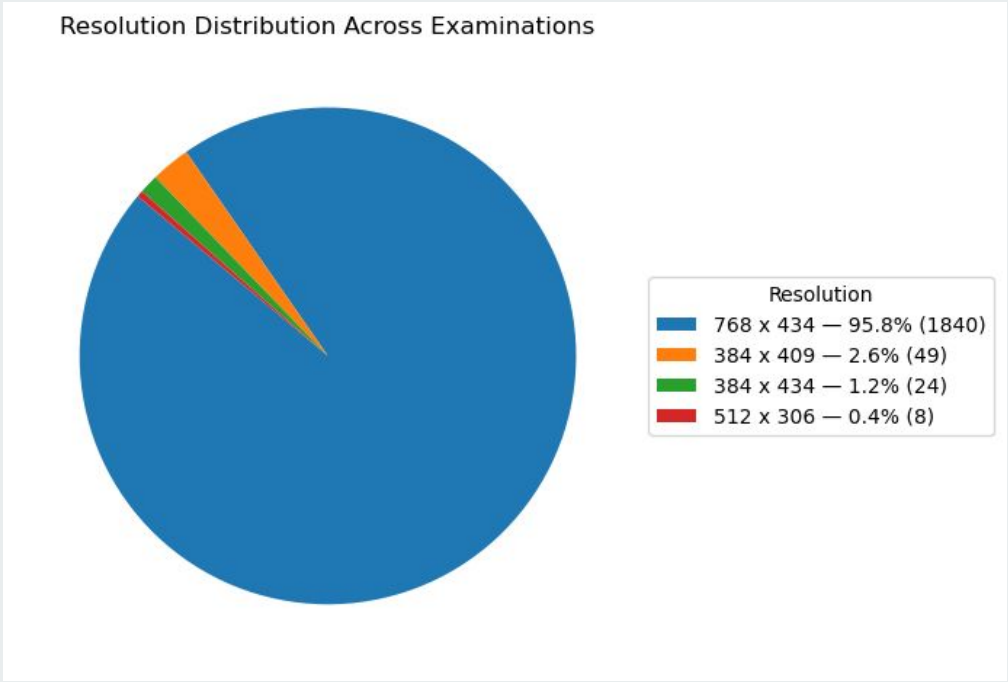
# Tymo’s progress

Week 1

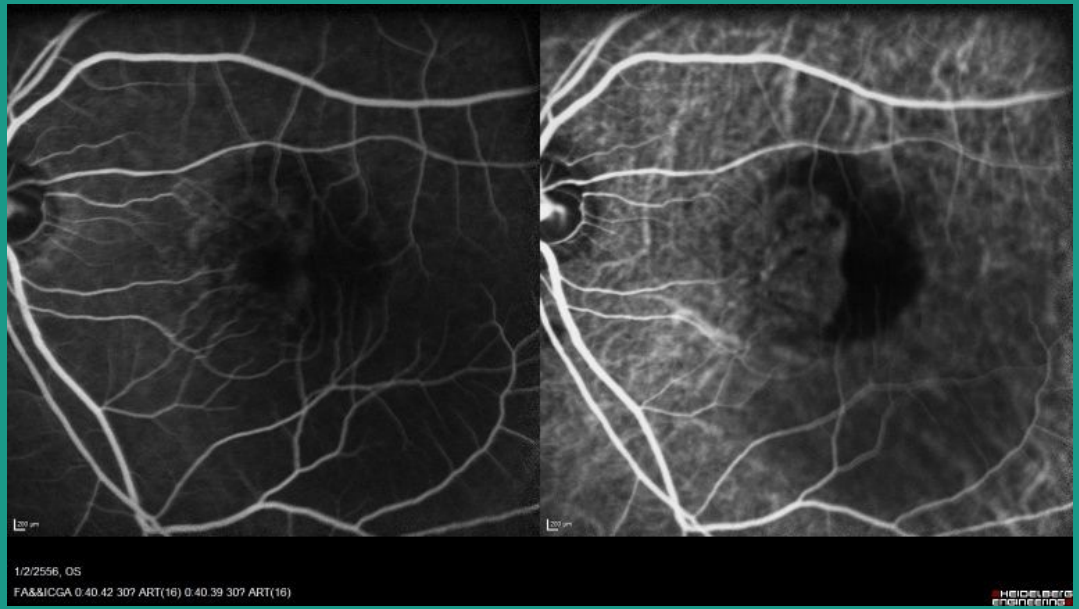
# Goal

*Become one with the data & Mednet*


Resolution Distribution Across Examinations
Resolution
768 x 434 — 95.8% (1840)
384 x 409 — 2.6% (49)
384 x 434 — 1.2% (24)
512 x 306 — 0.4% (8)

# 95.8%

**Resolution: 768 x 434 — 0_L/0.jpg**



# 2.6%

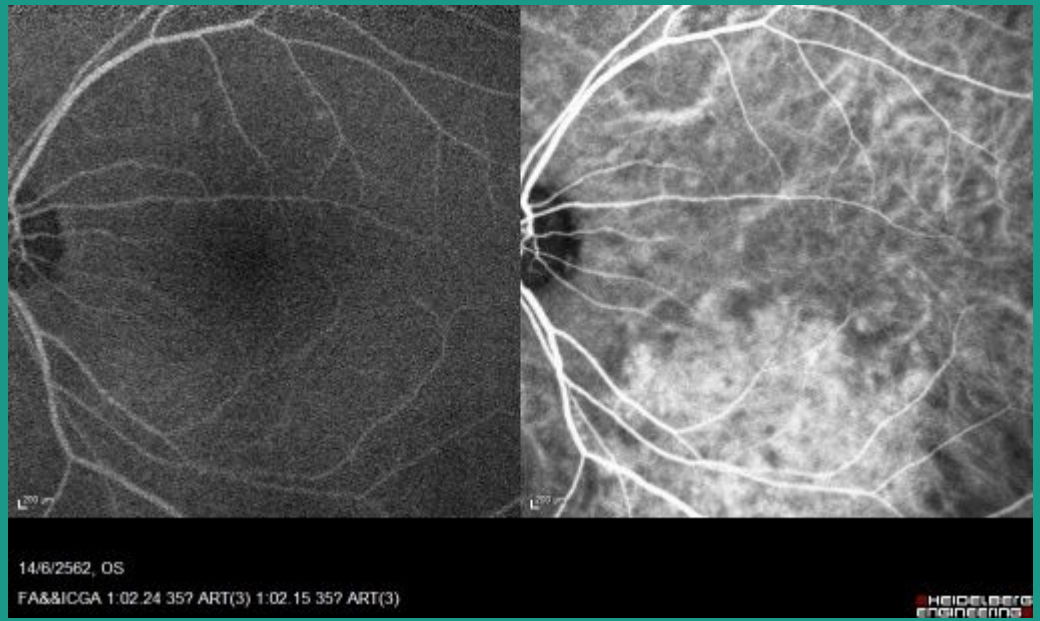
**Resolution: 384 x 434 — 1038_L/0.jpg**

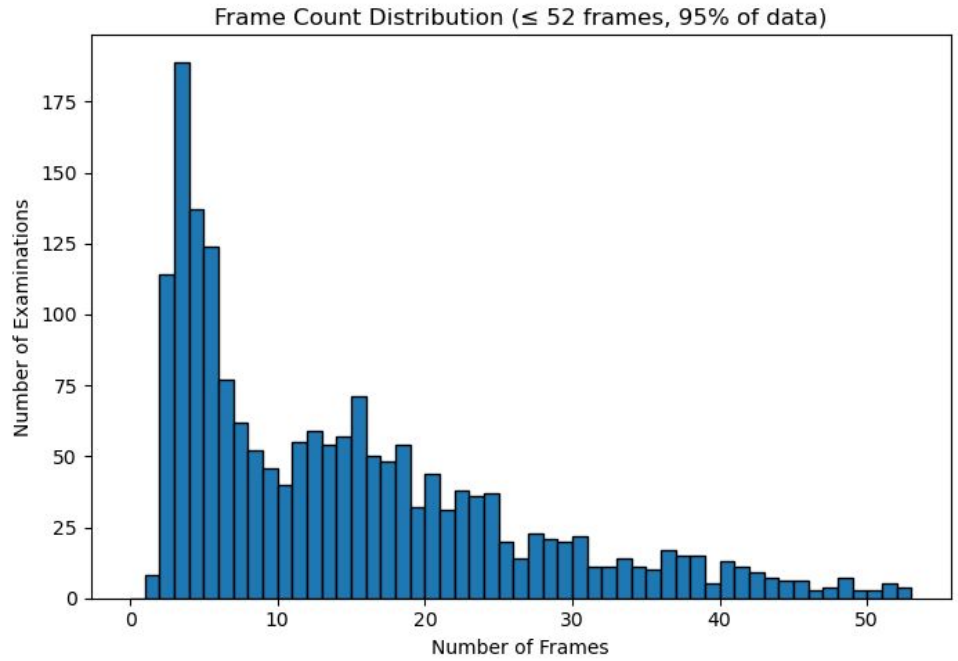
# 1.2%

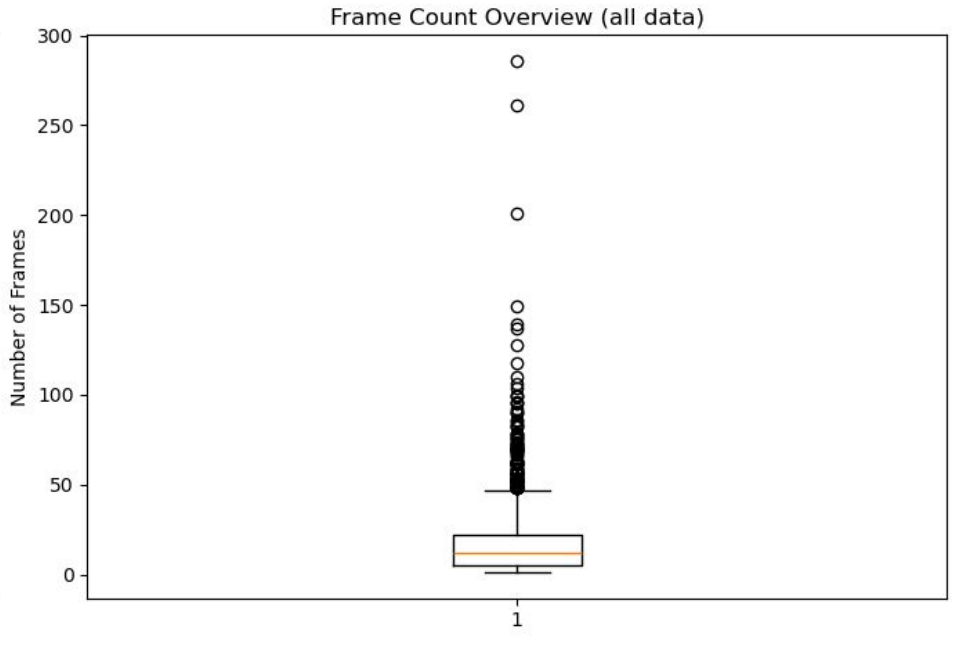
**Resolution: 384 x 409 — 1050_R/0.jpg**

# 0.4%

**Resolution: 512 x 306 — 1523_L/0.jpg**



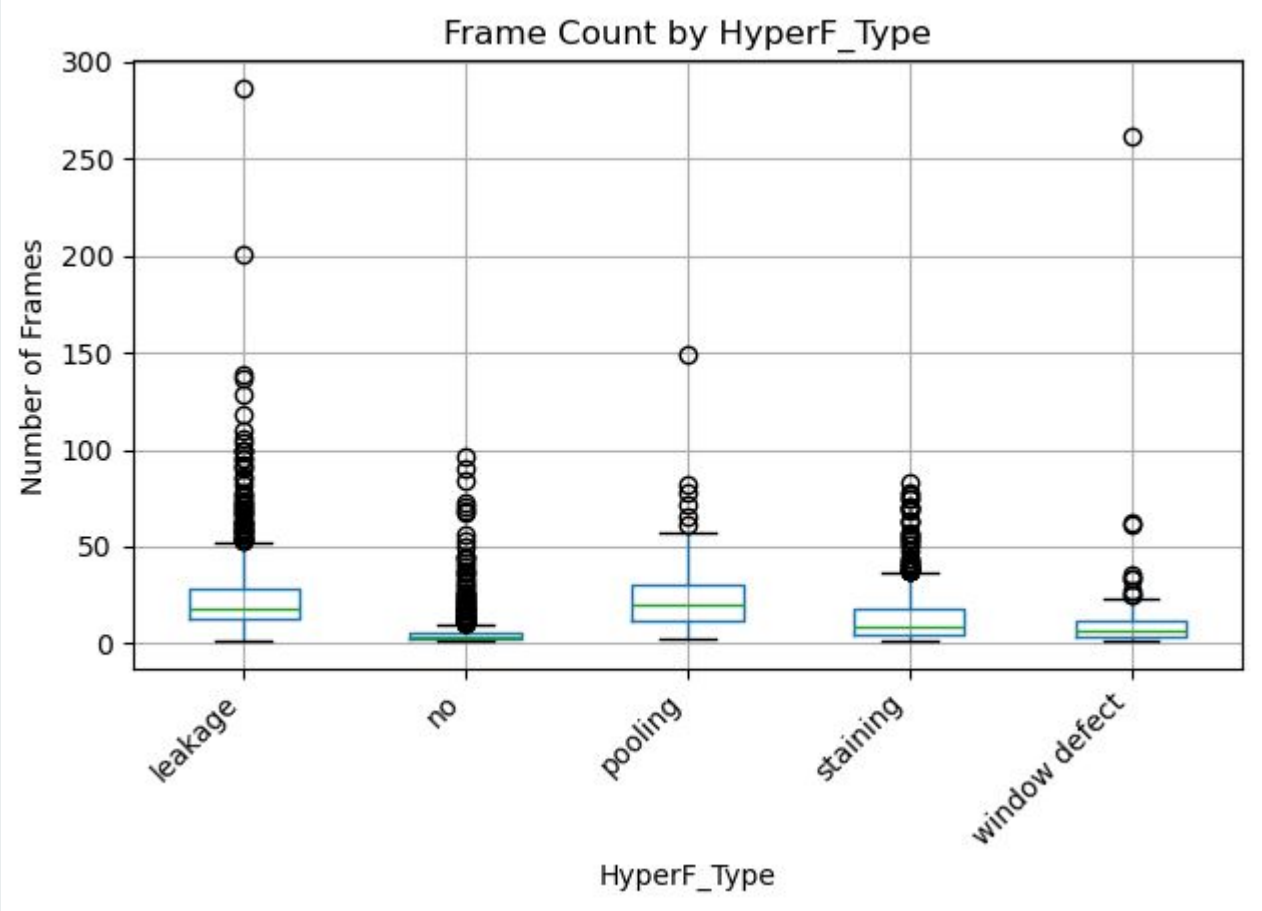

Frame Count Distribution (≤ 52 frames, 95% of data)
Number of Examinations
Number of Frames
0
25
50
75
100
125
150
175
0
10
20
30
40
50


Frame Count Overview (all data)
Number of Frames
0
50
100
150
200
250
300
1

# Frame statistics

Max: 286
Min: 1
Median: 12
Mean: 17.5
Mode: 3

The different HyperF_Types appear to be in the somewhat same range frame count wise. No big discoveries.



# (Full) examinations

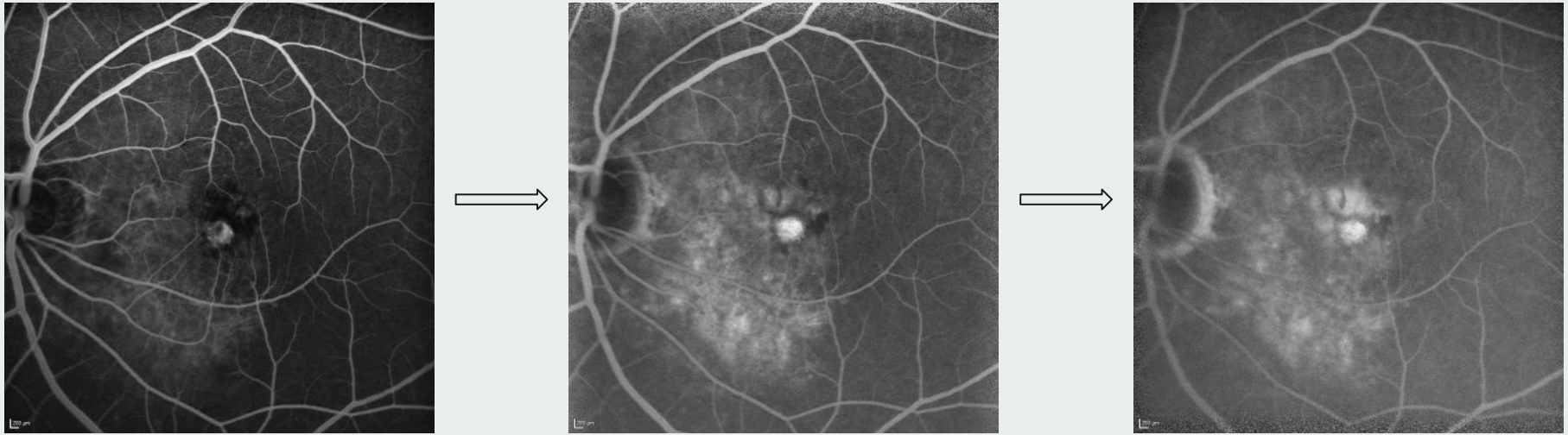
The following slides will contain:
First frame -> Middle frame -> Last frame
To get a feel for the way the FA evolves over time

# Leakage

Progressive hyperfluorescence that spreads and blurs over time, making spatio-temporal data essential to observe spatial expansion

# Leakage

Progressive hyperfluorescence that spreads and blurs over time, making spatio-temporal data essential to observe spatial expansion

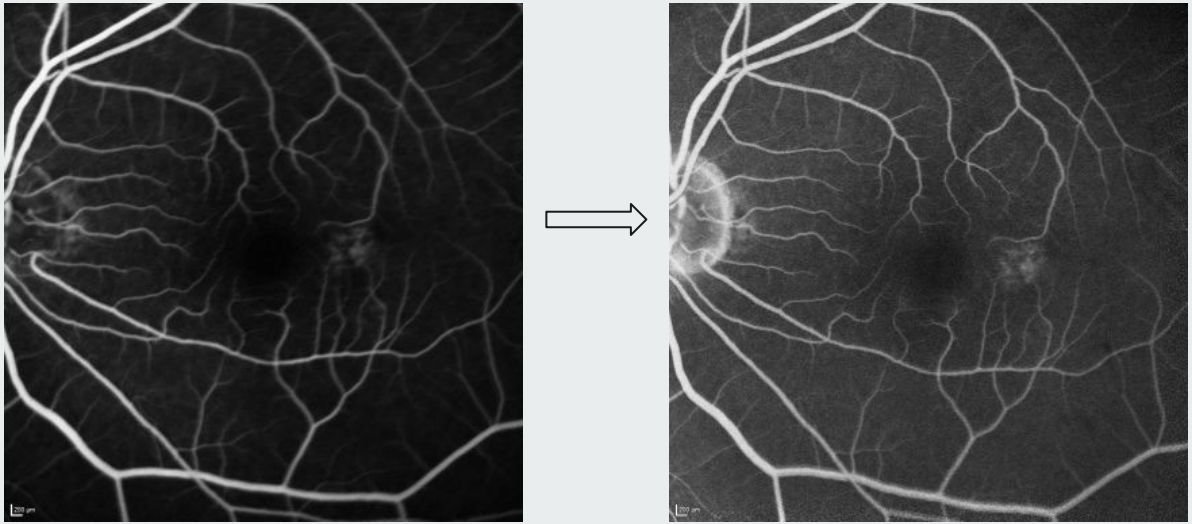


# Staining

Gradual increase in brightness with a stable shape and sharp borders, where spatio-temporal data is important to confirm spatial growth

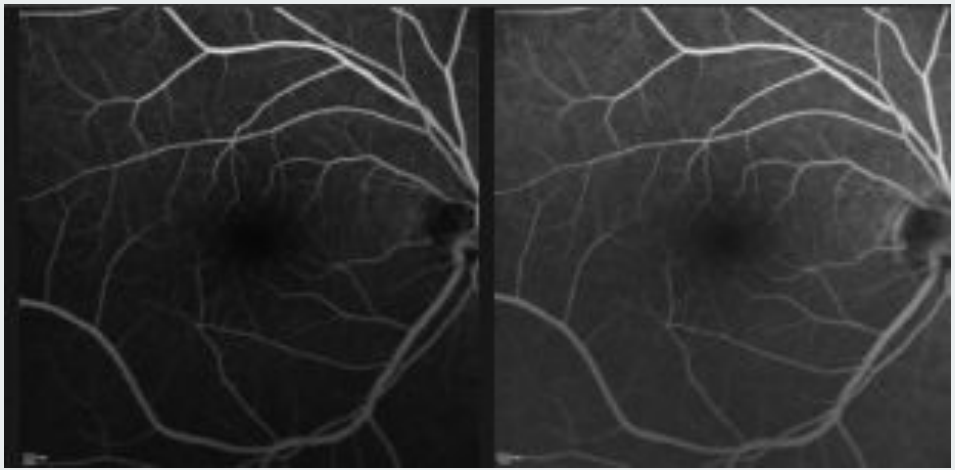
# Staining

Gradual increase in brightness with a stable shape and sharp borders, where spatio-temporal data is important to confirm spatial growth



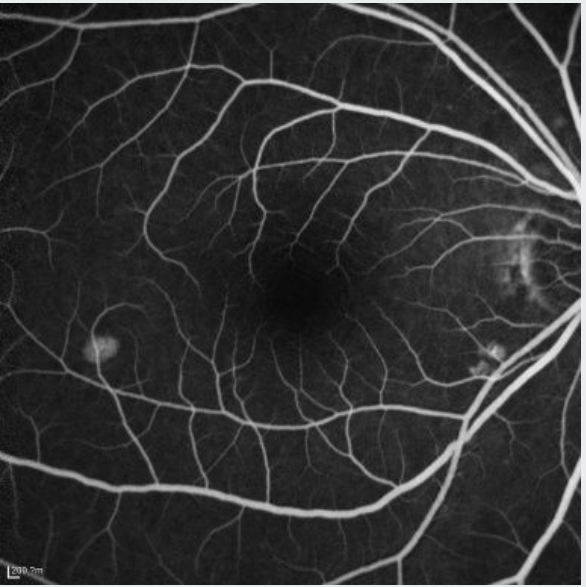
# Nothing

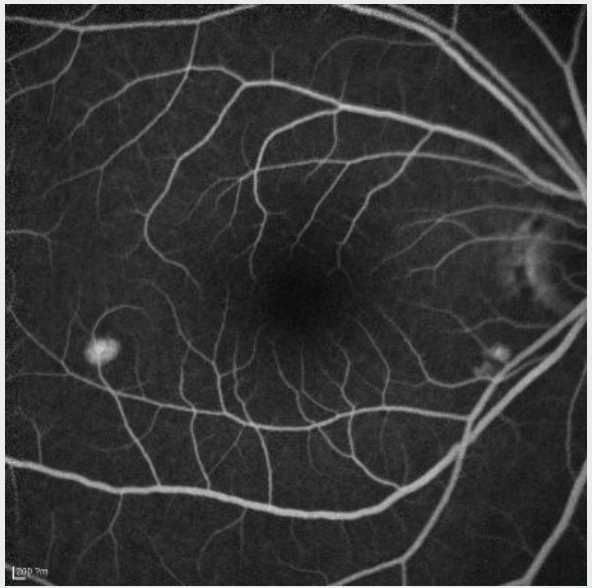
No abnormal hyperfluorescence at any time point, where spatio-temporal data is not necessary

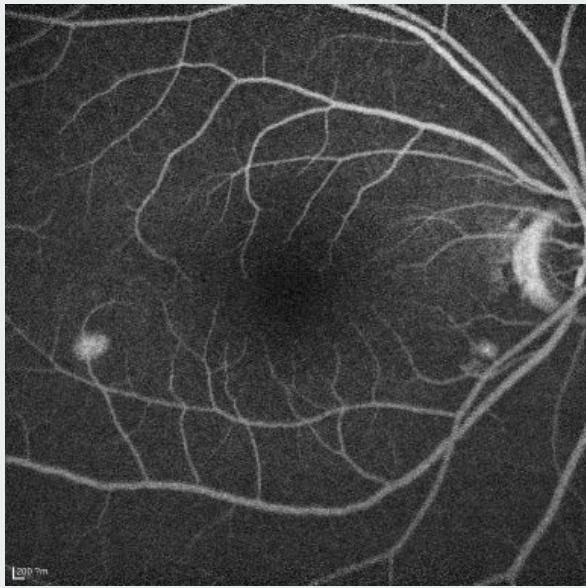


# Pooling

Hyperfluorescence that fills a well-defined anatomical space and remains sharply bounded, so spatio-temporal data is useful but not “really” required.

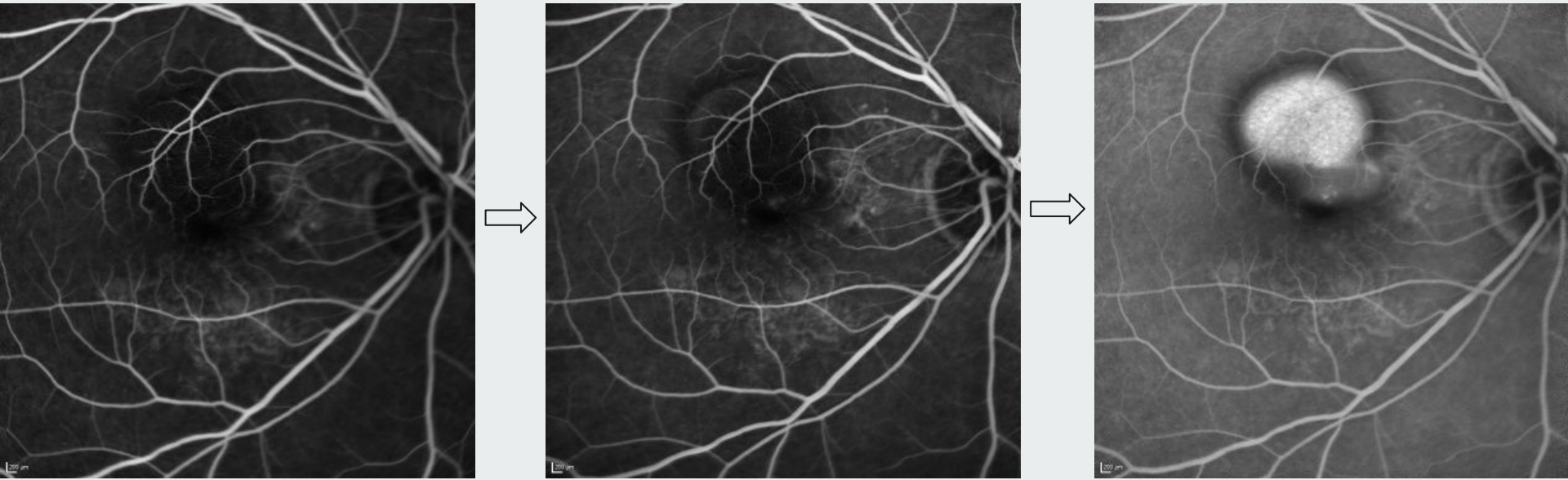






# Pooling

Hyperfluorescence that fills a well-defined anatomical space and remains sharply bounded, so spatio-temporal data is useful but not “really” required.

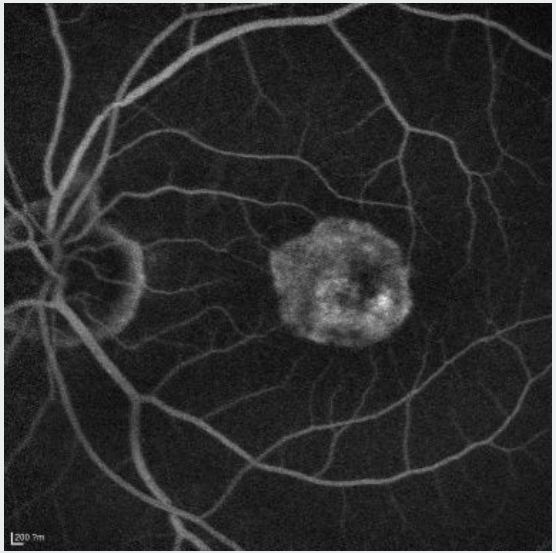


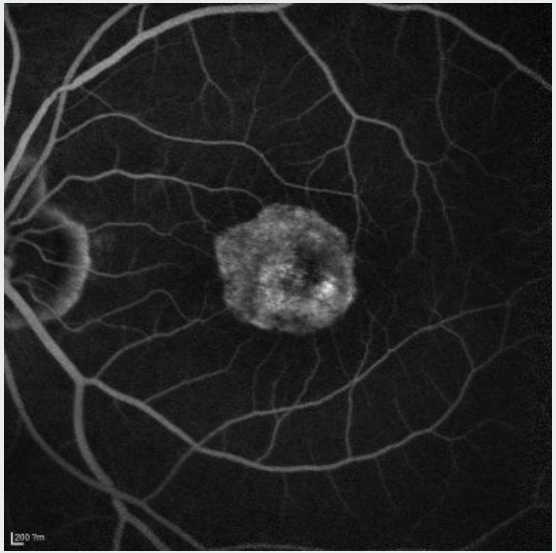
# Pooling

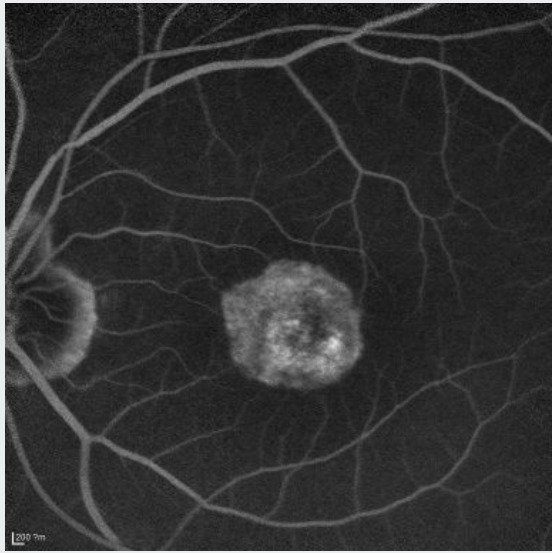
Hyperfluorescence that fills a well-defined anatomical space and remains sharply bounded, so spatio-temporal data is useful but not “really” required.

# Window Defect

Early, well defined hyperfluorescence that does not change in size or intensity, meaning spatio-temporal data might not be important

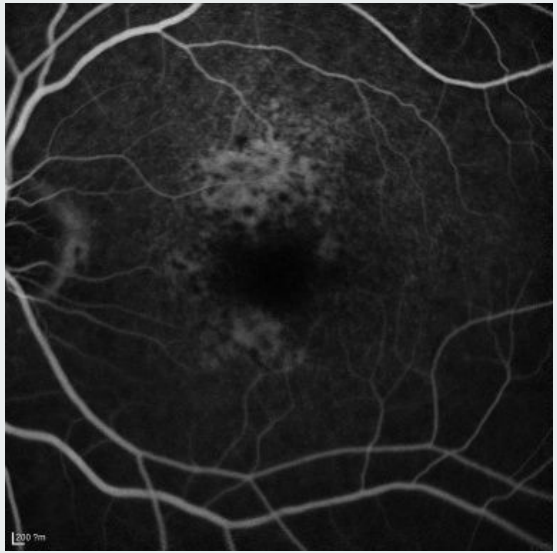


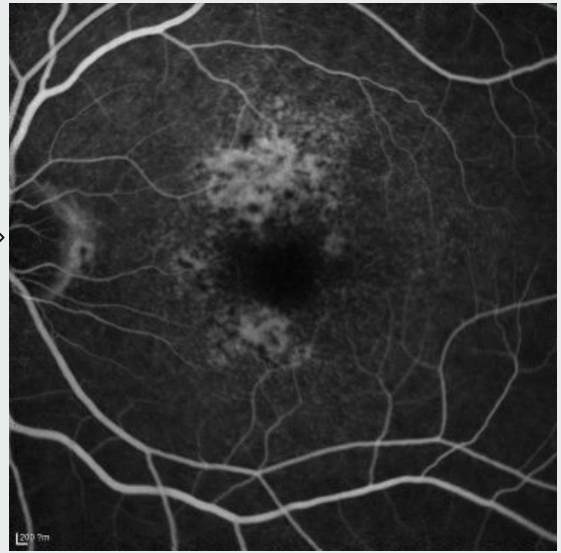


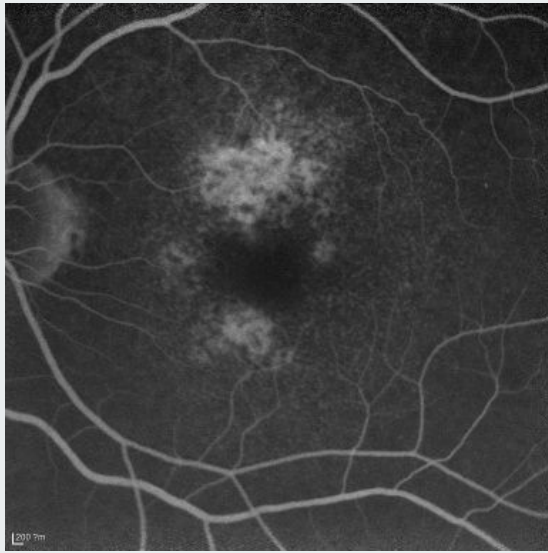


# Window Defect

Early, well defined hyperfluorescence that does not change in size or intensity, meaning spatio-temporal data might not be important







# Study

“Dynamic Contrast-Enhanced (DCE) MRI”

LINK

# Study

**Problem:** Doctors want to understand how blood and contrast move through tissue over time

- The brightness curve is noisy
- Influenced by scanner settings
- Influenced by timing
- Influenced by patient omvement

**Solution (approach):** Instead of treating each image separately, they treated it as a **time problem**

for each pixel:

- track how intensity changes over time
- Convert intensity into something more meaningful (contrast concentration)
- Fit a simple mathematical model that describes how contrast enters and leaves tissue

# Study; Challenges

1. Intensity is not straightforward: brightness doesn't directly equal contrast amount, there is some correction and calibration needed.
2. Timing matters: If the scans aren't evenly spaced or fast enough, details are missed.
3. Variability across hospitals: Different scanners, protocols will contribute in the curves looking different

# Study; Key Principles

The meaning is not in a single image

The meaning is in how the image changes over time

And: The pattern of change often tells more than how bright something is at one moment.

A time curve can be summarized in a few numberS:

- How fast it rises
- How high it peaks
- How fast it falls
- The total AUC

# Study; Our case

Leakage: gets brighter over time, spreads

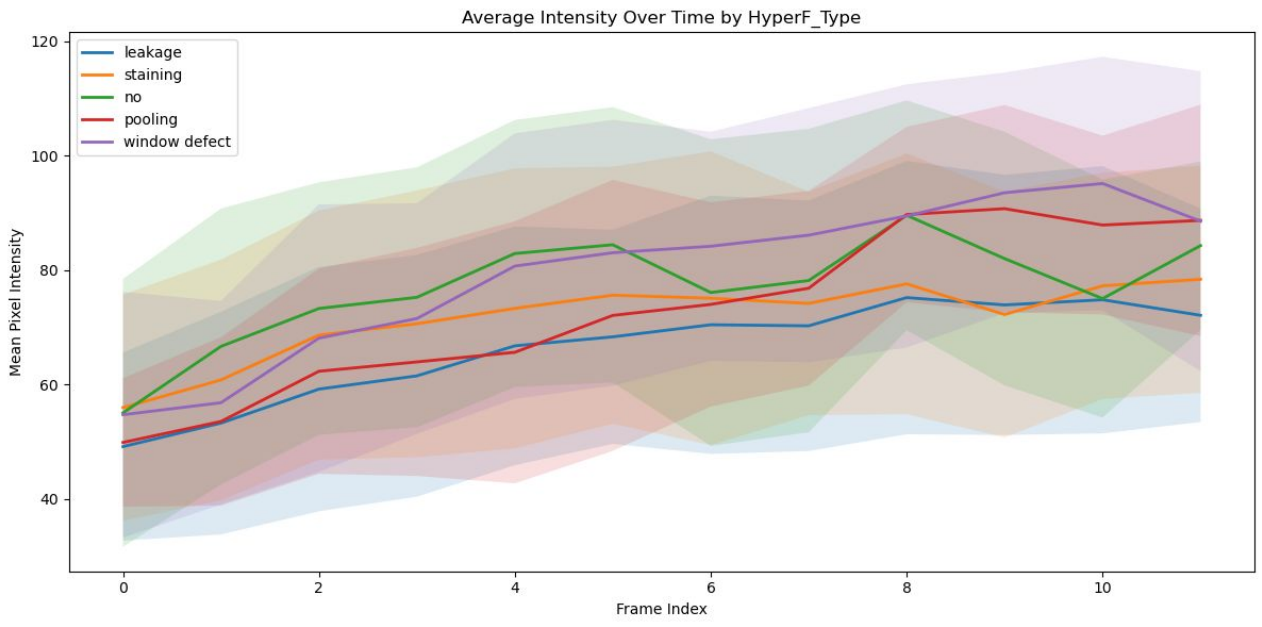
Window defect: Bright immediately, stays stable

Staining: Appears later, Persists but does not expand

If the embeddings of the frames are averaged, the curve is ignored

- So computing differences between embeddings could be helpful to give to models a better understanding of what is happening / changing during an exam. $\Delta$
- Should be thinking about intensity normalization across frames, using a certain baseline from the images, so manual capture changes don't affect the classification too much

# Intensity over time




There seems to be a temporal signal in the shapes of the curves, especially leakage (steep climb from low) vs no (starts high, flattens).

Overlap is large, The intensity alone isn't enough to separate classes.

This would motivate the need for a sequential classification model.

# Accuracy Per Frame Experiment

Extract RETFound-Green features for each examination.
Saves one .pt file per exam containing the full temporal sequence of features.
Each .pt file contains:

- features : tensor of shape (num_frames, 384)
- label : the HyperF_Type string
- folder : the exam folder name
- num_frames : number of frames in this exam

# Accuracy Per Frame Experiment

- The notebook loads all the ~1900.pt files into memory – this is instant since they're just small feature vectors, not images.
- Rebuilds the frame_data dictionary grouped by frame index
- Runs logistic regression at each frame position and cumulatively
- So there will be 2 graphs, the single frame and the cumulative, which will show us where and how the best accuracy is being measured across an examination.

# Accuracy Per Frame Experiment

For each frame, it took every exam's feature vector at that frame position, paired it with the HyperF_Type label.
Trains logistic regression, tests it with 5-fold cross validation

Freezing the feature extractor, only train a simple linear classification on top. So we know: "how much useful information is already present in these features for this task?"

# Technical part

Loading all the feature extracted .pt files, which were created using the extract_features.py script

```python
import torch
import glob
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score

# Load all .pt files
all_data = []
for pt_file in glob.glob('/idiap/temp/tvanrijn/Graduation_Project/features/*.pt'):
    all_data.append(torch.load(pt_file, weights_only=False))
print(f"Loaded {len(all_data)} exams")

# Only exams with all 12 frames
max_frames = 12
full_exams = [exam for exam in all_data if exam['num_frames'] >= max_frames]
print(f"Exams with {max_frames}+ frames: {len(full_exams)}")

# — Single frame linear probe —
single_accs = []

for i in range(max_frames):
    X = np.array([exam['features'][i].numpy() for exam in full_exams])
    y = np.array([exam['label'] for exam in full_exams])

    clf = LogisticRegression(max_iter=1000)
    scores = cross_val_score(clf, X, y, cv=5, scoring='accuracy')
    single_accs.append(scores.mean())
    print(f"Frame {i}: {scores.mean():.3f} (+/- {scores.std():.3f})")

# — Cumulative linear probe (average pooling) —
cumulative_accs = []

for i in range(max_frames):
    X = np.array([exam['features'][:i+1].mean(dim=0).numpy() for exam in full_exams])
    y = np.array([exam['label'] for exam in full_exams])

    clf = LogisticRegression(max_iter=1000)
    scores = cross_val_score(clf, X, y, cv=5, scoring='accuracy')
    cumulative_accs.append(scores.mean())
    print(f"Frames 0-{i} (avg): {scores.mean():.3f} (+/- {scores.std():.3f})")
```

# Technical part

To make it a somewhat good comparison, decision was made to only get the exams that had same amount of frames. So later on there wouldn't be a smaller n

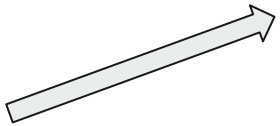
```python
import torch
import glob
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score

# Load all .pt files
all_data = []
for pt_file in glob.glob('/idiap/temp/tvanrijn/Graduation_Project/features/*.pt'):
    all_data.append(torch.load(pt_file, weights_only=False))
print(f"Loaded {len(all_data)} exams")

# Only exams with all 12 frames
max_frames = 12
full_exams = [exam for exam in all_data if exam['num_frames'] >= max_frames]
print(f"Exams with {max_frames}+ frames: {len(full_exams)}")

# ── Single frame linear probe ──
single_accs = []

for i in range(max_frames):
    X = np.array([exam['features'][i].numpy() for exam in full_exams])
    y = np.array([exam['label'] for exam in full_exams])

    clf = LogisticRegression(max_iter=1000)
    scores = cross_val_score(clf, X, y, cv=5, scoring='accuracy')
    single_accs.append(scores.mean())
    print(f"Frame {i}: {scores.mean():.3f} (+/- {scores.std():.3f})")

# ── Cumulative linear probe (average pooling) ──
cumulative_accs = []

for i in range(max_frames):
    X = np.array([exam['features'][:i+1].mean(dim=0).numpy() for exam in full_exams])
    y = np.array([exam['label'] for exam in full_exams])

    clf = LogisticRegression(max_iter=1000)
    scores = cross_val_score(clf, X, y, cv=5, scoring='accuracy')
    cumulative_accs.append(scores.mean())
    print(f"Frames 0-{i} (avg): {scores.mean():.3f} (+/- {scores.std():.3f})")
```

# Technical part

This loops through the frame indices 0 to 11.

Each iteration asks: “how well can we classify from frame i alone?”

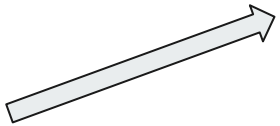
X is becoming the feature matrix, results in a matrix of shape (num_exams, 384) – each row is one exam’s features at that given frame



```python
import torch
import glob
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score

# Load all .pt files
all_data = []
for pt_file in glob.glob('/idiap/temp/tvanrijn/Graduation_Project/features/*.pt'):
    all_data.append(torch.load(pt_file, weights_only=False))
print(f"Loaded {len(all_data)} exams")

# Only exams with all 12 frames
max_frames = 12
full_exams = [exam for exam in all_data if exam['num_frames'] >= max_frames]
print(f"Exams with {max_frames}+ frames: {len(full_exams)}")

# — Single frame linear probe —
single_accs = []

for i in range(max_frames):
    X = np.array([exam['features'][i].numpy() for exam in full_exams])
    y = np.array([exam['label'] for exam in full_exams])

    clf = LogisticRegression(max_iter=1000)
    scores = cross_val_score(clf, X, y, cv=5, scoring='accuracy')
    single_accs.append(scores.mean())
    print(f"Frame {i}: {scores.mean():.3f} (+/- {scores.std():.3f})")

# — Cumulative linear probe (average pooling) —
cumulative_accs = []

for i in range(max_frames):
    X = np.array([exam['features'][:i+1].mean(dim=0).numpy() for exam in full_exams])
    y = np.array([exam['label'] for exam in full_exams])

    clf = LogisticRegression(max_iter=1000)
    scores = cross_val_score(clf, X, y, cv=5, scoring='accuracy')
    cumulative_accs.append(scores.mean())
    print(f"Frames 0-{i} (avg): {scores.mean():.3f} (+/- {scores.std():.3f})")
```

# Technical part

y is becoming the label vector. One label per exam, e.g., “leakage”, “pooling”, etc.



```python
import torch
import glob
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score

# Load all .pt files
all_data = []
for pt_file in glob.glob('/idiap/temp/tvanrijn/Graduation_Project/features/*.pt'):
    all_data.append(torch.load(pt_file, weights_only=False))
print(f"Loaded {len(all_data)} exams")

# Only exams with all 12 frames
max_frames = 12
full_exams = [exam for exam in all_data if exam['num_frames'] >= max_frames]
print(f"Exams with {max_frames}+ frames: {len(full_exams)}")

# — Single frame linear probe —
single_accs = []

for i in range(max_frames):
    X = np.array([exam['features'][i].numpy() for exam in full_exams])
    y = np.array([exam['label'] for exam in full_exams])

    clf = LogisticRegression(max_iter=1000)
    scores = cross_val_score(clf, X, y, cv=5, scoring='accuracy')
    single_accs.append(scores.mean())
    print(f"Frame {i}: {scores.mean():.3f} (+/- {scores.std():.3f})")

# — Cumulative linear probe (average pooling) —
cumulative_accs = []

for i in range(max_frames):
    X = np.array([exam['features'][:i+1].mean(dim=0).numpy() for exam in full_exams])
    y = np.array([exam['label'] for exam in full_exams])

    clf = LogisticRegression(max_iter=1000)
    scores = cross_val_score(clf, X, y, cv=5, scoring='accuracy')
    cumulative_accs.append(scores.mean())
    print(f"Frames 0-{i} (avg): {scores.mean():.3f} (+/- {scores.std():.3f})")
```

# Technical part

Creates a logistic regression classifier. This learns a linear decision boundary in the 384-dim feature space

The max_iter=1000 just gives the solver enough iterations to converge

```python
import torch
import glob
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score

# Load all .pt files
all_data = []
for pt_file in glob.glob('/idiap/temp/tvanrijn/Graduation_Project/features/*.pt'):
    all_data.append(torch.load(pt_file, weights_only=False))
print(f"Loaded {len(all_data)} exams")

# Only exams with all 12 frames
max_frames = 12
full_exams = [exam for exam in all_data if exam['num_frames'] >= max_frames]
print(f"Exams with {max_frames}+ frames: {len(full_exams)}")

# — Single frame linear probe —
single_accs = []

for i in range(max_frames):
    X = np.array([exam['features'][i].numpy() for exam in full_exams])
    y = np.array([exam['label'] for exam in full_exams])

    clf = LogisticRegression(max_iter=1000)
    scores = cross_val_score(clf, X, y, cv=5, scoring='accuracy')
    single_accs.append(scores.mean())
    print(f"Frame {i}: {scores.mean():.3f} (+/- {scores.std():.3f})")

# — Cumulative linear probe (average pooling) —
cumulative_accs = []

for i in range(max_frames):
    X = np.array([exam['features'][:i+1].mean(dim=0).numpy() for exam in full_exams])
    y = np.array([exam['label'] for exam in full_exams])

    clf = LogisticRegression(max_iter=1000)
    scores = cross_val_score(clf, X, y, cv=5, scoring='accuracy')
    cumulative_accs.append(scores.mean())
    print(f"Frames 0-{i} (avg): {scores.mean():.3f} (+/- {scores.std():.3f})")
```

# Technical part

This is where the 5-fold-cross-validation happens.

- splits the exams into 5 equal groups
- Fold 1: train on groups 2-5, test on group 1 -> get accuracy
- Train on groups 1,3,4,5, test on group 2 -> get accuracy etc. etc.
- Returns 5 accuracy scores

This ensures every exam is tested exactly once, and the model never sees its own test data during training.

```python
import torch
import glob
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score

# Load all .pt files
all_data = []
for pt_file in glob.glob('/idiap/temp/tvanrijn/Graduation_Project/features/*.pt'):
    all_data.append(torch.load(pt_file, weights_only=False))
print(f"Loaded {len(all_data)} exams")

# Only exams with all 12 frames
max_frames = 12
full_exams = [exam for exam in all_data if exam['num_frames'] >= max_frames]
print(f"Exams with {max_frames}+ frames: {len(full_exams)}")

# — Single frame linear probe —
single_accs = []

for i in range(max_frames):
    X = np.array([exam['features'][i].numpy() for exam in full_exams])
    y = np.array([exam['label'] for exam in full_exams])

    clf = LogisticRegression(max_iter=1000)
    scores = cross_val_score(clf, X, y, cv=5, scoring='accuracy')
    single_accs.append(scores.mean())
    print(f"Frame {i}: {scores.mean():.3f} (+/- {scores.std():.3f})")

# — Cumulative linear probe (average pooling) —
cumulative_accs = []

for i in range(max_frames):
    X = np.array([exam['features'][:i+1].mean(dim=0).numpy() for exam in full_exams])
    y = np.array([exam['label'] for exam in full_exams])

    clf = LogisticRegression(max_iter=1000)
    scores = cross_val_score(clf, X, y, cv=5, scoring='accuracy')
    cumulative_accs.append(scores.mean())
    print(f"Frames 0-{i} (avg): {scores.mean():.3f} (+/- {scores.std():.3f})")
```

# Technical part

Averages the 5 fold scores into one number.

```python
import torch
import glob
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score

# Load all .pt files
all_data = []
for pt_file in glob.glob('/idiap/temp/tvanrijn/Graduation_Project/features/*.pt'):
    all_data.append(torch.load(pt_file, weights_only=False))
print(f"Loaded {len(all_data)} exams")

# Only exams with all 12 frames
max_frames = 12
full_exams = [exam for exam in all_data if exam['num_frames'] >= max_frames]
print(f"Exams with {max_frames}+ frames: {len(full_exams)}")

# — Single frame linear probe —
single_accs = []

for i in range(max_frames):
    X = np.array([exam['features'][i].numpy() for exam in full_exams])
    y = np.array([exam['label'] for exam in full_exams])

    clf = LogisticRegression(max_iter=1000)
    scores = cross_val_score(clf, X, y, cv=5, scoring='accuracy')
    single_accs.append(scores.mean())
    print(f"Frame {i}: {scores.mean():.3f} (+/- {scores.std():.3f})")

# — Cumulative linear probe (average pooling) —
cumulative_accs = []

for i in range(max_frames):
    X = np.array([exam['features'][:i+1].mean(dim=0).numpy() for exam in full_exams])
    y = np.array([exam['label'] for exam in full_exams])

    clf = LogisticRegression(max_iter=1000)
    scores = cross_val_score(clf, X, y, cv=5, scoring='accuracy')
    cumulative_accs.append(scores.mean())
    print(f"Frames 0-{i} (avg): {scores.mean():.3f} (+/- {scores.std():.3f})")
```

# Technical part

Then for the cumulative version there is a small difference.

It grabs frames 0 through i. So when i = 0, it's just frame 0.

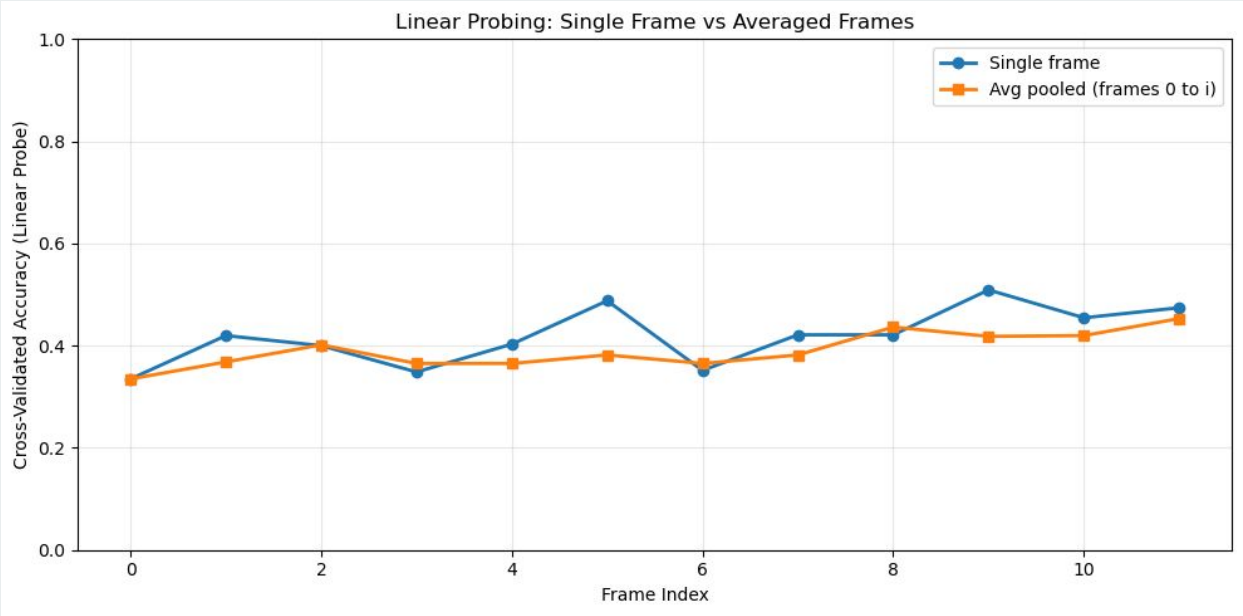
When i=3 it's frames 0,1,2,3. This is a tensor shape of (i+1, 384).

Then it takes averages across all those frames, collapsing them into a single 384-dim vector.

Result is still (num_exams, 384)

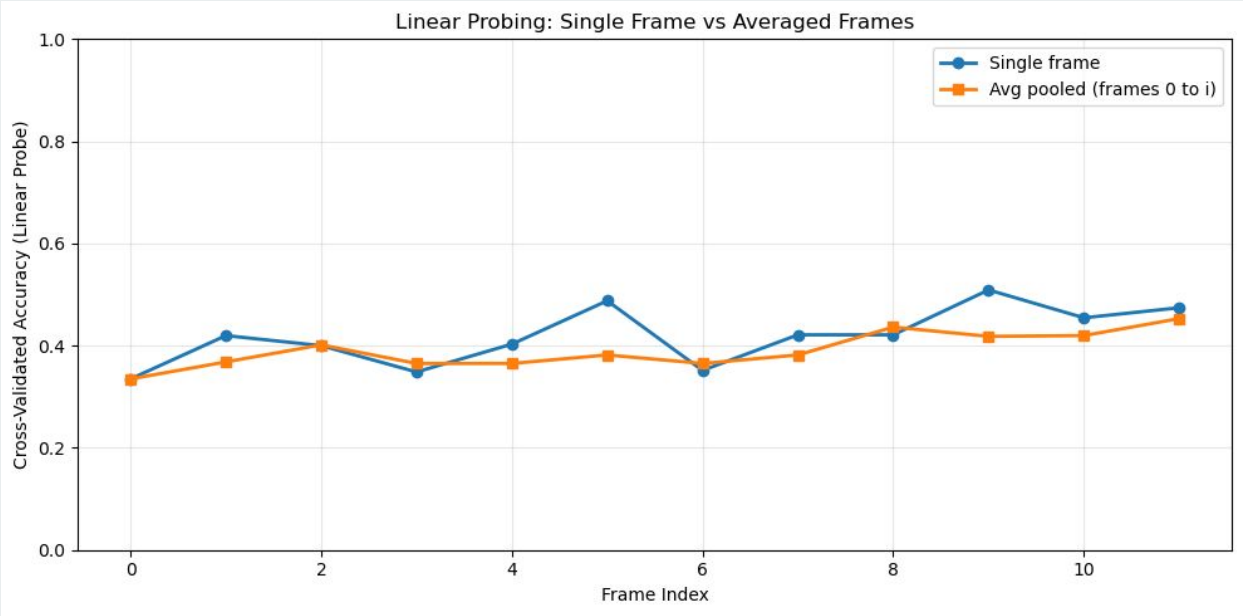
```
# — Cumulative linear probe (average pooling) —
cumulative_accs = []

for i in range(max_frames):
    X = np.array([exam['features'][:i+1].mean(dim=0).numpy() for exam in full_exams])
    y = np.array([exam['label'] for exam in full_exams])

    clf = LogisticRegression(max_iter=1000)
    scores = cross_val_score(clf, X, y, cv=5, scoring='accuracy')
    cumulative_accs.append(scores.mean())
    print(f"Frames 0-{i} (avg): {scores.mean():.3f} (+/- {scores.std():.3f})")
```

# Result




- Later frames carry more diagnostic information. The single frame line is climbing from 34% to 50%.
- Simple averaging doesn't beat the best single frame.
- Averaging might destroy temporal order
- It can't distinguish "intensity went up then down" from "intensity stayed flat"

# Ideas



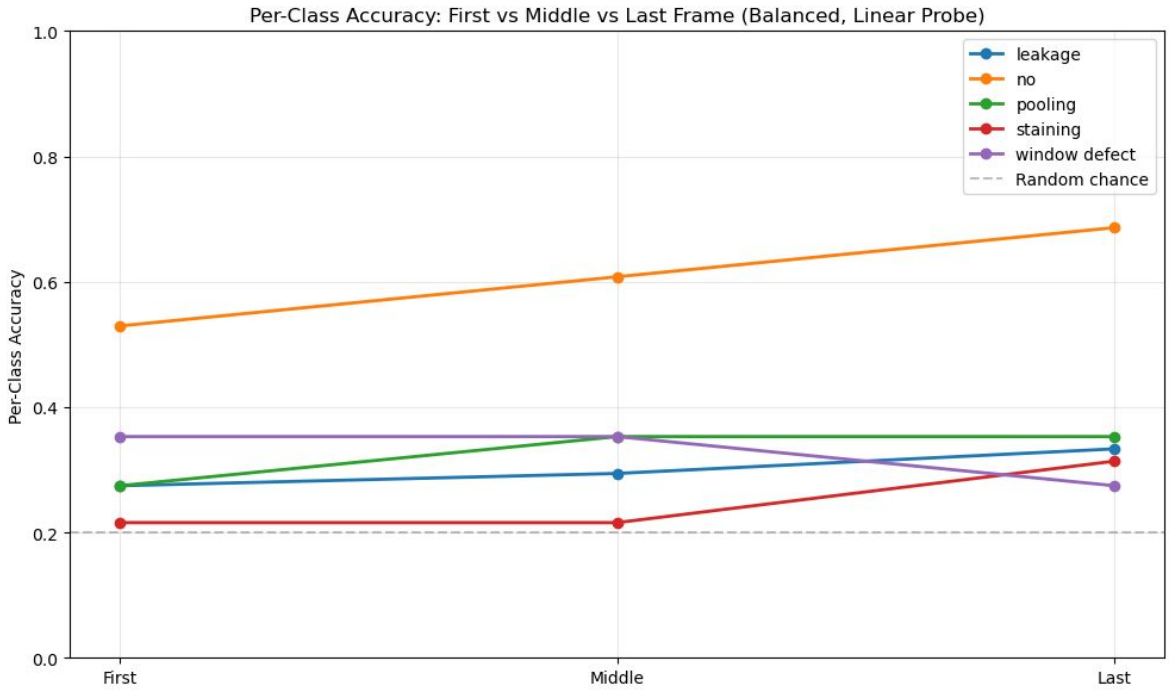

- An LTSM or Transformer can learn these temporal patterns, which neither a single frame nor a naive average can capture.
-

# Accuracy Per Class (over time) Experiment

To understand wether temporal information in fluorescein angiography sequences contributes to classification of HyperF_Type's, performed a linear probing on frozen RETFound-Green features.
Features where extracted at three temporal points: **first frame**, **middle frame** and **last frame**.
Expected to see accuracy improve over time for pathologies that develop progressively during angiography, such as **leakage** (where fluorescein gradually escapes from vessels).
Conversely, expected static findings like **window defects** – which are caused by structural photoreceptor loss and visible from the start – to show little temporal variation

# Results

- The “no(ne)” class shows a clear improvement from 53% to 69%
- The pathological classes (leakage, pooling, staining) remain difficult to separate with a linear classifier on frozen features.



*Absolute changes in per-class accuracy over time*

# Results

| *Hyper_F_Type* | *First* | *Middle* | *Last* |
|---|---|---|---|
| **Leakage** | 27.5% | 29.4% (+2.0%) | 33.3% (+3.9%) |
| **No** | 52.9% | 60.8% (+7.8%) | 68.6% (+7.8%) |
| **Pooling** | 27.5% | 35.3% (+7.8%) | 35.3% |
| **Staining** | 21.6% | 21.6% | 31.4% (+9.8%) |
| **Window defect** | 35.3% | 35.3% | 27.5% (-7.8%) |

# Results

Interestingly, window defect acc. decreases from first to last frame, suggesting that its diagnostic features are most prominent early in the angiography and become obscured as other fluorescein dynamics develop. This could suggest that when using more frames from an examination, the results of the metrics would improve

These opposing temporal profiles across classes motivate a sequential model with learned temporal attention, capable of focusing on different frames.

| *Hyper_F_Type* | *First* | *Middle* | *Last* |
|---|---|---|---|
| **Leakage** | 27.5% | 29.4% (+2.0%) | 33.3% (+3.9%) |
| **No** | 52.9% | 60.8% (+7.8%) | 68.6% (+7.8%) |
| **Pooling** | 27.5% | 35.3% (+7.8%) | 35.3% |
| **Staining** | 21.6% | 21.6% | 31.4% (+9.8%) |
| **Window defect** | 35.3% | 35.3% | 27.5% (-7.8%) |

# Disclaimer

These results should be interpreted with caution due to several limitations.

- RETFound-Green is a general-purpose ret. found. model pretrained on color fundus photographs via SSL.
- The extracted features therefore capture generic retinal structures rather than task-specific diagnostic patterns, which likely limits classification performance across all classes.
- The analysis was performed on a class-balanced subset, resulting in a reduced dataset.
- The linear probe (log. regr.) can only learn lin. decision boundaries in the feat. space, which may be insufficient for separating visually similar pathologies like leakage, pooling, and staining.

# Study

“Harnessing the power of longitudinal medical imaging for eye disease prognosis using Transformer-based sequence modeling”
Link

# Study

**Introducing LTSA (Longitudinal Transformer for Survival Analysis**

which tries to do prognosis – meaning it predicts *when* a disease is likely to develop, not just if it's present right now – by learning patterns in **sequences of images over time**.

! LTSA outperform a standard single-image baseline in 19/20 experiments for **AMD*** and 18/20 for **POAG*** when predicting future risk

! Transformers can handle irregular timing in sequences,  a temporal positional encoder embeds the time of each visit so the model knows the real interval between images, handling the fact that patients don't come in on regular schedule

! By inspecting the attention weights, the authors show the most recent image tends to be most informative BUT older images still contribute in a decaying way.

*AMD* = Age-related Macular Degeneration*
*POAG* = Primary Open Angle Glaucoma*

# Study; *How do they handle temporal data?*

> Usage of temporal positional encoding, to ensure ordinal sequence (?)

> The model uses **casual attention**, meaning it only attends to past and current images, not future ones. That mirrors how prognosis should work: you shouldn't peek into the future when making a prediction at a given point

> The model sees the entire sequence of per-visit embeddings at once, self-attention lets each visit look at earlier visits. Attention is casual, so no cheating by looking into the future.

> It remains a sequence of vectors, not one vector

# Study; *Our case*

1. Decide what “time” means for the frames in our examination
    a. Probably frame indexes in our case
    b. Could also be phase labels
2. Turn every frame into an embedding (one vector per frame)
    a. Resize/normalise
    b. Feed through a vision backbone (RETFound-Green probably)
    c. Take the pooled output as a vector

Now the exam is a sequence matrix: shape [T=n_frames, D]

3. Add temporal information to each vector
    a. Frame-index embedding
4. Feed the sequence into a temporal model (Temporal Transformer Encoder / GRU or LSTM first)

    Now there is Z = [Z_1, Z_n] -> Collapse the sequence into one exam-level representation to get a single vector, probably using CLS token (transformer styled).

# Defining a baseline

Based on Roberto's fm-overspecialization (which is based on Angioreport dataset and HyperF_Type classification) I wanted to try to get baseline results by running the experiment.py on my own machine, to get a feel for the mednet workflow

So I forked the repository and tried it myself, which led to the following baseline results.

# Defining the “baseline”

Roberto’s fm-overspecialization results with RETFound-Green (Test evaluation)

Using only the last frame of a FA examination

| Class | AUC | AP |
|---|---|---|
| Leakage | 0.84 | 0.71 |
| Staining | 0.74 | 0.48 |
| None | 0.95 | 0.84 |
| Pooling | 0.74 | 0.21 |
| Window Defect | 0.70 | 0.19 |

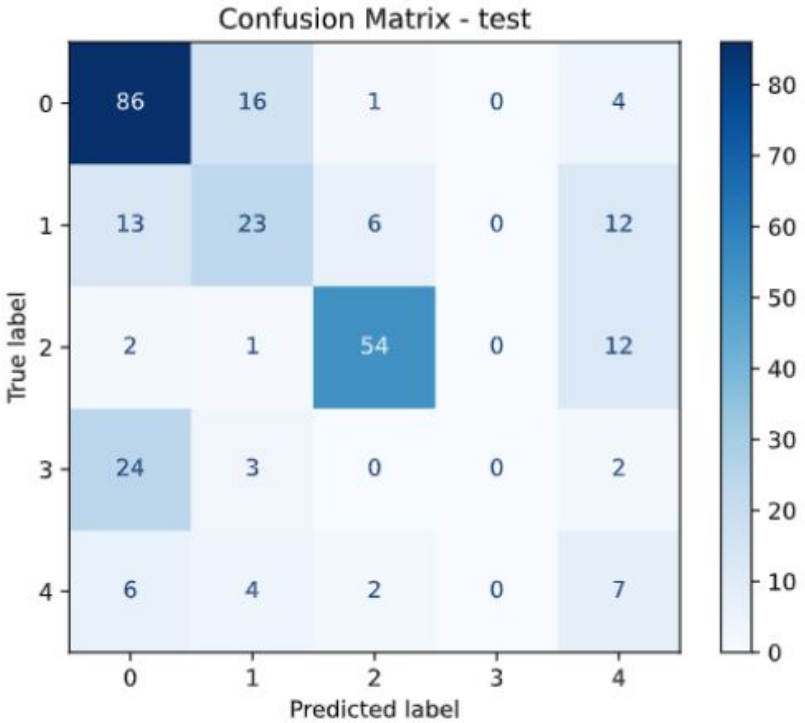
# Defining the “baseline”

Goal: figure out if using multiple frames from one examination can result in better results (using the same metrics) than just using the last frame.

| Class | AUC | AP |
|---|---|---|
| Leakage | 0.84 | 0.71 |
| Staining | 0.74 | 0.48 |
| None | 0.95 | 0.84 |
| Pooling | 0.74 | 0.21 |
| Window Defect | 0.70 | 0.19 |

# Defining the “baseline”

Confusion matrix from the same experiment.py run



# Study

“Temporal Segment Networks for Action Recognition in Videos”

Link

# Study

**Problem**: Videos are long. Processing every frame is expensive and redundant. Using only a single frame misses temporal dynamics (Same could apply to FA exams).

**Solution:**

1. Divide the full video into K temporal segments (uniformly across time)
2. Randomly sample one short simple (or frame) from each segment
3. Pass each snippet independently through the same CNN
4. Aggregate the predictions using a consensus function (usually average)
5. Backprop through the entire system end-to-end

Called Temporal Segment Networks (**TSN**)

# Study; Principles

**3 Key principles**

1. **Long-range coverage without heavy computation**
2. **Shared feature extractor**
3. **Late fusion via consensus** (*= Processing each temporal chunk separately, each chunk make sits own prediction, combining those predictions at the end give you the combination that is calle the “consensus”*

# Study; Our case

**The FA exam:**

- Variable length
- Strong early/late phase differences
- Often redundant adjacent frames
- Clinically interpreted as phases (link to phases)

**Step 1 - Temporal segmentation**

Split FA exam into K segments (buckets)

K = 3 -> early, mid, late

or K = 5 -> finer temporal resolution

This solves:

- Variable-length sequence problem
- Computational efficiency
- Noise from too many redundant frames

# Study; Our case

**Step 2 - Frame sampling per segment**

Sample:

- 1 representative frame

  OR

- 2-3 frames and average embeddings

This prevents overfitting to dense late-phase redundancy

**Step 3 - Shared backbone**

Using a shared backbone to get embeddings per sampled frame, now there are **K** embeddings per exam

**Step 4 - Consensus function**

Instead of average, possibly able to feed into a small transformer

# Study; Our case (Limitation)

**TSN** does not model temporal evolution explicitly. Which means it does not learn how things change over time, it only sees snapshots.

It only ensures coverage, will not learn “How does intensity change over time?”

So it might miss fine-grained temporal patterns.

Models that **DO** capture evolutions:

- RNN / LSTM
- Temporal transformers
- 3d CNNs
- Time-aware attention

They have the possibility to encode order and transition, TSN most likely will not.


Aptos Pipeline
measure the first image, if it's not 768x434, throw away entire folder
7.L
then crop the image from 768x434 to 384x434 then to 384x384
FA
ICGA
If metadata indicates left Eye, horizontal flip for the anatomical symmetry.
Resizing the frame, ViT operate on non-overlapping Patches. (usually 14)
384 x 384
392 x 392
Phase 1
Preprocessing.
GOal:
make sure the images are uniform and containing the correct frames (FA) only. And suitable for next steps.

# Phase 2

## Temporal Sampling

T=20, We don't want 20 frames all from the beginning, we need to span the timeline. So diVide the total duration of of the exam in T "buckets" (time intervals)


$$\Delta f(x,y) = div(grad(f))$$

We must select one representative frame per bucket, not random. Using Laplacian variance to call. image sharpness, acts as a de-blurring filter.

Ref: "temporal sampling strategies for video classification.

>20

60 frames → $\frac{60}{T} = 3$ → b1=0,1,2 b2=3,4,5 ... then from every bucket select 1 best image

<20

14 frames, take every single frame available, then add zero padding for the remaing slots. And apply masking so the transformer will know

Preprocessed examination

next step is to extract features from these frames, using RETFound Green "black box"

Convert the sequence of chosen frames with the metadata to a .pt, (torch.save)

[20,3,392,392]

RETFound Green

[20,384]

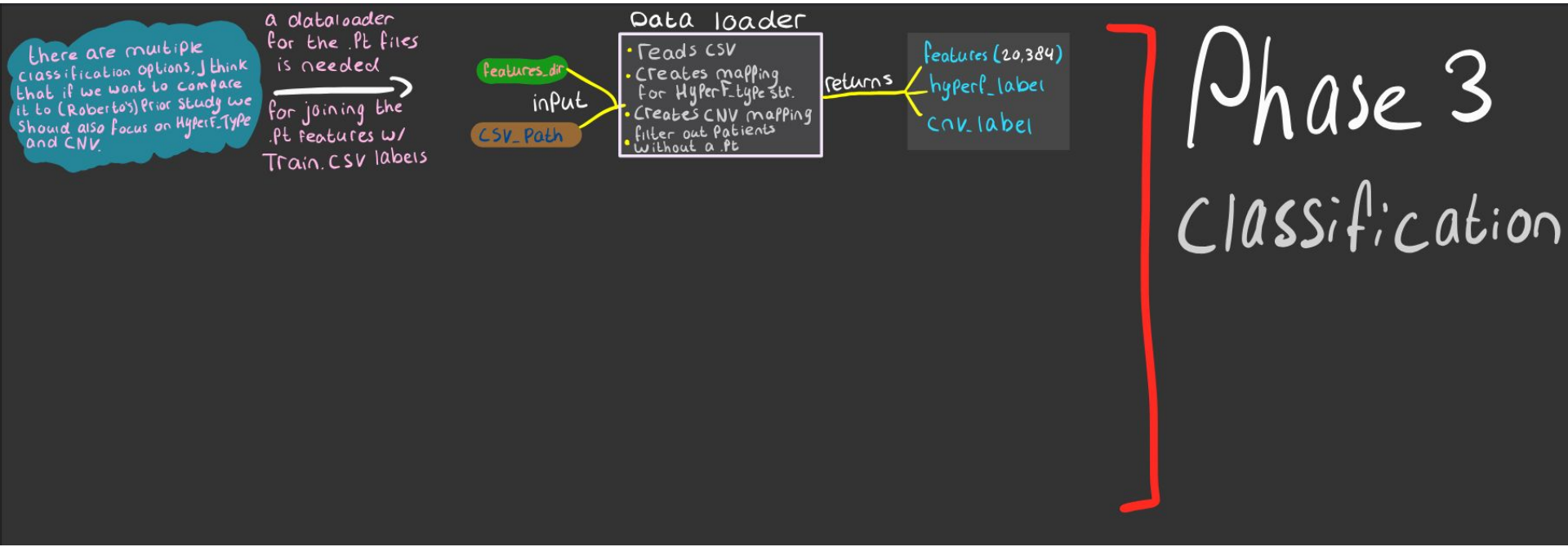
now each frame of an examination is represented by a vector of 384 numbers

Any downstream task will be faster. Transfer learning w/ RETFound

at the end of this phase you're left with (hopefully) $\nabla$ with encoded medical concepts, representing examinations. next step would be to train these .pt's

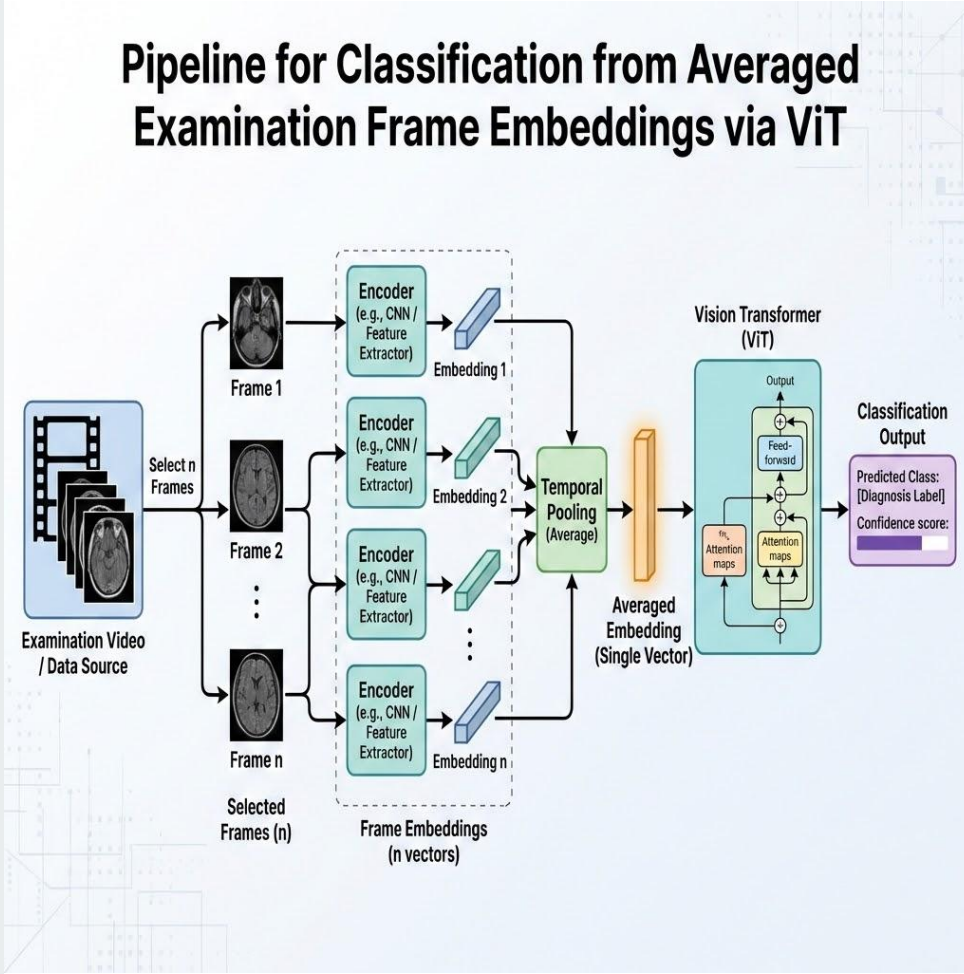
### Goal

thought out frame selection, Placing the spatio-temporal data in buckets. then encoding the vectors we just made by using a headless(?) transformer. Should conclude with medical embeddings for examinations.

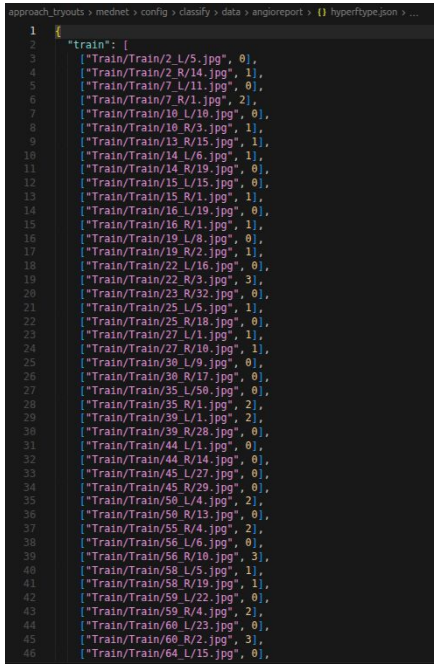

there are multiple classification options, I think that if we want to compare it to (Roberto's) prior study we should also focus on HyperF_Type and CNV.
a dataloader for the .Pt files is needed
for joining the .Pt features w/ Train.csv labels
features_dir
input
CSV_Path
Data loader
• reads CSV
• creates mapping for HyperF_type str.
• creates CNV mapping
• filter out patients without a .pt
returns
features (20,384)
hyperf_label
cnv_label
Phase 3
Classification

# Approach 1

Using averaged embeddings of the frames inside the examination.



# Approach 1




config/classifiy/data/angioreport/hyperftype.json had to be changed. So it was seeing the dirs only.

# Approach 1

Small script to make this happen, instead of manually doing it! ;)

scripts › json_paths_to_dirs.py › ...

```python
#!/usr/bin/env python3
"""Convert [path, label] entries to [dir_path, label] in angioreport JSON splits."""

import argparse
import json
from pathlib import Path

parser = argparse.ArgumentParser()
parser.add_argument("inputs", nargs="+", type=Path)
parser.add_argument("-o", "--output", type=Path)
parser.add_argument("--suffix", default="_dirs")
parser.add_argument("--dry-run", action="store_true")
args = parser.parse_args()

if args.output and len(args.inputs) > 1:
    parser.error("Cannot use -o with multiple inputs")

def to_dir(p): return p.rsplit("/", 1)[0] if "/" in p else p

for f in args.inputs:
    if not f.exists():
        print(f"  WARNING: {f} not found — skipping")
        continue
    data = json.loads(f.read_text())
    out = {k: [[to_dir(e[0]), e[1]] if isinstance(e, list) and len(e) >= 2 else e for e in v] for k, v in data.items()}
    if args.dry_run:
        print(f"  [dry-run] {f}"); continue
    dst = args.output if args.output else f.parent / f"{f.stem}{args.suffix}.json"
    dst.write_text(json.dumps(out, indent=2), encoding="utf-8")
    print(f"  Wrote {dst}")
```

Ctrl+L to chat, Ctrl+K to generate

# Approach 1

Added an argument to experiment.py to handle user input for the usage of averaged frames, so the pipeline could adapt accordingly.

Then I set a env. var, which I'm pretty sure is not ideal, but as far as I could understand at this point, it was ok.

```
    parser.add_argument(
        "--use-average-frames",
        action="store_true",
        help="If set, use averaged frames over the examination (hyperftype_dirs.json). Default: False.",
    )

    return parser


def main(args):
    # Set env var BEFORE any mednet CLI runs (config is loaded when datamodule is resolved)
    if args.use_average_frames:
        os.environ["MEDNET_USE_AVERAGE_FRAMES"] = "1"
    else:
        os.environ["MEDNET_USE_AVERAGE_FRAMES"] = "0"  # or .pop() to unset
    runner = CliRunner()

    pipeline = PIPELINE_DICT[args.modality]
    model = args.model
    task = args.task
    print(
        f"Running {args.modality} ({pipeline}) for...\n"
        f"Model: {model}\n"
        f"Task: {task}\n"
    )
```

# Approach 1

Following the existing pipeline, I made changes to data/classify/angioreport.py RawDataLoader function, to instead of just taking one image, make a list of all the (cropped) images in an exam.

The data loader knew what .json to use because of these modifications in hyperftype.py:

```python
# hyperftype.py
import os

use_avg = os.environ.get("MEDNET_USE_AVERAGE_FRAMES", "0") == "1"
split_file = "hyperftype_dirs.json" if use_avg else "hyperftype.json"

datamodule = DataModule(
    split_path=importlib.resources.files(__package__ or __name__.rsplit(".", 1)[0])
    / split_file,
    num_classes=5,
    problem_type="multiclass",
    use_average_features=use_avg,
)
```

```python
class RawDataLoader(BaseDataLoader):
    def sample(self, sample: typing.Any) -> Sample:

        if self.use_average_features:
            dir_path = self.datadir / sample[0]
            frame_files = sorted(
                [f for f in dir_path.iterdir() if f.suffix.lower() in (".jpg")],
                key=lambda f: int(f.stem) if f.stem.isdigit() else 0,
            )
            images = []
            for fpath in frame_files:
                img = PIL.Image.open(fpath).convert("L")
                width, height = img.size
                box = (
                    (0, 0, width - width / 2, height - 50)
                    if self.modality == "FA"
                    else (width - width / 2, 0, width, height - 50)
                )
                img = img.crop(box)
                img = to_dtype(to_image(img), torch.float32, scale=True)
                images.append(tv_tensors.Image(img))
            return dict[str, Any](image=images, target=self.target(sample), name=sample[0])
        else:
            # Load the image from the file
            image = PIL.Image.open(self.datadir / sample[0]).convert("L")
            # crop the image to get only FA/ICGA image
            width, height = image.size
            box = (
                (0, 0, width - width / 2, height - 50)
                if self.modality == "FA"
                else (width - width / 2, 0, width, height - 50)
            )
            image = image.crop(box)

            image = to_dtype(to_image(image), torch.float32, scale=True)
            image = tv_tensors.Image(image)

            return dict[str, Any](image=image, target=self.target(sample), name=sample[0])
```

# Approach 1 - Test 1

First test was to see if the results would change if the last **2** frames were used (with the averaged embeddings).

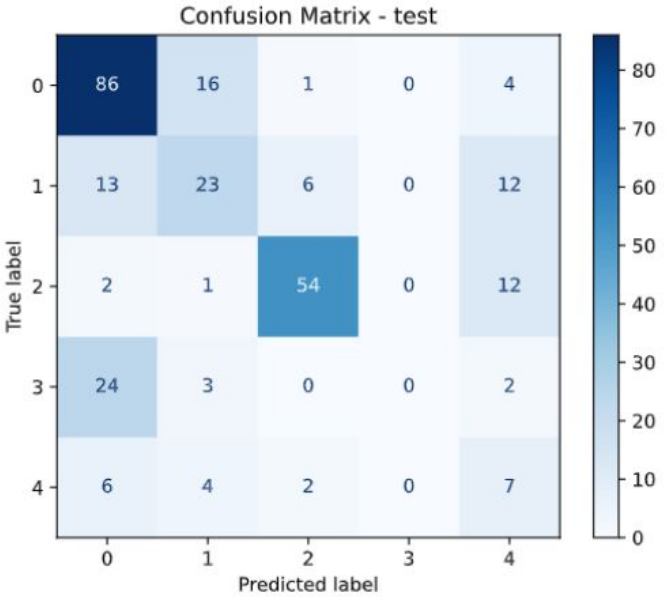
# Approach 1 - Test 1 (Results)

Results, compared to the baseline

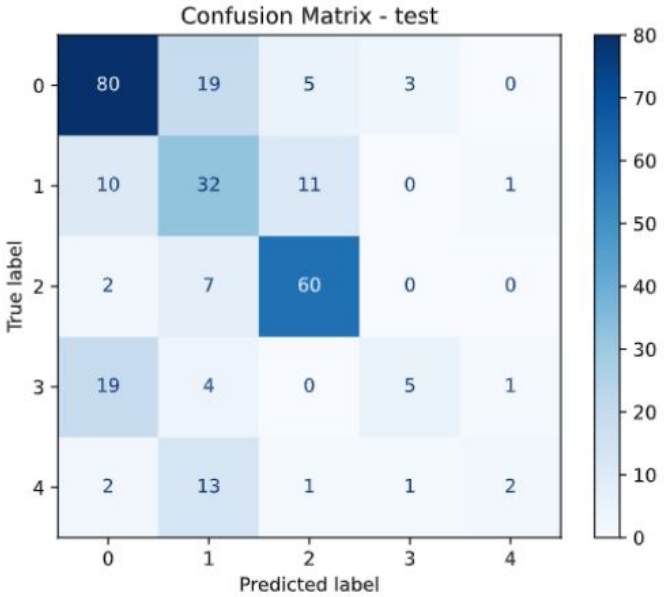
| Class | AUC | AP |
|---|---|---|
| Leakage | 0.85 (+0.01) | 0.73 (+0.02) |
| Staining | 0.75 (+0.01) | 0.40 (-0.08) |
| None | 0.95 | 0.83 (-0.01) |
| Pooling | 0.78 (+0.04) | 0.38 (+0.17) |
| Window Defect | 0.90 (+0.20) | 0.46 (+0.27) |

# Approach 1 - Test 1 (Results)

BASELINE




LAST 2 FRAMES



# Approach 1 - Test 1 (Thoughts)

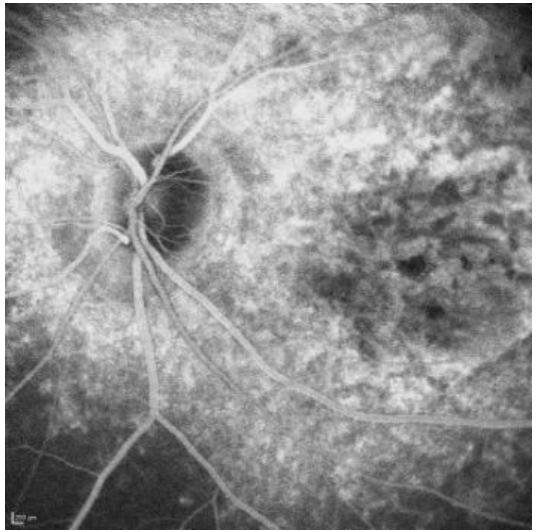
**Windows Defect (Largest Gain):** This class saw a massive increase of 0.20% AUC and +0.26% in AP. This could be because window defects are static anatomical features that do not change over time; averaging a signal across frames could help "lock in" the location and intensity of the defect while filtering out transient artifacts or noise.

**Pooling (Significant improvement):** Pooling also showed a strong positive shift with a +0.17 increase in AP. Since pooling involves the slow accumulation of dye in a physical space, averaging embeddings helps the model distinguish this consistent growth from the more variable patterns of staining or leakage.

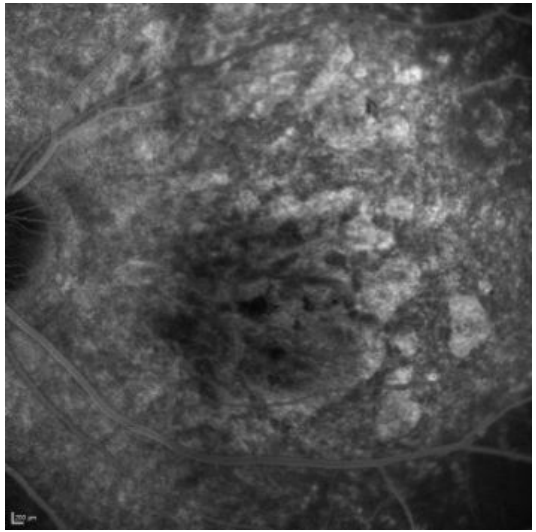
**Staining (The Negative Outlier): f**Interrestingly, while AUC rose slightly, the AP dropped by -0.08. Staining often relies on specific late-phase intensity changes; averaging might be diluting the specific temporal "peak" that the model uses to distinguish staining from other forms of hyperfluorescence.

# Approach 1 - Test 1 (Visual)

Notice the heavy “background noise” in the. Averaging embeddings here might help the model separate the persistent window defect from the mottled background texture.



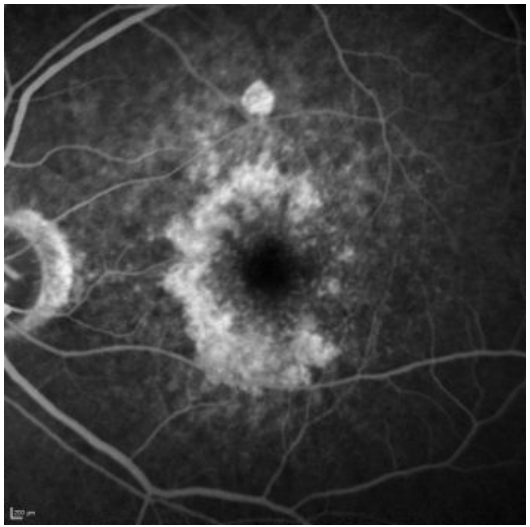
Second to last frame



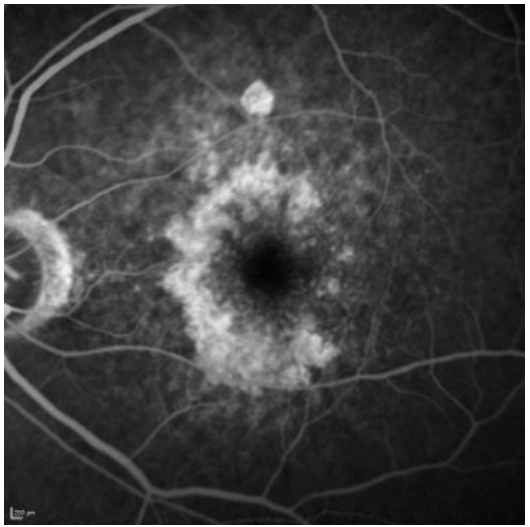
Last frame

# Approach 1 - Test 1 (Visual)

Complex window defect. Averaging could help the model ignore the slight variations in lightning between the two frames to measure the defect more accurately.



Second to last frame



Last frame

# Approach 1 - Test 1 (Visual)

Central window defect is very bright and clear in both of the frames. Averaging these embeddings reinforces the “stable” signal of the defect.

Second to last frame

Last frame

# Approach 1 - Test 2

Second test was to see if the results would change if the last **10** frames were used (with the averaged embeddings).

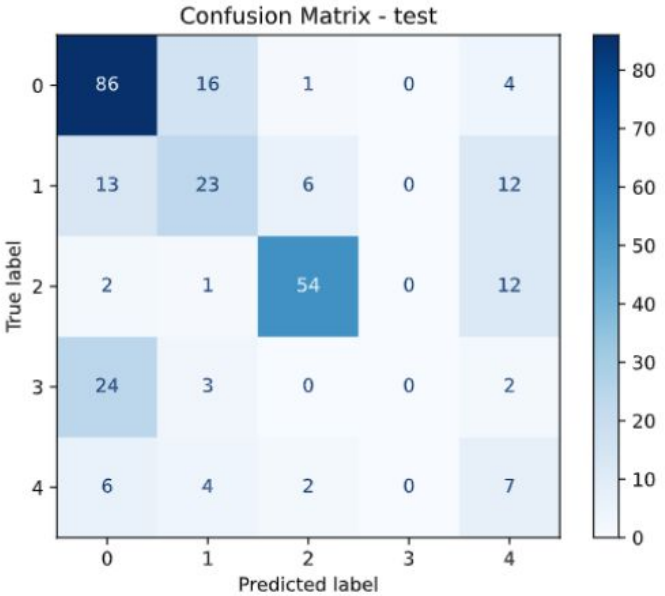
# Approach 1 - Test 2 (Results)

Results compared to the baseline

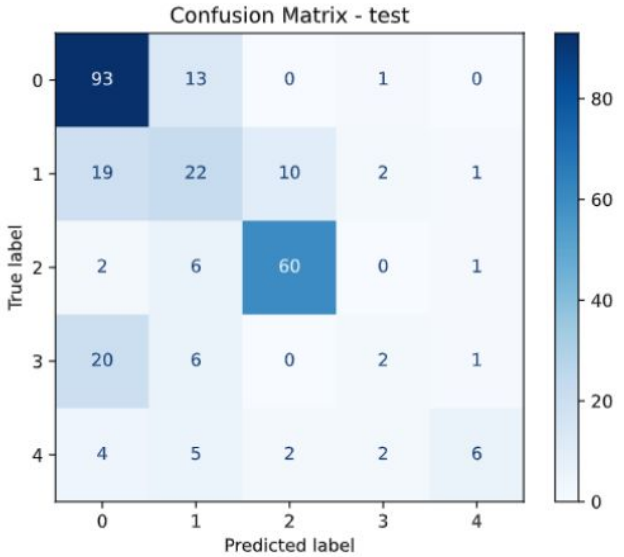
| Class | AUC | AP |
|---|---|---|
| Leakage | 0.86 (+0.02) | 0.74 (+0.03) |
| Staining | 0.70 (-0.04) | 0.44 (-0.04) |
| None | 0.96 (+0.01) | 0.86 (+0.02) |
| Pooling | 0.75 (+0.01) | 0.28 (+0.07) |
| Window Defect | 0.89 (+0.19) | 0.44 (+0.25) |

# Approach 1 - Test 2 (Results)

BASELINE




LAST 10 FRAMES



# Honourable mentions

**N_frames = 5** performed worse on almost all metrics

**N_frames = 20** performed worse on almost all metrics

# Test 1 vs Test 2 (Results)

| Class | AUC | AP |
|---|---|---|
| Leakage | 0.85 (+0.01) | 0.73 (+0.02) |
| Staining | 0.75 (+0.01) | 0.40 (-0.08) |
| None | 0.95 | 0.83 (-0.01) |
| Pooling | 0.78 (+0.04) | 0.38 (+0.17) |
| Window Defect | 0.90 (+0.20) | 0.46 (+0.27) |

Test 1 Results compared to baseline

| Class | AUC | AP |
|---|---|---|
| Leakage | 0.86 (+0.02) | 0.74 (+0.03) |
| Staining | 0.70 (-0.04) | 0.44 (-0.04) |
| None | 0.96 (+0.01) | 0.86 (+0.02) |
| Pooling | 0.75 (+0.01) | 0.28 (+0.07) |
| Window Defect | 0.89 (+0.19) | 0.44 (+0.25) |

Test 2 Results compared to baseline

# Week 1 Key Conclusions

Managed to do the average last *n* embeddings baseline, and it actually moved the needle (especially for **window defect** and **pooling**). But there are still plausible improvements laying around.

# Week 1 Key Conclusions

- Temporal information matters: Using multiple late frames improved performance compared to single-frame baseline for certain classes
- Simple averaging helps stable late-phase patterns but may dilute phase-specific signals
- Performance depends strongly on the choice of *n*; too few frames underutilize information, too many introduce noise or redundancy.
- The current method (mean embedding pooling) ensures temporal coverage but does not explicitly model temporal evolution
- Literature in dynamic contrast imaging (e.g., DCE-MRI) confirms that clinically relevant information often lies in temporal behavior, not just static intensity.
- Therefore, a next logical step is to move from uniform averaging toward weighted or change-aware temporal aggregation.

# Next week(s)

Ideas for the upcoming week(s)

# Next week(s)

“Low hanging fruits”

- **Replace the “dumb” average with a learnable weighted average (attention pooling)**
  - Minimal engineering, still end-to-end learnable, directly targets the observed issue: averaging can dilute peaky cues, while helping stable cues and it lets keep variable frame counts easily (mask + softmax).
- **Add “trend” features alongside the pooled embedding**
  - Computing one extra vector: (last embedding - first embedding) over the selected frames. This gives the classifier explicit information about if something changed, which averaging destroys).
- **Swap “last n frames” for simple buckets (phase segments) (K=3?)**
  - One frame from early third, one frim from middle third, one frame from late third
  - Could even use a laplacian function to figure out which image is the sharpest, and might be the best for a bucket