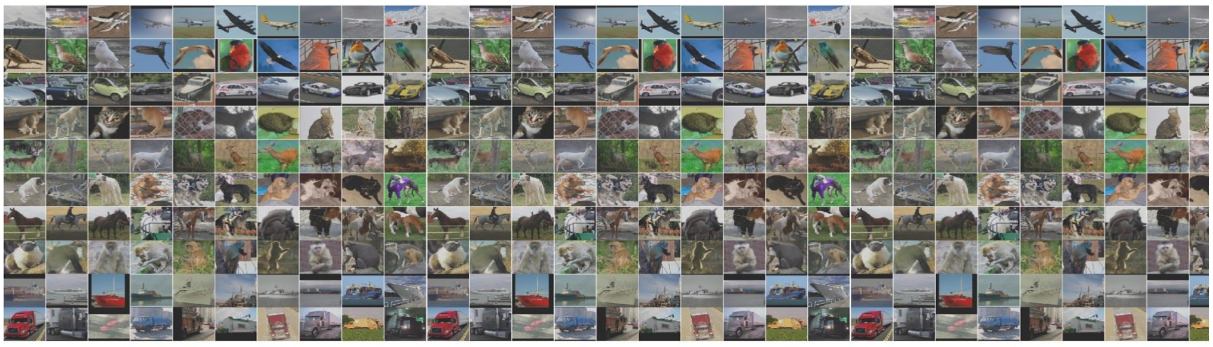
[PyTorch] Simple Contrastive Learning Representation using STL10 dataset



Hi there!
Today we will extensively talk about Simple Contrastive Learning Representation Framework developed by Google.

Agenda

We are going to implement SimCLR from scratch which will allow you better to understand how the framework is arranged using PyTorch and PyTorch Lightning Frameworks as well as such architectures as ResNet and EfficientNet.

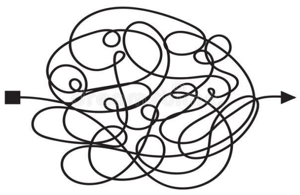
I kindly encourage you to familiarize yourself with both of the papers and original Google AI blog post as well which are indicated in the references at the end of the tutorial.

Before we will jump into explanations, I would like to remind and give a kind hint for you - DO NOT blindly copy the codebase! Always try to dive into the pipeline and keep asking yourself the questions!

So, fasten the belts and let's get started!

LETS GET STARTED

Introduction



As you know classical Machine Learning consists of various tasks. The most popular and heard are "Supervised", "Unsupervised" and "Reinforcement" tasks. But there are many more others such as "Semi-Supervised" and "Self-Supervised" tasks, for instance.

Literally, I would like to pay your attention to "Unsupervised" and "Self-Supervised". Try to answer what they differ from each other? Actually it is hot discussion topic nowadays as well, but from my point of view the main difference is that:

- ***Unsupervised***: there are no corresponding labels for the data. (mostly, term applied for Machine Learning tasks)
- ***Self-Supervised***: is a bit more complex. It relies on the unsupervised system but ***makes*** its own labels. (mostly, term applied for Computer Vision tasks)
  I.E.: Self-Supervised is a family of techniques for converting an unsupervised learning problem into a supervised one by creating surrogate labels from the unlabeled dataset.

So, SimCLR is a Self-Supervised algorithm, meaning you should keep in mind, as a part of the framework, we will generate labels for unlabeled STL10 dataset.

STL10 dataset

The STL-10 dataset is an image recognition dataset for developing unsupervised feature learning, deep learning, self-taught learning algorithms. It is inspired by the CIFAR-10 dataset but with some modifications.
In particular, each class has fewer labeled training examples than in CIFAR-10, but a very large set of unlabeled examples is provided to learn image models prior to supervised training. The primary challenge is to make use of the unlabeled data (which comes from a similar but different distribution from the labeled data) to build a useful prior.

SimCLR

In general, SimCLR is a simple framework for contrastive learning of visual representations. it's a set of fixed steps that one should follow in order to train good-quality image embeddings.

SimCLR Pipeline:

1. Take an input image
2. Generate 2 random augmentations on the image, including rotations, hue/saturation/brightness changes, zooming, cropping, etc.
3. Run ResNet50 to obtain image representations (embeddings) of those augmented images.
4. Run MLP (Multi Layer Perceptron) to project embeddings into another vector space.
5. Calculate the contrastive loss and run backpropagation through both networks.

SimCLR Loss:

The formal definition of the loss for a pair of positive examples (i) and (j) is as follows:

$$\ell_{i,j} = -\log \frac{\exp(\mathrm{sim}(\boldsymbol{z}_i, \boldsymbol{z}_j)/\tau)}{\sum_{k=1}^{2N} \mathbb{1}_{[k\neq i]} \exp(\mathrm{sim}(\boldsymbol{z}_i, \boldsymbol{z}_k)/\tau)}$$

The final loss is an arithmetic mean of the losses for all positive pairs in the batch:

$$\mathcal{L} = \frac{1}{2N} \sum_{k=1}^{N} [\ell(2k-1, 2k) + \ell(2k, 2k-1)]$$

Few additional notes to know about Contrastive Loss:

1. Contrastive loss decreases when projections of augmented images coming from the same input image are similar.
2. For two augmented images: (i), (j) (coming from the same input image—I will call them a "positive" pair later on), the contrastive loss for (i) tries to identify (j) among other images ("negative" examples) that are in the same batch.

Coding

Well, once we familiarize ourselves with the data to work with as well as the main concept of the SimCLR framework, let us start implementing the pipeline!

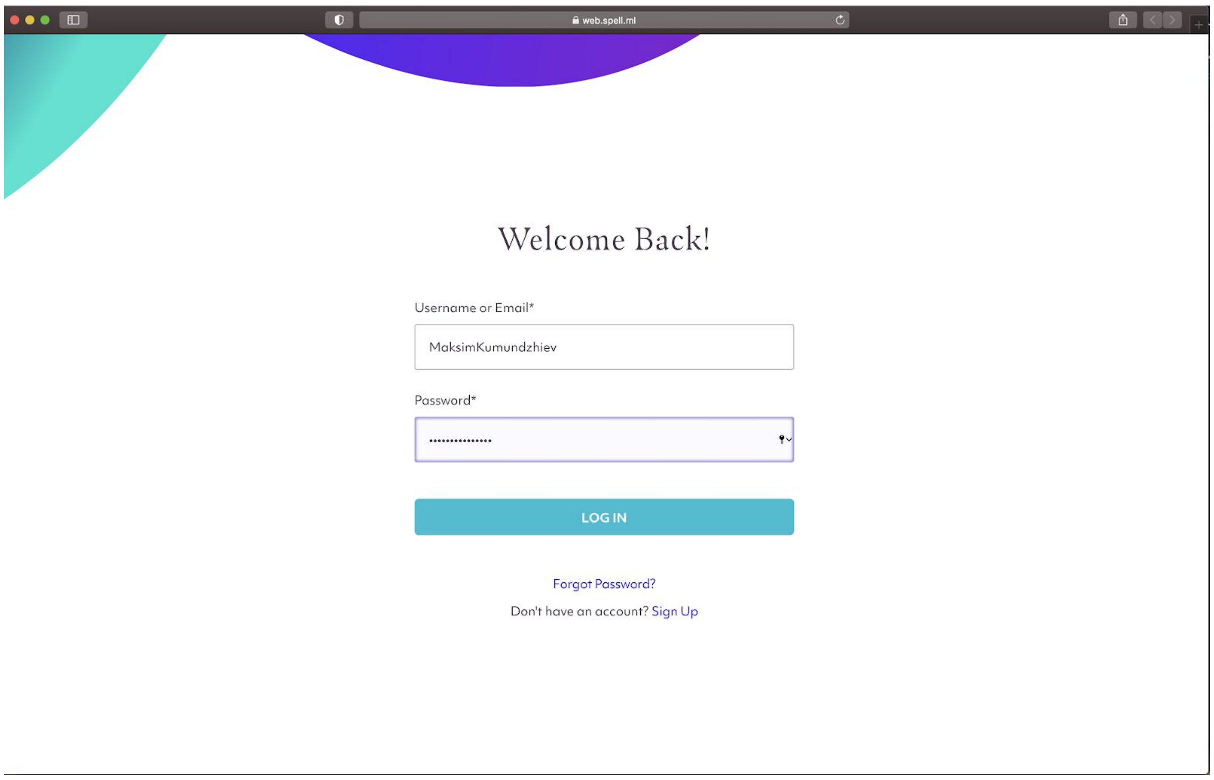
Firstly we need the environment to work in, including the GPU availability. For that we need create the new workflow within Spell.

To do that we have few options. In this tutorial we will use Spell CLI. If you would like to know and apply other approaches, check the tutorial I have written.

Open the terminal in the preferable location and follow steps below:

```
$ mkdir Spell-SimCLR && cd Spell-SimCLR

# Navigate to the GitHub account and create plaint repository named Spell-SimCLR
# Once you did it, follow steps below

$ echo "# Hey, SimCLR" >> README.md
$ git init
$ git add -A && git commit -m 'Initialised repository with README.'
$ git remote add origin https://github.com/<GitHub-Name>/Spell-SimCLR.git
$ git push -u origin master

# get help
$ spell jupyter --help

# create the workflow  within the just created github repository
$ spell jupyter --machine-type K80 --framework pytorch --pip
efficientnet_pytorch SimCLR-CLI
```

After you will see following window:



Welcome Back!

Username or Email*

MaksimKumundzhiev

Password*

Oaky, now we have prepared and launched the environment as well as GitHub repository to store the project. As simple as it is, you see!

Better to understand we will split our approach on to following sub-parts:

1. obtain and explore data
2. define the loss function
3. define the utility functions for image augmentations
4. define SimCLR neural network for embeddings + train
5. build an image classifier on top of SimCLR embeddings + train
6. evaluate

- Obtain and Explore Data

```
import random
import numpy as np
from argparse import Namespace

import torch
from torch import nn
import torch.nn.functional as F
from torch.optim import RMSprop
from torch.multiprocessing import cpu_count
from torch.utils.data import Dataset, DataLoader, SubsetRandomSampler,
SequentialSampler
from torch.optim.lr_scheduler import CosineAnnealingLR

from torchvision import transforms
from torchvision.datasets import STL10
import torchvision.transforms.functional as tvf

from efficientnet_pytorch import EfficientNet

import pytorch_lightning as pl


# Download data
stl10_unlabeled = STL10(".", split="unlabeled", download=True)

# plot few images
# Images are stored within PIL.Image.Image wrapper
for _ in range(1, 5):
    plt.figure(figsize=(7, 5))
    image_index = random.randint(10000)
    image_object = stl10_unlabeled[image_index][0]
    plt.title(f Category: {image_object.category} with
size:{image_object.size}')
    plt.imshow(image_object)
    plt.show()
```

- Define the Loss Function

```
class ContrastiveLoss(nn.Module):
   def __init__(self, batch_size, temperature=0.5):
       super().__init__()
       self.batch_size = batch_size
       self.register_buffer("temperature", torch.tensor(temperature))
       self.register_buffer("negatives_mask", (~torch.eye(batch_size * 2,
batch_size * 2, dtype=bool)).float())

   def forward(self, emb_i, emb_j):
       """
       emb_i and emb_j are batches of embeddings, where corresponding
indices are pairs
       z_i, z_j as per SimCLR paper
       """
       z_i = F.normalize(emb_i, dim=1)
       z_j = F.normalize(emb_j, dim=1)

       representations = torch.cat([z_i, z_j], dim=0)
       similarity_matrix = F.cosine_similarity(representations.unsqueeze(1),
representations.unsqueeze(0), dim=2)

       sim_ij = torch.diag(similarity_matrix, self.batch_size)
       sim_ji = torch.diag(similarity_matrix, -self.batch_size)
       positives = torch.cat([sim_ij, sim_ji], dim=0)

       nominator = torch.exp(positives / self.temperature)
       denominator = self.negatives_mask * torch.exp(similarity_matrix /
self.temperature)

       loss_partial = -torch.log(nominator / torch.sum(denominator, dim=1))
       loss = torch.sum(loss_partial) / (2 * self.batch_size)
       return loss
```

- Define the Utility Functions for Images Augmentation

```
class ResizedRotation():
   def __init__(self, angle, output_size=(96, 96)):
       self.angle = angle
       self.output_size = output_size

   def angle_to_rad(self, ang): return np.pi * ang / 180.0

   def __call__(self, image):
       w, h = image.size
       new_h = int(np.abs(w * np.sin(self.angle_to_rad(90 - self.angle))) +
np.abs(h * np.sin(self.angle_to_rad(self.angle))))
       new_w = int(np.abs(h * np.sin(self.angle_to_rad(90 - self.angle))) +
np.abs(w * np.sin(self.angle_to_rad(self.angle))))
       img = tvf.resize(image, (new_w, new_h))
       img = tvf.rotate(img, self.angle)
       img = tvf.center_crop(img, self.output_size)
       return img
```

```python
class WrapWithRandomParams():
   def __init__(self, constructor, ranges):
       self.constructor = constructor
       self.ranges = ranges

   def __call__(self, image):
       randoms = [float(np.random.uniform(low, high)) for _, (low, high) in
zip(range(len(self.ranges)), self.ranges)]
       return self.constructor(*randoms)(image)

def random_rotate(image):
   if random.random() > 0.5:
       return tvf.rotate(image, angle=random.choice((0, 90, 180, 270)))
   return image

class PretrainingDatasetWrapper(Dataset):
   def __init__(self, ds: Dataset, target_size=(96, 96), debug=False):
       super().__init__()
       self.ds = ds
       self.debug = debug
       self.target_size = target_size
       if debug:
           print("DATASET IN DEBUG MODE")


       self.preprocess = transforms.Compose([
               transforms.ToTensor(),
               transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225]),
       ])

       random_resized_rotation = WrapWithRandomParams(lambda angle:
ResizedRotation(angle, target_size), [(0.0, 360.0)])
       self.randomize = transforms.Compose([
           transforms.RandomResizedCrop(target_size, scale=(1/3, 1.0),
ratio=(0.3, 2.0)),
           transforms.RandomChoice([
               transforms.RandomHorizontalFlip(p=0.5),
               transforms.Lambda(random_rotate)
           ]),
           transforms.RandomApply([
               random_resized_rotation
           ], p=0.33),
           transforms.RandomApply([
               transforms.ColorJitter(brightness=0.5, contrast=0.5,
saturation=0.5, hue=0.2)
           ], p=0.8),
           transforms.RandomGrayscale(p=0.2)
```

```
        ])

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

    def __getitem_internal__(self, idx, preprocess=True):
        this_image_raw, _ = self.ds[idx]

        if self.debug:
            random.seed(idx)
            t1 = self.randomize(this_image_raw)
            random.seed(idx + 1)
            t2 = self.randomize(this_image_raw)
        else:
            t1 = self.randomize(this_image_raw)
            t2 = self.randomize(this_image_raw)

        if preprocess:
            t1 = self.preprocess(t1)
            t2 = self.preprocess(t2)
        else:
            t1 = transforms.ToTensor()(t1)
            t2 = transforms.ToTensor()(t2)

        return (t1, t2), torch.tensor(0)

    def __getitem__(self, idx):
        return self.__getitem_internal__(idx, True)

    def raw(self, idx):
        return self.__getitem_internal__(idx, False)

## Apply Augmentations
idx = 123
random_resized_rotation = WrapWithRandomParams(lambda angle:
ResizedRotation(angle), [(0.0, 360.0)])

dataset = PretrainingDatasetWrapper(stl10_unlabeled, debug=False)
tvf.to_pil_image(dataset[idx][0][0]) # augmented image
tvf.to_pil_image(dataset.raw(idx)[0][0]) # corresponding original image
```

- Define SimCLR Neural Network for Embeddings + Train

Here we define the ImageEmbedding neural network which is based on EfficientNet-b0 architecture. We change the last layer of pre-trained EfficientNet with identity function and add projection for image embeddings on top of it with "Linear-ReLU-Linear" layers.

```
class ImageEmbedding(nn.Module):
   class Identity(nn.Module):
       def __init__(self): super().__init__()

       def forward(self, x):
           return x

   def __init__(self, embedding_size=1024):
       super().__init__()

       base_model = EfficientNet.from_pretrained("efficientnet-b0")
       internal_embedding_size = base_model._fc.in_features
       base_model._fc = ImageEmbedding.Identity()

       self.embedding = base_model

       self.projection = nn.Sequential(
           nn.Linear(in_features=internal_embedding_size,
out_features=embedding_size),
           nn.ReLU(),
           nn.Linear(in_features=embedding_size,
out_features=embedding_size)
       )

   def calculate_embedding(self, image):
       return self.embedding(image)

   def forward(self, X):
       image = X
       embedding = self.calculate_embedding(image)
       projection = self.projection(embedding)
       return embedding, projection


class ImageEmbeddingModule(pl.LightningModule):
   """PyTorch-Lightning-based training module that orchestrates everything
together:
        hyper-parameters handling
        SimCLR ImageEmbedding network
        STL10 dataset
        optimizer
        forward step
   """

   def __init__(self, hparams):
       hparams = Namespace(**hparams) if isinstance(hparams, dict) else
hparams
       super().__init__()
       self.hparams = hparams
```

```
        self.model = ImageEmbedding()
        self.loss = ContrastiveLoss(hparams.batch_size)

    def total_steps(self):
        return len(self.train_dataloader()) // self.hparams.epochs

    def train_dataloader(self):
        return DataLoader(PretrainingDatasetWrapper(stl10_unlabeled,

debug=getattr(self.hparams, "debug", False)),
                         batch_size=self.hparams.batch_size,
                         num_workers=cpu_count(),

sampler=SubsetRandomSampler(list(range(hparams.train_size))),
                         drop_last=True)

    def val_dataloader(self):
        return DataLoader(PretrainingDatasetWrapper(stl10_unlabeled,

debug=getattr(self.hparams, "debug", False)),
                         batch_size=self.hparams.batch_size,
                         shuffle=False,
                         num_workers=cpu_count(),

sampler=SequentialSampler(list(range(hparams.train_size + 1,
hparams.train_size + hparams.validation_size))),
                         drop_last=True)

    def forward(self, X):
        return self.model(X)

    def step(self, batch, step_name = "train"):
        (X, Y), y = batch
        embX, projectionX = self.forward(X)
        embY, projectionY = self.forward(Y)
        loss = self.loss(projectionX, projectionY)
        loss_key = f"{step_name}_loss"
        tensorboard_logs = {loss_key: loss}

        return { ("loss" if step_name == "train" else loss_key): loss, 'log':
tensorboard_logs,
                       "progress_bar": {loss_key: loss}}

    def training_step(self, batch, batch_idx):
        return self.step(batch, "train")

    def validation_step(self, batch, batch_idx):
        return self.step(batch, "val")
```

```
   def validation_end(self, outputs):
       if len(outputs) == 0:
           return {"val_loss": torch.tensor(0)}
       else:
           loss = torch.stack([x["val_loss"] for x in outputs]).mean()
           return {"val_loss": loss, "log": {"val_loss": loss}}

   def configure_optimizers(self):
       optimizer = RMSprop(self.model.parameters(), lr=self.hparams.lr)
       return [optimizer], []

# define Neural Net parameters
hparams = Namespace(
    lr=1e-3,
    epochs=50,
    batch_size=160,
    train_size=10000,
    validation_size=1000
)

module = ImageEmbeddingModule(hparams)
trainer = pl.Trainer(gpus=1, max_epochs=hparams.epochs)

trainer.fit(module)
```

We could also use W&B Logging to keep track of the experiments and moreover Spell allows to use it out of the box. But today we will skip this option.
After the training finishes, embeddings are ready for usage for the downstream tasks.

- ## Building Image Classifier on Top of SimCLR embeddings

Once we trained the embeddings ,we can use them to train the classifier on top of them. We will freeze the base network with embeddings and learning linear classifiers on top of it.

```
# Save the weights
checkpoint_file = "efficientnet-b0-stl10-embeddings.ckpt"
trainer.save_checkpoint(checkpoint_file)

class SimCLRClassifier(nn.Module):
   def __init__(self, n_classes, freeze_base, embeddings_model_path,
hidden_size=512):
       super().__init__()

       base_model =
ImageEmbeddingModule.load_from_checkpoint(embeddings_model_path).model

       self.embeddings = base_model.embedding
```

```
        if freeze_base:
            print("Freezing embeddings")
            for param in self.embeddings.parameters():
                param.requires_grad = False

        self.classifier =
nn.Linear(in_features=base_model.projection[0].in_features,
                      out_features=n_classes if n_classes > 2 else 1)

    def forward(self, X, *args):
        emb = self.embeddings(X)
        return self.classifier(emb)

class SimCLRClassifierModule(pl.LightningModule):
    def __init__(self, hparams):
        super().__init__()
        hparams = Namespace(**hparams) if isinstance(hparams, dict) else
hparams
        self.hparams = hparams
        self.model = SimCLRClassifier(hparams.n_classes, hparams.freeze_base,
                                      hparams.embeddings_path,
                                      self.hparams.hidden_size)
        self.loss = nn.CrossEntropyLoss()

    def total_steps(self):
        return len(self.train_dataloader()) // self.hparams.epochs

    def preprocessing(seff):
        return transforms.Compose([
                transforms.ToTensor(),
                transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225]),
        ])

    def get_dataloader(self, split):
        return DataLoader(STL10(".", split=split,
transform=self.preprocessing()),
                          batch_size=self.hparams.batch_size,
                          shuffle=split=="train",
                          num_workers=cpu_count(),
                         drop_last=False)

    def train_dataloader(self):
        return self.get_dataloader("train")

    def val_dataloader(self):
        return self.get_dataloader("test")
```

```
   def forward(self, X):
       return self.model(X)

   def step(self, batch, step_name = "train"):
       X, y = batch
       y_out = self.forward(X)
       loss = self.loss(y_out, y)
       loss_key = f"{step_name}_loss"
       tensorboard_logs = {loss_key: loss}

       return { ("loss" if step_name == "train" else loss_key): loss, 'log':
tensorboard_logs,
                       "progress_bar": {loss_key: loss}}

   def training_step(self, batch, batch_idx):
       return self.step(batch, "train")

   def validation_step(self, batch, batch_idx):
       return self.step(batch, "val")

   def test_step(self, batch, batch_idx):
       return self.step(Batch, "test")

   def validation_end(self, outputs):
       if len(outputs) == 0:
           return {"val_loss": torch.tensor(0)}
       else:
           loss = torch.stack([x["val_loss"] for x in outputs]).mean()
           return {"val_loss": loss, "log": {"val_loss": loss}}

   def configure_optimizers(self):
       optimizer = RMSprop(self.model.parameters(), lr=self.hparams.lr)
       schedulers = [
           CosineAnnealingLR(optimizer, self.hparams.epochs)
       ] if self.hparams.epochs > 1 else []
       return [optimizer], schedulers

hparams_cls = Namespace(
   lr=1e-3,
   epochs=5,
   batch_size=160,
   n_classes=10,
   freeze_base=True,
   embeddings_path="./efficientnet-b0-stl10-embeddings.ckpt",
   hidden_size=512
)
module = SimCLRClassifierModule(hparams_cls)
```

```
trainer = pl.Trainer(gpus=1, max_epochs=hparams_cls.epochs)
trainer.fit(module)
```

SimCLRClassifier - is the custom module — it uses already existing embeddings and freezes the base model's weights on demand. Note that SimCLRClassifier.embeddings are only the EfficientNet part of the whole network used before—the projection head is discarded.

So, we have builded and trained an image classifier, let us train it and then evaluate.

- Evaluate

```
def evaluate(data_loader, module):
   with torch.no_grad():
       progress = ["/", "-", "\\", "|", "/", "-", "\\", "|"]
       module.eval().cuda()
       true_y, pred_y = [], []
       for i, batch_ in enumerate(data_loader):
           X, y = batch_
           print(progress[i % len(progress)], end="\r")
           y_pred = torch.argmax(module(X.cuda()), dim=1)
           true_y.extend(y.cpu())
           pred_y.extend(y_pred.cpu())
       print(classification_report(true_y, pred_y, digits=3))
       return true_y, pred_y

_ = evaluate(module.val_dataloader(), module)
```

After we will obtain the Confusion Matrix with the results

```
              precision    recall  f1-score   support

           0      0.931     0.813     0.876       800
           1      0.800     0.632     0.708       800
           2      0.956     0.849     0.899       800
           3      0.614     0.630     0.622       800
           4      0.581     0.810     0.676       800
           5      0.603     0.608     0.605       800
           6      0.768     0.733     0.750       800
           7      0.736     0.595     0.658       800
           8      0.801     0.943     0.866       800
           9      0.794     0.861     0.826       800

    accuracy                          0.810      8000
   macro avg      0.764     0.729     0.729      8000
weighted avg      0.764     0.729     0.729      8000
```

Conclusion

So, within this tutorial we had explored the SimCLR pre-training framework proposed by Google. We defined the SimCLR and its contrastive loss function step by step. We used SimCLR's pre-training routine to build image embeddings using EfficientNet network architecture, and finally, builded the classifier on top of it. And helped us in all this ***Spell*** functionalities, using which we set up our environment and workspace.

Used Spell workspace setup:
Details
Creator: MaksimKumundzhiev
Machine Type: K80
Status: running
Jupyter: Standard
Initialization Repository: Spell-SimCLR

Environment
Framework: pytorch
Pip: efficientnet_pytorch
Apt
Env Vars

References:

Google AI Blog
SimCLR research paper
EfficientNet research paper
SimCLR github repository (from here you can use pre trained model)