REALISATION

The making of a temporal pipeline (walkthrough)

Rijn, Tymo van

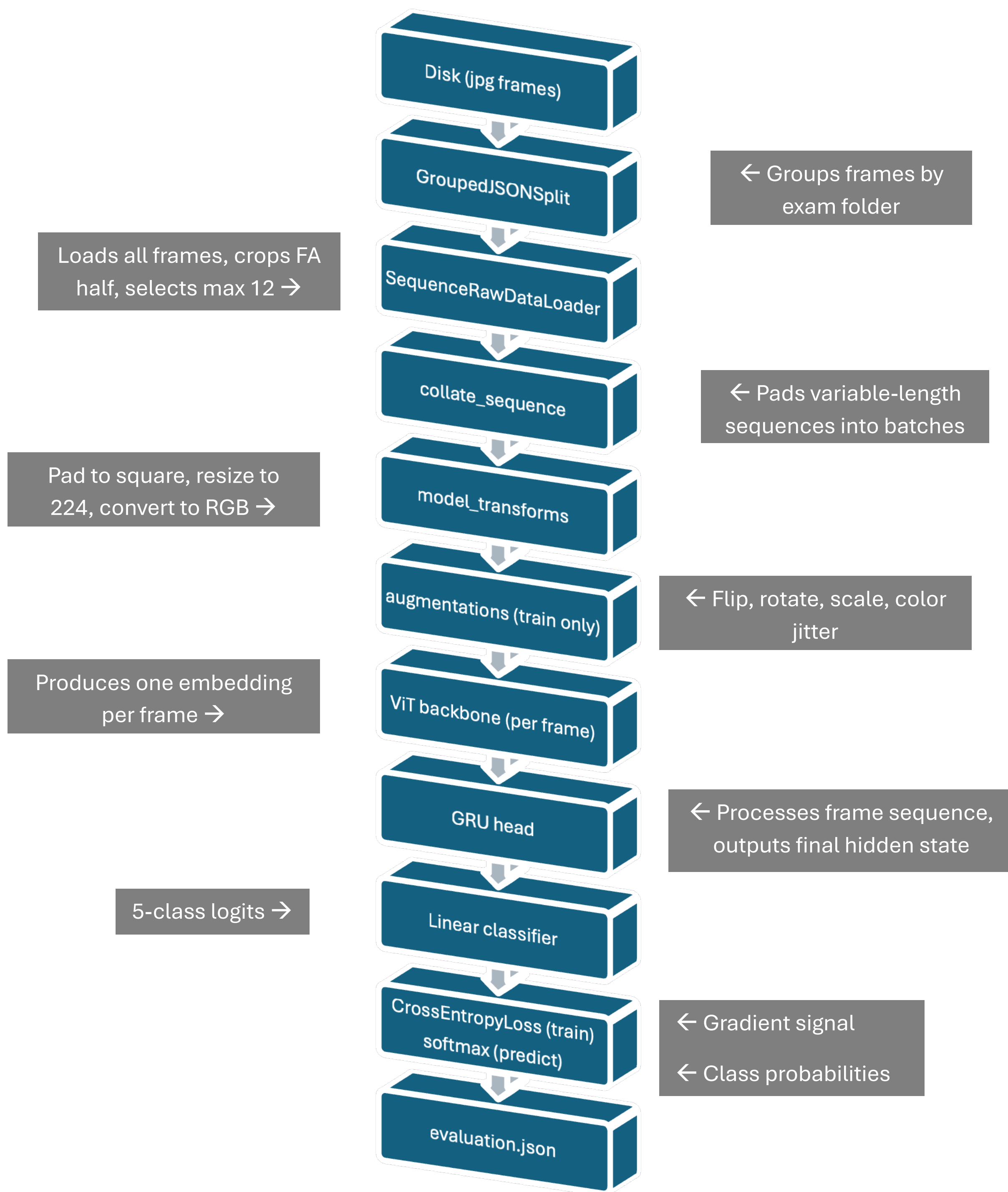
Idiap Research Institute | Rue de Marconi 19

What this project does

We have a dataset of **eye angiography exams** (AngioReport now, later HOJG, hospital). Each exam has multiple frames taken over time. The goal is to classify each exam into one of 5 categories based on the type of abnormal fluorescence seen. This task is called **HyperF_Type** (multiclass, 5 classes).

Instead of looking at just the last frame (like a standard single-image classifier), this pipeline uses the **entire sequence of frames** per exam and processed them temporally

Pipeline overview



Step 1 – The split JSON

File: config/hyperftype.json

This JSON file contains the train/validation/test split. Each entry is a list of [frame_path,label] pairs. The *frame_path* points to one specific frame inside an exam folder.

Example structure:

```
{
    "train": [
        ["Train/Train/0_L/7.jpg", 0],
        ["Train/Train/1_R/12.jpg", 2],
        ...
    ],
    "validation": [...],
    "test": [...]
}
```

Labels map to HyperF_Type classes:

- 0 → Leakage
- 1 → Staining
- 2 → No
- 3 → Pooling
- 4 → Window defect

Important note: The original single-frame loader used only the last frame path per exam. This temporal loader reads the exam folder from that path, then discovers all frames inside it.

Step 2 – Data config (what gets instatiated)

File: mednet/src/mednet/config/temporal/data/angioreport/hyperftype_seq.py

This is the config file that mednet discovers via its *angioreport-hyperftype-seq* entry point. It instantiates our *DataModule* with all parameters fixed for this task.

```
# From: mednet/src/mednet/config/temporal/data/angioreport/hyperftype_seq.py

import importlib.resources
from mednet.data.temporal.seq_angioreport import DataModule

datamodule = DataModule(
    split_path=importlib.resources.files("mednet.config.classify.data.angioreport")
    / "hyperftype.json",
    num_classes=5,
    problem_type="multiclass",
    modality="FA",       # crop left half of the side-by-side image
    max_frames=12,       # use at most 12 frames per exam
    max_frames_span="uniform",  # pick evenly across the timeline
)
```

Step 3 – Grouping frames by exam

File: mednet/src/mednet/data/temporal/seq_angioreport.py — class

GroupedJSONSplit

The standard split JSON maps each exam to its last frame path. I had to group by exam folder, not by frame.

Twhat *GroupedJSONSplit* does:

- Reads the JSON
- Extracts the **exam folder** from each frame path using *Path(frame_path).parent*
- Deduplicates so each exam appears exactly once
- Exposes (*exam_folder_rel, label*) tuples instead of (*frame_path, label*)

```
# From: mednet/src/mednet/data/temporal/seq_angioreport.py
# Lines 35–64

import json
import pathlib
import typing

class GroupedJSONSplit:
    """Read a frame-level split JSON and expose one sample per exam."""

    def __init__(self, path) -> None:
        with path.open() as f:
            raw = json.load(f)

        self._splits: dict[str, list[tuple[str, typing.Any]]] = {}
        for split_name, entries in raw.items():
            grouped: dict[str, typing.Any] = {}
            for frame_path, label in entries:
                # "Train/Train/0_L/7.jpg" → "Train/Train/0_L"
                exam_folder = str(pathlib.Path(frame_path).parent)
                grouped[exam_folder] = label
            self._splits[split_name] = list(grouped.items())

# After grouping, each split entry looks like:
# ("Train/Train/0_L", 0)  ← exam folder + class label
# instead of:
# ("Train/Train/0_L/7.jpg", 0)  ← single frame path + class label
```

Step 4 – Loading frames from disk

File: mednet/src/mednet/data/temporal/seq_angioreport.py — class

SequenceRawDataLoader

For each exam, this loader:

1. Lists all .jpg files in the exam folder and sorts them by frame number.
2. Selects at most *max_frames* of them (uniform subsampling or last *N*).
3. Loads ach frame as a grayscale PIL image.
4. Crops the **FA half** (left half, minus 50 px bottom strip. This is because the frame consists of a FA image on the left and a OCTG on the right, and we are just interested in the FA image).
5. Converts to a float32 tensor (*1, H, W)* in range [*0, 1*]
6. Pads frames to the same spatial size (some exams have different resolutions).
7. Stacks into a single tensor (*T, 1 , H, W*) where T = number of selected frames.

```
8.  # From: mednet/src/mednet/data/temporal/seq_angioreport.py
9.  # Lines 104-170 (key methods shown)
10.
11. import PIL.Image
12. import torch
13. from torchvision import tv_tensors
14. from torchvision.transforms.v2.functional import to_dtype, to_image
15.
16. # ---- Loading a single frame ----
17.
18. def _load_one_frame(self, path):
19.     image = PIL.Image.open(path).convert("L")   # greyscale
20.     width, height = image.size
21.     # Crop left half for FA modality (right half = ICGA)
22.     box = (0, 0, width - width / 2, height - 50)
23.     image = image.crop(box)
24.     image = to_dtype(to_image(image), torch.float32, scale=True)  # → [0, 1]
25.     return tv_tensors.Image(image)   # shape: (1, H, W)
26.
27. # ---- Selecting frames ----
28.
29. def _select_frames(self, frame_paths):
30.     if self.max_frames is None or len(frame_paths) <= self.max_frames:
31.         return frame_paths    # exam has fewer frames than cap → use all
32.
33.     if self.max_frames_span == "last":
34.         return frame_paths[-self.max_frames:]    # keep most recent frames
35.
36.     # "uniform": evenly spaced across the full timeline
37.     indices = [
38.         round(i * (len(frame_paths) - 1) / (self.max_frames - 1))
39.         for i in range(self.max_frames)
```

```
40.     ]
41.     return [frame_paths[i] for i in indices]
42.
43. # ---- Main sample() method ----
44.
45. def sample(self, sample):
46.     exam_folder_rel, _ = sample
47.     exam_dir = self.datadir / exam_folder_rel
48.
49.     # Sort by frame number (file stems are integers: 1.jpg, 2.jpg, ...)
50.     frame_paths = sorted(exam_dir.glob("*.jpg"), key=lambda p: int(p.stem))
51.     frame_paths = self._select_frames(frame_paths)
52.     frames = [self._load_one_frame(p) for p in frame_paths]
53.
54.     # Pad frames to same H,W then stack into (T, 1, H, W)
55.     images = tv_tensors.Image(self._pad_to_max_hw([torch.as_tensor(f) for f
    in frames]))
56.
57.     return dict(
58.         image=images,                # shape: (T, 1, H, W)  — the sequence
59.         target=self.target(sample), # shape: scalar long tensor
60.         lengths=len(frame_paths),   # how many real frames (T)
61.         name=exam_folder_rel,       # string ID for tracking
62.     )
```

Step 5 – Batching variable-length sequences

File: mednet/src/mednet/data/temporal/seq_angioreport.py — function

collate_sequence

Each exam can have a different number of frames, (T_i), PyTorch's default collate cannot stack tensors with different first dimensions.

My custom *collate_sequence* function:

1. Pads shorter sequences with zeros to match the longest sequence in the batch.
2. Stacks everything into a single (B, T_max, C, H, W) tensor.
3. Records the real lenghts (B,) so the GRU can ignore the padding.

```
4.  # From: mednet/src/mednet/data/temporal/seq_angioreport.py
5.  # Lines 181–189
6.
7.  import torch.nn.utils.rnn
8.
9.  def collate_sequence(batch):
10.     """Collate variable-length sequences with zero-padding on the time
    axis."""
11.     images = [torch.as_tensor(sample["image"]) for sample in batch]
12.     targets = torch.utils.data.default_collate([sample["target"] for sample
    in batch])
13.     lengths = torch.tensor([sample["lengths"] for sample in batch],
    dtype=torch.long)
14.     names = [sample["name"] for sample in batch]
15.
16.     # pad_sequence pads shorter sequences to the length of the longest one
17.     padded = torch.nn.utils.rnn.pad_sequence(images, batch_first=True)
18.     # padded shape: (B, T_max, C, H, W)
19.
20.     return dict(image=padded, target=targets, lengths=lengths, name=names)
21.
22. # Example batch result:
23. # {
24. #    "image":   tensor of shape (2, 12, 1, 224, 224)  ← 2 exams, 12 frames
    each (shorter one padded)
25. #    "target":  tensor([0, 2])                         ← class labels
26. #    "lengths": tensor([12, 6])                        ← exam 0 has 12 real
    frames, exam 1 has 6
27. #    "name":    ["Train/0_L", "Train/1_R"]              ← exam folder IDs
28. # }
```

Step 6 – Model transforms (preprocessing before model input)

File: mednet/src/mednet/config/temporal/models/vitgru.py

After loading, each frame goes through *model_transforms* before being fed to the ViT. These transforms are applied to every frame independently.

Three steps:

1. *SquareCenterPad* → pad image to a square (centre-alligned, fill with black) so ViT gets a consistent aspect ratio.
2. *Resize(224)* → resize to 224x224 as required by the ViT-Small architecture.
3. *RGB()* -> convert single grayscale channel to 3-channel RGB (ViT expects 3 channels)

```
4.  # From: mednet/src/mednet/config/temporal/models/vitgru.py
5.  # Lines 27–35
6.
7.  import torchvision.transforms
8.  import torchvision.transforms.v2
9.  import mednet.models.transforms
10.
11. model_transforms = [
12.     mednet.models.transforms.SquareCenterPad(),    # step 1: pad to square
13.     torchvision.transforms.v2.Resize(
14.         224,
15.         antialias=True,
16.         interpolation=torchvision.transforms.InterpolationMode.BICUBIC,
17.     ),                                             # step 2: resize to
    224×224
18.     torchvision.transforms.v2.RGB(),               # step 3: 1 channel → 3
    channels
19. ]
20.
21. # After model_transforms, each frame tensor has shape: (3, 224, 224)
22. # Batch after transforms:  (B, T, 3, 224, 224)
```

Step 7 – Data augmentations (training only)

File: seq_finetune/utils/augmentations.py

During training, augmentations are applied per-frame (each frame is augmented independently). They help the model generalise and avoid overfitting. During validation and prediction, no augmentations are applied.

```
# From: seq_finetune/utils/augmentations.py
# Lines 13–26

from torchvision.transforms import v2 as T

def get_augmentations():
    """Return default data augmentations for training."""
    cjitter = T.RandomApply([T.ColorJitter(brightness=0.33, contrast=0.33)],
p=0.25)
    slight_rotation = T.RandomApply([T.RandomRotation(degrees=25)], p=0.25)
    scale = T.RandomApply([T.RandomAffine(degrees=0, scale=(0.8, 1.2))], p=0.75)

    augmentation_list = [
        T.RandomHorizontalFlip(),           # mirror left-right (p=0.5)
        T.RandomVerticalFlip(p=0.1),        # mirror top-bottom (rare, p=0.1)
        scale,                              # random zoom in/out (p=0.75)
        slight_rotation,                    # small random rotation (p=0.25)
        cjitter,                            # brightness/contrast jitter (p=0.25)
    ]
    return augmentation_list


# How augmentations are applied per frame inside the model:
# (from mednet/src/mednet/models/temporal/vitgru.py, lines 148–154)
def _apply_augmentations(self, images):
    if not self.augmentation_transforms.transforms:
        return images
    batch_size, seq_len, channels, height, width = images.shape
    # Flatten time into batch dimension so each frame is treated independently
    flattened = images.view(batch_size * seq_len, channels, height, width)
    augmented = self.augmentation_transforms(flattened)
    return augmented.view(batch_size, seq_len, channels, height, width)
```

Step 8 – The model: ViTGRU

File: mednet/src/mednet/models/temporal/vitgru.py

The model has three components:

ViT Backbone (feature extractor)

- Architecture: *vit_small_patch16_224* from *timm*, pretrained on ImageNet-21K.
- Num_classes=0 means we use it as a feature extractor (no classification head).
- Each frame (3, 224, 224) → one embedding vector of size 384.

GRU (temporal head)

- Takes the sequence of per-frame embeddings (B, T, 384).
- Uses *pack_padded_sequence* so padded frames are skipped.
- Outputs a final hidden strrate: (B, 256) (hidden_dim=256)

Linear classifier

- Takes the final GRU hidden state (B, 256) → 5 logits (B,5)

```
# From: mednet/src/mednet/models/temporal/vitgru.py
# Lines 106–138 (construction) and 156–187 (forward pass)

import timm
import torch
import torch.nn

# ---- Building the model components ----

# 1. ViT backbone (feature extractor, no classification head)
backbone = timm.create_model(
    "vit_small_patch16_224.augreg_in21k",
    img_size=(224, 224),
    global_pool="token",     # use [CLS] token as frame representation
    num_classes=0,           # no head → outputs raw embedding
    pretrained=True,
)
# backbone.num_features → 384  (embedding size for vit_small)

# 2. GRU (temporal head)
gru = torch.nn.GRU(
    input_size=384,     # matches ViT embedding size
    hidden_size=256,    # hidden_dim
    num_layers=1,
    batch_first=True,
    bidirectional=False,
```

```
-   )
-
-   # 3. Classifier
-   classifier = torch.nn.Linear(256, 5)    # 256 → 5 classes
-   dropout_layer = torch.nn.Dropout(0.2)
-
-   # ---- Forward pass ----
-
-   def forward(self, images, lengths):
-       # images shape: (B, T_max, 3, 224, 224)
-       # lengths shape: (B,)  ← real number of frames per exam
-
-       batch_size, seq_len, channels, height, width = images.shape
-
-       # Step 1: flatten time into batch and normalise
-       flat_images = images.view(batch_size * seq_len, channels, height, width)
-       flat_images = self.normalizer(flat_images)  # z-normalise per channel
-
-       # Step 2: run ViT on every frame at once
-       frame_embeddings = self.backbone(flat_images)   # (B*T, 384)
-       frame_embeddings = frame_embeddings.view(batch_size, seq_len,
    self.embedding_dim)
-       # frame_embeddings shape: (B, T_max, 384)
-
-       # Step 3: pack so GRU ignores padded frames
-       packed = torch.nn.utils.rnn.pack_padded_sequence(
-           frame_embeddings,
-           lengths.cpu(),
-           batch_first=True,
-           enforce_sorted=False,
-       )
-
-       # Step 4: run GRU — we only need the final hidden state
-       _, hidden = self.gru(packed)   # hidden shape: (1, B, 256)
-       last_hidden = hidden[-1]       # (B, 256)
-
-       # Step 5: classify
-       return self.classifier(self.dropout_layer(last_hidden))  # (B, 5) logits
```

Step 9 – Model configuration (what gets instantiated)

File: mednet/src/mednet/config/temporal/models/vitgru.py

This is the config file that mednet discovers via its *temporal-vitgru* entry point. It wires together all model hyperparameters in one place.

```
# From: mednet/src/mednet/config/temporal/models/vitgru.py
# Complete file

import torch.nn
import torch.optim
import torchvision.transforms
import torchvision.transforms.v2
import mednet.models.temporal.vitgru
import mednet.models.transforms

model = mednet.models.temporal.vitgru.ViTGRU(
    architecture="vit_small_patch16_224.augreg_in21k",
    pretrained=True,
    img_size=224,
    global_pool="token",        # use [CLS] token
    hidden_dim=256,             # GRU hidden size
    num_layers=1,               # single GRU layer
    dropout=0.2,                # dropout before classifier
    bidirectional=False,        # unidirectional (chronological order)
    num_classes=5,              # HyperF_Type has 5 classes
    loss_type=torch.nn.CrossEntropyLoss,
    optimizer_type=torch.optim.AdamW,
    optimizer_arguments=dict(lr=0.0001, weight_decay=0.01),
    model_transforms=[
        mednet.models.transforms.SquareCenterPad(),
        torchvision.transforms.v2.Resize(
            224,
            antialias=True,
            interpolation=torchvision.transforms.InterpolationMode.BICUBIC,
        ),
        torchvision.transforms.v2.RGB(),
    ],
)
```

Step 10 – Base temporal model: training/validation steps

File: mednet/src/mednet/models/temporal/model.py

The base temporal model extends the standard MedNet classification base class. It overrides the training, validation, and prediction steps to pass *batch[“lengths”]* to the forward method (which the standard single-frame model does not need).

```
# From: mednet/src/mednet/models/temporal/model.py
# Lines 53–69

def training_step(self, batch, batch_idx):
    del batch_idx
    # Forward pass with both images and lengths, then compute loss
    return self.train_loss(
        self(batch["image"], batch["lengths"]),  # logits: (B, 5)
        batch["target"],                          # labels: (B,)
    )

def validation_step(self, batch, batch_idx, dataloader_idx=0):
    del batch_idx, dataloader_idx
    return self.validation_loss(
        self(batch["image"], batch["lengths"]),
        batch["target"],
    )

def predict_step(self, batch, batch_idx, dataloader_idx=0):
    del batch_idx, dataloader_idx
    # During prediction, apply softmax to get class probabilities (sums to 1)
    return torch.softmax(self(batch["image"], batch["lengths"]), dim=1)
    # output shape: (B, 5) ← probability for each of 5 classes
```

Step 11 – The experiment runner

File: seq_finetune/experiment.py

This is the main script you run. It orchestrates the full pipeline by calling MedNet's CLI commands (*train, predict, evaluate*) throught Python's *CliRunner.*

It also:

- Creates a *Train_N* folder per run so runs don't overwrite each other
- Saves a *Settings.json* inside so you know exactly what was run
- Generates *auimcp.pdf* after evaluation

```
# From: seq_finetune/experiment.py
# Showing the key parts of main()

import importlib

# 1. Load model and datamodule objects from mednet config modules
model =
importlib.import_module("mednet.config.temporal.models.vitgru").model
datamodule =
importlib.import_module("mednet.config.temporal.data.angioreport.hyperftype_
seq").datamodule
augmentations = get_augmentations()

# 2. Create a Train_N folder and save settings.json
output_folder = get_train_dir(output_folder, settings)
# → e.g. results/temporal-hyperftype/Train_1/

# 3. Run training via mednet CLI
result = runner.invoke(train, [
    "--model", model,
    "--datamodule", datamodule,
    "--epochs=40",
    "--batch-size=2",
    "--balance-classes",          # weight loss by class frequency
    "--augmentations", augmentations,
    "--checkpoint-metric", "min/loss",  # save best checkpoint when loss is
lowest
    ...
])

# 4. Run prediction on all splits
result = runner.invoke(predict, [
    "--model", model,
    "--datamodule", datamodule,
    "--weight", output_folder,    # loads the best checkpoint
    ...
])
# → writes predictions.json with per-sample probabilities
```

```
-
-   # 5. Run evaluation (computes AUC, F1, etc.)
-   result = runner.invoke(evaluate, [
-       "--predictions", output_folder / "predictions.json",
-       "--plot",
-       ...
-   ])
-   # → writes evaluation.json and evaluation.pdf
-
-   # 6. Compute AUIMCP and save as PDF
-   imcp_plots = compute_auimcp(output_folder / "predictions.json",
    "multiclass")
-   # → writes auimcp.pdf
```

Step 12 – Output folder structure

After a full run, the output folder looks like this:

```
results/temporal-hyperftype/
└── Train_1/
    ├── settings.json          ← all run parameters (model, epochs, seed, etc.)
    ├── train.meta.json        ← mednet metadata
    ├── last.ckpt              ← last epoch checkpoint
    ├── best-val_loss=....ckpt ← best checkpoint by validation loss
    ├── logs/                  ← tensorboard event files
    ├── train_log.pdf          ← training curves (loss over epochs)
    ├── predictions.json       ← per-sample (label, probability) for all splits
    ├── evaluation.json        ← AUC, F1, etc.
    ├── evaluation.pdf         ← evaluation plots
    └── auimcp.pdf             ← AUIMCP curve (IMCP metric for class imbalance)
```

Step 13 – Output: class probabilities

During *predict_step*, the model outputs class probabilities via softmax. These are saved to *predictions.json*

Example structure in *predictions.json:*

```
{
  "test": [
    ["Train/0_L", 0, [0.02, 0.10, 0.73, 0.08, 0.07]],
    ...
  ]
}
```

Each entry : [*exam_name, true_label, [prob_class_0, ..., prob_class_4]]*

These probabilities are then used by *evaluate* to compute:

- **AUC**
- **F1**
- **AUIMCP**