Contents

Explanation of Current Code Implementation **1**

Goals 1

Implementation 1

Environment 1

Dataset 2

Model(s) 2

Training 7

Evaluation 7

Results 7

Explanation of Current Code Implementation

######

10/15/25

disclaimer: As of now, some of the models are only trained on the *ImageNet* dataset, with *WordNet* labels. A new document and/or notebook will be made for the plankton dataset. Further, the implementation details are not set in stone, and changes will be made

- `Goals`
- `Implementation`
  - `Dataset`
  - `Model(s)`
    - `Classifier`
    - `ArcFace`
    - `SphereMulti`
    - `SphereMultiHier`
    - `ArcFaceMultiHead`

Goals

The goals of the thesis are a bit fuzzy, and there are many interesting directions to pursue. However, it must be limited at some point, and the research questions formulated. Currently, the goals / questions are:

1. Produce a strong classifier on plankton image data, leveraging deep metric learning.
   - The classifier should outperform the other available methods, according to some metric. It is not yet clear if "accuracy" is the most relevant for what I am aiming to do.
   -
2. The classifier should respect the underlying hierarchy of the data
   - The model predictions should be in accordance with the structure of the taxonomic-tree that defines the dataset.
   - Given an unseen sample, the model should be able to place it *somewhere* in the hierarchy where it belongs, preferably as far down as possible.
3. Dynamic confidence selection
   - The user should be able to decide how confident the model should be to produce a prediction. This is helpful for image annotation, where we might care more that the labels we have are correct, and are willing to accept non-complete annotations (i.e. a difficult *cat* image can be classified as *mammal*, if the model is uncertain)

Implementation

Below, I walk through the current implementations. Some code-blocks are included to illustrate, but it is not complete. I refer to the relevant files where appropriate. A full notebook can (and should) be created at some point (once the pipeline is better defined).

Environment

To handle dependencies and version across my machines and the server (BIRGET), I use anaconda (mini-conda). This choice was made early on, where almost all the coursework suggested using conda for this. If time allows, I would like to change over to something less cumbersome, like uv with a `reqs.txt`.

Dataset

The tiny-*ImageNet* dataset is a downscaled subset of the original *ImageNet*, and consists of 200 classes, each with 500 samples. The version I use, is a modified version of the official one, implemented by TJMoon. This upscales the images from 64x64 to 224x224. It also pre-defines a train-test-val split, but this is modified in the code. To get the full hierarchical labels, we use WordNet in combination with the leaf labels given from ImageNet to create the underlying graph. Since the labels of the ImageNet images are **WordNet-IDs (wnids)**, we can get the set of *synonyms*, `synset` from each of them. From here, we can generate the set of paths from the **root** `synset` to each leaf:

```
for id in self.wordnet_ids:
    synset = wordnet.synset_from_pos_and_offset(id[0], int(id[1:]))
    paths = synset.hypernym_paths()
```

Where `hypernym` is the *superset* of a given label. (i.e. *mammal* is a **hypernym** of *cat*) More details about the WordNet functions can be found here.

From these paths, we can then generate a DiGraph that shows us the underlying relations between the labels:

ImageNet Graph

To handle the graph, I use `rustworx`, since `networkx` was slow. For pure taxonomical data (i.e. where the underlying structure is a *tree*, a good option is `treelib`). As we notice from the plot, the data is represented by a DiGraph. To better mimic the taxonomical tree structure, we convert the graph into a tree. This is done by selecting only *one* parent for each node, dropping the others. We then get the following representation:

ImageNet Tree

We can further simplify the problem, by removing redundant or **dead** nodes. Notice from the plot that some of the nodes contain no information. Take for instance the `appliance` node. It has a single parent `durables`, and a single child `home_appliance`. What this essentially means, is that if we "know" `durables` to be true, `home_appliance` must directly follow. Indeed, and node with out-degree 1, gives no information that could not be gathered from the other nodes (This is assuming a hierarchical conforming model, where the probability of a parent never can be lower than a child). All such *dead nodes* are removed, and we are left with:

ImageNet Tree Pruned

Now, this does influence the underlying structure, and we loose something, especially in the step from DAG to tree. However, this dataset is more to benchmark methods, and the **final** classifier will be applied to the *plankton* dataset.

Implementation details of the graph can be found in `Utils/graph.py`.

Model(s)

The current model(s) uses a pretrained `EfficientNetV2_S` as a feature-extractor, with `IMAGENET1K_V1` weights. The feature-extractor layers are frozen, meaning they never get updated during backpropagation. This choice is made in the interest of time, and the final model will likely be defined with un-frozen layers, either with `efficientnet_v2_s`, or `ResNet`. All model architectures have a corresponding `config.yaml` file where pre-defined parameters can be found.

Next, we define an `embedding block`, with updatable parameters:

```
class DevNet(nn.Module):
    def __init__(self, embedding_size) -> None:
        super().__init__()
        # https//discuss.pytorch.org/t/how-to-extract-features-of-an-image-from-a-trained-model/119/2?u=risha
        # https://discuss.pytorch.org/t/module-children-vs-module-modules/4551
        pretrained = efficientnet_v2_s(EfficientNet_V2_S_Weights.IMAGENET1K_V1)
        # freeze feature extractor, online this seems to be the move
        for param in pretrained.parameters():
            param.requires_grad=False
        self.feature_extractor = nn.Sequential(*list(pretrained.children())[0:-1]) # remove classifier

        self.embedding = nn.Sequential(
            nn.Flatten(), # 1280
            nn.Linear(1280, 1280),
            nn.ReLU(),
            #nn.Dropout(0.2),
            nn.Linear(1280, embedding_size),
            nn.ReLU(),
            nn.Dropout(0.2), # gets passed to model head(s)
        )

    def forward(self, x):
        features = self.feature_extractor(x)
        embeddings = self.embedding(features)
        return embeddings
```

[There is much to explore here, but this model is simple, and trains fast with decent results. Again, the final model will in all likelihood be different. The full implementation can be found in `Backbone/DevNet.py`]

Classifier The classifier is the simplest model, and acts like a standard classifier. It receives the embedding, passes it through a single linear layer `nn.Linear(embedding_size, num_classes)`, and returns the *leaf logits*. During training, it gets updated by CrossEntropyLoss.

ArcFace `ArcFace` is similar to Classifier, in the sense that it predicts over leaf labels. However, we are now in the realm of metric learning. In my implementation, I define the process of applying the margin and scale as its own model head, as opposed to a custom loss function. This is more readable to me, and seems to work well.

To briefly explain, ArcFace is a hyper-spherical metric learning model, that uses **Additive Angular Margin Loss**, originally designed to create more discriminative embeddings for facial recognition tasks. It is *hyper-spherical*, since the embeddings are normalized to lay on a unit hypersphere, and *additive angular margin* refers to how it modifies the softmax-loss function:

$$L_{AAM} = -\frac{1}{N}\sum_{i=1}^{N} \log \frac{e^{s\cdot\cos(\theta_{y_i,i}+m)}}{e^{s\cdot\cos(\theta_{y_i,i}+m)} + \sum_{j\neq i} e^{s\cdot\cos(\theta_{j,i})}}$$

Where θ is the angle between the feature x_i, and *ground-truth class center* W_{y_i}. The *margin,* m is added to the positive/true class, which effectively means that the model most be *m-closer* to the true label to be considered correct, forcing tight intra-class clustering.

The name **ArcFace** (I think) refers to the geodesic distance (shortest path on a curved surface) being called Arc. The loss is attractive, since it is easy to implement, gets strong results (at least on facial recognition tasks), and has a clear geometric interpretation.

A more comprehensive explanation of the mathematics will be present in the final draft, but for a nice intuitive explanation, I recommend this survey, and blogpost.

Then, the implementation looks like:

```
class ArcFace(nn.Module):
    def __init__(self, m, s, num_classes, embedding_size, graph):
        super().__init__()
        self.m = m
        self.s = s
        self.num_classes = num_classes
        self.embedding_size = embedding_size
        self.W = nn.Parameter(torch.FloatTensor(self.num_classes, self.embedding_size))
        self.graph = graph
        nn.init.kaiming_uniform_(self.W)

    # Follow the paper illustration
    # Inspired by:
    # https://github.com/zakariaelaoufi/Arcface-Pytorch
    # https://github.com/odinhg/ArcFace-and-Triplet-Network-INF368A
    # https://github.com/deepinsight/insightface/issues/2126
    def forward(self, x, labels):
        device = x.device
        x = F.normalize(x).to(device)
        W = F.normalize(self.W).to(device)
        # Cosine simmilarity: cos(\theta)
        cosine_theta = F.linear(x,W)#torch.matmul(x, W.t()).clamp(-1+1e-8, 1-1e-8)
        if labels is None:
            return self.s*cosine_theta
        # with 0 bias, and magnitude W,x = 1, cos(theta) = xW^T => theta = arccos(xW^T):
        theta = torch.acos(torch.clamp(cosine_theta,-1+1e-8,1-1e-8)) # clamp for numerical stability

        # https://github.com/deepinsight/insightface/issues/2126
        # from https://github.com/KevinMusgrave/pytorch-metric-learning/pull/539/commits/10f202ee90b876ad964d
        # limits theta + m <= 180! important to keep monotonically decreasing function!
        cosine_theta_m = torch.cos(theta+self.m)
        cosine_theta_m = torch.where(theta <= np.deg2rad(180)-self.m,
                                     cosine_theta_m,
                                     cosine_theta - self.m*np.sin(self.m))
        # currently, I handle cases where there is no positive label as -1.
        # This causes error with F.one_hot.
        # A bad but functional solution for now:
        one_hot = labels

        logits =((one_hot * cosine_theta_m )+ ((1 - one_hot) * cosine_theta))
        logits *= self.s
        return logits
```

Here, `self.W` represents the class centers. As we can see, if no labels are passed to the module, we return the unscaled logits (no added margin), since we are at inference time. In the forward pass, we normalize both the embedding `x` and class-centers `self.W`. In addition, we have a check for extreme cases, where the angle between the class-center and embedding is >= 180. In these cases, the addition of a margin m $>$ 0 would actually **decrease** the loss (cosine is a periodic function [-1,1] with minima cos(π)). This should *almost* never happen, but depending on the scale and margin parameter, it can happen. This ensures that the model does not collapse.

Since I have implemented the ArcFace as a module head, the scaled logits with margins are then again passed to CrossEntropyLoss during training.

SphereMulti `SphereMulti` is an implementation of the Hyperspherical Multilabeled Classification paper, and differs from the previous two since we now predict ***all*** the labels of the hierarchy. This is done by representing the full path of the image as a multi-hot vector. Since there now are more than one true label, LAA-loss is no longer suitable. Luckily for us, in the referenced paper, the authors define a normalized hyperspherical loss that modifies BinaryCrossEntropy. The definitions are slightly different, but it is still easy to implement. In short, the loss is defined BCE:

$$L_{BCE} = -(\sum_{i \in Y}^{N} \log(p_i) + \sum_{i \notin Y}^{N} \log(1 - p_i))$$

Where the probability p_i now is modified to be:

$$\sigma(s \cdot \cos(\arccos(\theta) + m))$$

The strength of this method is that it handles the multi labeled case, while remaining hyperspherical. The python implementation looks like:

```
class SphereMulti(nn.Module):
    # Based on:
    # https://www.ecva.net/papers/eccv_2022/papers_ECCV/papers/136850038.pdf
    # NOTE: same backbone+head for "normal" multi-label classification,
    # and multi-label classification w/ respect to the hierarchy. See trainer.py for
    # implementation details of training loop and loss calculation.
    def __init__(self, s, m, num_classes, embedding_size, graph) -> None:
        super().__init__()
        self.s = s
        self.m = m
        self.num_classes = num_classes
        self.embedding_size = embedding_size
        self.W = nn.Parameter(torch.Tensor(self.num_classes, self.embedding_size))
        nn.init.xavier_normal_(self.W)
        self.graph = graph

    def forward(self, x, labels):
        device = x.device
        x = F.normalize(x)
        W = F.normalize(self.W).to(device)
        arc = torch.arccos(x @ W.t()) + self.m
        cos = torch.cos(arc)
        logits = self.s * cos
        return logits
```

One issue with `BCE`in general, is the inherent imbalance in positive and negative labels. In most cases, an image will contain only a few true positives, and the rest remain negative. This means that as the number of labels/samples grow, the loss gets artificially low, and slows down training. To mitigate this, we can change to `FocalLoss`, which is an extension of `BCE`, or as we do, `AsymmetricLoss`, wich further extends `FocalLoss`

The logits are then passed on to `BCEWithLogitsLoss` or `AsymmetricLoss`.

SphereMultiHier `SphereMultiHier` is an extension of SphereMulti, where we enforce hierarchical conformity. The network itself is identical to SphereMulti, but we modify the loss to attempt to include the hierarchy in a meaningful way. I have experimented with different ways to enforce hierarchy, but the current version with the best results is a simple extension, based on this paper. The loss works by calculating the BCE-loss, and adding a penalty for each violation. By **violation**, I am referring to cases where the model predicts / sigmoid output of a child node is 1, and its parent is 0. This generates an "impossible" label, in the sense that there is no cats that is not also a mammal. This is done as a local check, where for each node, we check if this property is violated, and punish accordingly. Further, I add a penalty based on the graph distance from the real label, to the predicted one. This is only applied to the lowest prediction. The final loss is then the combination of these:

```
# https://arxiv.org/pdf/2502.03591
# https://github.com/the-mercury/CIHMLC
# In short, add a penalty term to BCE, s.t. predictions
# that immidieatly violate the hierarchy gets punished.
# two approaches presented; fixed and data driven.
class hbce_loss(nn.Module):
    def __init__(
        self,
        graph,
        lineage_labels,
        config,
        reduction="mean",
        pos_weight=None,
        device=None,
    ) -> None:
        super().__init__()
        self.graph = graph
        self.lineage_labels = lineage_labels
        self.penalties = config["loss"]["hbce"]["penalties"]
        self.hier_penalty = config["loss"]["hbce"]["hier_penalty"]
        self.penalty_scale = config["loss"]["hbce"]["scale"]  # penalty scale
        self.threshold = config["model"]["threshold"]  # threshold t for predictions
        #self.reachable = reachable
        # BCE DOCS:
        # https://discuss.pytorch.org/t/anomaly-in-binary-cross-entropy-loss-for-batches-and-using-weights-to
        # NOTE: Try with graph level weights
        self.bce = nn.BCEWithLogitsLoss(
            reduction=reduction, weight=self.graph.level_weights.to(device),
        )  # HBCE = BCE + penalty


    def forward(self, logits, y_true):
        bce = self.bce(logits, y_true.float())
        probas = torch.sigmoid(logits)  # (B, num_labels)
        y_pred = torch.where(probas > self.threshold, 1, 0) #(probas > self.threshold).long()
        missclassified_mask = torch.where((y_pred != y_true), 1, 0)
        missclassified_indecies = torch.nonzero(missclassified_mask)

        G = self.graph.rxG
        penalties = 0.0  # calculate all hier penalties for batch

        for parent_node in G.nodes():
            if parent_node.is_root: continue # root is removed from pred (always 1)
            children = G.successors(parent_node.index)
            parent_idx = self.lineage_labels.index(parent_node.name)
            parent_pred = y_pred[:,parent_idx]
            children_idxs = [self.lineage_labels.index(child.name) for child in children]
            for child_idx in children_idxs:
                child_pred = y_pred[:,child_idx]
                penalty = torch.where((parent_pred == 0)&(child_pred == 1),
                                      self.hier_penalty *torch.ones_like(child_pred),
                                      torch.zeros_like(child_pred))
                penalties+= penalty
        penalties += self.get_distance_penalty(y_pred, y_true)
        loss = bce + penalties
        return loss.mean()

    def get_distance_penalty(self,y_pred, y_true):
        pred_idxs = torch.nonzero(y_pred, as_tuple=True)
        pred_level = torch.zeros(y_pred.shape, dtype=torch.float)
        pred_level[pred_idxs] = torch.tensor([self.graph.rxG[i].level for i in pred_idxs[1]], dtype=torch.flo
        lowest_pred_vals, lowest_pred_idxs = torch.max(pred_level, dim=1)

        true_idxs = torch.nonzero(y_true,as_tuple=True)
        true_level = torch.zeros(y_true.shape,dtype=torch.float)
        true_level[true_idxs] = torch.tensor([self.graph.rxG[i].level for i in true_idxs[1]], dtype=torch.flo
        lowest_true_vals, lowest_true_idxs = torch.max(true_level, dim=1)

        distance_to_pred = torch.tensor([self.graph.distance(lowest_pred_idxs[i], lowest_true_idxs[i]) for i

        return distance_to_pred.mean()
```

The implementation is not “clever” and could probably be optimized. As is, it is not too slow, so I have not bothered with it.

Note I have not yet tried asymmetric BCE loss for SphereMulti and SphereMultiHier, but this could potentially push the performance of these model higher! This will be implemented together with other todos.

ArcFaceMultiHead The latest version I am working on, is a multi headed version of the ArcFace implementation. Here, I split the dataset by the level they appear in the graph. Each head is then responsible for predicting a subset of the full labels. Since the WordNet labels differ in how specific they are, and the path from root to leaf differs, I decided to “move” all leafs to the same level.

Note:

Not (yet?) implemented

Hierarchical Attention One idea, not yet imp

Training

Setup

S

Evaluation

Results

The models are compared in `model_comparison.ipynb`. Not fully up to date, due to some server issue with hosting the notebook (once solved, this will be updated)