5/28 Presentation

Cross-stain Model Robustness: changing test data

	precision	recall	f1-score
0	0.75	0.87	0.81
1	0.93	0.86	0.89
accuracy			0.86
macro avg	0.84	0.87	0.85
weighted avg	0.87	0.86	0.87

	precision	recall	f1-score
0	0.76	0.89	0.82
1	0.94	0.86	0.90
accuracy			0.87
macro avg	0.85	0.88	0.86
weighted avg	0.88	0.87	0.87

Initial run

	precision	recall	f1-score
0	0.77	0.83	0.80
1	0.92	0.90	0.91
accuracy			0.88
macro avg	0.85	0.86	0.85
weighted avg	0.88	0.88	0.88

Cross-stain Model Robustness: adding data

More data

	precision	recall	f1-score
0	0.69	0.87	0.77
1	0.92	0.80	0.85
accuracy			0.82
macro avg	0.81	0.83	0.81
weighted avg	0.84	0.82	0.83

Initial run

	precision	recall	f1-score
0	0.77	0.83	0.80
1	0.92	0.90	0.91
accuracy			0.88
macro avg	0.85	0.86	0.85
weighted avg	0.88	0.88	0.88

Case 34, match 1 (B)
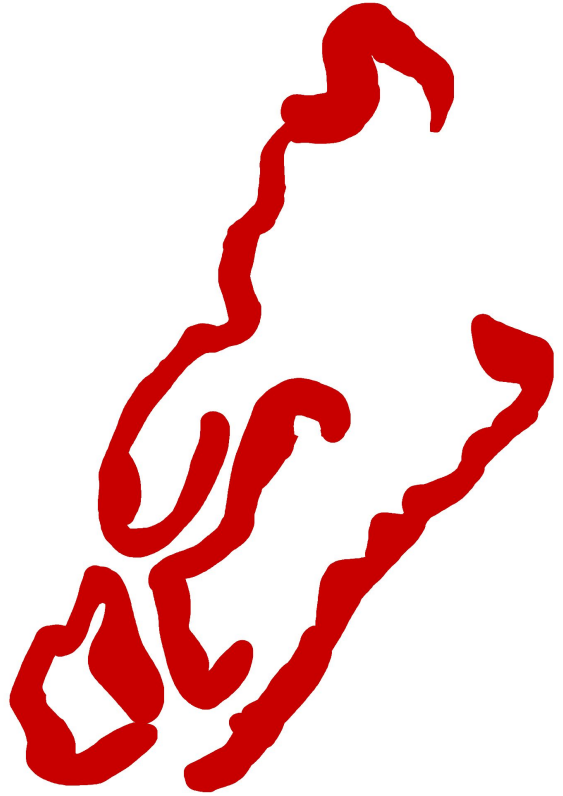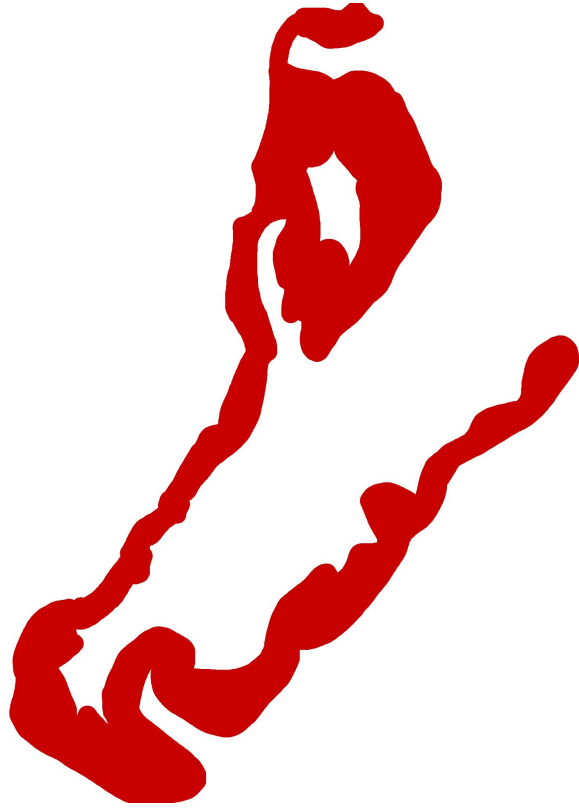
Final Presentation Workflows

Jeffrey and Harvey

Undersampling

- Undersample for high grade
- Subsample from slides with lots of patches

Promising Models So Far

AlexNet

Serial number	#Patch-level recall: Benign	#Patch-level recall: High-grade
33	92.9	90.8
34	0.83	0.9

ResNet with CBAM

Serial number	#Patch-level recall: Benign	#Patch-level recall: High-grade
27	0.7	0.79

CoAtNet

Serial number	#Patch-level recall: Benign	#Patch-level recall: High-grade
16	0.55	0.93

Resizing Workflow

1. Determine the best model for each stain type
   a. We test
      i. 3 most promising models per stain
      ii. Also resizing and pooling per stain
   b. 18 models total
2. Find the best-performing model for each stain and ensemble them
   a. Find best performing patch level model
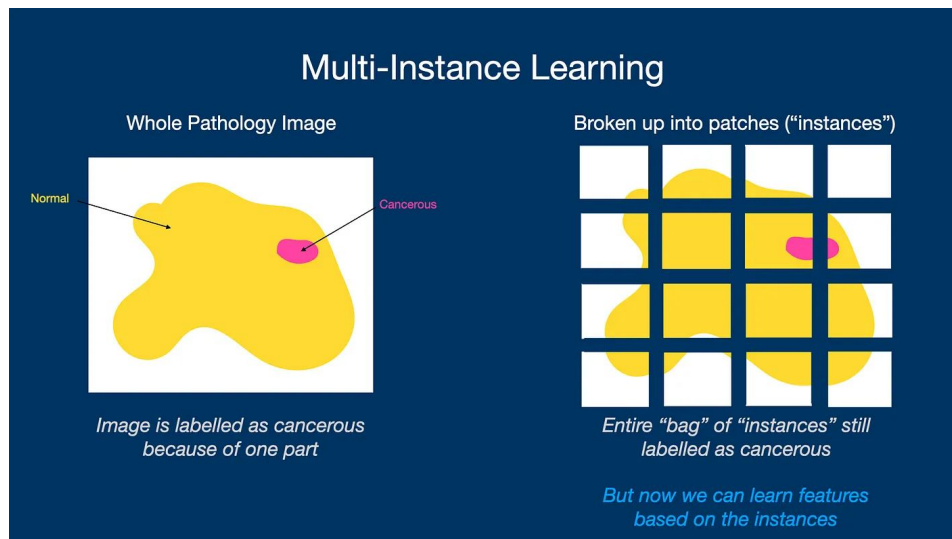   b. Patch level recall: High Grade and Benign

Ensemble Workflow

1. Pick best past models for each stain; ensemble these
   a. The ensemble splits all cases into test, train, and validation sets
   b. Each stain will get predicted by the best previously trained model per stain
   c. A majority vote is used to determine the case level prediction
2. Attempt to use Akhil's SVM Case Level Classifier
   a. Select top K confident predictions across all stains
   b. Select top K/3 confident predictions per stain
   c. Select most confident predictions per stain proportional to total distribution
   d. Select top K confident prediction per stain, majority voting across stains
3. Compare to human level accuracy and previous results

Multiple Instance Learning

Hannah

What is it? Why use it here?

MIL is a form of supervised learning where the input data is organized into labeled bags containing multiple instances. Instead of each instance being labeled individually, only the bag as a whole has a label.

Multi-Instance Learning

Whole Pathology Image

Normal

Cancerous

Image is labelled as cancerous because of one part

Broken up into patches ("instances")

Entire "bag" of "instances" still labelled as cancerous

But now we can learn features based on the instances

"Multiple Instance Learning (MIL) is designed to classify instances where class labels are associated with sets of instances, a common occurrence in biomedical data, especially when multiple images are derived from a single object measurement."

Data Representation

1. **Instances**: These are the individual data points within a bag. For example, in our case, instances would be the patches of the image.
2. **Bags:** Each bag is a collection of instances. In our case, the bag is the case and the patches or segments of the image are the instances within that bag.
3. **Labels:** Bags are labeled with a class label (e.g., "high grade" or "benign").

Application Process

1. Data Preparation:
   a. each case is a bag, and each patch is an instance (bag and instance construction

```python
def group_patches_recursive(root_dir):
    case_patches = {}
    for root, _, files in os.walk(root_dir):
        for filename in files:
            match = re.search(r"case_(\d+)", filename)
            if match and filename.endswith(".png"):
                case_num = int(match.group(1))
                if case_num not in case_patches:
                    case_patches[case_num] = []
                case_patches[case_num].append(os.path.join(root, filename))
    return case_patches

patches = group_patches_recursive(filtered_patches_dir)
```

Patches
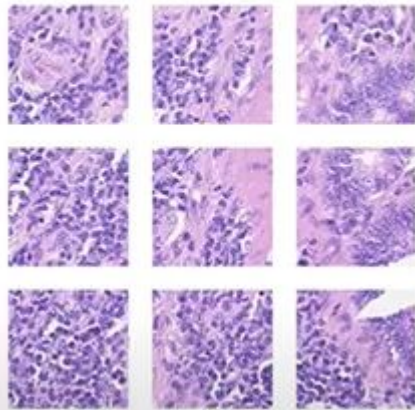
Application Process

1. Data Preparation:
   a. It builds a dataset where each item is a case (bag) made up of image patches (instances)
   b. self.bags holds lists of patch image paths, one list per case
   c. self.labels holds the corresponding bag-level (case-level) label.
   d. emergency_cap prevents memory issues by randomly sampling a fixed number of patches (e.g., 800) if a case has too many patches

```python
class MILDataset(Dataset):
    def __init__(self, case_patches, labels_df, transform=None, emergency_cap=800):
        self.transform = transform
        self.emergency_cap = emergency_cap  # only cap if massive
        self.bags, self.labels = [], []
        for case, paths in case_patches.items():
            raw = labels_df.loc[labels_df['Case'] == case, 'Class'].item()
            bag_lbl = 0 if raw == 1 else 1
            self.bags.append(paths)
            self.labels.append(bag_lbl)

    def __len__(self): return len(self.bags)

    def __getitem__(self, idx):
        paths = self.bags[idx]
        imgs = []
        for p in paths:
            try:
                img = Image.open(p).convert('RGB')
                if self.transform:
                    img = self.transform(img)
                imgs.append(img)
            except:
                continue

        if len(imgs) == 0:
            raise ValueError(f"No good patches in case {paths}")

        # Only sample if emergency_cap is set
        if self.emergency_cap is not None and len(imgs) > self.emergency_cap:
            imgs = random.sample(imgs, self.emergency_cap)

        return torch.stack(imgs), torch.tensor(self.labels[idx], dtype=torch.long)
```

Application Process

2. Feature Extraction

    a.   base_model.features: Convolutional layers from DenseNet extract features from each patch

    b.   AdaptiveAvgPool2d((2,2)): Each feature map is pooled into a 2×2 grid (4 vectors per patch)

    c.   patch_projector: Those 4 vectors are flattened and projected into a shared embedding space of dimension 512 (i.e., your embed_dim)

```python
class AttnMIL(nn.Module):
    def __init__(self, base_model, num_classes=2, embed_dim=512):
        super().__init__()
        # grabbing the convolutional feature extractor from the pretrained model
        self.features = base_model.features
        # applying adaptive average pooling to compress to feature map of 2x2 grid
        # you get 4 spatial vectors per patch
        self.pool = nn.AdaptiveAvgPool2d((2,2))  # richer than (1,1)
        # meaning that you'll get 4 vectors per patch which will then be flattened
        self.patch_projector = nn.Linear(base_model.classifier.in_features * 4, embed_dim)
        self.attention_pool = AttentionPool(embed_dim)
        self.classifier = nn.Linear(embed_dim, num_classes)


    def forward(self, x, return_patch_logits=False, return_attn_weights=False):
        if x.dim() == 4:
            x = x.unsqueeze(0)
        # typically after CNN you get 3D tensor with num channels, height and width of image
        # but we packed the patches into a bag by case (the tensor), so B is batch size, M is number of patches per bag
        B, M, C, H, W = x.shape
        x = x.view(B*M, C, H, W)

        features = self.features(x) # exxtracting cnn features for each patch
        pooled = self.pool(features).view(B*M, -1) # pool each feature map to a 2x2 grid and flatten
        embedded = self.patch_projector(pooled).view(B, M, -1) # project each patch into shared embedding space --
        # just ensuring all the patches are transformed into vectors of the same length for attention

        # in order to get patch level predictions
        if return_patch_logits:
            logits = self.classifier(embedded)  # (B, M, 2)
            return logits
        # returning attention weights for visualization
        if return_attn_weights:
            bag_emb, attn_weights = self.attention_pool(embedded, return_weights=True)
            logits = self.classifier(bag_emb)
            return logits, attn_weights  # bag prediction + per-patch attention scores

        # applying attention
        #computing a w̲e̲i̲g̲h̲t̲e̲d̲ ̲a̲v̲e̲r̲a̲g̲e̲ ̲o̲f̲ ̲p̲a̲t̲c̲h̲ ̲e̲mbeddings using attention, and then is passed through the classifier to get bag level prediction
        bag_emb = self       (variable) classifier: Any
        logits = self.classifier(bag_emb)
        return logits
```

Application Process

3. Attention pooling

    a.   Learns importance weights for each patch

    b.   Applies softmax to normalize across patches

    c.   Uses weighted average to get a single bag embedding

    d.   Outputs interpretable attention scores per patch

```python
# this pools the patch level features into single bag level representation for MIL
class AttentionPool(nn.Module):
    def __init__(self, input_dim, hidden_dim=128):
        super().__init__()
        # creates small neural network to compute attention scores for each patch
        # each patch embedding is passed through a linear layer, tanh for nonlinearity, and another linear layer to get a scalar score
        self.attention = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.Tanh(),
            nn.Linear(hidden_dim, 1)
        )

    def forward(self, x, return_weights=False):
        # x is of shape (B, M, D) where B is batch size or number of cases,
        # M is number of patches per bag, and D is the embedding dimensions for each patch
        weights = self.attention(x)      # (B, M, 1)
        weights = torch.softmax(weights, dim=1)
        # outputs attention scores for each patch and normalized with softmax
        # D is the embedding dimension which is size of feature vector for each patch after going through the patch classifier
        weighted_x = (weights * x).sum(dim=1)  # (B, D)
        # returning the raw attention weights per patch just to help with visualization of the weights for each patch
        if return_weights:
            return weighted_x, weights.squeeze(-1)  # (B, D), (B, M)
        return weighted_x
```

$$\text{Bag embedding} = \sum_{i=1}^{5} \alpha_i \cdot \mathbf{x}_i$$

where:

- $\mathbf{x}_i$ is the feature vector for the i-th patch

- α_i is the attention weight for the i-th patch (from softmax)

- $\sum \alpha_i = 1$

Next steps: Grouping by Slice instead of Case

1. Treat each slice as a bag during MIL
   a. I.e. case_9_unmatched_1 is one bag, case_9_unmatched_2 is another
2. Use MIL as usual: attention to get slice-level prediction
3. Aggregate slice predictions into case level prediction
   a. Average slice softmax probabilities
   b. Max probability for high-grade
   c. Majority vote over slices