ChatGPT

Полный процесс fine-tuning модели TinyTimeMixer (TTM R2) для временных рядов

Обзор модели и задачи fine-tuning

TinyTimeMixer (TTM R2) – это компактная предобученная модель IBM для многомерных временных рядов (менее 1 млн параметров). В исходном виде модель обучена **канально-независимо**, то есть она учит общие закономерности отдельно по каждому каналу (временному ряду) без учета взаимосвязей между ними [1]. Это обеспечивает универсальность, однако при адаптации к конкретному набору данных важно учесть взаимодействия между каналами (переменными). Процесс **fine-tuning** (дообучения) позволяет донастроить модель на вашем наборе данных, включая **взаимосвязи между каналами** и использование внешних признаков (экзогенных факторов).

Шаги fine-tuning TTM R2: [2]

1. **Подготовка данных.** Загрузите данные о временных рядах, разделите их на тренировочный/валидационный/тестовый наборы и обработайте с помощью утилиты `TimeSeriesPreprocessor`. На этапе fine-tuning можно добавлять дополнительные признаки: экзогенные переменные (значения которых известны наперед) и статические категориальные признаки [3] [4]. Предварительная обработка включает масштабирование (нормализацию) данных по каждому каналу [5].

2. **Загрузка предобученной модели.** Используйте предобученную модель TTM R2 из репозитория Hugging Face (`ibm-granite/granite-timeseries-ttm-r2`) с помощью метода `TinyTimeMixerForPrediction.from_pretrained`. При этом укажите конфигурацию под ваш набор данных: количество входных каналов, индексы целевых каналов (для прогноза) и экзогенных каналов, а также включите режим смешивания каналов (`decoder_mode="mix_channel"`) для учета межканальных зависимостей [6] [7]. Затем «заморозьте» параметры основной части модели (backbone), чтобы в ходе обучения изменялись главным образом параметры головы/декодера, ответственного за прогноз [8]. Это ускоряет обучение и сохраняет обобщающие способности модели, обученной на больших объемах данных.

3. **Настройка Trainer и обучение модели.** Создайте обучаемый dataset на основе подготовленных данных и настройте тренировочный процесс с помощью `HuggingFace Trainer`. Задайте `TrainingArguments` – число эпох, скорость обучения, размеры батчей и пр. [9]. Добавьте колбэки: например, `EarlyStoppingCallback` для ранней остановки (patience=10 эпох без улучшения) и `TrackingCallback` для логирования времени и метрик [10]. В качестве оптимизатора обычно применяют AdamW, а для адаптивного изменения learning rate – **OneCycleLR** (что быстро разгоняет и снижает шаг обучения в течение эпох) [11]. Инициализируйте `Trainer`, передав ему модель, датасеты (train/val), колбэки и оптимизатор с шедулером [12] [13]. Далее запустите обучение методом `trainer.train()` [14], наблюдая за динамикой ошибки на валидации.

4. **Оценка и прогнозирование.** После fine-tuning оцените модель на тестовом наборе: `trainer.evaluate(test_dataset)` вычисляет метрики, например `eval_loss` – среднеквадратичную ошибку (MSE) на нормализованных данных [15]. Затем используйте дообученную модель для прогнозирования новых значений: можно либо напрямую получить прогнозы моделью, либо воспользоваться удобным пайплайном `TimeSeriesForecastingPipeline` из библиотеки Granite-TSFM для автоматического формирования прогнозов [16]. Пайплайн берет на вход сырые тестовые данные (DataFrame) и выдаёт прогнозы в виде новой колонки, учитывая масштабирование/обратное масштабирование и прочие настройки из `TimeSeriesPreprocessor`.

Ниже подробно рассмотрены эти этапы, как если бы разработчик модели пояснял процесс с собственных позиций.

Подготовка и предварительная обработка данных

Начните с загрузки данных в DataFrame (например, с помощью `pandas.read_csv`) и убедитесь, что столбец времени приведён к типу даты-времени. Допустим, у нас есть временной ряд с временным столбцом `"date"` и три числовых столбца: `"value1"`, `"value2"` и `"value3"`. Предположим, первые два – целевые (то, что надо прогнозировать), а третий – экзогенный признак, значения которого известны наперёд (например, плановые показатели или погодные факторы). В этом случае мы зададим **target-колонки** и **control-колонки** следующим образом:

```
import pandas as pd
# Загрузка данных
data = pd.read_csv(dataset_path, parse_dates=["date"])

from tsfm_public import TimeSeriesPreprocessor
tsp = TimeSeriesPreprocessor(
    id_columns=[],                # ID групп (если несколько независимых
серий; здесь не используется)
    timestamp_column="date",      # Название столбца с меткой времени
    target_columns=["value1", "value2"],   # Целевые столбцы для прогноза
    control_columns=["value3"],   # Экзогенные столбцы (значения известны в
будущем)
    context_length=512,           # Длина истории (контекста) для модели
    prediction_length=96,         # Длина прогноза (горизонт в шагах
времени)
    scaling=True,                 # Масштабировать ли данные (нормализовать)
    scaling_type="standard"       # Тип нормализации: StandardScaler
(среднее=0, σ=1)
)
```

В этом примере мы указываем, что модель будет использовать 512 последних точек истории для предсказания 96 будущих точек по каждому из целевых рядов [17]. Параметр `scaling=True` включает автоматическое масштабирование каждого канала данных – по умолчанию стандартное (отнимаем среднее, делим на стандартное отклонение) [18]. **Важно:** модель TTM предполагает, что данные **нормализованы по каждому каналу** перед подачей на вход [5], поэтому масштабирование следует выполнять обязательно (либо вручную, либо через `TimeSeriesPreprocessor`).

Если у вас имеются **статические категориальные признаки** (например, идентификатор объекта, тип товара и т.д.), `TimeSeriesPreprocessor` может закодировать их при настройке `encode_categorical=True` [19]. В этом случае он сохранит размер словаря категорий, и эту информацию мы затем передадим модели. В нашем примере статических признаков нет (`id_columns=[]` означает, что все данные рассматриваются как один общий временной ряд, а не панель с разными объектами).

Далее необходимо **разбить данные** на обучающую, валидационную и тестовую выборки по времени. Разбиение можно выполнить вручную (например, по дате или долям выборки) либо воспользоваться утилитой `prepare_data_splits` из `tsfm_public.toolkit` [20]. Предположим, мы разделили данные и получили три DataFrame: `train_data`, `valid_data`, `test_data`. Теперь нужно обучить масштабирование на тренировочных данных и применить его ко всем выборкам:

```
# Обучение преобразований на тренировочных данных (вычисление средних/σ для
масштабирования, кодирование категорий)
tsp.train(train_data)

# Применение пре-процессинга (масштабирование, формирование входных массивов)
к каждому набору
train_proc = tsp.preprocess(train_data)
val_proc   = tsp.preprocess(valid_data)
test_proc  = tsp.preprocess(test_data)
```

Метод `tsp.train()` прогоняет тренировочный DataFrame через препроцессор, **настраивая scaler по тренировочным данным** (и, если нужно, собирая словари категорий) [21] [22]. Метод `tsp.preprocess(df)` затем применяет эти трансформации (нормализацию, кодирование и пр.) к любому DataFrame и возвращает преобразованные данные.

На выходе `TimeSeriesPreprocessor.preprocess` дает данные, готовые для формирования torch Dataset. Библиотека Granite-TSFM предоставляет класс `ForecastDFDataset`, оптимизированный для обучения модели прогнозирования [23]. Мы можем создать объекты датасета следующим образом:

```
from tsfm_public import ForecastDFDataset

# Подготовка параметров датасета (включая метаданные колонок и длины окон)
dataset_params = {
    "id_columns": [],
    "timestamp_column": "date",
    "target_columns": ["value1", "value2"],
    "control_columns": ["value3"],
    "context_length": 512,
    "prediction_length": 96,
    "frequency_token": tsp.get_frequency_token(tsp.freq)  # признак частоты
(например 'H' для hourly), если нужен
}
train_dataset = ForecastDFDataset(train_proc, **dataset_params)
```

```
val_dataset   = ForecastDFDataset(val_proc, **dataset_params)
test_dataset  = ForecastDFDataset(test_proc, **dataset_params)
```

Параметры `dataset_params` берутся в основном из тех же спецификаций, что мы передавали в `TimeSeriesPreprocessor` (они определяют структуру данных) [24]. `frequency_token` – это символ частоты временного шага (например, `'H'` для часовых данных, `'D'` для дневных и т.д.), который препроцессор может определить автоматически (`tsp.freq`) либо задать вручную. Он нужен для корректной обработки меток времени, особенно если используется механизм frequency prefix tuning в моделях (в релизе r2.1 появились дневные/недельные модели) [25].

Теперь наши данные подготовлены: у нас есть PyTorch-совместимые датасеты `train_dataset`, `val_dataset`, `test_dataset`, содержащие временные ряды в виде входных и целевых тензоров для модели. Каждый сэмпл включает нормализованные значения последних 512 точек истории (по всем каналам) и 96 следующих точек (для целевых каналов), которые модель должна предсказать во время обучения.

Загрузка предобученной модели TTM R2 с настройкой каналов

Для загрузки предобученной модели воспользуемся методом класса модели `TinyTimeMixerForPrediction.from_pretrained(...)`. Мы укажем путь модели на Hugging Face Hub – `"ibm-granite/granite-timeseries-ttm-r2"` – и передадим важные аргументы настройки:

- **num_input_channels**: общее число входных каналов, то есть сумма целевых и экзогенных временных рядов. В нашем примере 2 целевых + 1 экзогенный = **3 канала** [26].
- **prediction_channel_indices**: индексы каналов, соответствующие целевым рядам. Если препроцессор сформировал порядок каналов как `[value1, value2, value3]`, то целевые – это первые два, с индексами `[0, 1]` [26]. `tsp.prediction_channel_indices` предоставляет этот список автоматически.
- **exogenous_channel_indices**: индексы экзогенных (контрольных) каналов. В нашем примере экзогенный столбец `"value3"` – третий канал, индекс `[2]` [26], который доступен как `tsp.exogenous_channel_indices`.
- **decoder_mode**: режим декодера – ставим `"mix_channel"`, чтобы включить *channel-mixing* (перемешивание каналов) на этапе прогноза [27]. Именно этот флаг позволяет модели во время fine-tuning учиться учитывать корреляции между разными временными рядами, чего не было на этапе предобучения [28] [29]. В документации особо отмечено, что ключевым шагом fine-tuning является установка `decoder_mode="mix_channel"` для захвата межканальных зависимостей [27] [7].
- **context_length** и **prediction_length**: можно указать явно длины контекста и прогноза, чтобы убедиться, что загружается правильная версия модели, соответствующая этим параметрам. (Альтернатива: использовать утилиту `get_model()` для автоматического выбора ревизии модели [30], либо явно указать `revision` – например, `"512-96-r2"` – если нужна определенная ветка модели.)
- **Дополнительно для экзогенных и категориальных признаков:** Если мы используем экзогенные переменные **во время прогноза**, рекомендуется включить флаг `enable_forecast_channel_mixing=True`. Это активирует в модели специальный компонент – *exogenous mixer*, позволяющий модельному декодеру учесть влияние экзогенных каналов на прогнозируемые значения [31]. Также, если были статические

категориальные признаки, нужно передать параметр `categorical_vocab_size_list` – список размеров словарей для каждой категориальной переменной, который хранится внутри `tsp` (например, `tsp.categorical_vocab_size_list`) [32].

Применим все вышесказанное на практике:

```
from tsfm_public import TinyTimeMixerForPrediction

model = TinyTimeMixerForPrediction.from_pretrained(
    "ibm-granite/granite-timeseries-ttm-r2",
    num_input_channels=tsp.num_input_channels,               # =3 в нашем
случае
    prediction_channel_indices=tsp.prediction_channel_indices,  # = [0, 1]
    exogenous_channel_indices=tsp.exogenous_channel_indices,    # = [2]
    decoder_mode="mix_channel",                              # включаем
межканальное смешивание
    enable_forecast_channel_mixing=True,                     # учитываем
экзогенные при прогнозе
    # context_length=512, prediction_length=96,             # (можно указать
явн., модель r2 имеет эти параметры)
    # categorical_vocab_size_list=tsp.categorical_vocab_size_list  # если
есть статические категориальные
)
```

После загрузки у нас в объекте `model` находится предобученная модель TTM R2, настроенная на нужное число входных каналов. Она изначально обучена предсказывать каждый целевой ряд независимо; но благодаря параметрам `decoder_mode="mix_channel"` и `enable_forecast_channel_mixing=True` мы **добавили новые возможности** декодеру: учитывать связи между целевыми рядами и влияние экзогенных факторов при формировании прогноза [31]. Внутри модели появились дополнительные параметры (в весах декодера и exogenous mixer), которые изначально либо нулевые, либо случайно инициализированы – их-то мы и будем обучать на наших данных.

Теперь **заморозим параметры «спины» (backbone)** – основной части модели, ответственной за извлечение признаков. Backbone состоит из нескольких блоков TSMixer, обученных на большом объеме данных; его мы менять не хотим, чтобы не нарушить выученные паттерны. Вместо этого мы сфокусируем обучение на **голове модели**: декодере и выходном слое прогноза (а также компонентах, отвечающих за экзогенные входы). Согласно дизайну TTM, декодер и прогнозирующая голова составляют лишь 10–20% всех параметров модели [33], поэтому fine-tuning будет гораздо эффективнее, чем обучение всей модели с нуля. Мы проходимся по всем параметрам `model.backbone` и отключаем у них градиенты:

```
for param in model.backbone.parameters():
    param.requires_grad = False
```

Таким образом, **в ходе дообучения будут обновляться только параметры головы (decoder + forecast head)**, которые как раз и отвечают за адаптацию к вашим целевым рядам и их взаимодействиям [34]. Это подтверждается и идеологией статьи: TTM обучается в два этапа – на

этапе pre-training канал-независимо, а на этапе fine-tuning добавляются межканальные связи и экзогенные влияния для конкретных данных [1].

Настройка обучения: Trainer, параметры и колбэки

Модель и данные готовы – следующий шаг настроить процесс обучения (fine-tuning). Мы будем использовать встроенный в Hugging Face `Trainer`, который интегрируется с PyTorch. Это позволяет нам легко задать параметры обучения и автоматически выполнять цикл train/validation.

Начнём с задания гиперпараметров и настроек через объект `TrainingArguments`:

```
from transformers import Trainer, TrainingArguments, EarlyStoppingCallback

training_args = TrainingArguments(
    output_dir="ttm_finetune_output",   # папка для сохранения результатов и чекпойнтов
    overwrite_output_dir=True,
    learning_rate=1e-3,                 # скорость обучения (можно подбирать под ваш датасет)
    num_train_epochs=100,               # максимальное число эпох обучения
    per_device_train_batch_size=64,
    per_device_eval_batch_size=64,
    evaluation_strategy="epoch",        # выполнять evaluation на валидации после каждой эпохи
    save_strategy="epoch",
# (можно сохранять модель каждый эпоху или по лучшему результату)
    logging_strategy="epoch",
    load_best_model_at_end=True,        # по завершении загрузить лучшую модель (по метрике) на валидации
    metric_for_best_model="eval_loss", # метрика для определения лучшей модели
    greater_is_better=False             # для loss: меньше – лучше
)
```

Здесь мы выбрали базовые значения: LR 0.001, батч 64 и до 100 эпох обучения [9]. Параметры вроде размера батча и шага обучения могут потребовать настройки под ваш набор данных (например, для большого датасета LR можно снизить, или если данных мало – снизить batch size, увеличить epochs и т.д.) [35]. Мы настроили оценку на валидации каждый эпоху и сохраняем лучшую модель по минимальному `eval_loss`.

Добавим **Early Stopping** – чтобы модель не переобучалась впустую, остановим обучение если в течение 10 эпох качество не улучшилось:

```
early_stop = EarlyStoppingCallback(
    early_stopping_patience=10,
    early_stopping_threshold=0.0   # требуемое минимальное улучшение (0.0 =
```

```
любое улучшение)
)
```

И возьмем из библиотеки Granite-TSFM специальный колбэк `TrackingCallback` для мониторинга (он логирует время эпох, скорость и т.д.):

```
from tsfm_public import TrackingCallback
tracking = TrackingCallback()
```

Далее определим оптимизатор и scheduler для обучения. Хотя `Trainer` мог бы использовать свой дефолтный оптимизатор, разработчики Granite рекомендуют явно применять **AdamW** и схему смены Learning Rate по принципу 1cycle (OneCycleLR) [11]. Это позволяет сначала увеличить learning rate, а затем плавно его снизить к концу обучения, что часто ускоряет сходимость:

```
from torch.optim import AdamW
from torch.optim.lr_scheduler import OneCycleLR
import math

optimizer = AdamW(model.parameters(), lr=1e-3)
# Рассчитываем число шагов (итераций) на эпоху:
steps_per_epoch = math.ceil(len(train_dataset) /
training_args.per_device_train_batch_size)
scheduler = OneCycleLR(
    optimizer, max_lr=1e-3,
    epochs=training_args.num_train_epochs,
    steps_per_epoch=steps_per_epoch
)
```

Теперь создадим экземпляр `Trainer`, передав ему все компоненты:

```
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    callbacks=[early_stop, tracking],
    optimizers=(optimizer, scheduler)
)
```

Обратите внимание: мы передаем **нашу модель**, подготовленные датасеты и список колбэков в `Trainer`. В аргумент `optimizers` мы сразу вложили кортеж `(optimizer, scheduler)`, чтобы использовать нашу настройку OneCycle. Если бы мы этого не сделали, `Trainer` по умолчанию взял бы простой AdamW с постоянным шагом или косинусным спадом. Мы же явно задали оптимизацию по нашим предпочтениям [13].

Кроме того, с помощью `TrainingArguments` мы настроили автоматическое сохранение и выбор лучшей модели. По окончании обучения можно будет восстановить веса лучшей эпохи (`load_best_model_at_end=True`).

Fine-tuning модели (обучение на новых данных)

Все готово для запуска fine-tuning. Запустим обучение и будем наблюдать за выводимыми метриками:

```
trainer.train()
```

Во время обучения `Trainer` будет выводить информацию по эпохам – в частности, тренировочную и валидационную потерю (loss). Благодаря `TrackingCallback` вы также увидите время обучения на эпоху, скорость обработки сэмплов и т.п.

Учитывая небольшой размер модели TTM, **обучение проходит достаточно быстро** – как правило, считанные минуты на одной GPU (или десятки минут на CPU, в зависимости от объема данных). Например, fine-tuning на 520 обучающих образцах и 100 эпох занял у нас около 15 секунд на эпоху на GPU [36]. Часто модель сходится раньше максимального числа эпох, и Early Stopping остановит процесс, если улучшений не будет. В нашем примере patience=10: то есть если 10 эпох подряд валидационный loss не снижается, тренинг завершится досрочно.

Поскольку мы заморозили backbone, обучаются в основном веса декодера. Это несколько ограничивает риск переобучения и позволяет эффективно тренировать даже на небольшом объеме данных (**few-shot fine-tuning**). В официальном туториале демонстрируется, что можно дообучать модель всего на 5–10% данных и уже получать заметное улучшение [37]. Вы тоже можете экспериментально уменьшать обучающий датасет – например, взять подмножество (`Subset` или просто ранние точки серии) – и оценивать, как это влияет на качество.

После завершения `trainer.train()` наша модель наилучшей эпохи сохранена внутри `trainer.model`. Мы готовы проверить качество и строить прогнозы!

Оценка качества модели после fine-tuning

Чтобы оценить дообученную модель, воспользуемся тестовой выборкой. Можно просто вызвать:

```
metrics = trainer.evaluate(test_dataset)
print(metrics)
```

По умолчанию возвращается словарь метрик, включающий `eval_loss` – для задач регрессии (прогноза) это обычно среднеквадратичная ошибка (MSE) усредненная по всем точкам прогноза и всем рядам [15]. **Важно:** так как мы нормализовали данные, этот loss рассчитывается на **нормированных значениях**. Для интерпретации в реальных величинах вам следует перемасштабировать ошибки. Например, MSE на нормированных данных = 0.01 не означает, что ошибка прогноза 0.01 единицы исходного ряда – нужно умножить на дисперсию исходных данных. Поэтому лучше сразу вычислить метрики на исходном масштабе.

Мы можем самостоятельно посчитать метрики: например, средний MAE и RMSE. Для этого получим **прогнозы модели на тестовом наборе** и сравним с реальными значениями. Воспользуемся методом `Trainer.predict`, либо – еще удобнее – встроенным в Granite пайплайном прогнозирования, который сразу вернет нам результат в исходном масштабе.

Прогнозирование на новых данных (после fine-tuning)

Способ 1: Через Trainer/Predictor вручную. Если вы предпочитаете ручной подход, можно использовать `trainer.predict(test_dataset)`. Этот метод вернет объект `PredictionOutput` с полем `predictions` – содержащим прогнозируемые значения. Однако, в текущей реализации TTM, формат вывода не настолько прямой: модель может возвращать tensor, который надо интерпретировать. Кроме того, надо не забыть конвертировать результат обратно к исходному масштабу. Для обратного преобразования удобно использовать методы препроцессора. Например, `tsp.inv_transform(predictions)` (гипотетически) или хранить где-то `tsp.scalers` для каждого канала. В официальных примерах IBM поступают иначе – они используют специальный пайплайн.

Способ 2: Через TimeSeriesForecastingPipeline (рекомендуется). Библиотека Granite-TSFM предоставляет высокоуровневый класс `TimeSeriesForecastingPipeline`, упрощающий инференс. Он инкапсулирует модель и препроцессор: вы подаете на вход исходный DataFrame, а на выходе получаете DataFrame с добавленными колонками прогноза. Внутри он сам выполнит масштабирование входа, запустит модель и сделает обратное масштабирование прогноза.

Мы воспользуемся этим подходом для удобства и наглядности:

```
from tsfm_public import TimeSeriesForecastingPipeline

pipeline = TimeSeriesForecastingPipeline(
    model,               # наша fine-tuned модель (trainer.model)
    feature_extractor=tsp,   # наш препроцессор, чтобы pipeline понимал структуру и скейлинг
    device="cuda",           # используем GPU, если доступен (pipeline сам перенесет тензоры)
    batch_size=64            # батч для инференса, можно поставить больше/меньше в зависимости от памяти
)

# Получение прогноза на тестовых данных
forecast_df = pipeline(test_data)  # test_data тот же DataFrame, что мы передавали в препроцессор
```

На выходе мы получим DataFrame `forecast_df`, содержащий копию данных `test_data` и новые колонки с суффиксом `_prediction` для каждого целевого ряда [38]. Например, для `target_columns=["value1","value2"]` появятся столбцы `"value1_prediction"` и `"value2_prediction"`. Эти прогнозы уже преобразованы обратно в исходный масштаб (pipeline позаботился об инверсном преобразовании).

Теперь можно оценить качество: например, вычислить MAE, RMSE между `value1` и `value1_prediction` (и аналогично для `value2`). В IBM-примере определены функции для

расчета MSE, RMSE, MAE по полученным колонкам [39] [40]. Вы можете просто воспользоваться pandas/numpy для расчета ошибок. Предположим, для простоты наш целевой ряд один – `"value1"`. Тогда:

```
y_true = forecast_df["value1"].values
y_pred = forecast_df["value1_prediction"].values
mae = np.mean(np.abs(y_pred - y_true))
rmse = np.sqrt(np.mean((y_pred - y_true)**2))
print(f"MAE = {mae:.2f}, RMSE = {rmse:.2f}")
```

Полученные значения покажут качество прогноза в тех же единицах, что и исходные данные. В наших тестах, например, RMSE составлял порядка 1000 при уровне продаж в десятки тысяч (что довольно точно) [41].

Помимо численных метрик, полезно **визуализировать прогноз** вместе с фактическими значениями. В toolkit Granite есть функция `plot_predictions`, которая на одном графике показывает исторические данные, реальные значения и предсказания на интервале прогноза [42]. Ей нужно передать исходный DataFrame (или его часть) и DataFrame с прогнозом, а также указать, какие столбцы являются временным, целевым и т.д.:

```
from tsfm_public.toolkit.visualization import plot_predictions

plot_predictions(
    input_df   = test_data[test_data["date"] >= "2025-01-01"],  # например,
последние месяцы теста
    predictions_df = forecast_df[forecast_df["date"] >= "2025-01-01"],
    freq="H",                          # частота данных, например "H" для
часов (или "D" для дней и т.п.)
    timestamp_column="date",
    channel="value1"                   # какой ряд рисуем (если несколько
целевых, по одному за раз)
)
```

На графиках вы сможете наглядно убедиться, где модель дает точный прогноз, а где ошибается. Анализ таких ошибок может подсказать, какие внешние факторы добавить или как настроить модель.

Заключение

Мы рассмотрели полный процесс тонкой настройки (fine-tuning) модели **TinyTimeMixer R2** для задачи прогноза временных рядов – от подготовки данных до получения финального прогноза. Ключевые моменты, которые стоит подчеркнуть:

- **Совместимость данных с моделью:** необходимо задать правильные спецификации длины контекста и горизонта прогноза, а также структурировать данные (включая все целевые и экзогенные признаки) в формате, ожидаемом моделью. Библиотека Granite-TSFM предоставляет для этого удобный `TimeSeriesPreprocessor` [43] [44] и связанные утилиты.

- **Масштабирование (Normalization):** обязательно нормируйте каждый временной ряд (канал) отдельно перед подачей в модель [5], иначе результаты могут быть плохими. Препроцессор может автоматически стандартировать данные.
- **Использование предобученных весов:** TTM R2 – предобученная на огромном разнообразии данных модель, поэтому **fine-tuning** сводится к адаптации небольшой части её параметров под вашу задачу. Мы загружаем модель с Hub, включаем `mix_channel` режим декодера и замораживаем backbone. Это позволяет **учесть межканальные корреляции** в ходе дообучения, не разрушив уже выученные паттерны динамики [27] [7].
- **Экзогенные и категориальные данные:** Модель поддерживает добавление известных будущих факторов (экзогенных регрессоров) и статических категориальных признаков. При их наличии нужно корректно настроить препроцессор (параметры `control_columns`, `encode_categorical=True`) и передать модели индексы экзогенных каналов и размеры словарей категорий. Также следует установить `enable_forecast_channel_mixing=True`, чтобы эти факторы действительно влияли на прогноз [31].
- **Процесс обучения:** мы используем HuggingFace Trainer для удобства – он справляется с циклом обучения/валидации, применяет шедулер OneCycle для шага обучения и позволяет легко внедрить раннюю остановку [10] [45]. Благодаря небольшому числу обучаемых параметров, fine-tuning можно выполнять даже на CPU, хотя на GPU быстрее [46].
- **Прогнозирование:** после обучения мы получили специализированную модель, которую можно применять на новых данных. Для прогнозирования удобно применять `TimeSeriesForecastingPipeline`, возвращающий результаты в удобном формате DataFrame [16]. Модель TTM R2 способна предсказывать сразу несколько шагов вперёд (заданное `prediction_length`) одновременно, то есть формирует *многошаговый прогноз* одним вызовом. Если требуется прогноз на горизонт больше, чем `prediction_length`, можно использовать **роллинг-прогноз**: например, взять последние 512 точек, получить прогноз на 96 шагов, затем сдвинуть окно и снова подать модель и т.д. – либо воспользоваться утилитами `RecursivePredictor`/режимом `force_return="rolling"` в `get_model()` [47] [48].
- **Анализ результатов:** Оцените метрики MSE, MAE, RMSE на тестовых данных, убедитесь, что они улучшаются по сравнению с нулевым прогнозом или исходной моделью без дообучения. Визуализируйте несколько временных интервалов прогноза vs факта, чтобы проверить, адекватно ли модель отреагировала на паттерны (тренды, сезонность, всплески). Помните, что все ошибки оценивали на **обратном преобразовании к исходным единицам**, чтобы интерпретация была корректной.

Следуя этому пошаговому процессу, вы сможете с высокой степенью детализации и контроля выполнить fine-tuning модели TTM R2 на ваших данных и получить точные прогнозы. Этот подход разработан авторами модели с учетом многолетнего опыта работы с временными рядами, поэтому учитывает множество нюансов – от нормализации данных до специальных режимов работы декодера. При возникновении вопросов вы всегда можете обратиться к приведенному исследованию – в нем раскрыты все аспекты, необходимые для успешного дообучения и прогнозирования с помощью Granite Time Series TTM R2. Все этапы сопровождены соответствующими фрагментами кода и пояснениями, так что данное руководство может служить надежной опорой при реализации вашего проекта прогнозирования. Удачного моделирования!

Sources: Использована документация и примеры из репозитория Granite-TSFM [43] [6], официального руководства IBM по fine-tuning TTM [28] [9], а также дополнительные разъяснения

из статьи об архитектуре TTM [1] и обучающих материалов IBM [16] – все ссылки на первоисточники приведены по ходу текста.

---

[1] [2401.03955] Tiny Time Mixers (TTMs): Fast Pre-trained Models for Enhanced Zero/Few-Shot Forecasting of Multivariate Time Series
https://ar5iv.labs.arxiv.org/html/2401.03955

[2] [3] [4] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [17] [18] [26] [27] [28] [29] [35] [43] [44] [45] Fine tuning Guide for IBM Granite Time Series – IBM Granite
https://www.ibm.com/granite/docs/fine-tune/time-series/

[5] [25] [30] [37] [47] ibm-granite/granite-timeseries-ttm-r2 · Hugging Face
https://huggingface.co/ibm-granite/granite-timeseries-ttm-r2

[16] [19] [20] [21] [22] [23] [24] [31] [32] [33] [34] [36] [38] [39] [40] [41] [42] [46] Time Series Sales Forecasting with Granite | IBM
https://www.ibm.com/think/tutorials/time-series-sales-forecasting-with-granite

[48] Update README.md · ibm-granite/granite-timeseries-ttm-r2 at ...
https://huggingface.co/ibm-granite/granite-timeseries-ttm-r2/commit/07c373415737ba8eb1ab061f1fc75878f8360f01