Few-Shot Classifier

Overview

Few-shot classification tackles the challenge of classifying images with limited labeled data. In this scenario, we have a labeled support set containing a small number of examples per class (typically less than 10) and a query set of unseen images for which we want to predict labels. The model leverages the information from the support set to classify the query images effectively.

Metric-Based Approach: Prototypical Networks

Most few-shot classification methods employ a metric-based approach. Here's a breakdown of the process:

1. **Feature Extraction:** A Convolutional Neural Network (CNN) acts as a feature extractor. It takes an image as input and outputs a representation (embedding) in a specific feature space. The goal is for the CNN to embed images from the same class close together in this space, even for unseen classes.
2. **Prototype Computation:** Prototypical Networks compute a prototype for each class in the support set. This prototype is the mean of all embedding vectors corresponding to the support images of that class.
3. **Classification by Nearest Neighbor:** During classification, each query image is compared to the prototypes using a distance metric (like Euclidean distance). The query image is assigned the label of the closest prototype in the feature space.

Core Techniques

Most methods rely on metrics to compare image features. Here's the two-step process:

1. **Feature Extraction with CNNs:** A convolutional neural network (CNN) takes both support and query images and converts them into numerical representations in a feature space. Think of this space as a way to organize images based on their similarities. The key here is for the CNN to learn how to represent similar classes close together in this space, even for unseen classes.
2. **Similarity Comparison:** This is where Prototypical Networks come in. They calculate a prototype (average embedding) for each class in the support set. Then, each query image is assigned the label of the closest prototype in the feature space, based on distance (e.g., Euclidean distance)

Problems Faced

There was an 'AttributeError' in the training part. It's because the label method in train_set is incorrectly used right before the TaskSampler.

Things Learned :

- What a Prototypical Network is and how to implement one in 15 lines of code.
- How to use Omniglot to evaluate few-shot models
- How to use custom PyTorch objets to sample batches in the shape of a few-shot classification tasks.
- How to use meta-learning to train a few-shot algorithm

Process Overview

a. installing libraries: Pytorch

- a. nn -Neural Networks and optim- optimization algorithms like SGD or Adam
- b. Dataloader to combine dataset and sampler to provide iteration of dataset
- c. Transforms and torchvision- suites for image processing
  - i. Omniglot- for few-shot learning
  - ii. Resnet18- pre trained model on ImageNet data for Image Classification
  - iii. Tqdm- library that provides progress bar for loops
  - iv. Easy Fsl- tasksampler; custom library for few-shot learning
    - a. Plot_images- for visualization
    - b. Sliding_average- for smoothing out metrics over time

b. importing dataset: omniglot part of torchvision package, a popular MNIST-like benchmark for few-shot classification. It contains 1623 characters from 50 different alphabets. Each character has been written by 20 different people.

c. Segregating Test Set and Training Set from the Fetched Dataset

d. We initiate PrototypicalNetworks with a backbone. This is the feature extractor we were talking about. Here, we use as backbone a ResNet18 pre-trained on ImageNet, with its head chopped off and replaced by a Flatten layer. The output of the backbone, for an input image, will be a 512-dimensional feature vector.

e. The forward method doesn't only take one input tensor

h. in order to predict the labels of query images, we also need support images and labels as inputs of the model.

f. Here we create a data loader that will feed few-shot classification tasks to our model. But a regular PyTorch dataloader will feed batches of images, with no consideration for their label or whether they are supported or query. We need 2 specific features in our case.

1. We need images evenly distributed between a given number of classes.
2. We need them split between support and query sets.

For the first point, I wrote a custom sampler: it first samples n_way classes from the dataset, then it samples n_shot + n_query images for each class (for a total of n_way * (n_shot + n_query) images in each batch). For the second point, I have a custom collate function to replace the built-in PyTorch collate_fn . This baby feed each batch as the combination of 5 items:

1. support images
2. support labels between 0 and n_way
3. query images
4. query labels between 0 and n_way
5. a mapping of each label in range(n_way) to its true class id in the dataset (it's not used by the model but it's very useful for us to know what the true class is)

You can see that in PyTorch, a DataLoader is basically the combination of a sampler, a dataset and a collate function (and some multiprocessing voodoo): sampler says which items to fetch, the dataset says how to fetch them, and the collate function says how to present these items together. If you want to dive into these custom objects, they're here.

g. We created a data loader that will feed us with 5-way 5-shot tasks (the most common setting in the literature). Now, as every data scientist should do before launching opaque training scripts, let's take a look at our dataset.

h. With absolutely zero training on Omniglot images, and only 5 examples per class, we achieve around 86% accuracy! Isn't this a great start? Now that you know how to make Prototypical Networks work, you can see what happens if you tweak it a little bit (change the backbone, use other distances than euclidean...) or if you change the problem (more classes in each task, less or more examples in the support set, maybe even one example only, but keep in mind that in that case Prototypical Networks are just standard nearest neighbor). When you're done, you can scroll further down and learn how to meta-train this model, to get even better results.

i. Training Meta-Learning Algorithm: Let's use the "background" images of Omniglot as a training set. Here we prepare a data loader of 40 000 few-shot classification tasks on which we will train our model. The alphabets used in the training set are entirely separated from those used in the testing set. This guarantees that at test time, the model will have to classify characters that were not seen during training. Note that we don't set a validation set here to keep this notebook concise, but keep in mind that this is not good practice and you should always use validation when training a model for production.
ii. We will keep the same model. So our weights will be pre-trained on ImageNet. If you want to start a training from scratch, feel free to set pretrained=False in the definition of the ResNet. Here we define our loss and our optimizer (cross entropy and Adam, pretty standard), and a fit method. This method takes a classification task as input (support set and query set). It predicts the labels of the query set based on the information from the support set; then it compares the predicted labels to ground truth query labels, and this gives us a loss value. Then it uses this loss to update the parameters of the model. This is a meta-training loop.
iii. To train the model, we are just going to iterate over a large number of randomly generated few-shot classification tasks, and let the fit method update our model after each task. This is called episodic training. This took me 20mn on an RTX 2080 and I promised you that this whole tutorial would take 15mn. So if you don't want to run the training yourself, you can just skip the training and load the model that I trained using the exact same code.

Conclusion

Around 98.26%!

It's not surprising that the model performs better after being further trained on Omniglot images than it was with its ImageNet-based parameters. However, we have to keep in mind that the classes on which we just evaluated our model were still **not seen during training**, so 99% (with a 12% improvement over the model trained on ImageNet) seems like a decent performance.

Souradeep De

01st April 2024