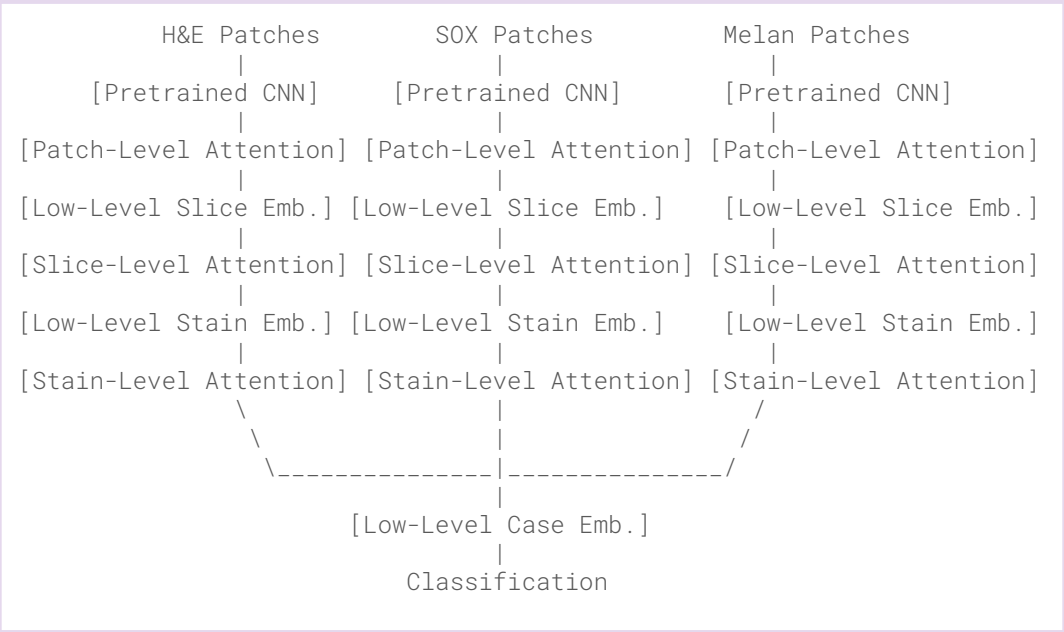
Presentation 5: Wrap-Up

STAT 390 | Project 1 | Fall 2025

Multi-Stain Model Overview



Part 1: Code Updates

1.1 Patch Visualization Updates

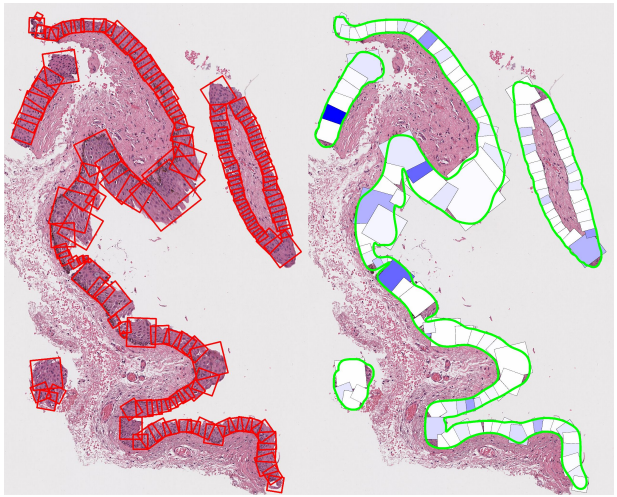
- Patch visualization code changes linked here
  - draw_squares: normalizes attentions core and shades patch based on effective attention
  - patching_export: creates a subset of weights for each slice

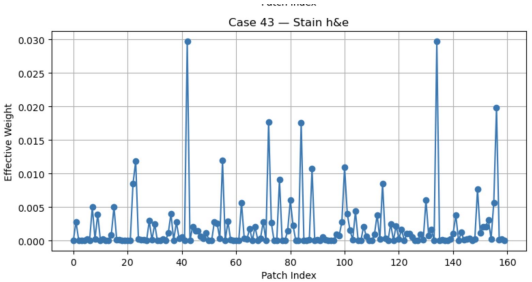
- Code verification checks
  - Compared printed patch index and effective attention to the patched image
  - Patch index extraction from file name verification: /projects/e32998/MIL_training/pres_4_runs/run_20251118_115222/attention_analysis/plots/patch_attention_selected_slices.csv
  - Compared patched slice results with and without attention gradients

1.1 Patch Visualization Updates

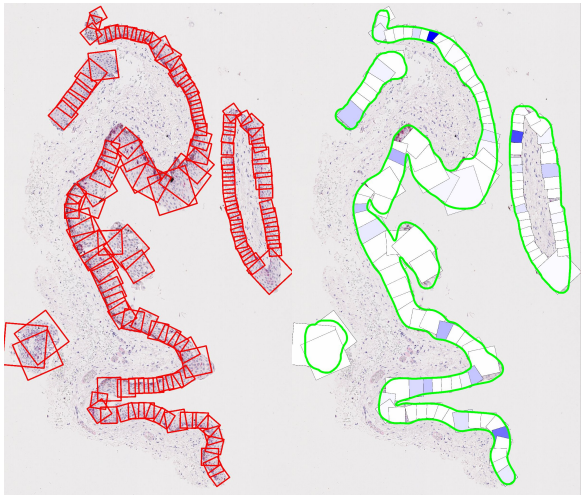
Benign cases: predicted correctly

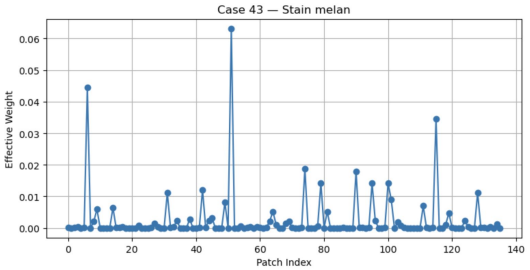
Case 43 H&E



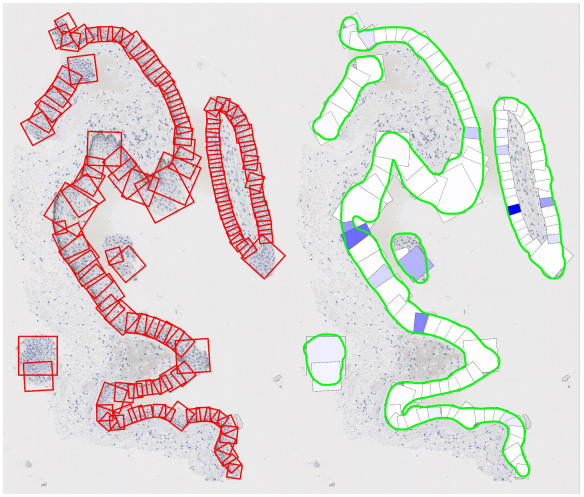


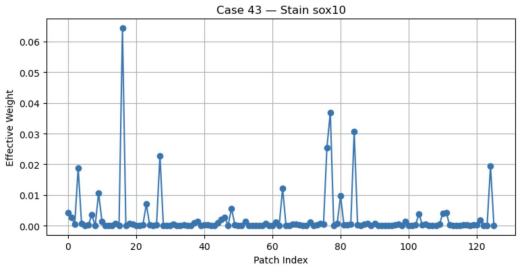
Case 43 Melan





Case 43 Sox 10

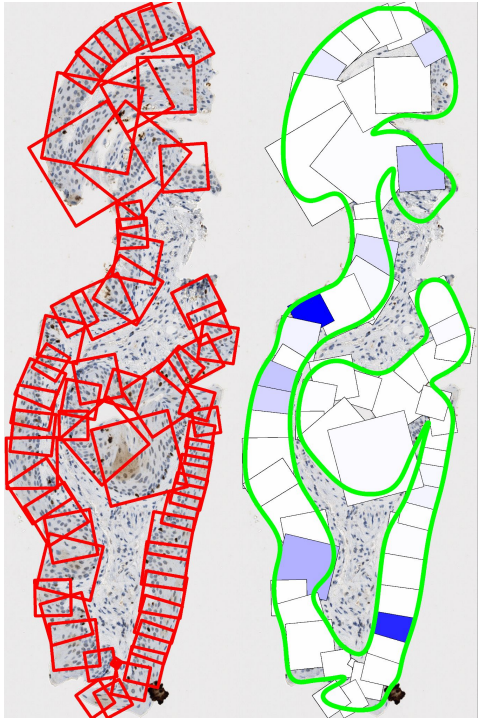


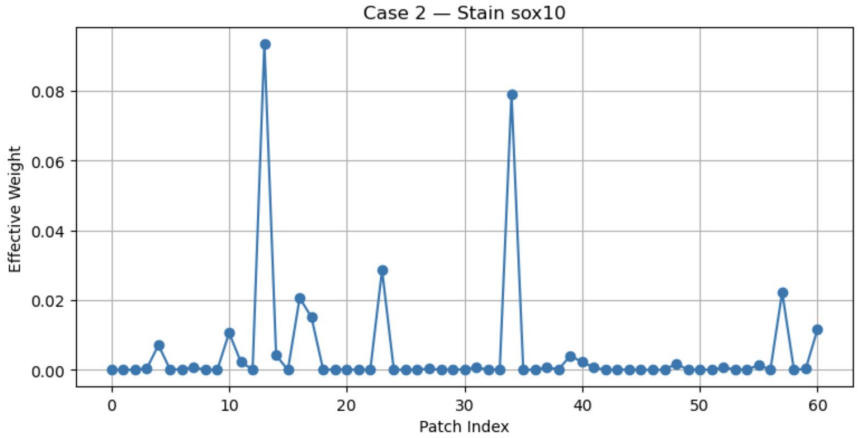


1.1 Patch Visualization Updates

Benign cases: predicted correctly

Case 2 Sox 10

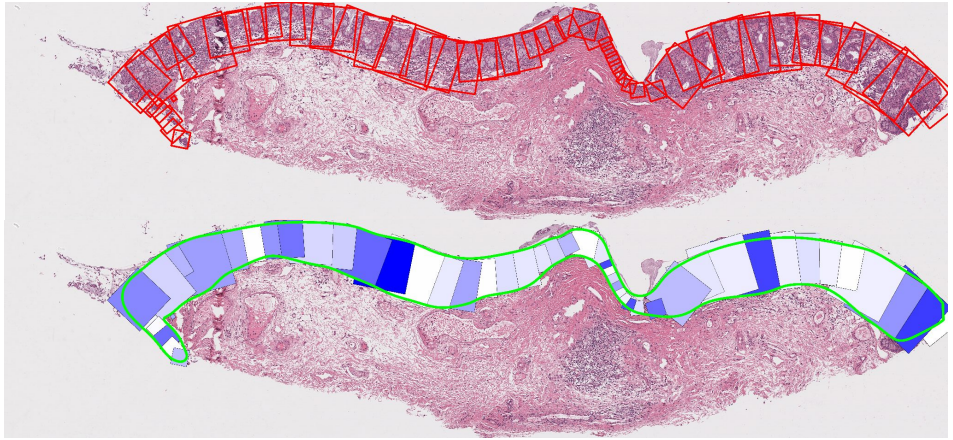


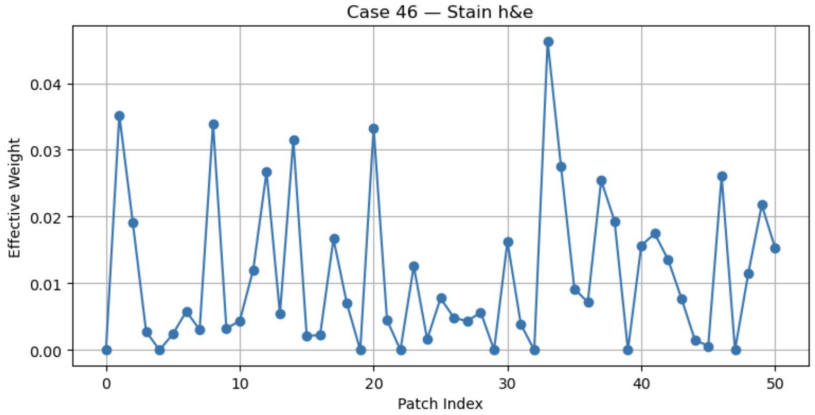


1.1 Patch Visualization Updates

Benign cases: predicted incorrectly

Case 46 H&E

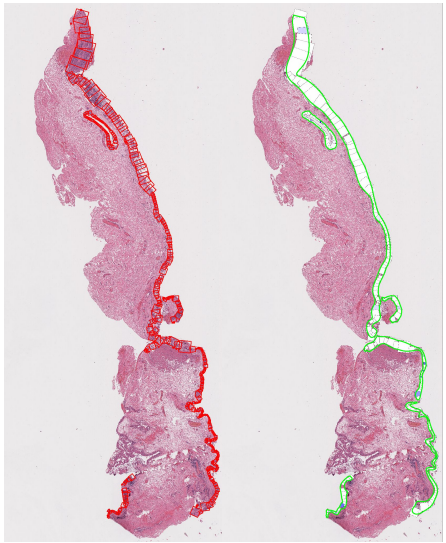


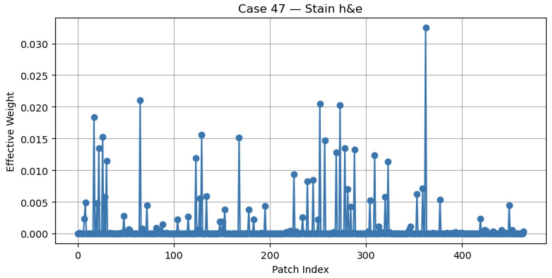


1.1 Patch Visualization Updates

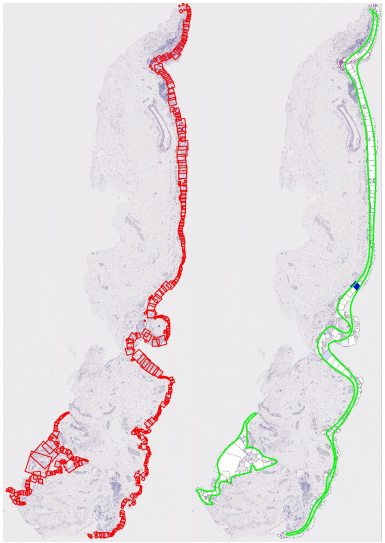
High grade cases: predicted correctly

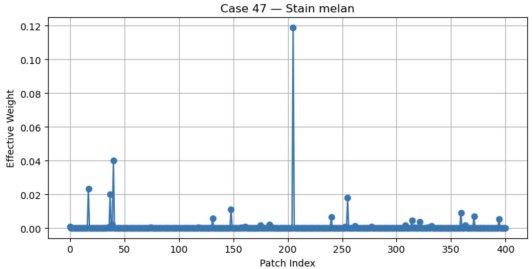
Case 47 H&E



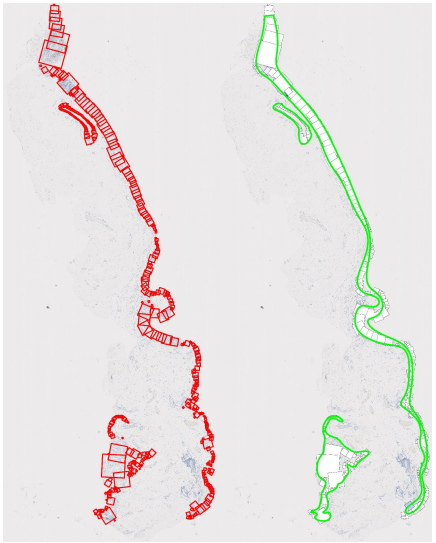


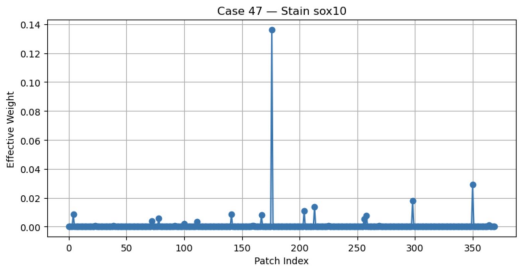
Case 47 Melan





Case 47 Sox 10

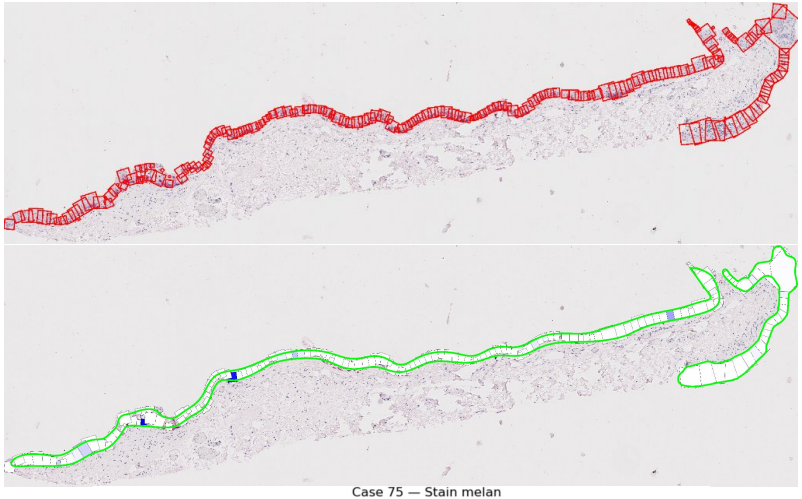


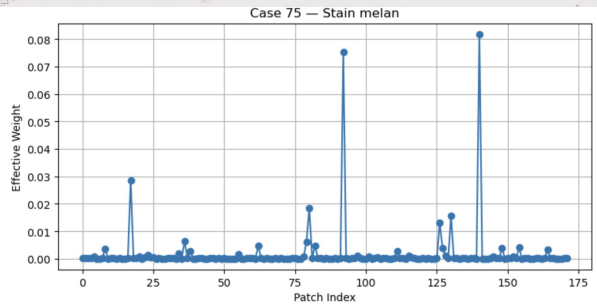


1.1 Patch Visualization Updates

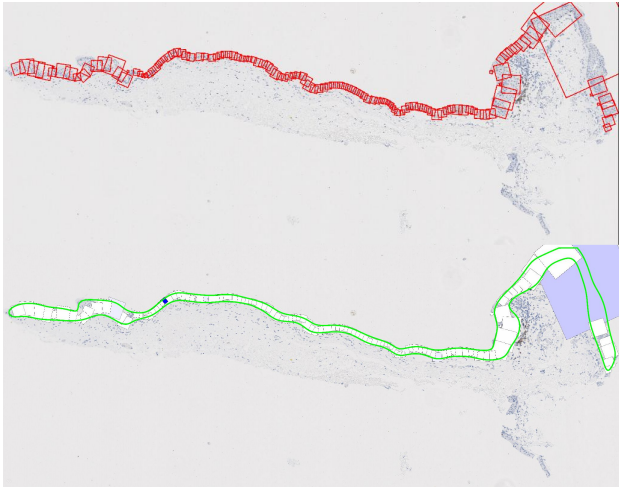
High grade cases: predicted correctly

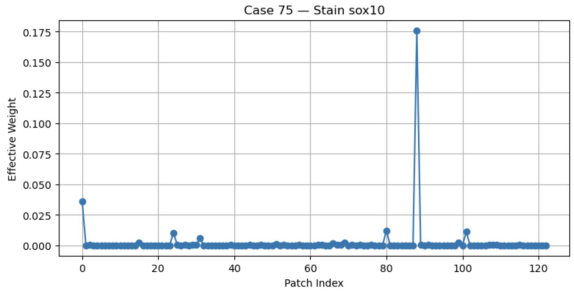
Case 75 Melan





Case 75 Sox 10

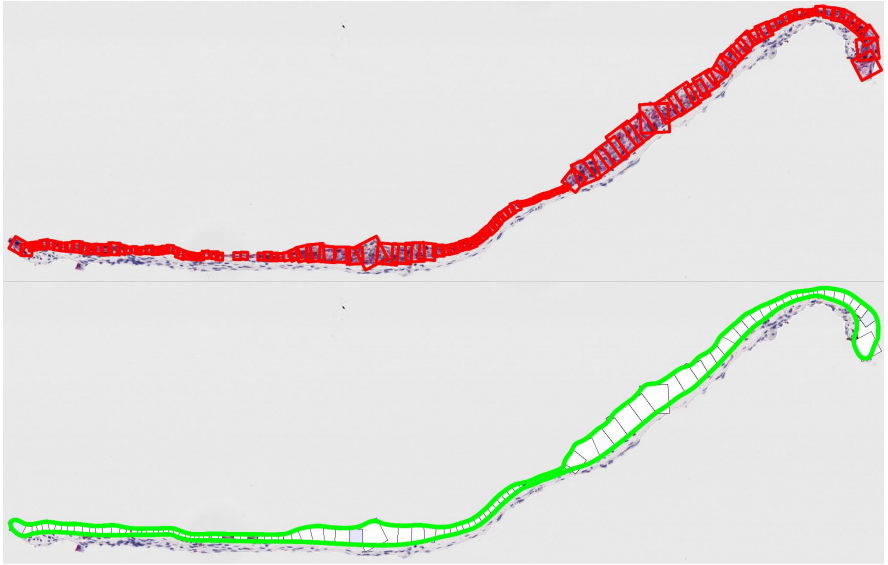


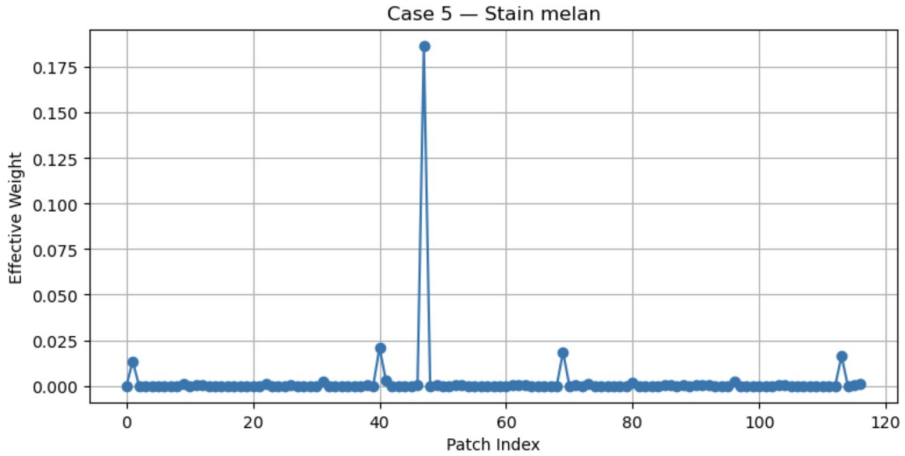


1.1 Patch Visualization Updates

High grade cases: predicted correctly

Case 5 Melan

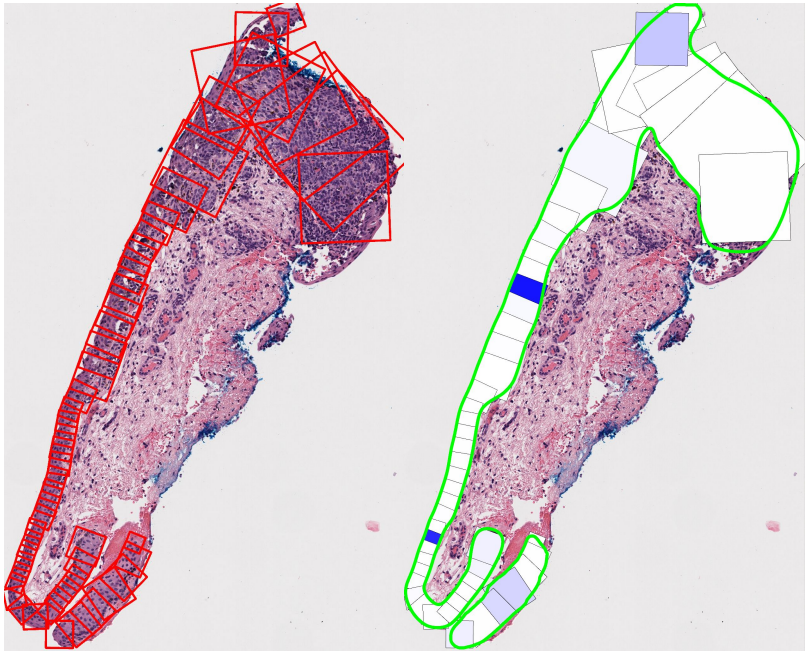


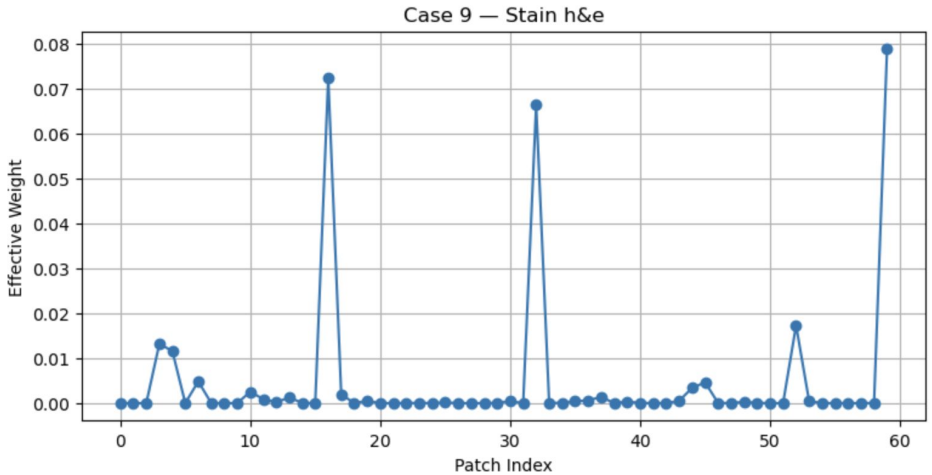


1.1 Patch Visualization Updates

High grade cases: predicted correctly

Case 9 H&E

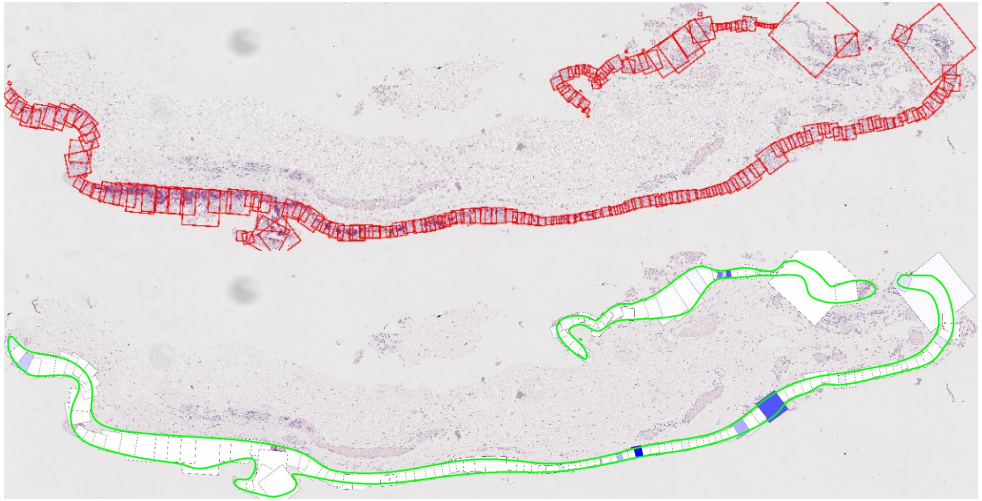


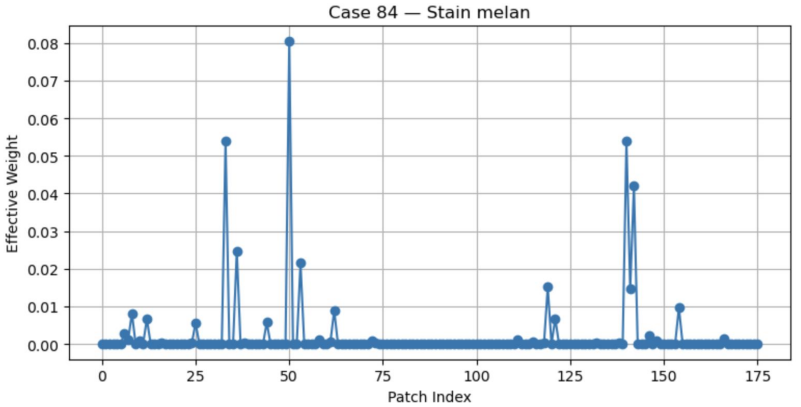


1.1 Patch Visualization Updates

High grade cases: predicted correctly

Case 84 Melan

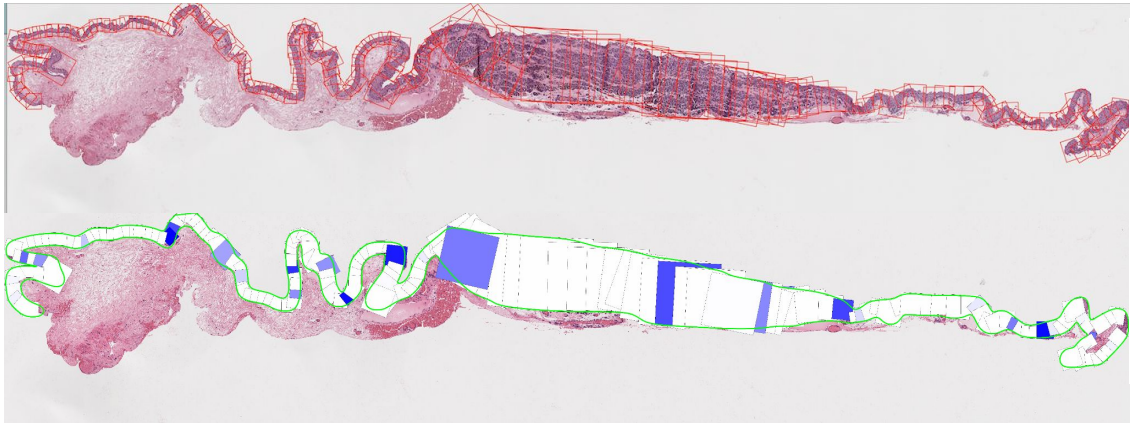


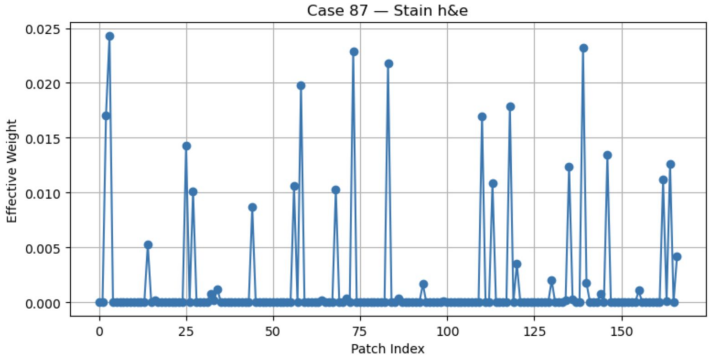


1.1 Patch Visualization Updates

High grade cases: predicted correctly

Case 87 H&E

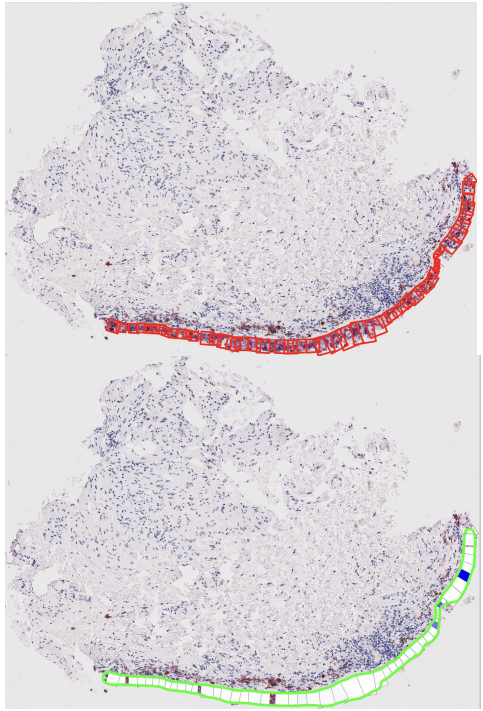




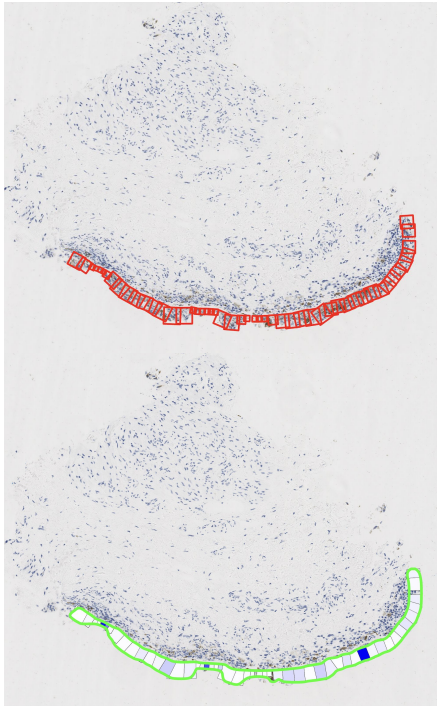
1.1 Patch Visualization Updates

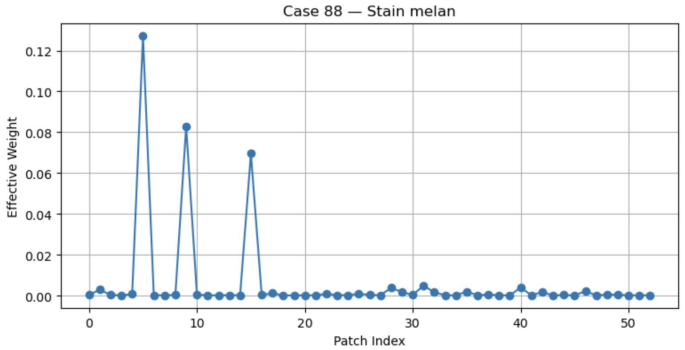
High grade cases: predicted correctly

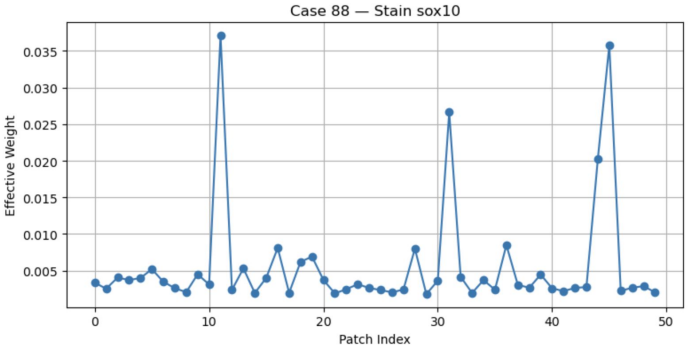
Case 88 Melan



Case 88 Sox 10

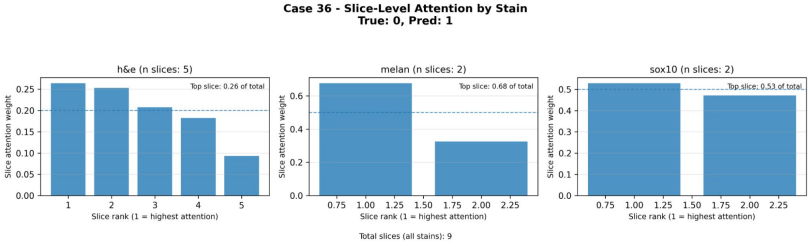


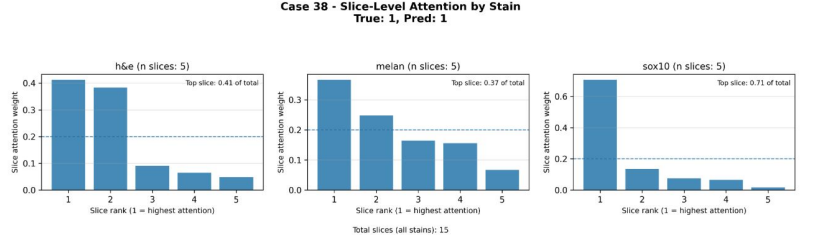


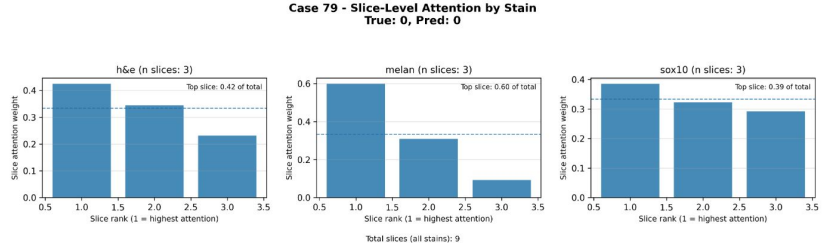


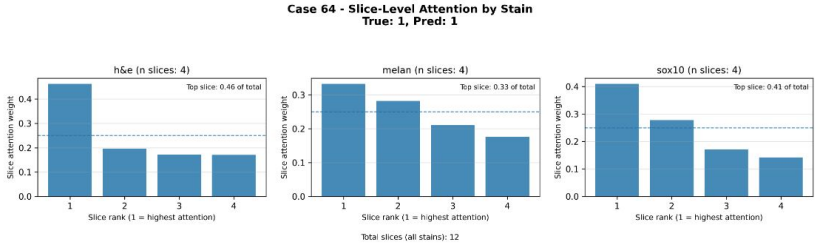
1.2 Slice-Level Attention Analysis

- **Goal**: identify equality of attention scores within a given case-stain combination
  - Indicates whether this level of attention is useful
- Added code printing the slice-level attention for every case-stain combination (code link to function)
- Slice-level attention distributions for select cases w/ multiple slices:









*note: the above graphs were based on training from a set of checkpoints for split_01

1.3 Train / Val / Test Split Update

- **Train / val / test split:**
  - Removed case 91 from all data_splits files (maintained all other case placements)
  - Updated .npz files can be found in **MIL_trainer_9Dec_JointFinal** in the **data_splits** folder

Part 2: Code Organization

2.1 Execution Flow

- When you run main.py…

1. Setup
2. Data Prep
3. Model Prep

- Parsing the arguments
  - Default: config.py
  - Overridden by: arguments provided at python execution
- Initializing devices (GPU/CPU)
- Creates output directories
- Codes: [main.py, utils.py]

- Find all .png patch images; get their case/slice/stain information
- Train-test-validation-split
- Based on PyTorch Datasets
- Convert images to tensors
- Codes: dataset.py

Build a hierarchical attention model:

1. DenseNet121 backbone
2. 3 levels of attention pooling (patch->slice->stain->case)
3. final classifier

Codes: model.py

2.1 Execution Flow

- Continued…

4. Training infrastructure

- Initialize Adam optimizer
- Weighted cross-entropy loss
- Learning rate scheduler
- Early stopping mechanism
- Codes: trainer.py

5. Training loop

For each epoch:

1. Train w/ all training cases, one by one
2. Compute cross-entropy loss
3. Back-propagate gradients
4. Update weights
5. Validate on validation set
6. Adjust learning rate if needed
7. Save checkpoint
8. Stop early if no improvement

Codes: trainer.py

6. Final evaluation

- Test trained model on the test set
- Save per-case predictions
- Attention analysis add-on
  - We have developed different add-on modules for attention analysis

Codes: trainer.py, attention_analysis.py

2.2 data_utils.py

- Main purpose: fundamental structure of the training/validation/test datasets
- Key components:
  - group_patches_by_slice()
    - Parse file names using regex -> get case/slice/stain information
    - Group patches by case and slice ID
    - Output a dictionary in the format of {(case_id, slice_id): [patch_paths]}
  - split_by_case_stratified()
    - Stratified split twice to achieve 60/20/20 split across train/val/test
  - build_case_dict() uses of the above information, and is then utilized by main.py to create dictionaries in the following formats:
    - case_dict = {42: {'h&e': [[paths for each patch...], …], 'melan': [[...]], 'sox10': [[...]]}, 43:..., …}
    - label_map = {42: 0, 43: 1, …}

2.3 dataset.py

- Purpose: convert the dictionary of file paths (created from the previous step) into actual image tensors

- Key component: StainBagCaseDataset
  a. Load the patch images needed for the current batch (memory efficient!)
  b. Shuffle the patches and cap to per_slice_cap (default: 500) patches per slice
     - In each epoch, the subset of slices being selected could be different
  c. Handle data augmentation
     - Training: resize, crop, flip, rotate, and color jitter
     - Validation: resize only
  d. Group tensors by h&e/melan/sox10 for the hierarchical model

- Output format is similar to case_dict from last step, with file paths replaced by tensors

2.4 models.py (pt. 1)

- Key components:
  - self.features: the pretrained DenseNet121 CNN model (frozen)
  - self.patch_projector: trainable projector that turns CNN features into a patch embedding
  - Three AttentionPool instances for patch/slice/stain levels
  - self.classifier: final single layer binary classifier -> result

- Pseudocode for the process:

```
for each stain (melan, sox10, h&e):
     for each slice in this stain:
          for each patch in this slice:
               patch_embeddings = patch_projector(CNN(patch))
               slice_embeddings = lvl_1_attention(patch_embeddings)
          stain_embeddings = lvl_2_attention(slice_embeddings)
     case_embeddings  = lvl_3_attention(stain_embeddings)
logits = classifier(case_embeddings)
```

2.4 models.py (pt. 2)

- Each attention module contains:

1. Input embedding (default n_dim=512)
2. Linear projector layer (default n_dim=128) to compress information
3. Tanh activation
4. Dropout
5. Linear layer, from 128 to 1 dimension. Single importance score
6. Softmax across all importance scores

The attention module architecture is the same across the entire model.

2.5 trainer.py (setup + training)

- Setup:
  - Optimizer: Adam (with adjustable learning rate & weight_decay)
  - Loss: CrossEntropyLoss (with adjustable class_weights)
  - LR Scheduler: ReduceLROnPlateau
  - Early stopping: stop if no improvement for n epochs

- Training process:
  - Pass a batch by loading from the Dataset
  - Calculate loss using `criterion`
  - Backward propagation of the loss

- Validation
  - model.eval() + torch.no_grad(), Returns loss and accuracy

2.5 trainer.py (training loop + evaluation)

- Training loop pseudocode:

```
for each epoch:
     train_loss = train_this_epoch()
     val_loss, val_acc = validate()
     scheduler(val_loss)

     if val_loss improved: epochs_without_improvement = 0
     else: epochs_without_improvement += 1; break if epochs_without_improvement >= n

     save_checkpoint()
```

- Evaluation
  - Runs on test set and saves:
    - Predictions.csv: per-case predictions with probabilities
    - Confusion_matrix.png: visualize results

2.6 helpers (config.py & utils.py)

- Input data:
  - Labels_csv: case_grade_match.csv
  - Patches_dir: folder with patch image files
  - Runs_dir: the following output paths will be created under runs_dir

- Output path structure:

```
./runs/run_YYYYMMDD_HHMMSS/
├── results.txt
├── predictions.csv
├── confusion_matrix.png
├── data_splits.npz
└── checkpoints/
    └── *.pth
```

Most edited in config.py:

- per_slice_cap: reduce if OOM
- learning_rate
  - increase if not learning
  - Decrease if wobbly loss reduction
- class_weights: adjust for class imbalance
  - Higher benign class weight: higher penalty for mistaking benign as …
- early_stopping_patience

Key utils in utils.py:

- set_seed(): reproducible results
- save/load_data_splits():
  - Saves data_splits.npz with exact case IDs in each split
  - Could be read for reproducibility

2.7 Possible modifications

Want to...	Where to look...
Hyperparameter tuning with Optuna	Wrap `main()` [main.py] in Optuna objective function, sample from `TRAINING_CONFIG` [config.py]
Split cases into sub-cases	Modify `build_case_dict()` [data_utils.py]: Create more case IDs w/ splitting
Change regularization	`TRAINING_CONFIG` [config.py]: `dropout`, `weight_decay` (modify hyperparameters) `AttentionPool` [models.py]: modify `dropout` or add other regularization methods
Different attention mechanisms	`AttentionPool` [models.py]: Replace MLP with transformer, gated, or multi-head attention
Custom loss functions	`MILTrainer.__init__()` [trainer.py]: Replace `CrossEntropyLoss`

2.8 Using SBATCH

- See project documentation

Part 3: Exploring Next Steps for WI2026

3.1 Making More Cases

(Idea from Krish)

- The sbatch file currently has parameters: `--per_slice_cap 500 --max_slices_per_stain 5`
  - This means:
    - If a slice has **>500** patches, in each epoch 500 patches will randomly be selected to represent the slice
    - If a stain has **>5** slices, 5 slices will randomly be selected to represent the stain (in all epochs)
- In practice, how much data is unused? (Code link (to run in terminal))
  - **0** patches from any slice are unused across all epochs
  - **28** case / stain combinations have >5 slices
    - 7 of those combinations were for **benign cases** w/ stain(s) that have >5 slices: 26, 22, 24, 25, 27, 82, 65
- The idea:
  - Split cases, especially those with unused slice data, into multiple cases
    - For a given case, partition patch numbers and take selected patch numbers from all slices

3.1 Making More Cases

- **Critique**: even if splitting patches using similar index cutoffs, there could be significant information overlap across sub-cases from the parent case
    - Patching function positions the first patch based on the first contour found; the initial patch could be placed differently across 2 slices that are relatively similar, and thus the same range of patch indices could contain largely overlapping information
- However, if we if we only partition cases in training, despite potential overlap, there would be **no data leakage**
- May be beneficial to try in training, **particularly for benign cases with unused slices**

3.2 Patch-by-Patch Predictions

(Idea from Krish)

- **Goal**: to understand what individual patches predict for the case
- **Method**: iteratively go through all patches and predict for the case using 100% attention on that one patch
    - Or just run the model on only the patch without the attention layers

3.3 Regularization

- Current approach:
  - weight decay: L2 regularization on weight updates during back propagation
    - patch_projector: keeps mapping smooth so similar image patches result in similar embeddings
    - attention: prevent one patch/slice/stain to dominate attention
    - classifier: prevent features with larger weight to dominate the classification process
  - Dropout: randomly zeros out some attention weights
  - Early stopping
  - Learning rate scheduler
- **Current goal: try to prevent overly sparse attention distribution caused by the hierarchical structure**
- Potential issue with current approach:
  - Most regularization are done on training process (early stopping, learning rate scheduler)
  - Configs are chosen and adjusted manually based on results from our runs
  - Dropout introduces noise to make attention mechanism more robust, but it does not directly affect attention distribution

3.3 Regularization

- **Proposal 1:** ElasticNet()
  - Implement in each attention layer before softmax function on attention logits
  - penalty = $\lambda_1 \|\text{logits}\|_1 + \lambda_2 \|\text{logits}\|_2^2$
  - Intuition: L1 encourages sparsity in attention, L2 prevents extremely large attention. This will help prevent the hierarchical structure amplifying extremes, while still make attention selective.
  - Ideas about tuning:
    - Tune L1:L2 ratio. L2 should be higher because our goal is prevent larger attentions keep getting larger.
    - Applying ElasticNet() on all three attention layer might a overkill. Experiment with applying just on the patch-level attention, or first two layers.
    - Along that line, could also try to regularize early layers for stability, and leave later layers freer for selective attention. Eg. apply L2 for patch-level and slice-level, apply L1 for stain-level

3.3 Regularization

- **Proposal 2:** entropy regularization
  - For each attention layer, Add penalty term based on entropy after softmax
  - Entropy formula $$H(w) = -\sum_i w_i \log w_i$$
    - If attention weights are more uniformly distributed, entropy is high
    - If we have very sparse attention distribution, entropy is low
  - Penalty = -λH(w)
    - This gives more penalty when entropy is low. In gradient descent, this pushes smaller attention weight up, and large attention weight down. aka penalize “large attention larger, smaller attention smaller”
  - Ideas about tuning:
    - Larger λ -> stability; smaller λ -> selectivity
    - Similar idea: gradually decrease λ as we moves from patch attention layer to stain layer (so that later layer are more selective)

3.3 Regularization

- **Proposal 3:** tuning weight_decay
  - Weight decay in attention helps to reduce sparsity, but currently, we manually selected the config
  - Ideas:
    - Involves weight_decay as a part of hyperparameter tuning (Optuna, Bayessearch)
    - Try adaptive weight decay (Nakamura & Hong, 2019)
      - Core idea: adjust decay based on gradient, higher gradient -> larger decay; lower gradient -> smaller decay instead of a constant value
      - Intuition: standard weight decay constantly shrinks large weights and lead attention distribution to flatten out; adaptive weight decay shrinks large weights when model is in the learning process (gradient is high), but maintain large weights after convergence
  - Standard weight decay formula
    - f as loss function $w_{t+1} = w_t - \eta \left(\nabla f(w_t) + \lambda w_t \right)$
    - lambda as weight_decay co[illegible]

3.3 Hyperparameter Tuning: Training Configuration

- Hyperparameter tuning: learning_rate, weight_decay
  - Currently have a learning rate scheduler but could try a properly tuned fixed learning rate
  - Tune weight_decay if not using adaptive weight decay (slide 36)

- Optuna

Benefits	● Learns from previous combinations to focus on high-performing hyperparameters ● Efficient (vs a grid search trying every possible combination) ● Can be implemented with current trainer structure
Downsides	● Compare Optuna results with adaptive weight decay, LR scheduler ● High computational cost
How it works	● Uses Bayesian Optimization to learn from past trials ● Plots optimization history and hyperparameter importance

3.3 Hyperparameter Tuning: Training Configuration

Optuna sample psuedocode:

Source: Optuna website example modified by ChatGPT

```
def objective(trial):
    lr = trial.suggest_float("lr", 1e-6, 1e-2, log=True)
    dropout = trial.suggest_float("dropout", 0.1, 0.7)

    model = MyNet(dropout=dropout)
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)

    val_loss = train_and_eval(model, optimizer)
    return val_loss

study = optuna.create_study(direction="minimize")
study.optimize(objective, n_trials=75)

print(study.best_params)
```

Key features:

- Define objective function to be maximized
- Suggest hyperparameter values
- Create a study object and invoke optimize method

References

Krish's ideas from last week

GenAI tools

Nakamura, Kensuke & Hong, Byung-Woo. (2019). Adaptive Weight Decay for Deep Neural Networks. IEEE Access. PP. 1-1. 10.1109/ACCESS.2019.2937139.