
  """Return distance to given holiday."""
```

```
  def _distance_to_day(index):
    holiday_date = holiday.dates(
        index - pd.Timedelta(days=MAX_WINDOW),
        index + pd.Timedelta(days=MAX_WINDOW),
    )
    assert (
        len(holiday_date) != 0  # pylint: disable=g-explicit-length-test
    ), f"No closest holiday for the date index {index} found."
    # It sometimes returns two dates if it is exactly half a year after the
    # holiday. In this case, the smaller distance (182 days) is returned.
    return (index - holiday_date[0]).days

  return _distance_to_day


EasterSunday = Holiday(
    "Easter Sunday", month=1, day=1, offset=[Easter(), Day(0)]
)
NewYearsDay = Holiday("New Years Day", month=1, day=1)
SuperBowl = Holiday(
    "Superbowl", month=2, day=1, offset=DateOffset(weekday=SU(1))
)
MothersDay = Holiday(
    "Mothers Day", month=5, day=1, offset=DateOffset(weekday=SU(2))
)
IndependenceDay = Holiday("Independence Day", month=7, day=4)
ChristmasEve = Holiday("Christmas", month=12, day=24)
ChristmasDay = Holiday("Christmas", month=12, day=25)
NewYearsEve = Holiday("New Years Eve", month=12, day=31)
BlackFriday = Holiday(
    "Black Friday",
    month=11,
    day=1,
    offset=[pd.DateOffset(weekday=TH(4)), Day(1)],
)
CyberMonday = Holiday(
    "Cyber Monday",
    month=11,
    day=1,
    offset=[pd.DateOffset(weekday=TH(4)), Day(4)],
)

HOLIDAYS = [
    EasterMonday,
    GoodFriday,
    USColumbusDay,
    USLaborDay,
```

```
    USMartinLutherKingJr,
    USMemorialDay,
    USPresidentsDay,
    USThanksgivingDay,
    EasterSunday,
    NewYearsDay,
    SuperBowl,
    MothersDay,
    IndependenceDay,
    ChristmasEve,
    ChristmasDay,
    NewYearsEve,
    BlackFriday,
    CyberMonday,
]


class TimeCovariates(object):
  """Extract all time covariates except for holidays."""

  def __init__(
      self,
      datetimes,
      normalized=True,
      holiday=False,
  ):
    """Init function.

    Args:
      datetimes: pandas DatetimeIndex (lowest granularity supported is min)
      normalized: whether to normalize features or not
      holiday: fetch holiday features or not

    Returns:
      None
    """
    self.normalized = normalized
    self.dti = datetimes
    self.holiday = holiday

  def _minute_of_hour(self):
    minutes = np.array(self.dti.minute, dtype=np.float32)
    if self.normalized:
      minutes = minutes / 59.0 - 0.5
    return minutes

  def _hour_of_day(self):
    hours = np.array(self.dti.hour, dtype=np.float32)
```

```
    if self.normalized:
      hours = hours / 23.0 - 0.5
    return hours

  def _day_of_week(self):
    day_week = np.array(self.dti.dayofweek, dtype=np.float32)
    if self.normalized:
      day_week = day_week / 6.0 - 0.5
    return day_week

  def _day_of_month(self):
    day_month = np.array(self.dti.day, dtype=np.float32)
    if self.normalized:
      day_month = day_month / 30.0 - 0.5
    return day_month

  def _day_of_year(self):
    day_year = np.array(self.dti.dayofyear, dtype=np.float32)
    if self.normalized:
      day_year = day_year / 364.0 - 0.5
    return day_year

  def _month_of_year(self):
    month_year = np.array(self.dti.month, dtype=np.float32)
    if self.normalized:
      month_year = month_year / 11.0 - 0.5
    return month_year

  def _week_of_year(self):
    week_year = np.array(self.dti.strftime("%U").astype(int), dtype=np.float32)
    if self.normalized:
      week_year = week_year / 51.0 - 0.5
    return week_year

  def _get_holidays(self):
    dti_series = self.dti.to_series()
    hol_variates = np.vstack([
        dti_series.apply(_distance_to_holiday(h)).values for h in tqdm(HOLIDAYS)
    ])
    # hol_variates is (num_holiday, num_time_steps), the normalization should be
    # performed in the num_time_steps dimension.
    return StandardScaler().fit_transform(hol_variates.T).T

  def get_covariates(self):
    """Get all time covariates."""
    moh = self._minute_of_hour().reshape(1, -1)
    hod = self._hour_of_day().reshape(1, -1)
    dom = self._day_of_month().reshape(1, -1)
```

```
dow = self._day_of_week().reshape(1, -1)
doy = self._day_of_year().reshape(1, -1)
moy = self._month_of_year().reshape(1, -1)
woy = self._week_of_year().reshape(1, -1)

all_covs = [
    moh,
    hod,
    dom,
    dow,
    doy,
    moy,
    woy,
]
columns = ["moh", "hod", "dom", "dow", "doy", "moy", "woy"]
if self.holiday:
  hol_covs = self._get_holidays()
  all_covs.append(hol_covs)
  columns += [f"hol_{i}" for i in range(len(HOLIDAYS))]

return pd.DataFrame(
    data=np.vstack(all_covs).transpose(),
    columns=columns,
    index=self.dti,
)
```

10) src/timesfm/timesfm_base.py

```
# Copyright 2024 Google LLC
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
"""Base class for TimesFM inference. This will be common to PAX and Pytorch."""

import collections
import dataclasses
import logging
import multiprocessing
from typing import Any, Literal, Sequence

import numpy as np
import pandas as pd

from utilsforecast.processing import make_future_dataframe

from . import xreg_lib

Category = xreg_lib.Category
XRegMode = xreg_lib.XRegMode

_TOL = 1e-6
DEFAULT_QUANTILES = (0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9)


def process_group(key, group, value_name, forecast_context_len):
  group = group.tail(forecast_context_len)
  return np.array(group[value_name], dtype=np.float32), key


def moving_average(arr, window_size):
  """Calculates the moving average using NumPy's convolution function."""
  # Pad with zeros to handle initial window positions
  arr_padded = np.pad(arr, (window_size - 1, 0), "constant")
  smoothed_arr = (np.convolve(arr_padded, np.ones(window_size), "valid") /
                  window_size)
```

```
  return [smoothed_arr, arr - smoothed_arr]


def freq_map(freq: str):
  """Returns the frequency map for the given frequency string."""
  freq = str.upper(freq)
  if (freq.endswith("H") or freq.endswith("T") or freq.endswith("MIN") or
      freq.endswith("D") or freq.endswith("B") or freq.endswith("U")):
    return 0
  elif freq.endswith(("W", "M", "MS")):
    return 1
  elif freq.endswith("Y") or freq.endswith("Q"):
    return 2
  else:
    raise ValueError(f"Invalid frequency: {freq}")


# Per time series normalization: forward.
def normalize(batch):
  stats = [
      (np.mean(x), np.where((w := np.std(x)) > _TOL, w, 1.0)) for x in batch
  ]
  new_batch = [(x - stat[0]) / stat[1] for x, stat in zip(batch, stats)]
  return new_batch, stats


# Per time series normalization: inverse.
def renormalize(batch, stats):
  return [x * stat[1] + stat[0] for x, stat in zip(batch, stats)]


@dataclasses.dataclass(kw_only=True)
class TimesFmHparams:
  """Hparams used to initialize a TimesFM model for inference.

  These are the sufficient subset of hparams to configure TimesFM inference
  agnostic to the checkpoint version, and are not necessarily the same as the
  hparams used to train the checkpoint.

  Attributes:
    context_len: Largest context length the model allows for each decode call.
      This technically can be any large, but practically should set to the
      context length the checkpoint was trained with.
    horizon_len: Forecast horizon.
    input_patch_len: Input patch len.
    output_patch_len: Output patch len. How many timepoints is taken from a
      single step of autoregressive decoding. Can be set as the training horizon
      of the checkpoint.
```

```
    num_layers: Number of transformer layers in the model.
    model_dims: Model dimension.
    per_core_batch_size: Batch size on each core for data parallelism.
    backend: One of "cpu", "gpu" or "tpu".
    quantiles: Which quantiles are output by the model.
  """

  context_len: int = 512
  horizon_len: int = 128
  input_patch_len: int = 32
  output_patch_len: int = 128
  num_layers: int = 20
  num_heads: int = 16
  model_dims: int = 1280
  per_core_batch_size: int = 32
  backend: Literal["cpu", "gpu", "tpu"] = "cpu"
  quantiles: Sequence[float] | None = DEFAULT_QUANTILES


@dataclasses.dataclass(kw_only=True)
class TimesFmCheckpoint:
  """Checkpoint used to initialize a TimesFM model for inference.

  Attributes:
    version: Version of the checkpoint, e.g. "jax", "torch", "tensorflow", etc.
      The factory will create the corresponding TimesFm inference class based on
      this version.
    path: Path to the checkpoint.
    type: If provided, type of the checkpoint used by the specific checkpoint
      loader per version.
    step: If provided, step of the checkpoint.
  """

  version: str = "jax"
  path: str | None = None
  huggingface_repo_id: str | None = None
  type: Any = None
  step: int | None = None


class TimesFmBase:
  """Base TimesFM forecast API for inference.

  This class is the scaffolding for calling TimesFM forecast. To properly use:
    1. Create an instance with the correct hyperparameters of a TimesFM model.
    2. Call `load_from_checkpoint` to load a compatible checkpoint.
    3. Call `forecast` for inference.
  """
```

```
def _logging(self, s):
  print(s)

def __post_init__(self) -> None:
  """Additional initialization for subclasses before checkpoint loading."""
  pass

def __init__(self, hparams: TimesFmHparams,
             checkpoint: TimesFmCheckpoint) -> None:
  """Initializes the TimesFM forecast API.

  Args:
    hparams: Hyperparameters of the model.
    checkpoint: Checkpoint to load. Notice `checkpoint.version` will decide
      which TimesFM version to use.
  """
  self.hparams = hparams

  # Expand hparams for conciseness within the model code.
  self.context_len = hparams.context_len
  self.horizon_len = hparams.horizon_len
  self.input_patch_len = hparams.input_patch_len
  self.output_patch_len = hparams.output_patch_len
  self.num_layers = hparams.num_layers
  self.model_dims = hparams.model_dims
  self.backend = hparams.backend
  self.quantiles = hparams.quantiles
  self.num_heads = hparams.num_heads

  # Rewrite these values in __post_init__ for SPMD.
  self.num_cores = 1
  self.per_core_batch_size = hparams.per_core_batch_size
  self.global_batch_size = hparams.per_core_batch_size

  self._horizon_start = self.context_len - self.input_patch_len
  self.__post_init__()
  self.load_from_checkpoint(checkpoint)

def load_from_checkpoint(self, checkpoint: TimesFmCheckpoint) -> None:
  """Loads a checkpoint and compiles the decoder."""
  raise NotImplementedError("`load_from_checkpoint` is not implemented.")

def _preprocess(self, inputs: Sequence[np.array],
                freq: Sequence[int]) -> tuple[np.array, np.array, int]:
  """Formats and pads raw inputs to feed into the model.

  This function both pads each time series to match the context length, and
```

```
pads the inputs to meet the SPMD shape requirement.

Args:
  inputs: A list of 1d JTensors. Each JTensor is the context time series of
    a single forecast task.
  freq: list of frequencies

Returns:
A tuple of:
- the padded input time series to meet the model required context.
- the padding indicator.
- the number of padded examples for SPMD so that each core has the same
    number (a multiple of `batch_size`) of examples.
"""

input_ts, input_padding, inp_freq = [], [], []

pmap_pad = ((len(inputs) - 1) // self.global_batch_size +
            1) * self.global_batch_size - len(inputs)

for i, ts in enumerate(inputs):
  input_len = ts.shape[0]
  padding = np.zeros(shape=(input_len + self.horizon_len,), dtype=float)
  if input_len < self.context_len:
    num_front_pad = self.context_len - input_len
    ts = np.concatenate([np.zeros(shape=(num_front_pad,), dtype=float), ts],
                        axis=0)
    padding = np.concatenate(
        [np.ones(shape=(num_front_pad,), dtype=float), padding], axis=0)
  elif input_len > self.context_len:
    ts = ts[-self.context_len:]
    padding = padding[-(self.context_len + self.horizon_len):]

  input_ts.append(ts)
  input_padding.append(padding)
  inp_freq.append(freq[i])

# Padding the remainder batch.
for _ in range(pmap_pad):
  input_ts.append(input_ts[-1])
  input_padding.append(input_padding[-1])
  inp_freq.append(inp_freq[-1])

return (
    np.stack(input_ts, axis=0),
    np.stack(input_padding, axis=0),
    np.array(inp_freq).astype(np.int32).reshape(-1, 1),
    pmap_pad,
```

```
  )

def forecast(
    self,
    inputs: Sequence[Any],
    freq: Sequence[int] | None = None,
    window_size: int | None = None,
    forecast_context_len: int | None = None,
    return_forecast_on_context: bool = False,
    truncate_negative: bool = False,
) -> tuple[np.array, np.array]:
  """Forecasts on a list of time series.

  Args:
    inputs: list of time series forecast contexts. Each context time series
      should be in a format convertible to JTensor by `jnp.array`.
    freq: frequency of each context time series. 0 for high frequency
      (default), 1 for medium, and 2 for low. Notice this is different from
      the `freq` required by `forecast_on_df`.
    window_size: window size of trend + residual decomposition. If None then
      we do not do decomposition.
    forecast_context_len: optional max context length.
    return_forecast_on_context: True to return the forecast on the context
      when available, i.e. after the first input patch.
    truncate_negative: truncate to only non-negative values if all the contexts
      have non-negative values.

  Returns:
  A tuple for JTensors:
  - the mean forecast of size (# inputs, # forecast horizon),
  - the full forecast (mean + quantiles) of size
      (# inputs,  # forecast horizon, 1 + # quantiles).

  Raises:
  ValueError: If the checkpoint is not properly loaded.
  """
  raise NotImplementedError("`forecast` is not implemented.")

def forecast_with_covariates(
    self,
    inputs: list[Sequence[float]],
    dynamic_numerical_covariates: (dict[str, Sequence[Sequence[float]]] |
                                   None) = None,
    dynamic_categorical_covariates: (dict[str, Sequence[Sequence[Category]]] |
                                     None) = None,
    static_numerical_covariates: dict[str, Sequence[float]] | None = None,
    static_categorical_covariates: (dict[str, Sequence[Category]] |
                                    None) = None,
```

```
    freq: Sequence[int] | None = None,
    window_size: int | None = None,
    forecast_context_len: int | None = None,
    xreg_mode: XRegMode = "xreg + timesfm",
    normalize_xreg_target_per_input: bool = True,
    ridge: float = 0.0,
    max_rows_per_col: int = 0,
    force_on_cpu: bool = False,
):
  """Forecasts on a list of time series with covariates.

  To optimize inference speed, avoid string valued categorical covariates.

  Args:
    inputs: A list of time series forecast contexts. Each context time series
      should be in a format convertible to JTensor by `jnp.array`.
    dynamic_numerical_covariates: A dict of dynamic numerical covariates.
    dynamic_categorical_covariates: A dict of dynamic categorical covariates.
    static_numerical_covariates: A dict of static numerical covariates.
    static_categorical_covariates: A dict of static categorical covariates.
    freq: frequency of each context time series. 0 for high frequency
      (default), 1 for medium, and 2 for low. Notice this is different from
      the `freq` required by `forecast_on_df`.
    window_size: window size of trend + residual decomposition. If None then
      we do not do decomposition.
    forecast_context_len: optional max context length.
    xreg_mode: one of "xreg + timesfm" or "timesfm + xreg". "xreg + timesfm"
      fits a model on the residuals of the TimesFM forecast. "timesfm + xreg"
      fits a model on the targets then forecasts on the residuals via TimesFM.
    normalize_xreg_target_per_input: whether to normalize the xreg target per
      input in the given batch.
    ridge: ridge penalty for the linear model.
    max_rows_per_col: max number of rows per column for the linear model.
    force_on_cpu: whether to force running on cpu for the linear model.

  Returns:
    A tuple of two lists. The first is the outputs of the model. The second is
    the outputs of the xreg.
  """

  # Verify and bookkeep covariates.
  if not (dynamic_numerical_covariates or dynamic_categorical_covariates or
          static_numerical_covariates or static_categorical_covariates):
    raise ValueError(
        "At least one of dynamic_numerical_covariates,"
        " dynamic_categorical_covariates, static_numerical_covariates,"
        " static_categorical_covariates must be set.")
```

```
# Track the lengths of (1) each input, (2) the part that can be used in the
# linear model, and (3) the horizon.
input_lens, train_lens, test_lens = [], [], []

for i, input_ts in enumerate(inputs):
  input_len = len(input_ts)
  input_lens.append(input_len)

  if xreg_mode == "timesfm + xreg":
    # For fitting residuals, no TimesFM forecast on the first patch.
    train_lens.append(max(0, input_len - self.input_patch_len))
  elif xreg_mode == "xreg + timesfm":
    train_lens.append(input_len)
  else:
    raise ValueError(f"Unsupported mode: {xreg_mode}")

  if dynamic_numerical_covariates:
    test_lens.append(
        len(list(dynamic_numerical_covariates.values())[0][i]) - input_len)
  elif dynamic_categorical_covariates:
    test_lens.append(
        len(list(dynamic_categorical_covariates.values())[0][i]) -
        input_len)
  else:
    test_lens.append(self.horizon_len)

  if test_lens[-1] > self.horizon_len:
    raise ValueError(
        "Forecast requested longer horizon than the model definition "
        f"supports: {test_lens[-1]} vs {self.horizon_len}.")

# Prepare the covariates into train and test.
train_dynamic_numerical_covariates = collections.defaultdict(list)
test_dynamic_numerical_covariates = collections.defaultdict(list)
train_dynamic_categorical_covariates = collections.defaultdict(list)
test_dynamic_categorical_covariates = collections.defaultdict(list)
for covariates, train_covariates, test_covariates in (
    (
        dynamic_numerical_covariates,
        train_dynamic_numerical_covariates,
        test_dynamic_numerical_covariates,
    ),
    (
        dynamic_categorical_covariates,
        train_dynamic_categorical_covariates,
        test_dynamic_categorical_covariates,
    ),
):
```

```
  if not covariates:
    continue
  for covariate_name, covariate_values in covariates.items():
    for input_len, train_len, covariate_value in zip(
        input_lens, train_lens, covariate_values):
      train_covariates[covariate_name].append(
          covariate_value[(input_len - train_len):input_len])
      test_covariates[covariate_name].append(covariate_value[input_len:])

# Fit models.
if xreg_mode == "timesfm + xreg":
  # Forecast via TimesFM then fit a model on the residuals.
  mean_outputs, _ = self.forecast(
      inputs,
      freq,
      window_size,
      forecast_context_len,
      return_forecast_on_context=True,
  )
  targets = [
      (np.array(input_ts)[-train_len:] -
       mean_output[(self._horizon_start - train_len):self._horizon_start])
      for input_ts, mean_output, train_len in zip(inputs, mean_outputs,
                                                  train_lens)
  ]
  per_instance_stats = None
  if normalize_xreg_target_per_input:
    targets, per_instance_stats = normalize(targets)
  xregs = xreg_lib.BatchedInContextXRegLinear(
      targets=targets,
      train_lens=train_lens,
      test_lens=test_lens,
      train_dynamic_numerical_covariates=train_dynamic_numerical_covariates,
      test_dynamic_numerical_covariates=test_dynamic_numerical_covariates,
      train_dynamic_categorical_covariates=
      train_dynamic_categorical_covariates,
      test_dynamic_categorical_covariates=
      test_dynamic_categorical_covariates,
      static_numerical_covariates=static_numerical_covariates,
      static_categorical_covariates=static_categorical_covariates,
  ).fit(
      ridge=ridge,
      one_hot_encoder_drop=None if ridge > 0 else "first",
      max_rows_per_col=max_rows_per_col,
      force_on_cpu=force_on_cpu,
      debug_info=False,
      assert_covariates=True,
      assert_covariate_shapes=True,
```

```
  )
  if normalize_xreg_target_per_input:
    xregs = renormalize(xregs, per_instance_stats)
  outputs = [
      (mean_output[self._horizon_start:(self._horizon_start + test_len)] +
       xreg)
      for mean_output, test_len, xreg in zip(mean_outputs, test_lens, xregs)
  ]

else:
  # Fit a model on the targets then forecast on the residuals via TimesFM.
  targets = [
      np.array(input_ts)[-train_len:]
      for input_ts, train_len in zip(inputs, train_lens)
  ]
  per_instance_stats = None
  if normalize_xreg_target_per_input:
    targets, per_instance_stats = normalize(targets)
  xregs, xregs_on_context, _, _, _ = xreg_lib.BatchedInContextXRegLinear(
      targets=targets,
      train_lens=train_lens,
      test_lens=test_lens,
      train_dynamic_numerical_covariates=train_dynamic_numerical_covariates,
      test_dynamic_numerical_covariates=test_dynamic_numerical_covariates,
      train_dynamic_categorical_covariates=
      train_dynamic_categorical_covariates,
      test_dynamic_categorical_covariates=
      test_dynamic_categorical_covariates,
      static_numerical_covariates=static_numerical_covariates,
      static_categorical_covariates=static_categorical_covariates,
  ).fit(
      ridge=ridge,
      one_hot_encoder_drop=None if ridge > 0 else "first",
      max_rows_per_col=max_rows_per_col,
      force_on_cpu=force_on_cpu,
      debug_info=True,
      assert_covariates=True,
      assert_covariate_shapes=True,
  )
  mean_outputs, _ = self.forecast(
      [
          target - xreg_on_context
          for target, xreg_on_context in zip(targets, xregs_on_context)
      ],
      freq,
      window_size,
      forecast_context_len,
      return_forecast_on_context=True,
```

```
    )
    outputs = [
        (mean_output[self._horizon_start:(self._horizon_start + test_len)] +
         xreg)
        for mean_output, test_len, xreg in zip(mean_outputs, test_lens, xregs)
    ]
    if normalize_xreg_target_per_input:
      outputs = renormalize(outputs, per_instance_stats)

  return outputs, xregs

def forecast_on_df(
    self,
    inputs: pd.DataFrame,
    freq: str,
    forecast_context_len: int = 0,
    value_name: str = "values",
    model_name: str = "timesfm",
    window_size: int | None = None,
    num_jobs: int = 1,
    verbose: bool = True,
) -> pd.DataFrame:
  """Forecasts on a list of time series.

  Args:
    inputs: A pd.DataFrame of all time series. The dataframe should have a
      `unique_id` column for identifying the time series, a `ds` column for
      timestamps and a value column for the time series values.
    freq: string valued `freq` of data. Notice this is different from the
      `freq` required by `forecast`. See `freq_map` for allowed values.
    forecast_context_len: If provided none zero, we take the last
      `forecast_context_len` time-points from each series as the forecast
      context instead of the `context_len` set by the model.
    value_name: The name of the value column.
    model_name: name of the model to be written into future df.
    window_size: window size of trend + residual decomposition. If None then
      we do not do decomposition.
    num_jobs: number of parallel processes to use for dataframe processing.
    verbose: output model states in terminal.

  Returns:
    Future forecasts dataframe.
  """
  if not ("unique_id" in inputs.columns and "ds" in inputs.columns and
          value_name in inputs.columns):
    raise ValueError(
        f"DataFrame must have unique_id, ds and {value_name} columns.")
  if not forecast_context_len:
```

```
  forecast_context_len = self.context_len
logging.info("Preprocessing dataframe.")
df_sorted = inputs.sort_values(by=["unique_id", "ds"])
new_inputs = []
uids = []
if num_jobs == 1:
  if verbose:
    print("Processing dataframe with single process.")
  for key, group in df_sorted.groupby("unique_id"):
    inp, uid = process_group(
        key,
        group,
        value_name,
        forecast_context_len,
    )
    new_inputs.append(inp)
    uids.append(uid)
else:
  if num_jobs == -1:
    num_jobs = multiprocessing.cpu_count()
  if verbose:
    print("Processing dataframe with multiple processes.")
  with multiprocessing.Pool(processes=num_jobs) as pool:
    results = pool.starmap(
        process_group,
        [(key, group, value_name, forecast_context_len)
         for key, group in df_sorted.groupby("unique_id")],
    )
  new_inputs, uids = zip(*results)
if verbose:
  print("Finished preprocessing dataframe.")
freq_inps = [freq_map(freq)] * len(new_inputs)
_, full_forecast = self.forecast(new_inputs,
                                 freq=freq_inps,
                                 window_size=window_size)
if verbose:
  print("Finished forecasting.")
fcst_df = make_future_dataframe(
    uids=uids,
    last_times=df_sorted.groupby("unique_id")["ds"].tail(1),
    h=self.horizon_len,
    freq=freq,
)
fcst_df[model_name] = full_forecast[:, 0:self.horizon_len, 0].reshape(-1, 1)

for i, q in enumerate(self.quantiles):
  q_col = f"{model_name}-q-{q}"
  fcst_df[q_col] = full_forecast[:, 0:self.horizon_len,
```

```
                                  1 + i].reshape(-1, 1)
  if q == 0.5:
    fcst_df[model_name] = fcst_df[q_col]
logging.info("Finished creating output dataframe.")
return fcst_df
```

11) src/timesfm/timesfm_jax.py

```
# Copyright 2024 Google LLC
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
"""TimesFM JAX forecast API for inference."""

import logging
import multiprocessing
import time
from os import path
from typing import Any, Sequence

import einshape as es
import jax
import jax.numpy as jnp
import numpy as np
from huggingface_hub import snapshot_download

from paxml import checkpoints, tasks_lib
from praxis import base_hyperparams, base_layer, pax_fiddle, py_utils, pytypes
from praxis.layers import normalizations, transformers
from timesfm import timesfm_base
from timesfm import patched_decoder

instantiate = base_hyperparams.instantiate
NestedMap = py_utils.NestedMap
JTensor = pytypes.JTensor

_TOL = 1e-6


class TimesFmJax(timesfm_base.TimesFmBase):
  """TimesFM forecast API for inference.

  This class is the scaffolding for calling TimesFM forecast. To properly use:
    1. Create an instance with the correct hyperparameters of a TimesFM model.
    2. Call `load_from_checkpoint` to load a compatible checkpoint.
```

3. Call `forecast` for inference.

Given the model size, this API does not shard the model weights for SPMD. All parallelism happens on the data dimension.

Compilation happens during the first time `forecast` is called and uses the `per_core_batch_size` to set and freeze the input signature. Subsequent calls to `forecast` reflect the actual inference latency.
"""

```
def _get_sample_inputs(self):
  return {
      "input_ts":
          jnp.zeros(
              (
                  self.per_core_batch_size,
                  self.context_len + self.output_patch_len,
              ),
              dtype=jnp.float32,
          ),
      "input_padding":
          jnp.zeros(
              (
                  self.per_core_batch_size,
                  self.context_len + self.output_patch_len,
              ),
              dtype=jnp.float32,
          ),
      "freq":
          jnp.zeros(
              (
                  self.per_core_batch_size,
                  1,
              ),
              dtype=jnp.int32,
          ),
  }

def __post_init__(self):
  self.num_cores = jax.local_device_count(self.backend)
  self.global_batch_size = self.per_core_batch_size * self.num_cores
  self._eval_context = base_layer.JaxContext.HParams(do_eval=True)
  self._pmapped_decode = None
  self._model = None
  self._train_state = None

def load_from_checkpoint(
    self,
```

```
    checkpoint: timesfm_base.TimesFmCheckpoint,
) -> None:
  """Loads a checkpoint and compiles the decoder."""
  checkpoint_type = (checkpoints.CheckpointType.FLAX
                     if checkpoint.type is None else checkpoint.type)
  checkpoint_path = checkpoint.path
  step = checkpoint.step
  repo_id = checkpoint.huggingface_repo_id
  if checkpoint_path is None:
    checkpoint_path = path.join(snapshot_download(repo_id), "checkpoints")
  # Rewrite the devices for Jax.
  self.mesh_shape = [1, self.num_cores, 1]
  self.mesh_name = ["replica", "data", "mdl"]

  self.model_p = pax_fiddle.Config(
      patched_decoder.PatchedTimeSeriesDecoder,
      name="patched_decoder",
      horizon_len=self.output_patch_len,
      patch_len=self.input_patch_len,
      model_dims=self.model_dims,
      hidden_dims=self.model_dims,
      residual_block_tpl=pax_fiddle.Config(patched_decoder.ResidualBlock),
      quantiles=self.quantiles,
      use_freq=True,
      stacked_transformer_params_tpl=pax_fiddle.Config(
          transformers.StackedTransformer,
          num_heads=self.num_heads,
          num_layers=self.num_layers,
          transformer_layer_params_tpl=pax_fiddle.Config(
              transformers.Transformer,
              ln_tpl=pax_fiddle.Config(normalizations.RmsNorm,),
          ),
      ),
  )

  self._key1, self._key2 = jax.random.split(jax.random.PRNGKey(42))
  self._model = None
  self._train_state = None
  self._pmapped_decode = None
  self._eval_context = base_layer.JaxContext.HParams(do_eval=True)
  try:
    multiprocessing.set_start_method("spawn")
  except RuntimeError:
    print("Multiprocessing context has already been set.")
  # Download the checkpoint from Hugging Face Hub if not given

  #  Initialize the model weights.
  self._logging("Constructing model weights.")
```

```
    start_time = time.time()
    self._model = instantiate(self.model_p)
    var_weight_hparams = self._model.abstract_init_with_metadata(
        self._get_sample_inputs(), do_eval=True)
    train_state_partition_specs = tasks_lib.create_state_partition_specs(
        var_weight_hparams,
        mesh_shape=self.mesh_shape,
        mesh_axis_names=self.mesh_name,
        discard_opt_states=True,
        learners=None,
    )
    train_state_local_shapes = tasks_lib.create_state_unpadded_shapes(
        var_weight_hparams,
        discard_opt_states=True,
        learners=None,
    )
    self._logging(
        f"Constructed model weights in {time.time() - start_time:.2f} seconds.")

    # Load the model weights.
    self._logging(f"Restoring checkpoint from {checkpoint_path}.")
    start_time = time.time()
    self._train_state = checkpoints.restore_checkpoint(
        train_state_local_shapes,
        checkpoint_dir=checkpoint_path,
        checkpoint_type=checkpoint_type,
        state_specs=train_state_partition_specs,
        step=step,
    )
    self._logging(
        f"Restored checkpoint in {time.time() - start_time:.2f} seconds.")
    self.jit_decode()

  def jit_decode(self):
    """Jitting decoding function."""

    # Initialize and jit the decode fn.
    def _decode(inputs):
      assert self._model is not None
      assert self._train_state is not None
      return self._model.apply(
          self._train_state.mdl_vars,
          inputs,
          horizon_len=self.horizon_len,
          output_patch_len=self.output_patch_len,
          max_len=self.context_len,
          return_forecast_on_context=True,
          rngs={
```

```
          base_layer.PARAMS: self._key1,
          base_layer.RANDOM: self._key2,
      },
      method=self._model.decode,
  )

self._logging("Jitting decoding.")
start_time = time.time()
self._pmapped_decode = jax.pmap(
    _decode,
    axis_name="batch",
    devices=jax.devices(self.backend),
    backend=self.backend,
    axis_size=self.num_cores,
)
with base_layer.JaxContext.new_context(hparams=self._eval_context):
  _ = self._pmapped_decode(
      NestedMap({
          "input_ts":
              jnp.zeros(
                  (
                      self.num_cores,
                      self.per_core_batch_size,
                      self.context_len,
                  ),
                  dtype=jnp.float32,
              ),
          "input_padding":
              jnp.zeros(
                  (
                      self.num_cores,
                      self.per_core_batch_size,
                      self.context_len + self.horizon_len,
                  ),
                  dtype=jnp.float32,
              ),
          "date_features":
              None,
          "freq":
              jnp.zeros(
                  (self.num_cores, self.per_core_batch_size, 1),
                  dtype=jnp.int32,
              ),
      }))
self._logging(f"Jitted decoding in {time.time() - start_time:.2f} seconds.")

def forecast(
    self,
```

```
    inputs: Sequence[Any],
    freq: Sequence[int] | None = None,
    window_size: int | None = None,
    forecast_context_len: int | None = None,
    return_forecast_on_context: bool = False,
    truncate_negative: bool = False,
) -> tuple[np.ndarray, np.ndarray]:
  """Forecasts on a list of time series.

  Args:
    inputs: list of time series forecast contexts. Each context time series
      should be in a format convertible to JTensor by `jnp.array`.
    freq: frequency of each context time series. 0 for high frequency
      (default), 1 for medium, and 2 for low. Notice this is different from
      the `freq` required by `forecast_on_df`.
    window_size: window size of trend + residual decomposition. If None then
      we do not do decomposition.
    forecast_context_len: optional max context length.
    return_forecast_on_context: True to return the forecast on the context
      when available, i.e. after the first input patch.
    truncate_negative: truncate to only non-negative values if all the contexts
      have non-negative values.

  Returns:
  A tuple for JTensors:
  - the mean forecast of size (# inputs, # forecast horizon),
  - the full forecast (mean + quantiles) of size
      (# inputs,  # forecast horizon, 1 + # quantiles).

  Raises:
  ValueError: If the checkpoint is not properly loaded.
  """
  if not self._train_state or not self._model:
    raise ValueError(
        "Checkpoint not loaded. Call `load_from_checkpoint` before"
        " `forecast`.")
  if forecast_context_len is None:
    fcontext_len = self.context_len
  else:
    fcontext_len = forecast_context_len
  inputs = [np.array(ts)[-fcontext_len:] for ts in inputs]
  inp_min = np.min([np.min(ts) for ts in inputs])

  if window_size is not None:
    new_inputs = []
    for ts in inputs:
      new_inputs.extend(timesfm_base.moving_average(ts, window_size))
    inputs = new_inputs
```

```
if freq is None:
  logging.info("No frequency provided via `freq`. Default to high (0).")
  freq = [0] * len(inputs)

input_ts, input_padding, inp_freq, pmap_pad = self._preprocess(inputs, freq)
with base_layer.JaxContext.new_context(hparams=self._eval_context):
  mean_outputs = []
  full_outputs = []
  assert input_ts.shape[0] % self.global_batch_size == 0
  for i in range(input_ts.shape[0] // self.global_batch_size):
    input_ts_in = jnp.array(input_ts[i * self.global_batch_size:(i + 1) *
                                     self.global_batch_size])
    input_padding_in = jnp.array(
        input_padding[i * self.global_batch_size:(i + 1) *
                      self.global_batch_size],)
    inp_freq_in = jnp.array(
        inp_freq[i * self.global_batch_size:(i + 1) *
                 self.global_batch_size, :],
        dtype=jnp.int32,
    )
    pmapped_inputs = NestedMap({
        "input_ts":
            es.jax_einshape(
                "(db)...->db...",
                input_ts_in,
                d=self.num_cores,
            ),
        "input_padding":
            es.jax_einshape(
                "(db)...->db...",
                input_padding_in,
                d=self.num_cores,
            ),
        "date_features":
            None,
        "freq":
            es.jax_einshape(
                "(db)...->db...",
                inp_freq_in,
                d=self.num_cores,
            ),
    })
    mean_output, full_output = self._pmapped_decode(pmapped_inputs)
    if not return_forecast_on_context:
      mean_output = mean_output[:, :, self._horizon_start:, ...]
      full_output = full_output[:, :, self._horizon_start:, ...]
    mean_output = es.jax_einshape("db...->(db)...",
```

```
                                  mean_output,
                                  d=self.num_cores)
    full_output = es.jax_einshape("db...->(db)...",
                                  full_output,
                                  d=self.num_cores)
    mean_output = np.array(mean_output)
    full_output = np.array(full_output)
    mean_outputs.append(mean_output)
    full_outputs.append(full_output)

mean_outputs = np.concatenate(mean_outputs, axis=0)
full_outputs = np.concatenate(full_outputs, axis=0)

if pmap_pad > 0:
  mean_outputs = mean_outputs[:-pmap_pad, ...]
  full_outputs = full_outputs[:-pmap_pad, ...]

if window_size is not None:
  mean_outputs = mean_outputs[0::2, ...] + mean_outputs[1::2, ...]
  full_outputs = full_outputs[0::2, ...] + full_outputs[1::2, ...]
if inp_min >= 0 and truncate_negative:
  mean_outputs = np.maximum(mean_outputs, 0.0)
  full_outputs = np.maximum(full_outputs, 0.0)
return mean_outputs, full_outputs
```

12) src/timesfm/timesfm_torch.py

```
# Copyright 2024 Google LLC
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
"""TimesFM pytorch forecast API for inference."""

import logging
from os import path
from typing import Any, Sequence

import numpy as np
import torch
from huggingface_hub import snapshot_download
from timesfm import timesfm_base

from . import pytorch_patched_decoder as ppd

_TOL = 1e-6


class TimesFmTorch(timesfm_base.TimesFmBase):
  """TimesFM forecast API for inference."""

  def __post_init__(self):
    self._model_config = ppd.TimesFMConfig(
        num_layers=self.num_layers,
        num_heads=self.num_heads,
        hidden_size=self.model_dims,
        intermediate_size=self.model_dims,
        patch_len=self.input_patch_len,
        horizon_len=self.output_patch_len,
        head_dim=self.model_dims // self.num_heads,
        quantiles=self.quantiles,
    )
    self._model = None
    self.num_cores = 1
    self.global_batch_size = self.per_core_batch_size
```

```
  self._device = torch.device("cuda:0" if (
      torch.cuda.is_available() and self.backend == "gpu") else "cpu")

def load_from_checkpoint(
    self,
    checkpoint: timesfm_base.TimesFmCheckpoint,
) -> None:
  """Loads a checkpoint and compiles the decoder."""
  checkpoint_path = checkpoint.path
  repo_id = checkpoint.huggingface_repo_id
  if checkpoint_path is None:
    checkpoint_path = path.join(snapshot_download(repo_id),
                                "torch_model.ckpt")
  self._model = ppd.PatchedTimeSeriesDecoder(self._model_config)
  loaded_checkpoint = torch.load(checkpoint_path, weights_only=True)
  logging.info("Loading checkpoint from %s", checkpoint_path)
  self._model.load_state_dict(loaded_checkpoint)
  logging.info("Sending checkpoint to device %s", f"{self._device}")
  self._model.to(self._device)
  self._model.eval()
  # TODO: add compilation.

def forecast(
    self,
    inputs: Sequence[Any],
    freq: Sequence[int] | None = None,
    window_size: int | None = None,
    forecast_context_len: int | None = None,
    return_forecast_on_context: bool = False,
    truncate_negative: bool = False,
) -> tuple[np.ndarray, np.ndarray]:
  """Forecasts on a list of time series.

      Args:
        inputs: list of time series forecast contexts. Each context time series
          should be in a format convertible to JTensor by `jnp.array`.
        freq: frequency of each context time series. 0 for high frequency
          (default), 1 for medium, and 2 for low. Notice this is different from
          the `freq` required by `forecast_on_df`.
        window_size: window size of trend + residual decomposition. If None then
          we do not do decomposition.
        forecast_context_len: optional max context length.
        return_forecast_on_context: True to return the forecast on the context
          when available, i.e. after the first input patch.
        truncate_negative: truncate to only non-negative values if all the contexts
          have non-negative values.

      Returns:
```

```
    A tuple for JTensors:
    - the mean forecast of size (# inputs, # forecast horizon),
    - the full forecast (mean + quantiles) of size
        (# inputs,  # forecast horizon, 1 + # quantiles).

    Raises:
    ValueError: If the checkpoint is not properly loaded.
    """
  if not self._model:
    raise ValueError(
        "Checkpoint not loaded. Call `load_from_checkpoint` before"
        " `forecast`.")
  if forecast_context_len is None:
    fcontext_len = self.context_len
  else:
    fcontext_len = forecast_context_len
  inputs = [np.array(ts)[-fcontext_len:] for ts in inputs]
  inp_min = np.min([np.min(ts) for ts in inputs])

  if window_size is not None:
    new_inputs = []
    for ts in inputs:
      new_inputs.extend(timesfm_base.moving_average(ts, window_size))
    inputs = new_inputs

  if freq is None:
    logging.info("No frequency provided via `freq`. Default to high (0).")
    freq = [0] * len(inputs)

  input_ts, input_padding, inp_freq, pmap_pad = self._preprocess(inputs, freq)
  with torch.no_grad():
    mean_outputs = []
    full_outputs = []
    assert input_ts.shape[0] % self.global_batch_size == 0
    for i in range(input_ts.shape[0] // self.global_batch_size):
      input_ts_in = torch.from_numpy(
          np.array(input_ts[i * self.global_batch_size:(i + 1) *
                            self.global_batch_size],
                   dtype=np.float32)).to(self._device)
      input_padding_in = torch.from_numpy(
          np.array(input_padding[i * self.global_batch_size:(i + 1) *
                                 self.global_batch_size],
                   dtype=np.float32)).to(self._device)
      inp_freq_in = torch.from_numpy(
          np.array(inp_freq[
              i * self.global_batch_size:(i + 1) * self.global_batch_size,
              :,
          ],
```

```
            dtype=np.int32)).long().to(self._device)
      mean_output, full_output = self._model.decode(
          input_ts=input_ts_in,
          paddings=input_padding_in,
          freq=inp_freq_in,
          horizon_len=self.horizon_len,
          return_forecast_on_context=return_forecast_on_context,
      )
      mean_output = mean_output.detach().cpu().numpy()
      full_output = full_output.detach().cpu().numpy()
      mean_output = np.array(mean_output)
      full_output = np.array(full_output)
      mean_outputs.append(mean_output)
      full_outputs.append(full_output)

    mean_outputs = np.concatenate(mean_outputs, axis=0)
    full_outputs = np.concatenate(full_outputs, axis=0)

    if pmap_pad > 0:
      mean_outputs = mean_outputs[:-pmap_pad, ...]
      full_outputs = full_outputs[:-pmap_pad, ...]

    if window_size is not None:
      mean_outputs = mean_outputs[0::2, ...] + mean_outputs[1::2, ...]
      full_outputs = full_outputs[0::2, ...] + full_outputs[1::2, ...]
    if inp_min >= 0 and truncate_negative:
      mean_outputs = np.maximum(mean_outputs, 0.0)
      full_outputs = np.maximum(full_outputs, 0.0)
    return mean_outputs, full_outputs
```

13) src/timesfm/xreg_lib.py

```
# Copyright 2024 Google LLC
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
"""Helper functions for in-context covariates and regression."""

import itertools
import math
from typing import Any, Iterable, Literal, Mapping, Sequence

import jax
import jax.numpy as jnp
import numpy as np
from sklearn import preprocessing

Category = int | str

_TOL = 1e-6
XRegMode = Literal["timesfm + xreg", "xreg + timesfm"]


def _unnest(nested: Sequence[Sequence[Any]]) -> np.ndarray:
  return np.array(list(itertools.chain.from_iterable(nested)))


def _repeat(elements: Iterable[Any], counts: Iterable[int]) -> np.ndarray:
  return np.array(
      list(
          itertools.chain.from_iterable(map(itertools.repeat, elements,
                                            counts))))


def _to_padded_jax_array(x: np.ndarray) -> jax.Array:
  if x.ndim == 1:
    (i,) = x.shape
    di = 2**math.ceil(math.log2(i)) - i
    return jnp.pad(x, ((0, di),), mode="constant", constant_values=0.0)
```

```
  elif x.ndim == 2:
    i, j = x.shape
    di = 2**math.ceil(math.log2(i)) - i
    dj = 2**math.ceil(math.log2(j)) - j
    return jnp.pad(x, ((0, di), (0, dj)), mode="constant", constant_values=0.0)
  else:
    raise ValueError(f"Unsupported array shape: {x.shape}")


class BatchedInContextXRegBase:
  """Helper class for in-context regression covariate formatting.

  Attributes:
    targets: List of targets (responses) of the in-context regression.
    train_lens: List of lengths of each target vector from the context.
    test_lens: List of lengths of each forecast horizon.
    train_dynamic_numerical_covariates: Dict of covariate names mapping to the
      dynamic numerical covariates of each forecast task on the context. Their
      lengths should match the corresponding lengths in `train_lens`.
    train_dynamic_categorical_covariates: Dict of covariate names mapping to the
      dynamic categorical covariates of each forecast task on the context. Their
      lengths should match the corresponding lengths in `train_lens`.
    test_dynamic_numerical_covariates: Dict of covariate names mapping to the
      dynamic numerical covariates of each forecast task on the horizon. Their
      lengths should match the corresponding lengths in `test_lens`.
    test_dynamic_categorical_covariates: Dict of covariate names mapping to the
      dynamic categorical covariates of each forecast task on the horizon. Their
      lengths should match the corresponding lengths in `test_lens`.
    static_numerical_covariates: Dict of covariate names mapping to the static
      numerical covariates of each forecast task.
    static_categorical_covariates: Dict of covariate names mapping to the static
      categorical covariates of each forecast task.
  """

  def __init__(
      self,
      targets: Sequence[Sequence[float]],
      train_lens: Sequence[int],
      test_lens: Sequence[int],
      train_dynamic_numerical_covariates: (
          Mapping[str, Sequence[Sequence[float]]] | None) = None,
      train_dynamic_categorical_covariates: (
          Mapping[str, Sequence[Sequence[Category]]] | None) = None,
      test_dynamic_numerical_covariates: (
          Mapping[str, Sequence[Sequence[float]]] | None) = None,
      test_dynamic_categorical_covariates: (
          Mapping[str, Sequence[Sequence[Category]]] | None) = None,
      static_numerical_covariates: Mapping[str, Sequence[float]] | None = None,
```

```
    static_categorical_covariates: (Mapping[str, Sequence[Category]] |
                                    None) = None,
) -> None:
  """Initializes with the exogenous covariate inputs.

  Here we use model fitting language to refer to the context as 'train' and
  the horizon as 'test'. We assume batched inputs. To properly format the
  request:

   - `train_lens` represents the contexts in the batch. Targets and all train
   dynamic covariates should have the same lengths as the corresponding
   elements
   in `train_lens`. Notice each `train_len` can be different from the exact
   length of the corresponding context depending on how much of the context is
   used for fitting the in-context model.
   - `test_lens` represents the horizon lengths in the batch. All tesdt
   dynamic
   covariates should have the same lengths as the corresponding elements in
   `test_lens`.
   - Static covariates should be one for each input.
   - For train and test dynamic covariates, they should have the same
   covariate
   names.

   Pass an empty dict {} for a covariate type if it is not present.

   Example:
     Here is a set of valid inputs whose schema can be used for reference.
     ```
     targets = [
         [0.0, 0.1, 0.2],
         [0.0, 0.1, 0.2, 0.3],
     ] # Two inputs in this batch.
     train_lens = [3, 4]
     test_lens = [2, 5] # Forecast horizons 2 and 5 respectively.
     train_dynamic_numerical_covariates = {
         "cov_1_dn": [[0.0, 0.5, 1.0], [0.0, 0.5, 1.0, 1.5]],
         "cov_2_dn": [[0.0, 1.5, 1.0], [0.0, 1.5, 1.0, 2.5]],
     } # Each train dynamic covariate has 3 and 4 elements respectively.
     test_dynamic_numerical_covariates = {
         "cov_1_dn": [[0.1, 0.6], [0.1, 0.6, 1.1, 1.6, 2.4]],
         "cov_2_dn": [[0.1, 1.1], [0.1, 1.6, 1.1, 2.6, 10.0]],
     } # Each test dynamic covariate has 2 and 5 elements respectively.
     train_dynamic_categorical_covariates = {
         "cov_1_dc": [[0, 1, 0], [0, 1, 2, 3]],
         "cov_2_dc": [["good", "bad", "good"], ["good", "good", "bad",
         "bad"]],
     }
```
```

```
  test_dynamic_categorical_covariates = {
      "cov_1_dc": [[1, 0], [1, 0, 2, 3, 1]],
      "cov_2_dc": [["bad", "good"], ["bad", "bad", "bad", "bad", "bad"]],
  }
  static_numerical_covariates = {
      "cov_1_sn": [0.0, 3.0],
      "cov_2_sn": [2.0, 1.0],
      "cov_3_sn": [1.0, 2.0],
  }  # Each static covariate has 1 element for each input.
  static_categorical_covariates = {
      "cov_1_sc": ["apple", "orange"],
      "cov_2_sc": [2, 3],
  }
  ```

Args:
  targets: List of targets (responses) of the in-context regression.
  train_lens: List of lengths of each target vector from the context.
  test_lens: List of lengths of each forecast horizon.
  train_dynamic_numerical_covariates: Dict of covariate names mapping to the
    dynamic numerical covariates of each forecast task on the context. Their
    lengths should match the corresponding lengths in `train_lens`.
  train_dynamic_categorical_covariates: Dict of covariate names mapping to
    the dynamic categorical covariates of each forecast task on the context.
    Their lengths should match the corresponding lengths in `train_lens`.
  test_dynamic_numerical_covariates: Dict of covariate names mapping to the
    dynamic numerical covariates of each forecast task on the horizon. Their
    lengths should match the corresponding lengths in `test_lens`.
  test_dynamic_categorical_covariates: Dict of covariate names mapping to
    the dynamic categorical covariates of each forecast task on the horizon.
    Their lengths should match the corresponding lengths in `test_lens`.
  static_numerical_covariates: Dict of covariate names mapping to the static
    numerical covariates of each forecast task.
  static_categorical_covariates: Dict of covariate names mapping to the
    static categorical covariates of each forecast task.
"""
self.targets = targets
self.train_lens = train_lens
self.test_lens = test_lens
self.train_dynamic_numerical_covariates = (
    train_dynamic_numerical_covariates or {})
self.train_dynamic_categorical_covariates = (
    train_dynamic_categorical_covariates or {})
self.test_dynamic_numerical_covariates = (test_dynamic_numerical_covariates
                                          or {})
self.test_dynamic_categorical_covariates = (
    test_dynamic_categorical_covariates or {})
self.static_numerical_covariates = static_numerical_covariates or {}
```
```

```
  self.static_categorical_covariates = static_categorical_covariates or {}

def _assert_covariates(self, assert_covariate_shapes: bool = False) -> None:
  """Verifies the validity of the covariate inputs."""

  # Check presence.
  if (self.train_dynamic_numerical_covariates and
      not self.test_dynamic_numerical_covariates) or (
          not self.train_dynamic_numerical_covariates and
          self.test_dynamic_numerical_covariates):
    raise ValueError(
        "train_dynamic_numerical_covariates and"
        " test_dynamic_numerical_covariates must be both present or both"
        " absent.")

  if (self.train_dynamic_categorical_covariates and
      not self.test_dynamic_categorical_covariates) or (
          not self.train_dynamic_categorical_covariates and
          self.test_dynamic_categorical_covariates):
    raise ValueError(
        "train_dynamic_categorical_covariates and"
        " test_dynamic_categorical_covariates must be both present or both"
        " absent.")

  # Check keys.
  for dict_a, dict_b, dict_a_name, dict_b_name in (
      (
          self.train_dynamic_numerical_covariates,
          self.test_dynamic_numerical_covariates,
          "train_dynamic_numerical_covariates",
          "test_dynamic_numerical_covariates",
      ),
      (
          self.train_dynamic_categorical_covariates,
          self.test_dynamic_categorical_covariates,
          "train_dynamic_categorical_covariates",
          "test_dynamic_categorical_covariates",
      ),
  ):
    if w := set(dict_a.keys()) - set(dict_b.keys()):
      raise ValueError(
          f"{dict_a_name} has keys not present in {dict_b_name}: {w}")
    if w := set(dict_b.keys()) - set(dict_a.keys()):
      raise ValueError(
          f"{dict_b_name} has keys not present in {dict_a_name}: {w}")

  # Check shapes.
  if assert_covariate_shapes:
```

```
if len(self.targets) != len(self.train_lens):
  raise ValueError(
      "targets and train_lens must have the same number of elements.")

if len(self.train_lens) != len(self.test_lens):
  raise ValueError(
      "train_lens and test_lens must have the same number of elements.")

for i, (target, train_len) in enumerate(zip(self.targets,
                                            self.train_lens)):
  if len(target) != train_len:
    raise ValueError(
        f"targets[{i}] has length {len(target)} != expected {train_len}.")

for key, values in self.static_numerical_covariates.items():
  if len(values) != len(self.train_lens):
    raise ValueError(
        f"static_numerical_covariates has key {key} with number of"
        f" examples {len(values)} != expected {len(self.train_lens)}.")

for key, values in self.static_categorical_covariates.items():
  if len(values) != len(self.train_lens):
    raise ValueError(
        f"static_categorical_covariates has key {key} with number of"
        f" examples {len(values)} != expected {len(self.train_lens)}.")

for lens, dict_cov, dict_cov_name in (
    (
        self.train_lens,
        self.train_dynamic_numerical_covariates,
        "train_dynamic_numerical_covariates",
    ),
    (
        self.train_lens,
        self.train_dynamic_categorical_covariates,
        "train_dynamic_categorical_covariates",
    ),
    (
        self.test_lens,
        self.test_dynamic_numerical_covariates,
        "test_dynamic_numerical_covariates",
    ),
    (
        self.test_lens,
        self.test_dynamic_categorical_covariates,
        "test_dynamic_categorical_covariates",
    ),
):
```

```
    for key, cov_values in dict_cov.items():
      if len(cov_values) != len(lens):
        raise ValueError(
            f"{dict_cov_name} has key {key} with number of examples"
            f" {len(cov_values)} != expected {len(lens)}.")
      for i, cov_value in enumerate(cov_values):
        if len(cov_value) != lens[i]:
          raise ValueError(
              f"{dict_cov_name} has key {key} with its {i}-th example"
              f" length {len(cov_value)} != expected {lens[i]}.")

def create_covariate_matrix(
    self,
    one_hot_encoder_drop: str | None = "first",
    use_intercept: bool = True,
    assert_covariates: bool = False,
    assert_covariate_shapes: bool = False,
) -> tuple[np.ndarray, np.ndarray, np.ndarray]:
  """Creates target vector and covariate matrices for in context regression.

  Here we use model fitting language to refer to the context as 'train' and
  the horizon as 'test'.

  Args:
    one_hot_encoder_drop: Which drop strategy to use for the one hot encoder.
    use_intercept: Whether to prepare an intercept (all 1) column in the
      matrices.
    assert_covariates: Whether to assert the validity of the covariate inputs.
    assert_covariate_shapes: Whether to assert the shapes of the covariate
      inputs when `assert_covariates` is True.

  Returns:
    A tuple of the target vector, the covariate matrix for the context, and
    the covariate matrix for the horizon.
  """
  if assert_covariates:
    self._assert_covariates(assert_covariate_shapes)

  x_train, x_test = [], []

  # Numerical features.
  for name in sorted(self.train_dynamic_numerical_covariates):
    x_train.append(
        _unnest(self.train_dynamic_numerical_covariates[name])[:, np.newaxis])
    x_test.append(
        _unnest(self.test_dynamic_numerical_covariates[name])[:, np.newaxis])

  for covs in self.static_numerical_covariates.values():
```

```
      x_train.append(_repeat(covs, self.train_lens)[:, np.newaxis])
      x_test.append(_repeat(covs, self.test_lens)[:, np.newaxis])

    if x_train:
      x_train = np.concatenate(x_train, axis=1)
      x_test = np.concatenate(x_test, axis=1)

      # Normalize for robustness.
      x_mean = np.mean(x_train, axis=0, keepdims=True)
      x_std = np.where((w := np.std(x_train, axis=0, keepdims=True)) > _TOL, w,
                       1.0)
      x_train = [(x_train - x_mean) / x_std]
      x_test = [(x_test - x_mean) / x_std]

    # Categorical features. Encode one by one.
    one_hot_encoder = preprocessing.OneHotEncoder(
        drop=one_hot_encoder_drop,
        sparse_output=False,
        handle_unknown="ignore",
    )
    for name in sorted(self.train_dynamic_categorical_covariates.keys()):
      ohe_train = _unnest(
          self.train_dynamic_categorical_covariates[name])[:, np.newaxis]
      ohe_test = _unnest(
          self.test_dynamic_categorical_covariates[name])[:, np.newaxis]
      x_train.append(np.array(one_hot_encoder.fit_transform(ohe_train)))
      x_test.append(np.array(one_hot_encoder.transform(ohe_test)))

    for covs in self.static_categorical_covariates.values():
      ohe = one_hot_encoder.fit_transform(np.array(covs)[:, np.newaxis])
      x_train.append(_repeat(ohe, self.train_lens))
      x_test.append(_repeat(ohe, self.test_lens))

    x_train = np.concatenate(x_train, axis=1)
    x_test = np.concatenate(x_test, axis=1)

    if use_intercept:
      x_train = np.pad(x_train, ((0, 0), (1, 0)), constant_values=1.0)
      x_test = np.pad(x_test, ((0, 0), (1, 0)), constant_values=1.0)

    return _unnest(self.targets), x_train, x_test

  def fit(self) -> Any:
    raise NotImplementedError("Fit is not implemented.")


class BatchedInContextXRegLinear(BatchedInContextXRegBase):
  """Linear in-context regression model."""
```

```
def fit(
    self,
    ridge: float = 0.0,
    one_hot_encoder_drop: str | None = "first",
    use_intercept: bool = True,
    force_on_cpu: bool = False,
    max_rows_per_col: int = 0,
    max_rows_per_col_sample_seed: int = 42,
    debug_info: bool = False,
    assert_covariates: bool = False,
    assert_covariate_shapes: bool = False,
) -> (list[np.ndarray] | tuple[list[np.ndarray], list[np.ndarray], jax.Array,
                               jax.Array, jax.Array]):
  """Fits a linear model for in-context regression.

  Args:
    ridge: A non-negative value for specifying the ridge regression penalty.
      If 0 is provided, fallback to ordinary least squares. Note this penalty
      is added to the normalized covariate matrix.
    one_hot_encoder_drop: Which drop strategy to use for the one hot encoder.
    use_intercept: Whether to prepare an intercept (all 1) column in the
      matrices.
    force_on_cpu: Whether to force execution on cpu for accelerator machines.
    max_rows_per_col: How many rows to subsample per column. 0 for no
      subsampling. This is for speeding up model fitting.
    max_rows_per_col_sample_seed: The seed for the subsampling if needed by
      `max_rows_per_col`.
    debug_info: Whether to return debug info.
    assert_covariates: Whether to assert the validity of the covariate inputs.
    assert_covariate_shapes: Whether to assert the shapes of the covariate
      inputs when `assert_covariates` is True.

  Returns:
    If `debug_info` is False:
      The linear fits on the horizon.
    If `debug_info` is True:
      A tuple of:
      - the linear fits on the horizon,
      - the linear fits on the context,
      - the flattened target vector,
      - the covariate matrix for the context, and
      - the covariate matrix for the horizon.
  """
  flat_targets, x_train_raw, x_test = self.create_covariate_matrix(
      one_hot_encoder_drop=one_hot_encoder_drop,
      use_intercept=use_intercept,
      assert_covariates=assert_covariates,
```

```
    assert_covariate_shapes=assert_covariate_shapes,
  )

  x_train = x_train_raw.copy()
  if max_rows_per_col:
    nrows, ncols = x_train.shape
    if nrows > (w := ncols * max_rows_per_col):
      subsample = jax.random.choice(
          jax.random.PRNGKey(max_rows_per_col_sample_seed),
          nrows,
          (w,),
          replace=False,
      )
      x_train = x_train[subsample]
      flat_targets = flat_targets[subsample]

  device = jax.devices("cpu")[0] if force_on_cpu else None
  # Runs jitted version of the solvers which are quicker at the cost of
  # running jitting during the first time calling. Re-jitting happens whenever
  # new (padded) shapes are encountered.
  # Ocassionally it helps with the speed and the accuracy if we force single
  # thread execution on cpu for accelerator machines:
  # 1. Avoid moving data to accelarator memory.
  # 2. Avoid precision loss if any.
  with jax.default_device(device):
    x_train_raw = _to_padded_jax_array(x_train_raw)
    x_train = _to_padded_jax_array(x_train)
    flat_targets = _to_padded_jax_array(flat_targets)
    x_test = _to_padded_jax_array(x_test)
    beta_hat = (jnp.linalg.pinv(
        x_train.T @ x_train + ridge * jnp.eye(x_train.shape[1]),
        hermitian=True,
    ) @ x_train.T @ flat_targets)
    y_hat = x_test @ beta_hat
    y_hat_context = x_train_raw @ beta_hat if debug_info else None

  outputs = []
  outputs_context = []

  # Reconstruct the ragged 2-dim batched forecasts from flattened linear fits.
  train_index, test_index = 0, 0
  for train_index_delta, test_index_delta in zip(self.train_lens,
                                                 self.test_lens):
    outputs.append(np.array(y_hat[test_index:(test_index +
                                              test_index_delta)]))
    if debug_info:
      outputs_context.append(
          np.array(y_hat_context[train_index:(train_index +
```

```
                                           train_index_delta)]))
  train_index += train_index_delta
  test_index += test_index_delta

if debug_info:
  return outputs, outputs_context, flat_targets, x_train, x_test
else:
  return outputs
```

14) peft/readme.md (Fine-Tuning Pipeline)

This folder contains a generic fine-tuning pipeline designed to support multiple PEFT fine-tuning strategies.

Features

Supported Fine-Tuning Strategies:

Full Fine-Tuning: Adjusts all parameters of the model during training.

Linear Probing: Fine-tunes only the residual blocks and the embedding layer, leaving other parameters unchanged.

LoRA (Low-Rank Adaptation): A memory-efficient method that fine-tunes a small number of parameters by decomposing the weight matrices into low-rank matrices.

DoRA (Directional LoRA): An extension of LoRA that decomposes pre-trained weights into magnitude and direction components. It uses LoRA for directional adaptation, enhancing learning capacity and stability without additional inference overhead.

Usage

Fine-Tuning Script

The provided finetune.py script allows you to fine-tune a model with specific configurations. You can customize various parameters to suit your dataset and desired fine-tuning strategy.

Example Usage:

source finetune.sh

This script runs the finetune.py file with a predefined set of hyperparameters for the model. You can adjust the parameters in the script as needed.

Available Options

Run the script with the --help flag to see a full list of available options and their descriptions:

python3 finetune.py --help

Script Configuration You can modify the following key parameters directly in the finetune.sh script: Fine-Tuning Strategy: Toggle between full fine-tuning, LoRA [--use-lora], DoRA [[--use-dora]], or Linear Probing [--use-linear-probing].

Performance Comparison

The figure below compares the performance of LoRA/DoRA against Linear Probing under the following conditions:

Method	# Trainable Parameters	ETTM1 MSE	ETTM1 MAE	ETTH1 MSE	ETTH1 MAE	Exchange Rate MSE	Exchange Rate MAE
Base Model	–	0.4149	0.4143	0.4887	0.4504	0.1551	0.2776
Full Fine-Tuning	203.56M	0.3391	0.3692	0.4574	0.4377	0.1311	0.2564
Linear Probing	6.72M	0.3345	0.3647	0.4539	0.4366	0.1354	0.2606
LoRA *(r=1)*	0.31M	0.3333	0.3696	0.4555	0.4377	**0.1253**	**0.2488**
LoRA *(r=2)*	0.61M	0.3306	0.3638	0.4549	0.4366	0.1306	0.2541
LoRA *(r=4)*	1.23M	0.3304	0.3660	0.4572	0.4367	0.1300	0.2544
LoRA *(r=8)*	2.46M	**0.3271**	0.3639	**0.4531**	**0.4361**	0.1288	0.2539
DoRA *(r=1)*	0.46M	0.3341	0.3701	0.4552	0.4377	**0.1253**	**0.2489**
DoRA *(r=2)*	0.77M	0.3304	**0.3636**	0.4548	0.4365	0.1306	0.2540
DoRA *(r=4)*	1.38M	0.3299	0.3659	0.4563	0.4365	0.1304	0.2549
DoRA *(r=8)*	2.61M	**0.3270**	**0.3638**	**0.4537**	**0.4362**	0.1290	0.2543

15) peft/finetune.py

```
# Copyright 2024 The Google Research Authors.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""
Finetune pipeline.
"""
import gc
import logging
import warnings
from datetime import datetime
from typing import Tuple

import jax
import jax.numpy as jnp
import numpy as np
import pandas as pd
import typer
import wandb
from jax import numpy as jnp
from paxml import checkpoint_types, checkpoints, learners, tasks_lib, trainer_lib
from praxis import optimizers, pax_fiddle, py_utils, schedules
from rich import print
from tqdm import tqdm
from typing_extensions import Annotated

from adapter.utils import get_adapter_params, load_adapter_layer
from timesfm import TimesFm, data_loader, patched_decoder

NestedMap = py_utils.NestedMap


warnings.filterwarnings("ignore")
cmdstanpy_logger = logging.getLogger("cmdstanpy")
absl_logger = logging.getLogger("absl")
cmdstanpy_logger.disabled = True
```

```
absl_logger.disabled = True

"""
TimesFM model config. These are fixed since pre-training was done
with this configuration.
"""
INPUT_PATCH_LEN = 32
OUTPUT_PATCH_LEN = 128
NUM_LAYERS = 20
MODEL_DIMS = 1280

QUANTILES = list(np.arange(1, 10) / 10.0)
EPS = 1e-7
RANDOM_SEED = 1234


def finetune(
    *,
    model_name: Annotated[
        str, typer.Option(help="Specify the name of the huggingface model.")
    ] = "google/timesfm-1.0-200m",
    checkpoint_path: Annotated[
        str, typer.Option(help="The path to the local model checkpoint.")
    ] = None,
    datetime_col: Annotated[str, typer.Option(help="Column having datetime.")] = "ds",
    ts_cols: Annotated[
        list[str], typer.Option(help="Columns of time-series features.")
    ] = [],
    normalize: Annotated[
        bool, typer.Option(help="Normalize data for eval or not")
    ] = True,
    context_len: Annotated[int, typer.Option(help="Length of the context window")],
    horizon_len: Annotated[int, typer.Option(help="Prediction length.")],
    freq: Annotated[
        str,
        typer.Option(
            ...,
            help="Frequency Map Str",
        ),
    ],
    data_path: Annotated[str, typer.Option(help="Path to dataset csv")],
    boundaries: Annotated[
        Tuple[int, int, int],
        typer.Option(
            help="boundaries of dataset to train, val, test",
        ),
    ] = (0, 0, 0),
    backend: Annotated[str, typer.Option(help="Backend device: cpu, gpu, tpu")],
```

```
    batch_size: Annotated[
        int, typer.Option(help="Batch size for the randomly sampled batch")
    ] = 16,
    num_epochs: Annotated[int, typer.Option(help="Number of epochs")],
    learning_rate: Annotated[float, typer.Option(help="adam optimizer learning rate")],
    adam_epsilon: Annotated[float, typer.Option(help="adam optimizer epsilon")],
    adam_clip_threshold: Annotated[
        float, typer.Option(help="adam optimizer clip threshold")
    ],
    cos_initial_decay_value: Annotated[
        float, typer.Option(help="cosine initial decay value")
    ],
    cos_final_decay_value: Annotated[
        float, typer.Option(help="cosine final decay value")
    ],
    cos_decay_steps: Annotated[int, typer.Option(help="Number of cosine decay steps")],
    ema_decay: Annotated[float, typer.Option(help="Exponential moving average decay")],
    early_stop_patience: Annotated[
        int, typer.Option(..., help="Early stopping patience")
    ] = 5,
    use_lora: Annotated[
        bool,
        typer.Option(
            help="Train low rank adapters for stacked transformer block",
        ),
    ] = False,
    lora_rank: Annotated[
        int,
        typer.Option(
            help="LoRA Rank",
        ),
    ] = 8,
    lora_target_modules: Annotated[
        str,
        typer.Option(
            help="LoRA target modules of the transformer block. Allowed values: [all, attention,
mlp]"
        ),
    ] = "all",
    use_dora: Annotated[
        bool,
        typer.Option(
            help="Apply DoRA strategy along with LoRA.",
        ),
    ] = False,
    use_linear_probing: Annotated[
        bool,
        typer.Option(
```

```
            help="Linear Probing. Train only input/output and embedding params. Freeze
params in stack transformer block.",
        ),
    ] = False,
    checkpoint_dir: Annotated[
        str, typer.Option(help="Checkpoint directory")
    ] = "./checkpoints",
    wandb_project: Annotated[
        str, typer.Option(help="Weights & Biases project name")
    ] = "google_timesfm_finetune",
) -> None:
    key = jax.random.PRNGKey(seed=RANDOM_SEED)
    wandb.init(project=wandb_project, config=locals())

    data_df = pd.read_csv(open(data_path, "r"))

    if boundaries == (0, 0, 0):
        # Default boundaries: train 60%, val 20%, test 20%
        boundaries = [
            int(len(data_df) * 0.6),
            int(len(data_df) * 0.8),
            len(data_df) - 1,
        ]

    ts_cols = [col for col in data_df.columns if col != datetime_col]

    dtl = data_loader.TimeSeriesdata(
        data_path=data_path,
        datetime_col=datetime_col,
        num_cov_cols=None,
        cat_cov_cols=None,
        ts_cols=np.array(ts_cols),
        train_range=[0, boundaries[0]],
        val_range=[boundaries[0], boundaries[1]],
        test_range=[boundaries[1], boundaries[2]],
        hist_len=context_len,
        pred_len=horizon_len,
        batch_size=batch_size,
        freq=freq,
        normalize=normalize,
        epoch_len=None,
        holiday=False,
        permute=False,
    )

    train_batches = dtl.tf_dataset(mode="train", shift=1).batch(batch_size)
    val_batches = dtl.tf_dataset(mode="val", shift=horizon_len)
```

```
for tbatch in tqdm(train_batches.as_numpy_iterator()):
    pass

tfm = TimesFm(
    context_len=context_len,
    horizon_len=horizon_len,
    input_patch_len=INPUT_PATCH_LEN,
    output_patch_len=OUTPUT_PATCH_LEN,
    num_layers=NUM_LAYERS,
    model_dims=MODEL_DIMS,
    backend=backend,
    per_core_batch_size=batch_size,
    quantiles=QUANTILES,
)

if checkpoint_path:
    tfm.load_from_checkpoint(
        checkpoint_path=checkpoint_path,
        checkpoint_type=checkpoints.CheckpointType.FLAX,
    )
else:
    tfm.load_from_checkpoint(
        repo_id=model_name,
        checkpoint_type=checkpoints.CheckpointType.FLAX,
    )

model = pax_fiddle.Config(
    patched_decoder.PatchedDecoderFinetuneModel,
    name="patched_decoder_finetune",
    core_layer_tpl=tfm.model_p,
)

if use_lora:
    load_adapter_layer(
        mdl_vars=tfm._train_state.mdl_vars,
        model=model.core_layer_tpl,
        lora_rank=lora_rank,
        lora_target_modules=lora_target_modules,
        use_dora=use_dora,
    )

@pax_fiddle.auto_config
def build_learner() -> learners.Learner:
    bprop_variable_inclusion = []
    bprop_variable_exclusion = []
    if use_lora:
        bprop_variable_inclusion.append(r"^.*lora.*$")
        if use_dora:
```

```
        bprop_variable_inclusion.append(r"^.*dora.*$")
    elif use_linear_probing:
      bprop_variable_exclusion = [".*/stacked_transformer_layer/.*"]

    return pax_fiddle.Config(
      learners.Learner,
      name="learner",
      loss_name="avg_qloss",
      optimizer=optimizers.Adam(
        epsilon=adam_epsilon,
        clip_threshold=adam_clip_threshold,
        learning_rate=learning_rate,
        lr_schedule=pax_fiddle.Config(
          schedules.Cosine,
          initial_value=cos_initial_decay_value,
          final_value=cos_final_decay_value,
          total_steps=cos_decay_steps,
        ),
        ema_decay=ema_decay,
      ),
      bprop_variable_exclusion=bprop_variable_exclusion,
      bprop_variable_inclusion=bprop_variable_inclusion,
    )

task_p = tasks_lib.SingleTask(
    name="ts-learn",
    model=model,
    train=tasks_lib.SingleTask.Train(
      learner=build_learner(),
    ),
)

task_p.model.ici_mesh_shape = [1, 1, 1]
task_p.model.mesh_axis_names = ["replica", "data", "mdl"]

DEVICES = np.array(jax.devices()).reshape([1, 1, 1])
jax.sharding.Mesh(DEVICES, ["replica", "data", "mdl"])

num_devices = jax.local_device_count()
print(f"num_devices: {num_devices}")
print(f"device kind: {jax.local_devices()[0].device_kind}")

jax_task = task_p
key, init_key = jax.random.split(key)

def process_train_batch(batch):
    past_ts = batch[0].reshape(batch_size * len(ts_cols), -1)
    actual_ts = batch[3].reshape(batch_size * len(ts_cols), -1)
```

```
    return NestedMap(input_ts=past_ts, actual_ts=actual_ts)

def process_eval_batch(batch):
    past_ts = batch[0]
    actual_ts = batch[3]
    return NestedMap(input_ts=past_ts, actual_ts=actual_ts)

jax_model_states, _ = trainer_lib.initialize_model_state(
    jax_task,
    init_key,
    process_train_batch(tbatch),
    checkpoint_type=checkpoint_types.CheckpointType.GDA,
)
jax_model_states.mdl_vars["params"]["core_layer"] = tfm._train_state.mdl_vars[
    "params"
]
gc.collect()

jax_task = task_p

def train_step(states, prng_key, inputs):
    return trainer_lib.train_step_single_learner(jax_task, states, prng_key, inputs)

def eval_step(states, prng_key, inputs):
    states = states.to_eval_state()
    return trainer_lib.eval_step_single_learner(jax_task, states, prng_key, inputs)

key, train_key, eval_key = jax.random.split(key, 3)
train_prng_seed = jax.random.split(train_key, num=jax.local_device_count())
eval_prng_seed = jax.random.split(eval_key, num=jax.local_device_count())

p_train_step = jax.pmap(train_step, axis_name="batch")
p_eval_step = jax.pmap(eval_step, axis_name="batch")

replicated_jax_states = trainer_lib.replicate_model_state(jax_model_states)

def reshape_batch_for_pmap(batch, num_devices):
    def _reshape(input_tensor):
        bsize = input_tensor.shape[0]
        residual_shape = list(input_tensor.shape[1:])
        nbsize = bsize // num_devices
        return jnp.reshape(input_tensor, [num_devices, nbsize] + residual_shape)

    return jax.tree.map(_reshape, batch)

patience = 0
best_eval_loss = 1e7
```

```
    checkpoint_dir =
f"{checkpoint_dir}/run_{datetime.now().strftime('%Y%m%d_%H%M%S')}_{wandb.run.id}"
    for epoch in range(num_epochs):
        if patience >= early_stop_patience:
            print("Early stopping.")
            break
        print(f"Epoch: {epoch + 1}")
        train_its = train_batches.as_numpy_iterator()
        train_losses = []
        for batch in tqdm(train_its):
            tbatch = process_train_batch(batch)
            tbatch = reshape_batch_for_pmap(tbatch, num_devices)
            replicated_jax_states, step_fun_out = p_train_step(
                replicated_jax_states, train_prng_seed, tbatch
            )
            train_losses.append(step_fun_out.loss[0])
            wandb.log({"train_step_loss": step_fun_out.loss[0]})

        avg_train_loss = np.mean(train_losses)

        print("Starting eval.")
        val_its = val_batches.as_numpy_iterator()
        eval_losses = []
        for ev_batch in tqdm(val_its):
            ebatch = process_eval_batch(ev_batch)
            ebatch = reshape_batch_for_pmap(ebatch, num_devices)
            _, step_fun_out = p_eval_step(replicated_jax_states, eval_prng_seed, ebatch)
            eval_losses.append(step_fun_out.loss[0])
            wandb.log({"eval_step_loss": step_fun_out.loss[0]})

        avg_eval_loss = np.mean(eval_losses)

        print(f"Train Loss: {avg_train_loss}, Val Loss: {avg_eval_loss}")

        wandb.log(
            {
                "epoch": epoch + 1,
                "avg_train_loss": avg_train_loss,
                "avg_val_loss": avg_eval_loss,
            }
        )

        if avg_eval_loss < best_eval_loss or np.isnan(avg_eval_loss):
            best_eval_loss = avg_eval_loss
            print("Saving checkpoint.")
            jax_state_for_saving = py_utils.maybe_unreplicate_for_fully_replicated(
                replicated_jax_states
            )
```

```
            if use_lora:
                adapter_params = get_adapter_params(
                    params=jax_state_for_saving.mdl_vars,
                    lora_target_modules=lora_target_modules,
                    num_layers=NUM_LAYERS,
                    use_dora=use_dora,
                )
                jax_state_for_saving.mdl_vars["params"] = adapter_params

            checkpoints.save_checkpoint(
                jax_state_for_saving, checkpoint_dir, overwrite=True
            )

            patience = 0
            del jax_state_for_saving
            gc.collect()
        else:
            patience += 1
            print(f"patience: {patience}")
    print("Fine-tuning completed.")


if __name__ == "__main__":
    typer.run(finetune)
```

16) peft/usage.ipynb

```
#!/usr/bin/env python
# coding: utf-8

# ## Load Base Model

# In[ ]:


from timesfm import TimesFm, freq_map, data_loader
from adapter.utils import load_adapter_checkpoint
from tqdm import tqdm
import numpy as np
import pandas as pd


tfm = TimesFm(
    context_len=512,
    horizon_len=128,
    input_patch_len=32,
    output_patch_len=128,
    num_layers=20,
    model_dims=1280,
    backend="cpu",
)
tfm.load_from_checkpoint(repo_id="google/timesfm-1.0-200m")


# In[ ]:


DATA_DICT = {
    "ettm2": {
        "boundaries": [34560, 46080, 57600],
        "data_path": "../datasets/ETT-small/ETTm2.csv",
        "freq": "15min",
    },
    "ettm1": {
        "boundaries": [34560, 46080, 57600],
        "data_path": "../datasets/ETT-small/ETTm1.csv",
        "freq": "15min",
    },
    "etth2": {
        "boundaries": [8640, 11520, 14400],
        "data_path": "../datasets/ETT-small/ETTh2.csv",
        "freq": "H",
    },
```

```
    "etth1": {
        "boundaries": [8640, 11520, 14400],
        "data_path": "../datasets/ETT-small/ETTh1.csv",
        "freq": "H",
    },
    "elec": {
        "boundaries": [18413, 21044, 26304],
        "data_path": "../datasets/electricity/electricity.csv",
        "freq": "H",
    },
    "traffic": {
        "boundaries": [12280, 14036, 17544],
        "data_path": "../datasets/traffic/traffic.csv",
        "freq": "H",
    },
    "weather": {
        "boundaries": [36887, 42157, 52696],
        "data_path": "../datasets/weather/weather.csv",
        "freq": "10min",
    },
}


# ## Load Adapter Checkpoint
#
# Specify the adapter checkpoint path, rank and the modules used to train the adapters and whether dora was employed or not.

# In[ ]:


load_adapter_checkpoint(
    model=tfm,
    adapter_checkpoint_path="./checkpoints/run_20240716_163900_lyo4psz3",
    lora_rank=1,
    lora_target_modules="all",
    use_dora=True,
)


# ## Test Performance

# In[ ]:


dataset = "ettm1"
data_path = DATA_DICT[dataset]["data_path"]
freq = DATA_DICT[dataset]["freq"]
```

```
int_freq = freq_map(freq)
boundaries = DATA_DICT[dataset]["boundaries"]

data_df = pd.read_csv(open(data_path, "r"))

ts_cols = [col for col in data_df.columns if col != "date"]
num_cov_cols = None
cat_cov_cols = None

context_len = 512
pred_len = 96

num_ts = len(ts_cols)
batch_size = 16

dtl = data_loader.TimeSeriesdata(
    data_path=data_path,
    datetime_col="date",
    num_cov_cols=num_cov_cols,
    cat_cov_cols=cat_cov_cols,
    ts_cols=np.array(ts_cols),
    train_range=[0, boundaries[0]],
    val_range=[boundaries[0], boundaries[1]],
    test_range=[boundaries[1], boundaries[2]],
    hist_len=context_len,
    pred_len=pred_len,
    batch_size=num_ts,
    freq="15min",
    normalize=True,
    epoch_len=None,
    holiday=False,
    permute=True,
)


# In[ ]:


test_batches = dtl.tf_dataset(mode="test", shift=pred_len)


# In[ ]:


mae_losses = []
for batch in tqdm(test_batches.as_numpy_iterator()):
    past = batch[0]
    actuals = batch[3]
```

```
    _, forecasts = tfm.forecast(list(past), [0] * past.shape[0])
    forecasts = forecasts[:, 0 : actuals.shape[1], 5]
    mae_losses.append(np.abs(forecasts - actuals).mean())

print(f"MAE: {np.mean(mae_losses)}")
```

17) tests/test_timesfm.py

```
# Copyright 2024 The Google Research Authors.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.


from datetime import datetime, timedelta

import numpy as np
import pandas as pd
import pytest

import timesfm


def create_sample_dataframe(
    start_date: datetime, end_date: datetime, freq: str = "D"
) -> pd.DataFrame:
    """
    Create a sample DataFrame with time series data.

    Args:
        start_date (datetime): Start date of the time series.
        end_date (datetime): End date of the time series.
        freq (str): Frequency of the time series (default: "D" for daily).

    Returns:
        pd.DataFrame: DataFrame with columns 'unique_id', 'ds', and 'ts'.
    """
    date_range = pd.date_range(start=start_date, end=end_date, freq=freq)
    ts_data = np.random.randn(len(date_range))
    df = pd.DataFrame({"unique_id": "ts-1", "ds": date_range, "ts": ts_data})
    return df


@pytest.mark.parametrize("context_length", [128, 256, 512])
@pytest.mark.parametrize("prediction_length", [96, 128, 256])
```

```
@pytest.mark.parametrize("freq", ["D", "H", "W"])
def test_timesfm_forecast_on_df(
    context_length: int,
    prediction_length: int,
    freq: str,
) -> None:
    model = timesfm.TimesFm(
        context_len=context_length,
        horizon_len=prediction_length,
        input_patch_len=32,
        output_patch_len=128,
        num_layers=20,
        model_dims=1280,
        backend="cpu",
    )
    model.load_from_checkpoint(repo_id="google/timesfm-1.0-200m")

    end_date = datetime.now()
    start_date = end_date - timedelta(days=context_length)
    input_df = create_sample_dataframe(start_date, end_date, freq)

    forecast_df = model.forecast_on_df(
        inputs=input_df,
        freq=freq,
        value_name="ts",
        num_jobs=-1,
    )

    assert (
        len(forecast_df) == prediction_length
    ), f"Expected forecast length of {prediction_length}, but got {len(forecast_df)}"
    assert (
        "timesfm" in forecast_df.columns
    ), "Forecast DataFrame should contain 'timesfm' column"

    last_input_date = input_df["ds"].max()
    first_forecast_date = forecast_df["ds"].min()
    expected_first_forecast_date = last_input_date + pd.Timedelta(1, unit=freq)
    assert (
        first_forecast_date == expected_first_forecast_date
    ), f"Forecast should start from {expected_first_forecast_date}, but starts from
{first_forecast_date}"

    print(
        f"Successful forecast with context_length={context_length},
prediction_length={prediction_length}, freq={freq}"
    )
```

18) notebooks/finetuning.ipynb

```
# ## Importing relevant packages for finetuning
import os

os.environ['XLA_PYTHON_CLIENT_PREALLOCATE'] = 'false'
os.environ['JAX_PMAP_USE_TENSORSTORE'] = 'false'

import timesfm
import gc
import numpy as np
import pandas as pd
from timesfm import patched_decoder
from timesfm import data_loader
from tqdm import tqdm
import dataclasses
import IPython
import IPython.display
import matplotlib as mpl
import matplotlib.pyplot as plt

mpl.rcParams['figure.figsize'] = (8, 6)
mpl.rcParams['axes.grid'] = False

# ## Loading TimesFM pretrained checkpoint

tfm = timesfm.TimesFm(
    context_len=512,
    horizon_len=128,
    input_patch_len=32,
    output_patch_len=128,
    num_layers=20,
    model_dims=1280,
    backend="gpu",
)
tfm.load_from_checkpoint(repo_id="google/timesfm-1.0-200m")

# ## Evaluating pretrained checkpoint on ETT datasets
DATA_DICT = {
    "ettm2": {
        "boundaries": [34560, 46080, 57600],
        "data_path": "../datasets/ETT-small/ETTm2.csv",
        "freq": "15min",
    },
    "ettm1": {
        "boundaries": [34560, 46080, 57600],
        "data_path": "../datasets/ETT-small/ETTm1.csv",
        "freq": "15min",
```

```
    },
    "etth2": {
        "boundaries": [8640, 11520, 14400],
        "data_path": "../datasets/ETT-small/ETTh2.csv",
        "freq": "H",
    },
    "etth1": {
        "boundaries": [8640, 11520, 14400],
        "data_path": "../datasets/ETT-small/ETTh1.csv",
        "freq": "H",
    },
    "elec": {
        "boundaries": [18413, 21044, 26304],
        "data_path": "../datasets/electricity/electricity.csv",
        "freq": "H",
    },
    "traffic": {
        "boundaries": [12280, 14036, 17544],
        "data_path": "../datasets/traffic/traffic.csv",
        "freq": "H",
    },
    "weather": {
        "boundaries": [36887, 42157, 52696],
        "data_path": "../datasets/weather/weather.csv",
        "freq": "10min",
    },
}

dataset = "ettm1"
data_path = DATA_DICT[dataset]["data_path"]
freq = DATA_DICT[dataset]["freq"]
int_freq = timesfm.freq_map(freq)
boundaries = DATA_DICT[dataset]["boundaries"]

data_df = pd.read_csv(open(data_path, "r"))

ts_cols = [col for col in data_df.columns if col != "date"]
num_cov_cols = None
cat_cov_cols = None

context_len = 512
pred_len = 96

num_ts = len(ts_cols)
batch_size = 16

dtl = data_loader.TimeSeriesdata(
      data_path=data_path,
```

```
        datetime_col="date",
        num_cov_cols=num_cov_cols,
        cat_cov_cols=cat_cov_cols,
        ts_cols=np.array(ts_cols),
        train_range=[0, boundaries[0]],
        val_range=[boundaries[0], boundaries[1]],
        test_range=[boundaries[1], boundaries[2]],
        hist_len=context_len,
        pred_len=pred_len,
        batch_size=num_ts,
        freq=freq,
        normalize=True,
        epoch_len=None,
        holiday=False,
        permute=True,
    )

train_batches = dtl.tf_dataset(mode="train", shift=1).batch(batch_size)
val_batches = dtl.tf_dataset(mode="val", shift=pred_len)
test_batches = dtl.tf_dataset(mode="test", shift=pred_len)

for tbatch in tqdm(train_batches.as_numpy_iterator()):
    pass
print(tbatch[0].shape)

# ### MAE on the test split for the pretrained TimesFM model
mae_losses = []
for batch in tqdm(test_batches.as_numpy_iterator()):
    past = batch[0]
    actuals = batch[3]
    _, forecasts = tfm.forecast(list(past), [0] * past.shape[0])
    forecasts = forecasts[:, 0 : actuals.shape[1], 5]
    mae_losses.append(np.abs(forecasts - actuals).mean())

print(f"MAE: {np.mean(mae_losses)}")

# ## Finetuning the model on the ETT dataset
import jax
from jax import numpy as jnp
from praxis import pax_fiddle
from praxis import py_utils
from praxis import pytypes
from praxis import base_model
from praxis import optimizers
from praxis import schedules
from praxis import base_hyperparams
from praxis import base_layer
from paxml import tasks_lib
```

```
from paxml import trainer_lib
from paxml import checkpoints
from paxml import learners
from paxml import partitioning
from paxml import checkpoint_types

# PAX shortcuts
NestedMap = py_utils.NestedMap
WeightInit = base_layer.WeightInit
WeightHParams = base_layer.WeightHParams
InstantiableParams = py_utils.InstantiableParams
JTensor = pytypes.JTensor
NpTensor = pytypes.NpTensor
WeightedScalars = pytypes.WeightedScalars
instantiate = base_hyperparams.instantiate
LayerTpl = pax_fiddle.Config[base_layer.BaseLayer]
AuxLossStruct = base_layer.AuxLossStruct

AUX_LOSS = base_layer.AUX_LOSS
template_field = base_layer.template_field

# Standard prng key names
PARAMS = base_layer.PARAMS
RANDOM = base_layer.RANDOM

key = jax.random.PRNGKey(seed=1234)

model = pax_fiddle.Config(
    patched_decoder.PatchedDecoderFinetuneModel,
    name='patched_decoder_finetune',
    core_layer_tpl=tfm.model_p,
)

# ### We will hold the transformer layers fixed while finetuning, while training all other
components.
@pax_fiddle.auto_config
def build_learner() -> learners.Learner:
  return pax_fiddle.Config(
      learners.Learner,
      name='learner',
      loss_name='avg_qloss',
      optimizer=optimizers.Adam(
          epsilon=1e-7,
          clip_threshold=1e2,
          learning_rate=1e-2,
          lr_schedule=pax_fiddle.Config(
              schedules.Cosine,
              initial_value=1e-3,
```

```
            final_value=1e-4,
            total_steps=40000,
        ),
        ema_decay=0.9999,
      ),
      # Linear probing i.e we hold the transformer layers fixed.
      bprop_variable_exclusion=['.*/stacked_transformer_layer/.*'],
  )

task_p = tasks_lib.SingleTask(
    name='ts-learn',
    model=model,
    train=tasks_lib.SingleTask.Train(
        learner=build_learner(),
    ),
)

task_p.model.ici_mesh_shape = [1, 1, 1]
task_p.model.mesh_axis_names = ['replica', 'data', 'mdl']

DEVICES = np.array(jax.devices()).reshape([1, 1, 1])
MESH = jax.sharding.Mesh(DEVICES, ['replica', 'data', 'mdl'])

num_devices = jax.local_device_count()
print(f'num_devices: {num_devices}')
print(f'device kind: {jax.local_devices()[0].device_kind}')

jax_task = task_p
key, init_key = jax.random.split(key)

# To correctly prepare a batch of data for model initialization (now that shape
# inference is merged), we take one devices*batch_size tensor tuple of data,
# slice out just one batch, then run the prepare_input_batch function over it.

def process_train_batch(batch):
    past_ts = batch[0].reshape(batch_size * num_ts, -1)
    actual_ts = batch[3].reshape(batch_size * num_ts, -1)
    return NestedMap(input_ts=past_ts, actual_ts=actual_ts)


def process_eval_batch(batch):
    past_ts = batch[0]
    actual_ts = batch[3]
    return NestedMap(input_ts=past_ts, actual_ts=actual_ts)


jax_model_states, _ = trainer_lib.initialize_model_state(
    jax_task,
```

```
    init_key,
    process_train_batch(tbatch),
    checkpoint_type=checkpoint_types.CheckpointType.GDA,
)

# ### Setting the initial model weights to the pretrained TimesFM parameters.
jax_model_states.mdl_vars['params']['core_layer'] = tfm._train_state.mdl_vars['params']
jax_vars = jax_model_states.mdl_vars
gc.collect()


# ### Training loop
jax_task = task_p

def train_step(states, prng_key, inputs):
  return trainer_lib.train_step_single_learner(
      jax_task, states, prng_key, inputs
  )

def eval_step(states, prng_key, inputs):
  states = states.to_eval_state()
  return trainer_lib.eval_step_single_learner(
      jax_task, states, prng_key, inputs
  )

key, train_key, eval_key = jax.random.split(key, 3)
train_prng_seed = jax.random.split(train_key, num=jax.local_device_count())
eval_prng_seed = jax.random.split(eval_key, num=jax.local_device_count())

p_train_step = jax.pmap(train_step, axis_name='batch')
p_eval_step = jax.pmap(eval_step, axis_name='batch')

replicated_jax_states = trainer_lib.replicate_model_state(jax_model_states)
replicated_jax_vars = replicated_jax_states.mdl_vars

best_eval_loss = 1e7
step_count = 0
patience = 0
NUM_EPOCHS = 100
PATIENCE = 5
TRAIN_STEPS_PER_EVAL = 1000
CHECKPOINT_DIR='/home/senrajat_google_com/ettm1_finetune'

def reshape_batch_for_pmap(batch, num_devices):
  def _reshape(input_tensor):
    bsize = input_tensor.shape[0]
    residual_shape = list(input_tensor.shape[1:])
    nbsize = bsize // num_devices
```

```
    return jnp.reshape(input_tensor, [num_devices, nbsize] + residual_shape)

  return jax.tree.map(_reshape, batch)

for epoch in range(NUM_EPOCHS):
    print(f"___________________Epoch: {epoch}___________________", flush=True)
    train_its = train_batches.as_numpy_iterator()
    if patience >= PATIENCE:
        print("Early stopping.", flush=True)
        break
    for batch in tqdm(train_its):
        train_losses = []
        if patience >= PATIENCE:
            print("Early stopping.", flush=True)
            break
        tbatch = process_train_batch(batch)
        tbatch = reshape_batch_for_pmap(tbatch, num_devices)
        replicated_jax_states, step_fun_out = p_train_step(
            replicated_jax_states, train_prng_seed, tbatch
        )
        train_losses.append(step_fun_out.loss[0])
        if step_count % TRAIN_STEPS_PER_EVAL == 0:
            print(
                f"Train loss at step {step_count}: {np.mean(train_losses)}",
                flush=True,
            )
            train_losses = []
            print("Starting eval.", flush=True)
            val_its = val_batches.as_numpy_iterator()
            eval_losses = []
            for ev_batch in tqdm(val_its):
                ebatch = process_eval_batch(ev_batch)
                ebatch = reshape_batch_for_pmap(ebatch, num_devices)
                _, step_fun_out = p_eval_step(
                    replicated_jax_states, eval_prng_seed, ebatch
                )
                eval_losses.append(step_fun_out.loss[0])
            mean_loss = np.mean(eval_losses)
            print(f"Eval loss at step {step_count}: {mean_loss}", flush=True)
            if mean_loss < best_eval_loss or np.isnan(mean_loss):
                best_eval_loss = mean_loss
                print("Saving checkpoint.")
                jax_state_for_saving = py_utils.maybe_unreplicate_for_fully_replicated(
                    replicated_jax_states
                )
                checkpoints.save_checkpoint(
                    jax_state_for_saving, CHECKPOINT_DIR, overwrite=True
                )
```

```
            patience = 0
            del jax_state_for_saving
            gc.collect()
        else:
            patience += 1
            print(f"patience: {patience}")
    step_count += 1


# ## Loading and evaluating the best (according to validation loss) finetuned checkpoint
train_state = checkpoints.restore_checkpoint(jax_model_states, CHECKPOINT_DIR)
print(train_state.step)
tfm._train_state.mdl_vars['params'] = train_state.mdl_vars['params']['core_layer']
tfm.jit_decode()

mae_losses = []
for batch in tqdm(test_batches.as_numpy_iterator()):
    past = batch[0]
    actuals = batch[3]
    _, forecasts = tfm.forecast(list(past), [0] * past.shape[0])
    forecasts = forecasts[:, 0 : actuals.shape[1], 5]
    mae_losses.append(np.abs(forecasts - actuals).mean())

print(f"MAE: {np.mean(mae_losses)}")
```

19) notebooks/covariates.ipynb

```
#!/usr/bin/env python
# coding: utf-8

# # TimesFM with Covariates
#
# This toturial notebook demonstrates how to utilize exogenous covariates with TimesFM when making forecasts. Before running this notebook, make sure:
#
# - You've read through the README of TimesFM.
# - A local kernel with Python 3.10 is up and running.

# ## Setup the environment and install TimesFM.

# In[ ]:


import os
os.environ['XLA_PYTHON_CLIENT_PREALLOCATE'] = 'false'
os.environ['JAX_PMAP_USE_TENSORSTORE'] = 'false'


# In[ ]:


get_ipython().system('pip install timesfm')
import timesfm


# ## Load the checkpoint
#
# **Notice:** Please set up the backend as per your machine ("cpu", "gpu" or "tpu"). This notebook will run by default on CPU.
#
# We load the 1.0-200m model checkpoint from HuggingFace.

# In[ ]:


timesfm_backend = "cpu"  # @param

from jax._src import config
config.update(
    "jax_platforms", {"cpu": "cpu", "gpu": "cuda", "tpu": ""}[timesfm_backend]
)

model = timesfm.TimesFm(
```

```
    context_len=512,
    horizon_len=128,
    input_patch_len=32,
    output_patch_len=128,
    num_layers=20,
    model_dims=1280,
    backend=timesfm_backend,
)
model.load_from_checkpoint(repo_id="google/timesfm-1.0-200m")


# # Covariates
#
# Let's take a toy example of forecasting sales for a grocery store:
#
# **Task:** Given the observed the daily sales of this week (7 days), forecast the daily sales of next week (7 days).
#
# ```
# Product: ice cream
# Daily_sales: [30, 30, 4, 5, 7, 8, 10]
# Category: food
# Base_price: 1.99
# Weekday: [0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4, 5, 6]
# Has_promotion: [Yes, Yes, No, No, No, Yes, Yes, No, No, No, No, No, No, No]
# Daily_temperature: [31.0, 24.3, 19.4, 26.2, 24.6, 30.0, 31.1, 32.4, 30.9, 26.0, 25.0, 27.8, 29.5, 31.2]
# ```
#
# ```
# Product: sunscreen
# Daily_sales: [5, 7, 12, 13, 5, 6, 10]
# Category: skin product
# Base_price: 29.99
# Weekday: [0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4, 5, 6]
# Has_promotion: [No, No, Yes, Yes, No, No, No, Yes, Yes, Yes, Yes, Yes, Yes, Yes]
# Daily_temperature: [31.0, 24.3, 19.4, 26.2, 24.6, 30.0, 31.1, 32.4, 30.9, 26.0, 25.0, 27.8, 29.5, 31.2]
# ```
#
# In this example, besides the `Daily_sales`, we also have covariates `Category`, `Base_price`, `Weekday`, `Has_promotion`, `Daily_temperature`. Let's introduce some concepts:
#
# **Static covariates** are covariates for each time series.
# - In our example, `Category` is a **static categorical covariate**,
# - `Base_price` is a **static numerical covariates**.
#
```

```python
# **Dynamic covariates** are covaraites for each time stamps.
# - Date / time related features can be usually treated as dynamic covariates.
# - In our example, `Weekday` and `Has_promotion` are **dynamic categorical covariates**.
# - `Daily_temperate` is a **dynamic numerical covariate**.
#
# **Notice:** Here we make it mandatory that the dynamic covariates need to cover both the forecasting context and horizon. For example, all dynamic covariates in the example have 14 values: the first 7 correspond to the observed 7 days, and the last 7 correspond to the next 7 days.

# # TimesFM with Covariates
#
#
# The strategy we take here is to treat covariates as batched in-context exogenous regressors (XReg) and fit linear models on them outside of TimesFM. The final forecast will be the sum of the TimesFM forecast and the linear model forecast.
#
#  In simple words, we consider these two options.
#
# **Option 1:** Get the TimesFM forecast, and fit the linear model regressing the residuals on the covariates ("timesfm + xreg").
#
# **Option 2:** Fit the linear model of the time series itself on the covariates, then forecast the residuals using TimesFM  ("xreg + timesfm").
#
# Let's take a code at the example of Electricity Price Forecasting (EPF).
#

# In[ ]:


import pandas as pd
import numpy as np
from collections import defaultdict


# In[ ]:


df = pd.read_csv('https://datasets-nixtla.s3.amazonaws.com/EPF_FR_BE.csv')
df['ds'] = pd.to_datetime(df['ds'])
df


# This dataset has a few covariates beside the hourly target `y`:
#
# - `unique_id`: a static categorical covariate indicating the country.
```

```
# - `gen_forecast`: a dynamic numerical covariate indicating the estimated electricity to be
generated.
# - `system_load`: the observed system load. Notice that this **CANNOT** be considered as
a dynamic numerical covariate because we cannot know its values over the forecasting
horizon in advance.
# - `weekday`: a dynamic categorical covariate.\
#
# Let's now make some forecasting tasks for TimesFM based on this dataset. For simplicity
we create forecast contexts of 120 time points (hours) and forecast horizons of 24 time
points.

# In[ ]:


# Data pipelining
def get_batched_data_fn(
    batch_size: int = 128,
    context_len: int = 120,
    horizon_len: int = 24,
):
  examples = defaultdict(list)

  num_examples = 0
  for country in ("FR", "BE"):
    sub_df = df[df["unique_id"] == country]
    for start in range(0, len(sub_df) - (context_len + horizon_len), horizon_len):
      num_examples += 1
      examples["country"].append(country)
      examples["inputs"].append(sub_df["y"][start:(context_end := start + context_len)].tolist())
      examples["gen_forecast"].append(sub_df["gen_forecast"][start:context_end +
horizon_len].tolist())
      examples["week_day"].append(sub_df["week_day"][start:context_end +
horizon_len].tolist())
      examples["outputs"].append(sub_df["y"][context_end:(context_end +
horizon_len)].tolist())

  def data_fn():
    for i in range(1 + (num_examples - 1) // batch_size):
      yield {k: v[(i * batch_size) : ((i + 1) * batch_size)] for k, v in examples.items()}

  return data_fn


# In[ ]:


# Define metrics
def mse(y_pred, y_true):
```

```
  y_pred = np.array(y_pred)
  y_true = np.array(y_true)
  return np.mean(np.square(y_pred - y_true), axis=1, keepdims=True)

def mae(y_pred, y_true):
  y_pred = np.array(y_pred)
  y_true = np.array(y_true)
  return np.mean(np.abs(y_pred - y_true), axis=1, keepdims=True)


# Now let's try `model.forecast_with_covariates`.
#
# In particular, the output is a tuple whose first element is the new forecast.

# In[ ]:


# Benchmark
batch_size = 128
context_len = 120
horizon_len = 24
input_data = get_batched_data_fn(batch_size = 128)
metrics = defaultdict(list)
import time

for i, example in enumerate(input_data()):
  raw_forecast, _ = model.forecast(
      inputs=example["inputs"], freq=[0] * len(example["inputs"])
  )
  start_time = time.time()
  # Forecast with covariates
  # Output: new forecast, forecast by the xreg
  cov_forecast, ols_forecast = model.forecast_with_covariates(
      inputs=example["inputs"],
      dynamic_numerical_covariates={
          "gen_forecast": example["gen_forecast"],
      },
      dynamic_categorical_covariates={
          "week_day": example["week_day"],
      },
      static_numerical_covariates={},
      static_categorical_covariates={
          "country": example["country"]
      },
      freq=[0] * len(example["inputs"]),
      xreg_mode="xreg + timesfm",              # default
      ridge=0.0,
      force_on_cpu=False,
```

```
    normalize_xreg_target_per_input=True,    # default
  )
  print(
    f"\rFinished batch {i} linear in {time.time() - start_time} seconds",
    end="",
  )
  metrics["eval_mae_timesfm"].extend(
    mae(raw_forecast[:, :horizon_len], example["outputs"])
  )
  metrics["eval_mae_xreg_timesfm"].extend(mae(cov_forecast, example["outputs"]))
  metrics["eval_mae_xreg"].extend(mae(ols_forecast, example["outputs"]))
  metrics["eval_mse_timesfm"].extend(
    mse(raw_forecast[:, :horizon_len], example["outputs"])
  )
  metrics["eval_mse_xreg_timesfm"].extend(mse(cov_forecast, example["outputs"]))
  metrics["eval_mse_xreg"].extend(mse(ols_forecast, example["outputs"]))

print()

for k, v in metrics.items():
  print(f"{k}: {np.mean(v)}")

# My output:
# eval_mae_timesfm: 6.762283045916956
# eval_mae_xreg_timesfm: 5.39219617611074
# eval_mae_xreg: 37.15275842572484
# eval_mse_timesfm: 166.7771466306823
# eval_mse_xreg_timesfm: 120.64757721021306
# eval_mse_xreg: 1672.2116821201796


# You should see results close to
# ```
# eval_mae_timesfm: 6.762283045916956
# eval_mae_xreg_timesfm: 5.39219617611074
# eval_mae_xreg: 37.15275842572484
# eval_mse_timesfm: 166.7771466306823
# eval_mse_xreg_timesfm: 120.64757721021306
# eval_mse_xreg: 1672.2116821201796
# ```
#
# With the covariates, the TimesFM forecast Mean Absolute Error improves from 6.76 to 5.39, and Mean Squred Error from 166.78 to 120.65. The results of purely fitting the linear model are also provided for reference.

# ## Formatting Your Request
#
```

It is quite crucial to get the covariates properly formatted so that we can call this `model.forecast_with_covariates`. Please see its docstring for details. Here let's also grab a batch from a toy data input pipeline for quick explanations.

In[ ]:

toy_input_pipeline = get_batched_data_fn(batch_size=2, context_len=5, horizon_len=2)
print(next(toy_input_pipeline()))

You should see something similar to this
```
{
'country': ['FR', 'FR'],
'inputs': [[53.48, 51.93, 48.76, 42.27, 38.41], [48.76, 42.27, 38.41, 35.72, 32.66]],
'gen_forecast': [[76905.0, 75492.0, 74394.0, 72639.0, 69347.0, 67960.0, 67564.0], [74394.0, 72639.0, 69347.0, 67960.0, 67564.0, 67277.0, 67019.0]],
'week_day': [[3, 3, 3, 3, 3, 3, 3], [3, 3, 3, 3, 3, 3, 3]],
'outputs': [[35.72, 32.66], [32.83, 30.06]],
}
```
#
Notice:
- We have two examples in this batch.
- For each example we support different context lengths and horizon lengths just as `model.forecast`. Although it is not demonstrated in this dataset.
- If dynamic covariates are present, the horizon lengths will be inferred from them, e.g. how many values are provided in additional to the ones corresponding to the inputs. Make sure all your dynamic covariates have the same length per example.
- The static covariates are one per example.
#
#

More Applications
#
Past Dynamic Covariates
#
Past dynamic covariates are covariates that are only available for the context. For instance in our example `system_load` is a past dynamic covariate. Time series models generally can handle this, however it is something the batched in context regression cannot address, because these regressors are not available in the future. If you do have those covariates and consider them very meaningful, there are two hacky options to try immediately:
#
1. Shift and repeat these past dynamic covariates to use their delayed version. For example, if you think the `system_load` for this week is meaningful for forecasting next week, you can create a `delay_7_system_load` by shifting 7 timestamps and use this as one dynamic numerical covariate for TimesFM.

2. Bootstrap, that is to run TimesFM once to forecast these past dynamic covariates into the horizon, then call TimesFM again using these forecasts as the future part for these dynamic covariates.
#
Multivariate Time Series
#
For multivariate time series, if we need univariate forecast, we can try treating the main time series as the target and use the rest as the dynamic covariates.