Week 8 Report

Thomas Deatherage
NFHM/BioCosmos
11 October 2024

Time Slot

1) What progress did you make in the last week?

- Weekly meeting with GaTech NFHM collaborators + Dr Porto.
- More progress on fine-tuning florence-2's model for trait grounding

2) What are you planning on working on next?

- Explore InternVL and/or other LLM/VLMs
    - Focus on trait grounding/referring aspects and performance
    - Research fine-tuning said LLMs

3) Is anything blocking you from getting work done?

- Very little documentation and examples of fine tuning florence-2, especially the REFERRING_EXPRESSION_SEGMENTATION task

Abstracts & Summaries

Lit review replaced by journal club:
https://humanaugmente-e7j6563.slack.com/archives/C07358BJDEY/p1728494700718729

Scripts and Code Blocks

This week and last week, I've been working on fine tuning the florence-2 model. Specifically, I've been focusing on fine tuning the REFERRING_EXPRESSION_SEGMENTATION, which I've been using to some success for trait grounding purposes. Although I am able, in a certain sense, to "tune" the model. I've been having some difficulty improving the quality of my performance benchmarks. In fact, the quality of the outputs tends to go down after fine-tuning. There's only scant documentation available on fine-tuning the model's various VLM tasks. Roboflow has a pretty thorough tutorial on fine-tuning object detection. However, my attempts at porting those concepts over to segmentation have been pretty mixed. Part of the difficulty is that running a single epoch is very time-consuming. Consequently, my feedback loop is very slow. Although it's tricky to manage, I'm going to explore provisioning multiple PACE VMs simultaneously so I can more quickly iterate on changes.

Below, I've repeat the full finetune script as I have it now:

```
# Imports etc . . .

class FishVistaDataset(Dataset):
    def __init__(self, csv_file, image_dir, seg_dir, trait_map_path,
processor):
        self.data = pd.read_csv(csv_file)
        self.image_dir = image_dir
        self.seg_dir = seg_dir
        self.processor = processor
#         self.resize = Resize((512, 512))  # Resize


        with open(trait_map_path, 'r') as f:
            self.trait_map = json.load(f)

        # Filter out rows with missing image or segmentation files
        valid_rows = []
        for idx, row in self.data.iterrows():
            img_path = os.path.join(self.image_dir, row['filename'])
            seg_path = os.path.join(self.seg_dir,
os.path.splitext(row['filename'])[0] + '.png')
            if os.path.exists(img_path) and os.path.exists(seg_path):
                valid_rows.append(idx)
            else:
                print(f"Skipping row {idx}: Image or segmentation file not
found.")

        self.data = self.data.iloc[valid_rows].reset_index(drop=True)
        print(f"Dataset contains {len(self.data)} valid images.")

    def __len__(self):
        return len(self.data)
    def __getitem__(self, idx):
        try:
            img_name = self.data.iloc[idx]['filename']
            img_path = os.path.join(self.image_dir, img_name)
            seg_name = os.path.splitext(img_name)[0] + '.png'
            seg_path = os.path.join(self.seg_dir, seg_name)

            image = Image.open(img_path)
            image_np = np.array(image)
            seg_mask = np.array(Image.open(seg_path))

            # Get all traits present in the image
            traits = [self.trait_map[str(i)] for i in np.unique(seg_mask)
if str(i) in self.trait_map and i != 0]

            results = []
            for trait in traits:
                # Create a prompt for the REFERRING_EXPRESSION_SEGMENTATION
task
                prompt = f"<REFERRING_EXPRESSION_SEGMENTATION> a fish
{trait}"

                # Create polygon for the trait
                trait_id = [k for k, v in self.trait_map.items() if v ==
trait][0]
                trait_mask = (seg_mask == int(trait_id)).astype(np.uint8)
                contours, _ = cv2.findContours(trait_mask,
cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

                polygons = []
                for contour in contours:
                    if len(contour) >= 2:
                        epsilon = 0.01 * cv2.arcLength(contour, True)
                        approx = cv2.approxPolyDP(contour, epsilon, True)

                        if len(approx) >= 2:
                            h, w = seg_mask.shape[:2]
                            polygon = approx.reshape(-1).tolist()
                            polygon = [max(0, min(999, int(p * 999 / h)))
if i % 2 == 0 else max(0, min(999, int(p * 999 / w))) for i, p in
enumerate(polygon)]

#                               poly_str = f"{trait}"
```

```
                            poly_str = ""
                            for i in range(0, len(polygon), 2):
                                poly_str +=
f"<loc_{polygon[i+1]}><loc_{polygon[i]}>"
                            polygons.append(poly_str)

                target = ' '.join(polygons)

                try:
                    inputs = self.processor(text=prompt, images=image_np,
return_tensors="pt", padding=True)
                    inputs = {k: v.squeeze(0) for k, v in inputs.items()}
# Remove batch dimension
                    results.append((inputs, target, image))
                except Exception as e:
                    print(f"Error processing trait {trait} for image
{img_path}: {str(e)}")

            return results

        except Exception as e:
            print(f"Error processing item {idx}: {str(e)}")
            print(f"Image path: {img_path}")
            print(f"Segmentation mask path: {seg_path}")
            return None


def collate_fn(batch, processor):
    # Flatten the batch of results
    flattened_batch = [item for sublist in batch if sublist is not None for
item in sublist]

    if len(flattened_batch) == 0:
        return None

    input_ids = pad_sequence([item[0]['input_ids'] for item in
flattened_batch], batch_first=True, padding_value=0)
    attention_mask = pad_sequence([item[0]['attention_mask'] for item in
flattened_batch], batch_first=True, padding_value=0)
    pixel_values = torch.stack([item[0]['pixel_values'] for item in
flattened_batch])

    targets = [item[1] for item in flattened_batch]
    images = [item[2] for item in flattened_batch]
    target_encoding = processor(text=targets, images=images,
return_tensors="pt", padding=True)
    target_ids = target_encoding.input_ids

    print("Sample target:", targets[0])
    print("Sample target_ids:", target_ids[0][:10])

    return {
        'input_ids': input_ids,
        'attention_mask': attention_mask,
        'pixel_values': pixel_values,
        'labels': target_ids
    }


def train(model, train_dataloader, test_dataloader, optimizer, scheduler,
device, num_epochs):
    model.train()
    for name, param in model.named_parameters():
        if param.requires_grad:
            print(f"Parameter {name} requires gradient")

    for epoch in range(num_epochs):
        total_loss = 0
        for batch_idx, batch in enumerate(tqdm(train_dataloader,
desc=f"Epoch {epoch + 1}")):
            if batch is None:
                continue  # Skip this batch if it's None

            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            pixel_values = batch['pixel_values'].to(device)
            labels = batch['labels'].to(device)

            outputs = model(input_ids=input_ids,
attention_mask=attention_mask, pixel_values=pixel_values, labels=labels)
            loss = outputs.loss

            print(f"Batch {batch_idx}, Loss: {loss.item()}")
            print(f"Input shape: {input_ids.shape}, Label shape:
{labels.shape}")
            print(f"Sample input: {input_ids[0][:10]}")
            print(f"Sample label: {labels[0][:10]}")

            total_loss += loss.item()

            loss.backward()
            optimizer.step()
            for name, param in model.named_parameters():
                if param.requires_grad:
                    print(f"Parameter {name} grad norm:
{param.grad.norm().item() if param.grad is not None else 'None'}")

            scheduler.step()
            optimizer.zero_grad()

            if batch_idx % 10 == 0:
                print(f"Epoch {epoch + 1}, Batch {batch_idx}, Loss:
{loss.item()}")

        avg_loss = total_loss / len(train_dataloader)
        print(f"Epoch {epoch + 1}, Average Loss: {avg_loss}")

        # Evaluation loop
        model.eval()
        total_eval_loss = 0
        with torch.no_grad():
            for batch in tqdm(test_dataloader, desc=f"Evaluation after
Epoch {epoch + 1}"):
                if batch is None:
                    continue  # Skip this batch if it's None

                input_ids = batch['input_ids'].to(device)
                attention_mask = batch['attention_mask'].to(device)
                pixel_values = batch['pixel_values'].to(device)
                labels = batch['labels'].to(device)

                outputs = model(input_ids=input_ids,
attention_mask=attention_mask, pixel_values=pixel_values, labels=labels)
                loss = outputs.loss
                total_eval_loss += loss.item()

        avg_eval_loss = total_eval_loss / len(test_dataloader)
        print(f"Epoch {epoch + 1}, Average Evaluation Loss:
{avg_eval_loss}")
```

```
    # Return the final training and evaluation loss
    return avg_loss, avg_eval_loss


def main():
    # Set up device
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    # Load Florence-2 model and processor
    model_id = "microsoft/Florence-2-base"
    processor = AutoProcessor.from_pretrained(model_id)
    model = AutoModelForCausalLM.from_pretrained(model_id,
trust_remote_code=True).to(device)

    print("Processor version:", processor.__class__.__name__)
#     print("Tokenizer config:", processor.tokenizer.get_config())

    # Set up LoRA
    lora_config = LoraConfig(
        r=8,
        lora_alpha=8,
        target_modules=["q_proj", "o_proj", "k_proj", "v_proj", "linear",
"Conv2d", "lm_head", "fc2"],
        task_type="CAUSAL_LM",
        lora_dropout=0.05,
        bias="none",
        inference_mode=False,
        use_rslora=True,
        init_lora_weights="gaussian",
        revision="refs/pr/6"
    )
    model = get_peft_model(model, lora_config)

    # Set up dataset and dataloader
    train_dataset = FishVistaDataset(
        csv_file="./fish-vista/segmentation_train.csv",
        image_dir="./fish-vista/AllImages",
        seg_dir="./fish-vista/segmentation_masks/images",

trait_map_path="./fish-vista/segmentation_masks/seg_id_trait_map.json",
        processor=processor
    )
    train_dataloader = DataLoader(train_dataset, batch_size=2,
shuffle=True, collate_fn=lambda batch: collate_fn(batch, processor))

    test_dataset = FishVistaDataset(
        csv_file="./fish-vista/segmentation_test.csv",
        image_dir="./fish-vista/AllImages",
        seg_dir="./fish-vista/segmentation_masks/images",

trait_map_path="./fish-vista/segmentation_masks/seg_id_trait_map.json",
        processor=processor
    )

    test_dataloader = DataLoader(test_dataset, batch_size=2, shuffle=True,
collate_fn=lambda batch: collate_fn(batch, processor))


    avg_traits_per_image = 7  # estimate given the range of 1-9 traits

    # Calculate estimated number of training steps
    num_images = len(train_dataset)
    batch_size = 4
    num_epochs = 1
    estimated_steps_per_epoch = (num_images * avg_traits_per_image) //
batch_size
    num_training_steps = num_epochs * estimated_steps_per_epoch

    # Set up optimizer and scheduler
    optimizer = AdamW(model.parameters(), lr=5e-5)
    scheduler = get_linear_schedule_with_warmup(optimizer,
num_warmup_steps=0, num_training_steps=num_training_steps)


#     # Set up optimizer and scheduler
#     optimizer = AdamW(model.parameters(), lr=5e-5)
#     num_epochs = 2
#     num_training_steps = num_epochs * len(train_dataloader)
#     scheduler = get_linear_schedule_with_warmup(optimizer,
num_warmup_steps=0, num_training_steps=num_training_steps)

    # Train the model
    train(model, train_dataloader, test_dataloader, optimizer, scheduler,
device, num_epochs)
```

```
    # Save the fine-tuned model
    model.save_pretrained("./fine_tuned_florence2")
    processor.save_pretrained("./fine_tuned_florence2")

if __name__ == "__main__":
    main()
```

Flow Charts/Diagrams

Nothing new to show here.

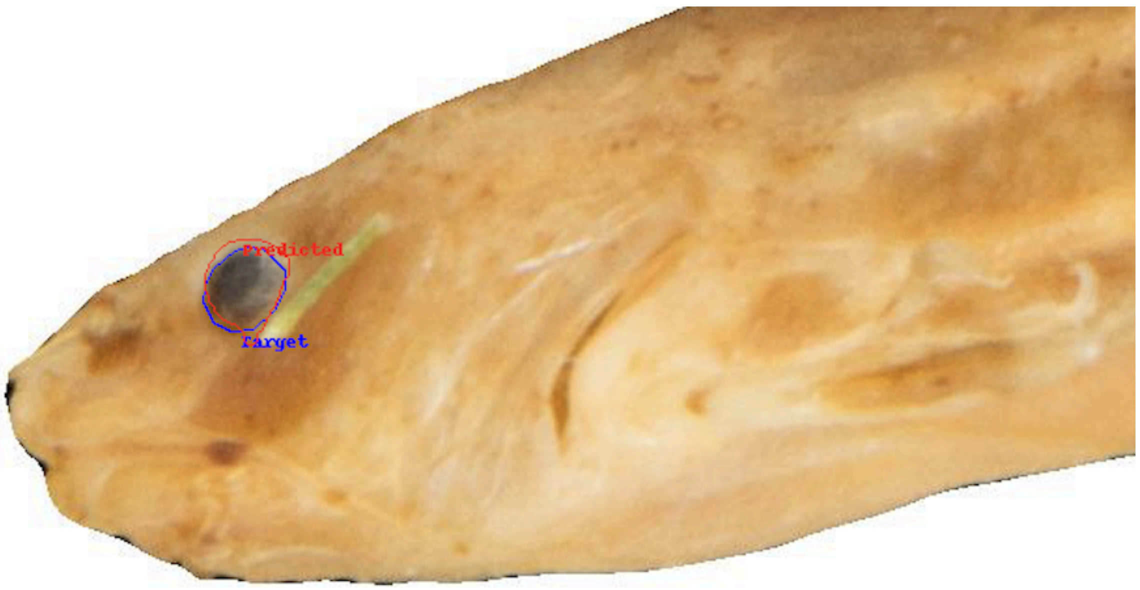
Documentation

Nothing new this week.

Results Visualization + Proof of Work

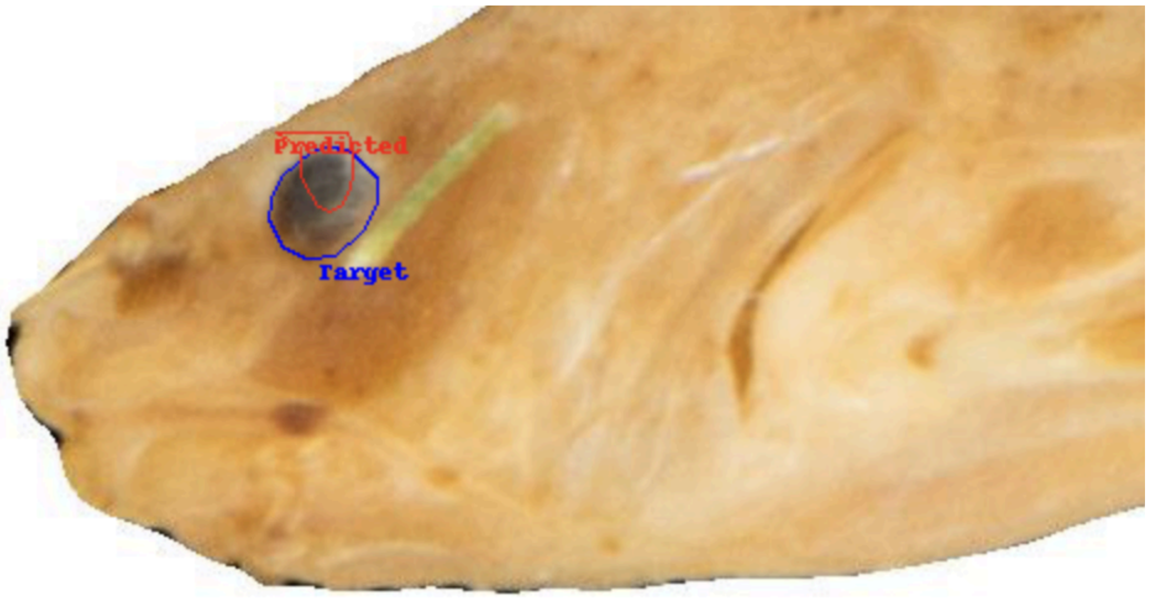
Here's some examples of the florence-2's large model vs my fine-tuned model. For context, the fine-tuned model is built with the florence-2's base model with 3 epochs of training using the fish-vista dataset. My goal is to benchmark the base model, the large model, and variously finetuned models.

Target vs predicted with florence-2's **large** model in a cropped section of a fish:

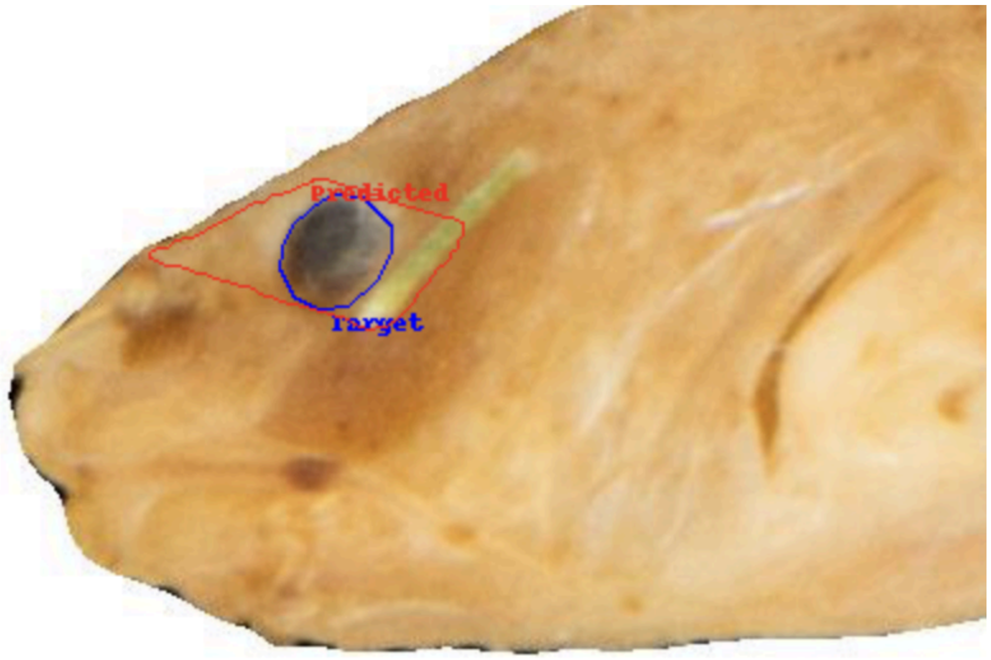


As you can see, the two circles overlap pretty closely. This is one of the better examples. Still, it shows that the large florence-2 model is by default not bad.

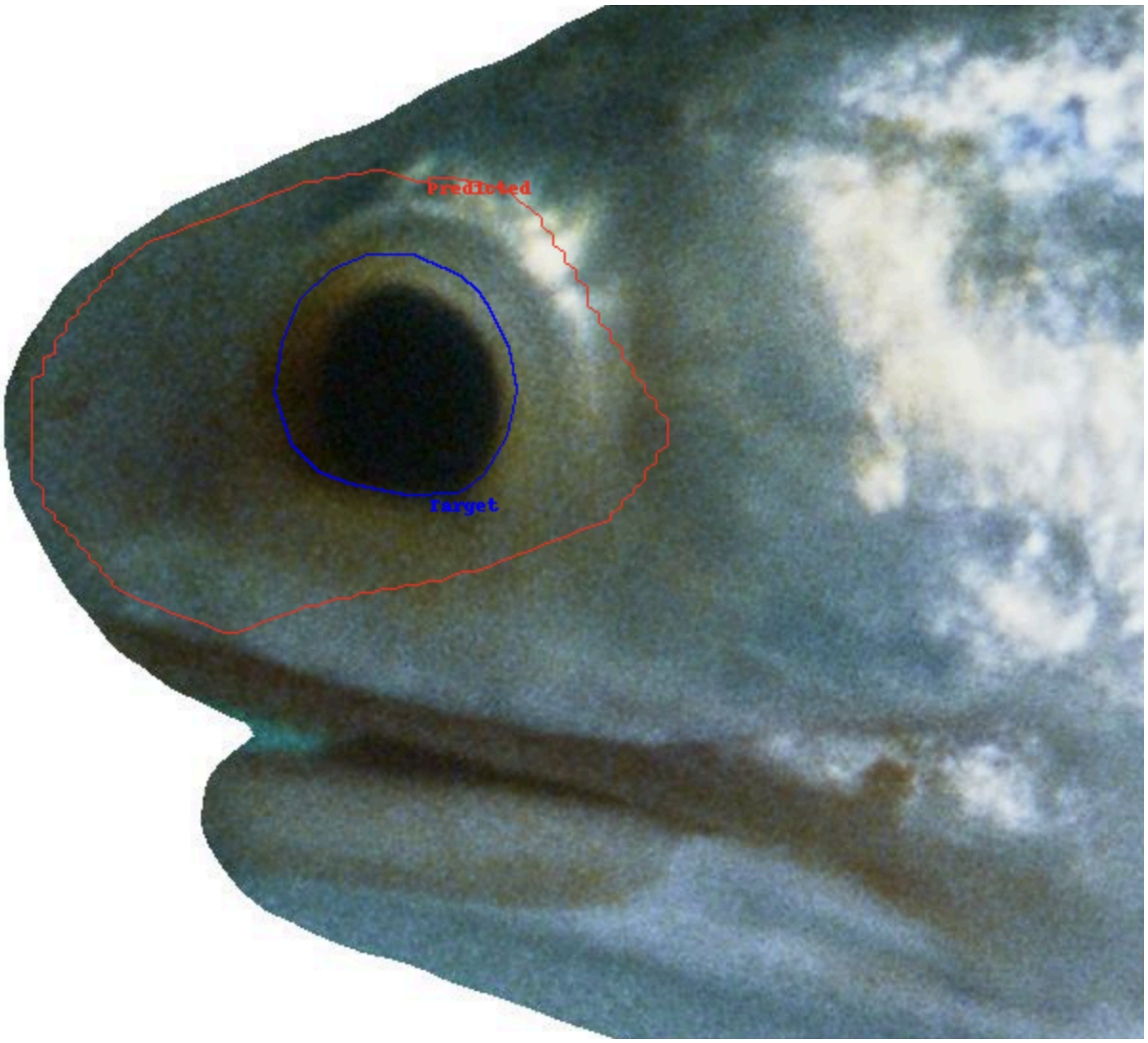
Here's the same fish, but with the prediction from the **fine-tuned** model:

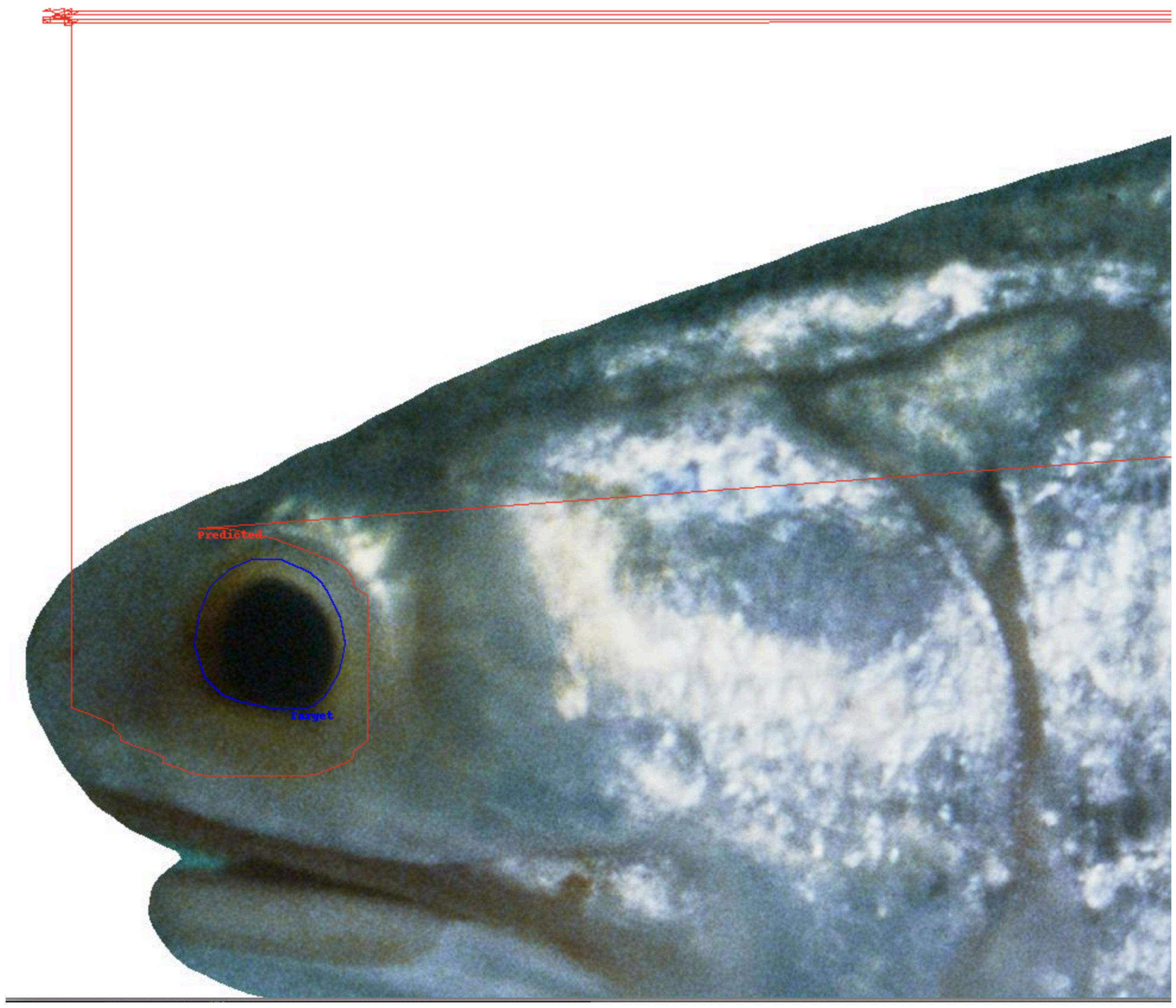


As you can see from a visual comparison of the red-circles, the performance has gotten worse compared to the large model. OK, but then you might ask how does it compare against the base florence-2 model, which after all is probably the better benchmark?. Well, here's what the florence-2 **base** model predicts:



Now that doesn't look so bad. In fact, subjectively, it looks like the fine-tuned model may have in fact improved the performance. However, there are other examples where the fine-tuned model's results are way out of whack compared to the base model. Consider these next two cropped examples. I'll let you guess which model produced which:


Predicted
Target



So at best, pretty mixed-results. Measured overall – using average intersection over union – my attempts at fine-tuning resulted in worse performance. About a 40% drop in performance for segmenting eyes, specifically. Other traits also showed worse performance with the fine-tuned version.

Next Week's Proposal

- InterVL and/or other LLM/VLMs
  - Topics: Fine-tuning, trait referral, trait grounding.