Week 11 Report

Thomas Deatherage
NFHM/BioCosmos
1 November 2024

Time Slot

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

- Weekly catchup with UF collaborators.
- Another deep dive into the UNICOM paper https://arxiv.org/pdf/2304.05884
- Used last week's InternVL captioned images to create clusters/pseudo labels per the process described in the Unicom paper.
- Used those cluster prototypes and the VLM4Bio data to train CLIP using the Unicom codebase.

2) What are you planning on working on next?

- Benchmarking performance of the VLM4Bio trained Unicom clip model that I'm training this week
- Work out any kinks
- Understand the specific training hyperparameters better
- Potentially move on to using larger datasets

3) Is anything blocking you from getting work done?

- Nope.

Abstracts & Summaries

Lit review replaced by journal club

Scripts and Code Blocks

This week was largely centered around testing and modifying the Unicom code to work with the VLM4Bio dataset. To this end, I first generated clusters or pseudo-classes for the 30k image/caption pairs. (Note the captions were generated last week via InternVL). The Unicom

code can be found at their Github repo. The most relevant files are retrieval.py and partial_fc.py.

Here's a relevant sample of the clustering code: Takes the 30,000 image/text captions and generates 3000 clusters from the average of each image<>caption embedding pair.

```
def combine_and_cluster(n_clusters=3000):
    """
    Load all .mat files, combine embeddings, and perform clustering
    """
    # Get all mat files -- Performance??????
    mat_files = sorted(glob.glob('embeddings_batch_*.mat'))  # Sort to
maintain order

    # Lists to store all embeddings and metadata
    all_image_embeddings = []
    all_text_embeddings = []
    all_image_paths = []
    all_scientific_names = []
    all_categories = []

    # Load each file
    for mat_file in mat_files:
        print(f"Loading {mat_file}")
        data = loadmat(mat_file)
        all_image_embeddings.extend(data['image_embeddings'])
        all_text_embeddings.extend(data['text_embeddings'])
        all_image_paths.extend(data['image_paths'])
        all_scientific_names.extend(data['scientific_names'])
        all_categories.extend(data['categories'])

    # Convert to numpy arrays
    image_embeddings = np.array(all_image_embeddings)
    text_embeddings = np.array(all_text_embeddings)

    # Create joint embeddings
    joint_embeddings = (image_embeddings + text_embeddings) / 2 # Average

    # Perform clustering
    n, d = joint_embeddings.shape
    n_clusters = min(n_clusters, n)
```

This I had to add to retrieval.py as a class-adaptor for our VLM4Bio clustered dataset.

```
# Used to load VLM4BIO-based clustered dataset
class CustomClusteredDataset(Dataset):
    def __init__(self, cluster_results_path, transform=None):
        self.df = pd.read_csv(cluster_results_path)
        self.transform = transform
        self.num_classes = self.df['cluster_id'].nunique()

        # Clean and convert paths
        base_dir = os.path.dirname(os.path.abspath(cluster_results_path))
        self.df['image_path'] = self.df['image_path'].apply(
            lambda x: os.path.abspath(os.path.join(base_dir, x.strip()))
        )

        # Debugging: Print first few paths to verify
        print("First few image paths:")
        print(self.df['image_path'].head())

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

    def __getitem__(self, idx):
        row = self.df.iloc[idx]
        image_path = row['image_path']
        cluster_id = row['cluster_id']

        try:
            image = PIL.Image.open(image_path)
            if self.transform:
                image = self.transform(image)
            return image, cluster_id
        except Exception as e:
            print(f"Error loading image {image_path}: {e}")
            raise
```

I was having difficulty with the main function, which is the entrypoint for the retrieval.py script. So I re-wrote it from scratch:

```
def main(config=None):
    print("Current working directory:", os.getcwd())
    if config is None:
```

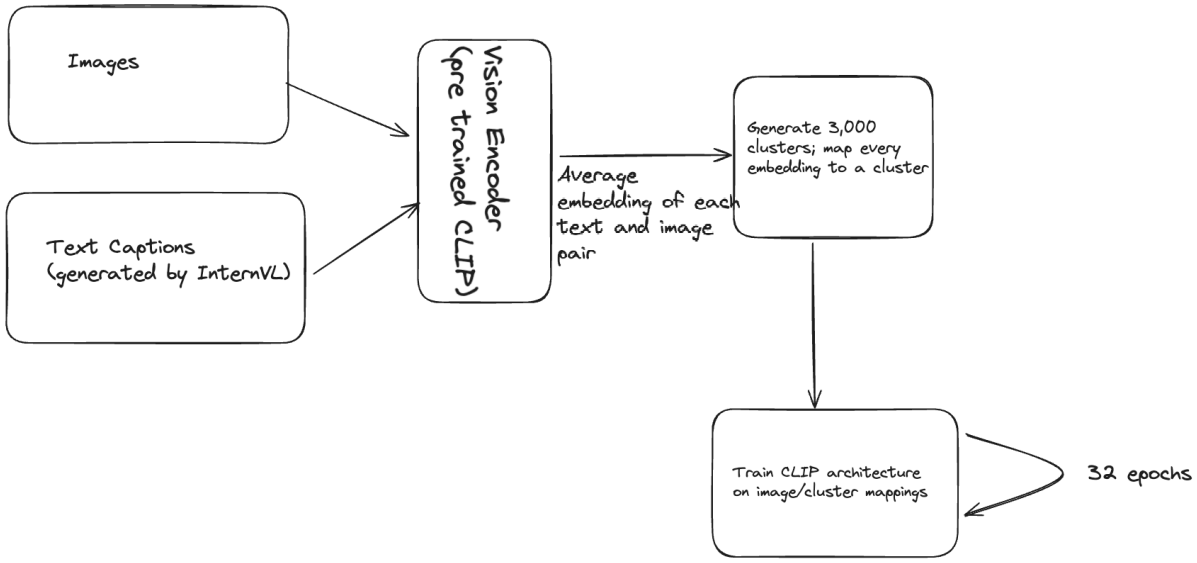
```
        config = Config()

    # Make config globally available as args
    global args
    args = config

    # Initialize distributed processing with a single process
    os.environ['MASTER_ADDR'] = 'localhost'
    os.environ['MASTER_PORT'] = '12355'
    distributed.init_process_group(backend='nccl', world_size=1, rank=0)


    # Load model and initialize
    model, transform_clip = clip.load(args.model_name)
    # Use only the vision encoder part of CLIP
    model = model.visual
    model = model.float()  # Convert to full precision because that's what
the rest of the unicom seems to want

    model = WarpModule(model)
    model.train()
    model.cuda()

    # Create dataset and loader
    dataset_train = CustomClusteredDataset(
        cluster_results_path='clustering_results.csv',
        transform=transform_clip
    )

    loader_train = DataLoader(
        dataset_train,
        batch_size=args.batch_size,
        num_workers=args.num_workers,
        pin_memory=True,
        drop_last=True,
        shuffle=True
    )

    backbone = model

    margin_loss = CombinedMarginLoss(
        args.margin_loss_s,
        args.margin_loss_m1,
```

```
        args.margin_loss_m2,
        args.margin_loss_m3,
        args.margin_loss_filter
    )

    module_partial_fc = PartialFC_V2(
        margin_loss,
        args.output_dim,
        dataset_train.num_classes,
        args.sample_rate,
        False,
        sample_num_feat=args.num_feat,
#         use_distributed=False
    )
    module_partial_fc.train().cuda()

    opt = torch.optim.AdamW(
        params=[
            {"params": backbone.parameters()},
            {"params": module_partial_fc.parameters(), "lr": args.lr *
args.lr_pfc_weight}
        ],
        lr=args.lr,
        weight_decay=args.weight_decay
    )

    steps_per_epoch = len(dataset_train) // args.batch_size + 1
    lr_scheduler = optim.lr_scheduler.OneCycleLR(
        optimizer=opt,
        max_lr=[args.lr, args.lr * args.lr_pfc_weight],
        steps_per_epoch=steps_per_epoch,
        epochs=args.epochs,
        pct_start=0.1,
    )

    callback_func = SpeedCallBack(10, args.epochs * steps_per_epoch,
args.batch_size)
    auto_scaler =
torch.cuda.amp.grad_scaler.GradScaler(growth_interval=200)
    global_step = 0

    # Main training loop - removed train_sampler check
    for epoch in range(args.epochs):
```

```
        for _, (img, local_labels) in enumerate(loader_train):
            img = img.cuda()
            local_labels = local_labels.long().cuda()

            with torch.cuda.amp.autocast(False):
                local_embeddings = backbone(img)
            local_embeddings.float()

            loss = module_partial_fc(local_embeddings, local_labels)
            auto_scaler.scale(loss).backward()

            if global_step % args.gradient_acc == 0:
                auto_scaler.step(opt)
                auto_scaler.update()
                opt.zero_grad()

            lr_scheduler.step()
            global_step += 1

            with torch.no_grad():
                callback_func(
                    lr_scheduler,
                    float(loss),
                    global_step,
                    auto_scaler.get_scale()
                )

        print(f"Completed epoch {epoch}")

    print("Training completed")
```

Otherwise, all the code provided by Unicom basically remained the same.

Flow Charts/Diagrams



Documentation

No documentation to add really just yet

Results Visualization + Proof of Work

The results output clearly shows minimization of the loss-function output (see the **bolded** output below), which is a good sign!

```
rank:73 total:73 lr:[0.00000040][0.00000402] step:20 amp:65536
required:2.9 hours 18.461
rank:82 total:82 lr:[0.00000040][0.00000405] step:30 amp:65536
required:2.7 hours 18.532
rank:73 total:73 lr:[0.00000041][0.00000409] step:40 amp:65536
required:2.7 hours 18.396
rank:64 total:64 lr:[0.00000041][0.00000414] step:50 amp:65536
required:2.7 hours 18.344
.
```

.
.
```
rank:86 total:86 lr:[0.00000000][0.00000000] step:20620
amp:66461399789245793645190353014017228 8 required:0.0 hours 0.001
rank:85 total:85 lr:[0.00000000][0.00000000] step:20630
amp:66461399789245793645190353014017228 8 required:0.0 hours 0.001
rank:93 total:93 lr:[0.00000000][0.00000000] step:20640
amp:66461399789245793645190353014017228 8 required:0.0 hours 0.001
```

One thing that was kind of dumb on my part was that I forgot to actually save the generated model weights (oops) after running the 32-epoch, multi-hour training session. Consequently, I don't have time to do any performance benchmarking this week. Regardless, the decreasing loss output looks very promising!

Next Week's Proposal

- Understand the various "hyperparameters" better. For this week, I really just cargo-culted the default parameters that were in the Unicom code.
- Re-run the training script – but save the model weights!
- Performance benchmarking – I want to compare this unicom-trained VLM4Bio-based model against an out-of-the-box pretrained CLIP model on a few bio-related datasets.