!pip install easyfsl

Installs Easy Few-Shot Learning repository.

import torch

from torch import nn, optim

from torch.utils.data import DataLoader

from torchvision import transforms

from torchvision.datasets import Omniglot

from torchvision.models import resnet18

from tqdm import tqdm

from easyfsl.samplers import TaskSampler

from easyfsl.utils import plot\_images, sliding\_average

Imports Neural Network [ nn ] and Optimization [ optim ] module from pytorch library for easy access neural network functionality and optimization algorithms.

Imports "transforms" module from "torchvision" for image transformation or data augmentation.

To handle Dataset, "DataLoader" was imported from "torch.utils.data" library.

For the baseline CNN model, "resnet18" model which is pre-trained on ImageNet dataset was imported.

Omniglot dataset is being used in the experiment.

Importing "TaskSampler" to create Meta tasks with support and query datasets.

To convert a Tensor array to Images "plot\_images" & to calculate loss functions "sliding\_average " is imported from "easyfsl.utils".

image\_size = 28

train\_set = Omniglot(

root="./data",

background=True,

transform=transforms.Compose(

[

transforms.Grayscale(num\_output\_channels=3),

transforms.RandomResizedCrop(image\_size),

transforms.RandomHorizontalFlip(),

transforms.ToTensor(),

]

),

download=True,

)

test\_set = Omniglot(

root="./data",

background=False,

transform=transforms.Compose(

[

# Omniglot images have 1 channel, but our model will expect 3-channel images

transforms.Grayscale(num\_output\_channels=3),

transforms.Resize([int(image\_size \* 1.15), int(image\_size \* 1.15)]),

transforms.CenterCrop(image\_size),

transforms.ToTensor(),

]

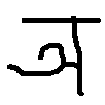
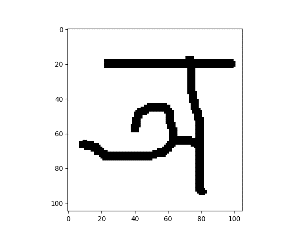
),

download=True,

)

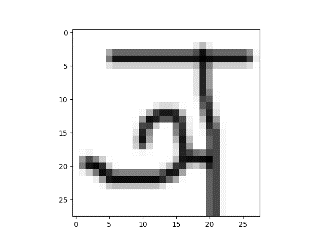
In the above code the train and test part of the Omniglot dataset is downloaded and data augmentation is done to improve model performance. The code converts the images of Omniglot from grayscale to RGB format, resizes them to 28x28 pixels, and randomly flips them horizontally. Finally, with the "transforms.ToTensor()" function images are converted to a Tensor array.

* transforms.Grayscale(num\_output\_channels=3)

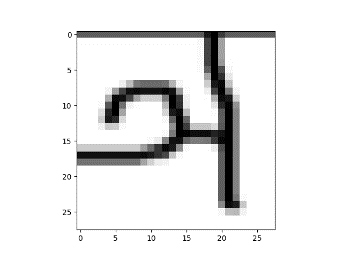
* transforms.RandomResizedCrop(image\_size),

![A black symbol with a curved line

Description automatically generated with medium confidence](data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAGkAAABpAQAAAAAR+TCXAAAAB3RJTUUH3woUEi86ccVK8gAAALlJREFUOI3l0jsOwyAMBmBHGRi9duMoPRqpeoBeid6EI2TMgHCBFMivllaZqirePoGMH5BsY6LDMPB7eirBSLWHvHlXYRn/yPEjr3t46ZNY7o2exi1npKVBbONEJM5UkrFAcTI3KmDIPFfyKxegafQcE/WpYyLg0meML6QTklbegKFMq1AnukrOwzP2SZWHV/q1abF+KP36lMtVyvrRkVzpEnVlrCBurHLJhwIdHZAayUCPDHg5xc/4ADxvzA8GFfbuAAAAAElFTkSuQmCC) 

* transforms.RandomHorizontalFlip(),

![A black symbol with a curved line

Description automatically generated with medium confidence](data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAGkAAABpAQAAAAAR+TCXAAAAB3RJTUUH3woUEi86ccVK8gAAALlJREFUOI3l0jsOwyAMBmBHGRi9duMoPRqpeoBeid6EI2TMgHCBFMivllaZqirePoGMH5BsY6LDMPB7eirBSLWHvHlXYRn/yPEjr3t46ZNY7o2exi1npKVBbONEJM5UkrFAcTI3KmDIPFfyKxegafQcE/WpYyLg0meML6QTklbegKFMq1AnukrOwzP2SZWHV/q1abF+KP36lMtVyvrRkVzpEnVlrCBurHLJhwIdHZAayUCPDHg5xc/4ADxvzA8GFfbuAAAAAElFTkSuQmCC) 

* transforms.ToTensor(),

A black and white image of a number

Description automatically generated

tensor([[[0.3686, 0.3686, 0.3686, ..., 0.3686, 0.3686, 0.3686],

[0.9961, 0.9961, 0.9961, ..., 0.9961, 0.9961, 0.9961],

[1.0000, 1.0000, 1.0000, ..., 1.0000, 1.0000, 1.0000],

...,

[1.0000, 1.0000, 1.0000, ..., 1.0000, 1.0000, 1.0000],

[1.0000, 1.0000, 1.0000, ..., 1.0000, 1.0000, 1.0000],

[1.0000, 1.0000, 1.0000, ..., 1.0000, 1.0000, 1.0000]],

[[0.3686, 0.3686, 0.3686, ..., 0.3686, 0.3686, 0.3686],

[0.9961, 0.9961, 0.9961, ..., 0.9961, 0.9961, 0.9961],

[1.0000, 1.0000, 1.0000, ..., 1.0000, 1.0000, 1.0000],

...,

[1.0000, 1.0000, 1.0000, ..., 1.0000, 1.0000, 1.0000],

[1.0000, 1.0000, 1.0000, ..., 1.0000, 1.0000, 1.0000],

[1.0000, 1.0000, 1.0000, ..., 1.0000, 1.0000, 1.0000]],

[[0.3686, 0.3686, 0.3686, ..., 0.3686, 0.3686, 0.3686],

[0.9961, 0.9961, 0.9961, ..., 0.9961, 0.9961, 0.9961],

[1.0000, 1.0000, 1.0000, ..., 1.0000, 1.0000, 1.0000],

...,

[1.0000, 1.0000, 1.0000, ..., 1.0000, 1.0000, 1.0000],

[1.0000, 1.0000, 1.0000, ..., 1.0000, 1.0000, 1.0000],

[1.0000, 1.0000, 1.0000, ..., 1.0000, 1.0000, 1.0000]]])

class PrototypicalNetworks(nn.Module):

def \_\_init\_\_(self, backbone: nn.Module):

super(PrototypicalNetworks, self).\_\_init\_\_()

self.backbone = backbone

def forward(

self,

support\_images: torch.Tensor,

support\_labels: torch.Tensor,

query\_images: torch.Tensor,

) -> torch.Tensor:

"""

Predict query labels using labeled support images.

"""

# Extract the features of support and query images

z\_support = self.backbone.forward(support\_images)

z\_query = self.backbone.forward(query\_images)

# Infer the number of different classes from the labels of the support set

n\_way = len(torch.unique(support\_labels))

# Prototype i is the mean of all instances of features corresponding to labels == i

z\_proto = torch.cat(

[

z\_support[torch.nonzero(support\_labels == label)].mean(0)

for label in range(n\_way)

]

)

# Compute the euclidean distance from queries to prototypes

dists = torch.cdist(z\_query, z\_proto)

# And here is the super complicated operation to transform those distances into classification scores!

scores = -dists

return scores

convolutional\_network = resnet18(pretrained=True)

convolutional\_network.fc = nn.Flatten()

print(convolutional\_network)

model = PrototypicalNetworks(convolutional\_network).cuda()

The above code defines a pytorch neural network model called '**PrototypicalNetworks**'. It takes a '**backbone'** as an argument. In this case '**resnet18**' pretrained model is used without its fully connected layers which is done by the "**nn.Flatten()**" function.

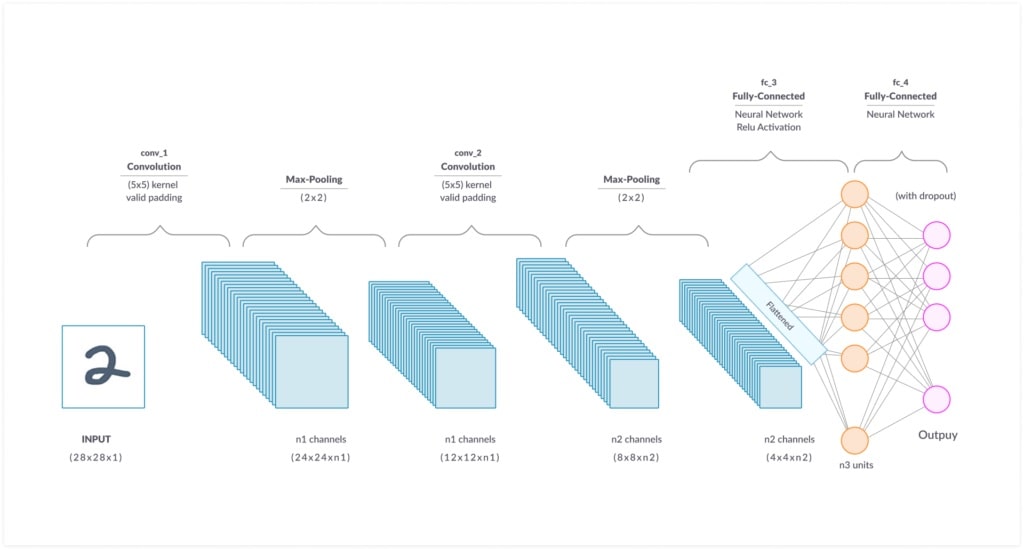
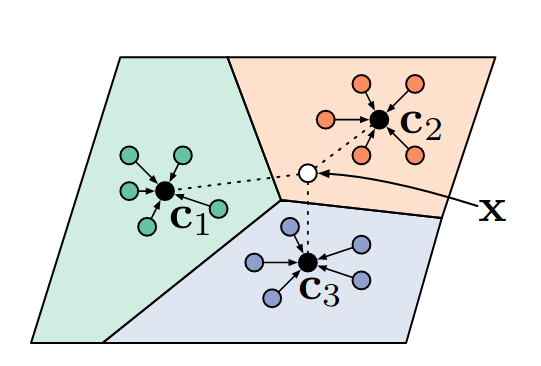


Image Features are extracted from support and query set using the '**resnet18**' network and stored in '**z\_support**' and '**z\_query**'tensors.

Number of Unique classes '**n\_way**' is calculated from the unique labels in '**support\_labels**'.

Prototypes are generated for each class by calculating the mean of all feature vectors that belong to that class in the support set. These prototypes are concatenated into 'z\_proto' tensor.

Using the negative Euclidean distance "**torch.cdist()**" from queries '**z\_query**' to prototypes '**z\_proto**' we calculate the similarity score.



N\_WAY = 5 # Number of classes in a task

N\_SHOT = 5 # Number of images per class in the support set

N\_QUERY = 10 # Number of images per class in the query set

N\_EVALUATION\_TASKS = 100

# The sampler needs a dataset with a "get\_labels" method. Check the code if you have any doubt!

test\_set.get\_labels = lambda: [

instance[1] for instance in test\_set.\_flat\_character\_images

]

test\_sampler = TaskSampler(

test\_set, n\_way=N\_WAY, n\_shot=N\_SHOT, n\_query=N\_QUERY, n\_tasks=N\_EVALUATION\_TASKS

)

test\_loader = DataLoader(

test\_set,

batch\_sampler=test\_sampler,

num\_workers=12,

pin\_memory=True,

collate\_fn=test\_sampler.episodic\_collate\_fn,

)

(

example\_support\_images,

example\_support\_labels,

example\_query\_images,

example\_query\_labels,

example\_class\_ids,

) = next(iter(test\_loader))

plot\_images(example\_support\_images, "support images", images\_per\_row=N\_SHOT)

plot\_images(example\_query\_images, "query images", images\_per\_row=N\_QUERY)

model.eval()

example\_scores = model(

example\_support\_images.cuda(),

example\_support\_labels.cuda(),

example\_query\_images.cuda(),

).detach()

\_, example\_predicted\_labels = torch.max(example\_scores.data, 1)

print("Ground Truth / Predicted")

for i in range(len(example\_query\_labels)):

print(

f"{test\_set.\_characters[example\_class\_ids[example\_query\_labels[i]]]} / {test\_set.\_characters[example\_class\_ids[example\_predicted\_labels[i]]]}"

)

The above code set's task to 5-Way 5-Shot. Number of images per query set is set to 10. For initial sample evaluation we do 100 tasks.

We use the '**TaskSampler'** module from **'easyfsl'** to create '**test\_sampler**' which sample images for Few-shot learning tasks on the test data set.

We then create '**test\_loader**' using **'DataLoader'** from **'torch'** which takes the following parameters 'test\_set' as 'dataset', 'test\_sampler' as 'batch\_sampler', '12' as 'num\_workers' which defines how many subprocesses to use for data loading, 'pin\_memory' is set to 'True' for CUDA acceleration, 'test\_sampler.episodic\_collate\_fn' as 'collate\_fn'. **episodic\_collate\_fn** is a method defined in **easyfsl**.

'**next(iter(test\_loader))**' is used to fetch next batch of tasks from '**test\_loader**'. And visualized using '**plot\_images**'.

We then set the model to evaluation mode with '**model.eval( )**' so that it won't update its weights during forward passes in evaluation.

We compute classification scores for the query images based on the support images and stored into '**example\_score**'. Then computation graph is detached from '**example\_scores**' using '**example\_scores.detach( )**'.

'**\_, example\_predicted\_labels = torch.max(example\_scores.data, 1)**' is used to find the index of the maximum score along dimension 1 (which corresponds to class scores). which effectively gives the predicted class labels for the query images, and stored in '**example\_predicted\_labels**' as predicted results.

Then the code prints out the Ground Truth vs Predicted Truth.

def evaluate\_on\_one\_task(

support\_images: torch.Tensor,

support\_labels: torch.Tensor,

query\_images: torch.Tensor,

query\_labels: torch.Tensor,

) -> [int, int]:

"""

Returns the number of correct predictions of query labels, and the total number of predictions.

"""

return (

torch.max(

model(support\_images.cuda(), support\_labels.cuda(), query\_images.cuda())

.detach()

.data,

1,

)[1]

== query\_labels.cuda()

).sum().item(), len(query\_labels)

def evaluate(data\_loader: DataLoader):

# We'll count everything and compute the ratio at the end

total\_predictions = 0

correct\_predictions = 0

model.eval()

with torch.no\_grad():

for episode\_index, (

support\_images,

support\_labels,

query\_images,

query\_labels,

class\_ids,

) in tqdm(enumerate(data\_loader), total=len(data\_loader)):

correct, total = evaluate\_on\_one\_task(

support\_images, support\_labels, query\_images, query\_labels

)

total\_predictions += total

correct\_predictions += correct

print(

f"Model tested on {len(data\_loader)} tasks. Accuracy: {(100 \* correct\_predictions/total\_predictions):.2f}%")

evaluate(test\_loader)

The above code performs the same evaluation but overall Omniglot data set.

N\_TRAINING\_EPISODES = 40000

N\_VALIDATION\_TASKS = 100

train\_set.get\_labels = lambda: [instance[1] for instance in train\_set.\_flat\_character\_images]

train\_sampler = TaskSampler(

train\_set, n\_way=N\_WAY, n\_shot=N\_SHOT, n\_query=N\_QUERY, n\_tasks=N\_TRAINING\_EPISODES

)

train\_loader = DataLoader(

train\_set,

batch\_sampler=train\_sampler,

num\_workers=12,

pin\_memory=True,

collate\_fn=train\_sampler.episodic\_collate\_fn,

)

We create 40k training tasks from the training dataset.

We make the '**train\_loader'** using '**DataLoader'** from Torch where **'Train\_Sampler'** is created for image sampling using **'TaskSampler'** from easyfsl & the **episodic\_collate\_fn** is also used here for collate function.

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters(), lr=0.001)

def fit(

support\_images: torch.Tensor,

support\_labels: torch.Tensor,

query\_images: torch.Tensor,

query\_labels: torch.Tensor,

) -> float:

optimizer.zero\_grad()

classification\_scores = model(

support\_images.cuda(), support\_labels.cuda(), query\_images.cuda()

)

loss = criterion(classification\_scores, query\_labels.cuda())

loss.backward()

optimizer.step()

return loss.item()

We choose **CrossEntropyLoss()** from **nn** module as the loss function & **Adam optimizer** as an optimization algorithm for the model with a learning rate of 0.001**.**

The fit function calculates the loss, computes gradients, and updates the model's parameters using the Adam optimizer.

log\_update\_frequency = 10

all\_loss = []

model.train()

with tqdm(enumerate(train\_loader), total=len(train\_loader)) as tqdm\_train:

for episode\_index, (

support\_images,

support\_labels,

query\_images,

query\_labels,

\_,

) in tqdm\_train:

loss\_value = fit(support\_images, support\_labels, query\_images, query\_labels)

all\_loss.append(loss\_value)

if episode\_index % log\_update\_frequency == 0:

tqdm\_train.set\_postfix(loss=sliding\_average(all\_loss, log\_update\_frequency))

The code performs training for few-shot learning model in an episodic manner. Each episode corresponds to a batch data from the **train\_loader.** It keeps track of the training loss.