PROJECT REPORT

Mingqi Yuan

ABSTRACT

Python implementation of the UniNet for iris recognition which is proposed in [Zhao and Kumar(2017)].

Keywords Iris recognition · Python · Pytorch

1 Brief summary

Dear Professor, thanks again for the valuable opportunity! This project is challenging and interesting, and I learn a lot from that. Because I have never been involved in projects about the iris recognition before, so it takes me some time to learn the corresponding theory. Moreover, because the TensorFlow can not realize the special gradients of ETL loss, I have to learn the PyTorch temporarily and rewrite the whole project. No matter whether I can pass the test, I sincerely hope that you can help me point out the shortcomings. And I'd appreciate it much if you can provide your codes for me. Thank you very much!

I have uploaded the project (contains no training data) to the GitHub, and you can check the details in the following url.

`https://github.com/Mingqi-Yuan/UniNet`

2 Project structure

This project is organized as follows:

- **dataset**: contains the training data, validation data and test data.
- **reference**: contains the reference materials.
- **report**: contains the LaTex files of the project report.
- **snapshots**: for saving model weights.
- **static**: contains the pretrained model (MaskNet) and so on.
- **data.py**: for constructing the data generator.
- **eval.py**: for calculating TAR, FAR, EER.
- **loss.py**: the implementation of the *Extended Triplet Loss*.
- **mask.py**: for predicting masks for all the images in dataset using the pretrained MaskNet.
- **match.py**: functions for the iris matching (e.g. Hanmming distance).
- **model.py**: model class of the UniNet.
- **network.py**: the implementation of the FeatNet and the MaskNet.
- **train.py**: training file.

3 class ETLoss

In this section, the implementation of the *Extended Triplet Loss* are elaborated in detail. The ETLoss class contains four major functions:

1) **shiftbits(self, fa, noshifts)** This function is used to calculate the shifted features.
2) **fd(self, f1, f2, mask1, mask2)** This function is used to calculate the *Fractional Distance* between two features.

```python
def shiftbits(self, fa, noshifts):
    fnew = fa.clone()
    width = fa.shape[2]
    s = 2 * np.abs(noshifts)
    p = width - s

    # Shift
    if noshifts == 0:
        return fa

    elif noshifts < 0:
        fnew[:, :, 0:p] = fa[:, :, s:p + s]
        fnew[:, :, p:width] = fa[:, :, 0:s]

    else:
        fnew[:, :, s:width] = fa[:, :, 0:p]
        fnew[:, :, 0:s] = fa[:, :, p:width]

    return fnew
```

```python
''' Fractional Distance '''
def fd(self, f1, f2, mask1, mask2):
    batch_size = f1.shape[0]
    batch_fd = torch.zeros(size=(batch_size, ))
    zero = torch.tensor(0.).to(self.device)

    for i in range(batch_size):
        M = torch.sum((mask1[i] == mask2[i]) & (mask1[i] == 1))
        fd = torch.where(
            ((mask1[i] == mask2[i]) & (mask1[i] == 1)),
            torch.square(f1[i] - f2[i]),
            zero)

        fd = torch.sum(fd) / M
        batch_fd[i] = fd

    return batch_fd
```

3) **mmsd(self, f1, f2, mask1, mask2)** This function is used to calculate the *Minimum Shifted and Masked Distance*.
4) **foward(self, fp, fa, fn, fp_mask, fa_mask, fn_mask)** This function is used to calculate the final loss.

```python
''' Minimum Shifted and Masked Distance '''
def mmsd(self, f1, f2, mask1, mask2):
    batch_size = f1.shape[0]
    fd_set = torch.zeros(size=(17, batch_size))

    for shifts in range(-8, 9):
        f1_s = self.shiftbits(f1, shifts)
        mask1_s = self.shiftbits(mask1, shifts)

        fd_set[shifts + 8] = self.fd(f1_s, f2, mask1_s, mask2)

    batch_min_fd = torch.min(fd_set, dim=0)

    return batch_min_fd.values, batch_min_fd.indices - 8
```

```python
def forward(self, fp, fa, fn, fp_mask, fa_mask, fn_mask):
    mmsd_fa_fp, offset_ap = self.mmsd(fa[:,0,:,:], fp[:,0,:,:], fa_mask, fp_mask)
    mmsd_fa_fn, offset_an = self.mmsd(fa[:,0,:,:], fn[:,0,:,:], fa_mask, fn_mask)

    etl_loss = mmsd_fa_fp - mmsd_fa_fn + self.alpha

    zero = torch.tensor(0.)
    etl_loss = torch.maximum(etl_loss, zero)
    etl_loss = torch.mean(etl_loss)

    return etl_loss, offset_ap, offset_an
```

4 Gradients of $\frac{\partial ETL}{\partial \mathbf{f}^P[x,y]}, \frac{\partial ETL}{\partial \mathbf{f}^A[x,y]}, \frac{\partial ETL}{\partial \mathbf{f}^N[x,y]}$

The ETL takes the shifted feature to get the final loss, which has special gradients definitions. When applying the back propagation, the AutoGrad tool of PyTorch will automatically calculate all the requisite gradients. But such gradients is different from the original definitions, which are needed to be replaced. So the following codes are leveraged to address the problem.

- etl_loss, b_AP, b_AN = self.etl_loss.forward(fp, fa, fn, img_ps_mask, img_as_mask, img_ns_mask) # get the etl loss, b_{AP}, b_{AN} for the batch features.
- fp.retain_grad() # obtain the gradients of f^P.
- fa.retain_grad() # obtain the gradients of f^A.
- fn.retain_grad() # obtain the gradients of f^N.
- etl_loss.backward()
- grad_etl2fp, grad_etl2fa, grad_etl2fn = self.get_grad(etl_loss, fp, fa, fn, img_ps_mask, img_as_mask, img_ns_mask, b_AP, b_AN) # get new gradients.

- fp.grad.data = grad_etl2fp.data # replace the old gradients.
- fa.grad.data = grad_etl2fa.data
- fn.grad.data = grad_etl2fn.data
- self.optimizer.step() # apply the BP process.

The **self.get_grad()** can be found in the **model.py**, which is used to calculate the gradients for the ETL.

5 Triplet input

The data generator defined in the data.py follows the selection method of the *facenet* repository in GitHub:

https://github.com/davidsandberg/facenet/blob/master/src/train_tripletloss.py

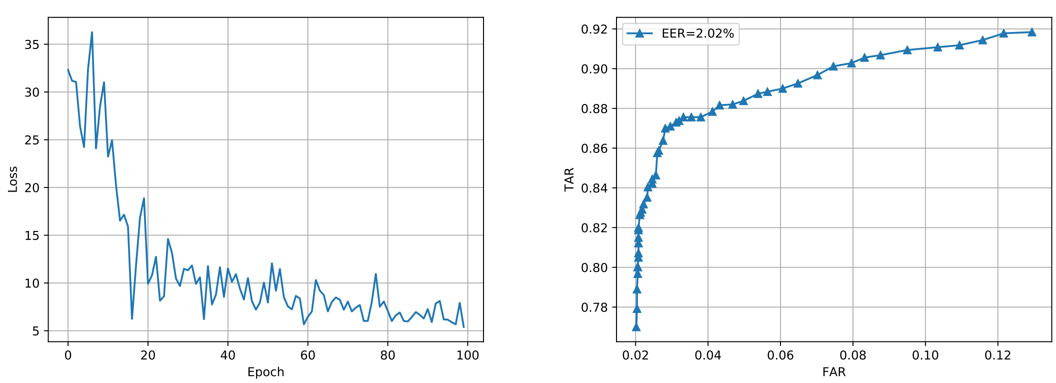
The only difference is the embedding form, which is a vector in facenet and a feature matrix in UniNet.

6 Training and evaluation

In view of the short time and limited devices (I have two RTX2080Ti GPU, each one only have 11Gb video memory.), so I have to use a part of the whole dataset to conduct the training, and the triplet input for each epoch is not comprehensive enough. The following results are based on the iris images of the former 50 people, which only aims to test the correctness the program. And the ROC is obtained with the randomly-selected pairs in validation data.

Table 1: Part of the training parameters

batch size	people number for per epoch	images number of per person	alpha
20	25	40	0.2



(a) Training loss

(b) ROC curve

Figure 1: Simulation results

References

[Zhao and Kumar(2017)] Zijing Zhao and Ajay Kumar. Towards more accurate iris recognition using deeply learned spatially corresponding features. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3809–3818, 2017.