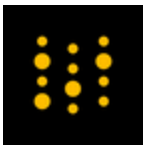
Transfer Learning Using PyTorch Lightning

wandb.ai/wandb/wandb-lightning/reports/Transfer-Learning-Using-PyTorch-Lightning--VmlldzoyODk2MjA

In this article, we have a brief introduction to transfer learning using PyTorch Lightning, building on the image classification example from a previous article.

Ayush Thakur

Last Updated: Nov 17, 2022


Ayush Thakur

4 comments

Share

5 hearts

Back to Fully Connected

Login to comment

Open in Colab

In the previous article, we built an image classification pipeline using PyTorch Lightning. In this article, we'll extend the pipeline to perform transfer learning with PyTorch Lightning.

Table of Contents

Introduction

Transfer Learning is a technique where the knowledge learned while training a model for "task" A can be used for "task" B. Here A and B can be the same deep learning tasks but on a different dataset.

Why?

The first few hidden layers of a deep neural network model learn general abstract features about the dataset. The later layers have task-specific knowledge. These learned features make for a good weight initialization.

Training neural networks from scratch can be expensive. Transfer learning has worked wonderfully so far by reducing the number of training hours and increasing the model's accuracy.

Usually, the dataset does not have many samples to learn from. Even data augmentation can push the accuracy so far. Transfer learning can come to the rescue.

How?

The most common workflow to use transfer learning in the context of deep learning is:

Take layers from a previously trained model. Usually, these models are trained on a large dataset.

Freeze them to avoid destroying any of the information they contain during future training rounds.

Add some new, trainable layers on top of the frozen layers. These new features will learn task-specific features. Train the new layers on your dataset.

An optional step is fine-tuning, which consists of unfreezing the entire model you obtained above and re-training it on the new data with a very low learning rate. The entire model can be unfrozen partially or in parts(unfreeze a few and train and so on).

This report requires some familiarity with PyTorch Lightning for the image classification task. You can check out my previous post on Image Classification using PyTorch Lightning to get started. Let us train a model with and without transfer learning on the Stanford Cars dataset and compare the results using Weights and Biases.

The Dataset

We will be using the Stanford Cars dataset to train our image classifier. The Cars dataset contains 16,185 images of 196 classes of cars. The data is split into 8,144 training images and 8,041 testing images, where each class has been split roughly in a 50-50 split. Classes are typically at the level of Make, Model, and Year, e.g. 2012 Tesla Model S or 2012 BMW M3 coupe.

With PyTorch Lighting's DataModule, one can define the download logic, preprocessing steps, augmentation policies, etc., in one class. It organizes the data pipeline into one shareable and reusable class. Learn more about DataModule here.

```
class StanfordCarsDataModule(pl.LightningDataModule):

    def __init__(self, batch_size, data_dir: str = './'):

        super().__init__()

        self.data_dir = data_dir

        self.batch_size = batch_size


        # Augmentation policy for training set

        self.augmentation = transforms.Compose([

              transforms.RandomResizedCrop(size=256, scale=(0.8, 1.0)),

              transforms.RandomRotation(degrees=15),

              transforms.RandomHorizontalFlip(),

              transforms.CenterCrop(size=224),

              transforms.ToTensor(),

              transforms.Normalize([0.485, 0.456, 0.406],[0.229, 0.224, 0.225])

        ])

        # Preprocessing steps applied to validation and test set.

        self.transform = transforms.Compose([

              transforms.Resize(size=256),

              transforms.CenterCrop(size=224),

              transforms.ToTensor(),

              transforms.Normalize([0.485, 0.456, 0.406],[0.229, 0.224, 0.225])

        ])


        self.num_classes = 196


    def prepare_data(self):

        pass
```

```
    def setup(self, stage=None):

        # build dataset

        dataset = StanfordCars(root=self.data_dir, download=True, split="train")

        # split dataset

        self.train, self.val = random_split(dataset, [6500, 1644])


        self.test = StanfordCars(root=self.data_dir, download=True, split="test")


        self.test = random_split(self.test, [len(self.test)])[0]


        self.train.dataset.transform = self.augmentation

        self.val.dataset.transform = self.transform

        self.test.dataset.transform = self.transform


    def train_dataloader(self):

        return DataLoader(self.train, batch_size=self.batch_size, shuffle=True,
num_workers=8)


    def val_dataloader(self):

        return DataLoader(self.val, batch_size=self.batch_size, num_workers=8)


    def test_dataloader(self):

        return DataLoader(self.test, batch_size=self.batch_size, num_workers=8)
```

LightingModule - Define the System

LightningModule is another class that organizes model definition, training, validation, and testing code in one place. Learn more about this here.

Let us look at the model definition to see how transfer learning can be used with PyTorch Lightning.

In the LitModel class, we can use the pre-trained model provided by Torchvision as a feature extractor for our classification model. Here we are using ResNet-18. A list of pre-trained models provided by PyTorch Lightning can be found here.

When pretrained=True, we use the pre-trained weights; otherwise, the weights are initialized randomly.

If .eval() is used, then the layers are frozen.

A single Linear layer is used as the output layer. We can have multiple layers stacked over the feature_extractor.

```
class LitModel(pl.LightningModule):

    def __init__(self, input_shape, num_classes, learning_rate=2e-4, transfer=False):

        super().__init__()


        # log hyperparameters

        self.save_hyperparameters()

        self.learning_rate = learning_rate

        self.dim = input_shape

        self.num_classes = num_classes


        # transfer learning if pretrained=True

        self.feature_extractor = models.resnet18(pretrained=transfer)


        if transfer:

            # layers are frozen by using eval()

            self.feature_extractor.eval()

            # freeze params

            for param in self.feature_extractor.parameters():

                param.requires_grad = False


        n_sizes = self._get_conv_output(input_shape)


        self.classifier = nn.Linear(n_sizes, num_classes)


        self.criterion = nn.CrossEntropyLoss()

        self.accuracy = Accuracy()
```

```
    # returns the size of the output tensor going into the Linear layer from the conv
block.

    def _get_conv_output(self, shape):

        batch_size = 1

        tmp_input = torch.autograd.Variable(torch.rand(batch_size, *shape))


        output_feat = self._forward_features(tmp_input)

        n_size = output_feat.data.view(batch_size, -1).size(1)

        return n_size


    # returns the feature tensor from the conv block

    def _forward_features(self, x):

        x = self.feature_extractor(x)

        return x


    # will be used during inference

    def forward(self, x):

       x = self._forward_features(x)

       x = x.view(x.size(0), -1)

       x = self.classifier(x)


       return x


    def training_step(self, batch):

        batch, gt = batch[0], batch[1]

        out = self.forward(batch)

        loss = self.criterion(out, gt)
```

```
        acc = self.accuracy(out, gt)

        self.log("train/loss", loss)
        self.log("train/acc", acc)

        return loss

    def validation_step(self, batch, batch_idx):
        batch, gt = batch[0], batch[1]
        out = self.forward(batch)
        loss = self.criterion(out, gt)

        self.log("val/loss", loss)

        acc = self.accuracy(out, gt)
        self.log("val/acc", acc)

        return loss

    def test_step(self, batch, batch_idx):
        batch, gt = batch[0], batch[1]
        out = self.forward(batch)
        loss = self.criterion(out, gt)

        return {"loss": loss, "outputs": out, "gt": gt}

    def test_epoch_end(self, outputs):
        loss = torch.stack([x['loss'] for x in outputs]).mean()
```

```
        output = torch.cat([x['outputs'] for x in outputs], dim=0)


        gts = torch.cat([x['gt'] for x in outputs], dim=0)


        self.log("test/loss", loss)
        acc = self.accuracy(output, gts)
        self.log("test/acc", acc)


        self.test_gts = gts
        self.test_output = output


    def configure_optimizers(self):
        return torch.optim.Adam(self.parameters(), lr=self.learning_rate)  .
.
```

While initializing the LitModule passing transfer=True will freeze the ResNet-18 backbone and use the pretrained weights instead for transfer learning.

Results

I have trained the defined model from scratch(ResNet-18 backbone trained from scratch) and with transfer learning(ResNet-18 pre-trained on ImageNet). Every other hyperparameter remains the same. Check out the colab notebook to reproduce the results.

Note that to train from scratch, you will have to pass pretrained arguments as False and comment out the line, self.feature_extractor.eval().

Let us compare the metrics to see the magic of transfer learning.

> We can see that the model trained with transfer learning(TransferLearning) performs far better than the model trained from scratch(FromScratch).

The test accuracy for TransferLearning is ~32%, while that of FromScratch is ~16%.

Conclusion and Resources

I hope you find this report helpful. I will encourage you to play with the code and train an image classifier with a dataset of your choice from scratch and using transfer learning.

To learn more about transfer learning check out these resources:

Write a comment...

Hi Ayush, Regarding the following snippet: # layers are frozen by using eval() self.feature_extractor.eval() In addition to calling .eval(), don't you also need to set requires_grad=False on all self.feature_extractor.parameters()?

1 reply

Thanks for this easy to follow through report Ayush. Transfer Learning is a crucial step in many machine learning workflow. PyTorch Lightning in my limited experience is the new Keras.

1 reply

Tags: Computer Vision, Domain Agnostic, Classification, Transfer Learning, PyTorch Lightning, Tutorial, ResNet, Plots, Slider, Caltech-101, ImageNet

Published with ❤️ on Weights & Biases. Read more reports in our community, Fully Connected.