A friendly introduction to Siamese Networks

You don't always need a lot of data to train your model, learn how to create a model with a tiny number of images per class

Sean Benhur J Sep 2, 2020 · 7 min read ★

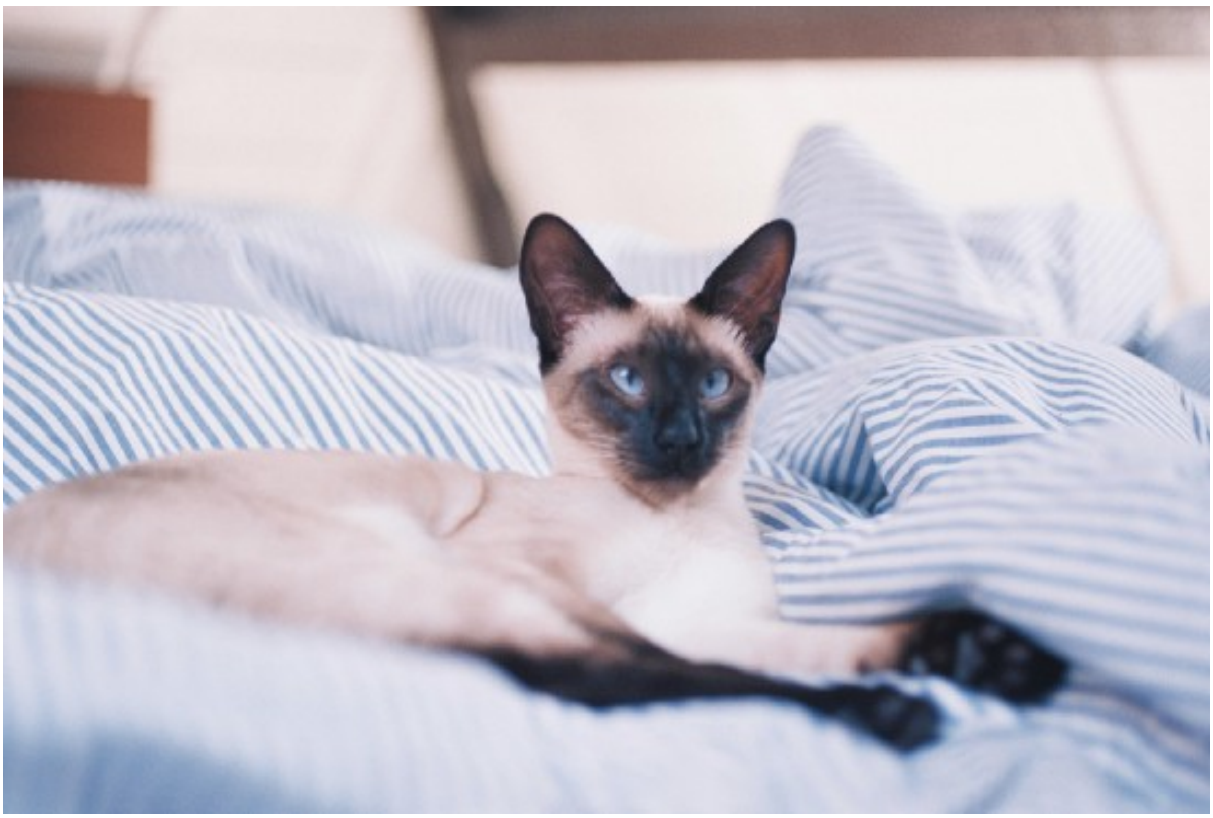
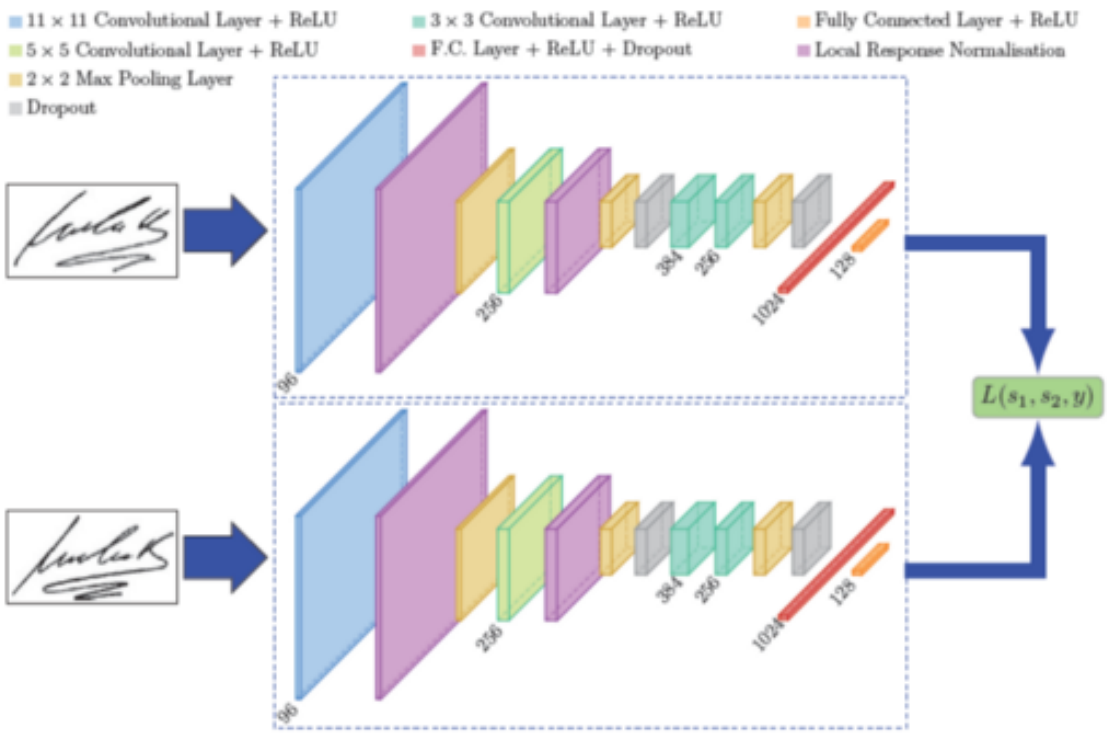


Photo by Lisa Algra on Unsplash

In the modern Deep learning era, Neural networks are almost good at every task, but these neural networks rely on more data to perform well. But, for certain problems like face recognition and signature verification, we can't always rely on getting more data, to solve this kind of tasks we have a new type of neural network architecture called Siamese Networks.

It uses only a few numbers of images to get better predictions. The ability to learn from very little data made Siamese networks more popular in recent years. In this article, we will explore what it is and how to develop a signature verification system with Pytorch using Siamese Networks.

What are Siamese Networks!?



Siamese network used in Signet

A Siamese Neural Network is a class of neural network architectures that **contain two or more *identical* subnetworks**. '*identical*' here means, they have the same configuration with the same parameters and weights. Parameter updating is mirrored across both sub-networks. It is used to find the similarity of the inputs by comparing its feature vectors, so these networks are used in many applications

Traditionally, a neural network learns to predict multiple classes. This poses a problem when we need to add/remove new classes to the data. In this case, we have to update the neural network and retrain it on the whole dataset. Also, deep neural networks need a large volume of data to train on. SNNs, on the other hand, learn a similarity function. Thus, we can train it to see if the two images are the same (which we will do here). This enables us to classify new classes of data without training the network again.

Pros and Cons of Siamese Networks:

The main advantages of Siamese Networks are,

- **More Robust to class Imbalance:** With the aid of One-shot learning, given a few images per class is sufficient for Siamese Networks to recognize those images in the future
- **Nice to an ensemble with the best classifier:** Given that its learning mechanism is somewhat different from Classification, simple averaging of it with a Classifier can do much better than average 2 correlated Supervised models (e.g. GBM & RF classifier)
- **Learning from Semantic Similarity:** Siamese focuses on learning embeddings (in the deeper layer) that place the same classes/concepts close together. Hence, can learn *semantic similarity*.

The downsides of the Siamese Networks can be,

- **Needs more training time than normal networks:** Since Siamese Networks involves quadratic pairs to learn from (to see all information available) it is slower than normal classification type of learning(pointwise learning)
- **Doesn't output probabilities:** Since training involves pairwise learning, it won't output the probabilities of the prediction, but the distance from each class

Loss functions used in Siamese Networks:

Contrastive Loss, Image created by Author

Since training of Siamese networks involves pairwise learning usual, Cross entropy loss cannot be used in this case, mainly two loss functions are mainly used in training these Siamese networks, they are

Triplet loss is a loss function where a baseline (anchor) input is compared to a positive (truthy) input and a negative (falsy) input. The distance from the baseline (anchor) input to the positive (truthy) input is minimized, and the distance from the baseline (anchor) input to the negative (falsy) input is maximized.

$$\mathcal{L}\left(A, P, N\right) = \max\left(\| \mathrm{f}(A) - \mathrm{f}(P)\|^2 - \| \mathrm{f}(A) - \mathrm{f}(N)\|^2 + \alpha, 0\right)$$

In the above equation, alpha is a margin term used to “stretch” the distance differences between similar and dissimilar pairs in the triplet, fa, fa, fn are the feature embeddings for the anchor, positive and negative images.

During the training process, an image triplet (anchor image, negative image, positive image)(anchor image, negative image, positive image) is fed into the model as a single sample. The idea behind this is that distance between the anchor and positive images should be smaller than that between the anchor and negative images.

Contrastive Loss: is a popular loss function used highly nowadays, It is a ***distance-based loss*** as opposed to more conventional ***error-prediction losses***. This loss is used to learn embeddings in which two similar points have a low

Euclidean distance and two dissimilar points have a large Euclidean distance.

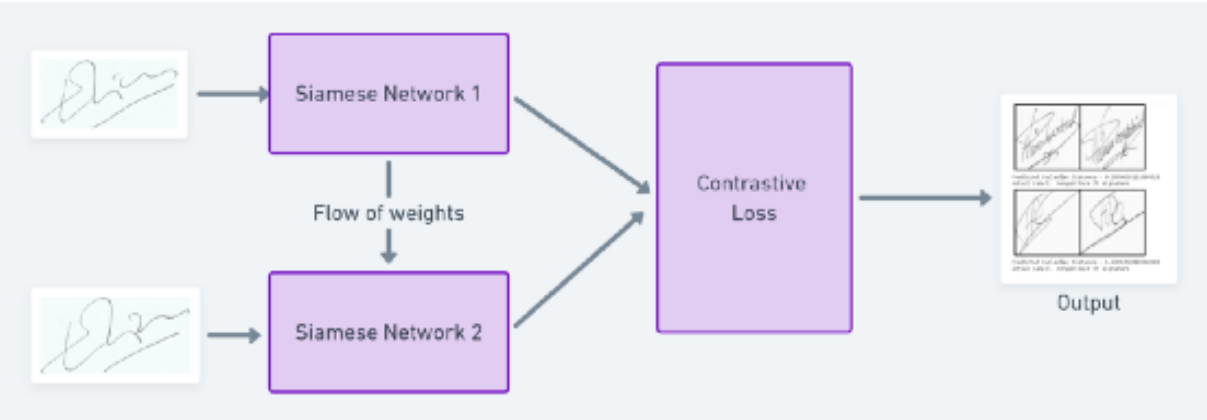
$$(1-Y)\frac{1}{2}(D_W)^2 + (Y)\frac{1}{2}\{max(0, m - D_W)\}^2$$

And we defined Dw which is just the Euclidean distance as :

$$\sqrt{\{G_W(X_1) - G_W(X_2)\}^2}$$

Gw is the output of our network for one image.

Signature verification with Siamese Networks:



Siamese Network for Signature Verification, Image created by Author

As Siamese networks are mostly used in verification systems such as face recognition, signature verification, etc…, Let's implement a signature verification system using Siamese neural networks on Pytorch

Dataset and Preprocessing the Dataset:

Signatures in ICDAR dataset, Image created by Author

We are going to use the ICDAR 2011 dataset which consists of the signatures of the dutch users both genuine and fraud, and the dataset itself is separated as train and folders, inside each folder, it consists of users folder separated as genuine and forgery, also the labels of the dataset is available as CSV files, you can download the dataset from here

Now to fed this raw data into our neural network, we have to turn all the images into tensors and add the labels from the CSV files to the images, to do this we can use the custom dataset class from Pytorch, here is how our full code will look like

```
#preprocessing and loading the dataset
class SiameseDataset():
    def __init__(self,training_csv=None,training_dir=None,transform=None):
        # used to prepare the labels and images path
        self.train_df=pd.read_csv(training_csv)
        self.train_df.columns =["image1","image2","label"]
        self.train_dir = training_dir
        self.transform = transform

    def __getitem__(self,index):
        # getting the image path
        image1_path=os.path.join(self.train_dir,self.train_df.iat[index,0])
        image2_path=os.path.join(self.train_dir,self.train_df.iat[index,1])
        # Loading the image
        img0 = Image.open(image1_path)
        img1 = Image.open(image2_path)
        img0 = img0.convert("L")
        img1 = img1.convert("L")
        # Apply image transformations
        if self.transform is not None:
            img0 = self.transform(img0)
            img1 = self.transform(img1)
        return img0, img1 , th.from_numpy(np.array([int(self.train_df.iat[ind
    def __len__(self):
```

Now after preprocessing the dataset, in PyTorch we have to load the dataset using Dataloader class, we will use the transforms function to reduce the image size into 105 pixels of height and width for computational purposes

```
# Load the the dataset from raw image folders
siamese_dataset = SiameseDataset(training_csv,training_dir,
                                        transform=transforms.Compose([transfor
                                                                      transfor
                                                                      ])
                                       )
```

dataloader_sign.py hosted with ❤ by **GitHub** **view raw**

Neural Network Architecture:

Now let's create a neural network in Pytorch, we will use the neural network architecture which will be similar, as described in the Signet paper

```
#create a siamese network
class SiameseNetwork(nn.Module):
    def __init__(self):
        super(SiameseNetwork, self).__init__()
        # Setting up the Sequential of CNN Layers
        self.cnn1 = nn.Sequential(
            nn.Conv2d(1, 96, kernel_size=11,stride=1),
            nn.ReLU(inplace=True),
            nn.LocalResponseNorm(5,alpha=0.0001,beta=0.75,k=2),
            nn.MaxPool2d(3, stride=2),

            nn.Conv2d(96, 256, kernel_size=5,stride=1,padding=2),
            nn.ReLU(inplace=True),
            nn.LocalResponseNorm(5,alpha=0.0001,beta=0.75,k=2),
            nn.MaxPool2d(3, stride=2),
            nn.Dropout2d(p=0.3),

            nn.Conv2d(256,384 , kernel_size=3,stride=1,padding=1),
            nn.ReLU(inplace=True),

            nn.Conv2d(384,256 , kernel_size=3,stride=1,padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(3, stride=2),
            nn.Dropout2d(p=0.3),
        )
        # Defining the fully connected layers
        self.fc1 = nn.Sequential(
            nn.Linear(30976, 1024),
            nn.ReLU(inplace=True),
            nn.Dropout2d(p=0.5),

            nn.Linear(1024, 128),
            nn.ReLU(inplace=True),

            nn.Linear(128,2))

    def forward_once(self, x):
        # Forward pass
        output = self.cnn1(x)
        output = output.view(output.size()[0], -1)
        output = self.fc1(output)
        return output

```

In the above code, we have created our network as follows, The first convolutional layers filter the 105*105 input signature image with 96 kernels of size 11 with a stride of 1 pixel. The second convolutional layer takes as input the(response-normalized and pooled) output of the first convolutional layer and filters it with 256 kernels of size 5. The third and fourth convolutional layers are connected to one another without any intervention of pooling or normalization of layers. The third layer has 384 kernels of size 3 connected to the (normalized, pooled, and dropout) output of the second convolutional layer. The fourth convolutional layer has 256 kernels of size 3 This leads to the neural network learning fewer lower level features for smaller receptive fields and more features for higher-level or more abstract features. The first fully connected layer has 1024 neurons, whereas the second fully connected layer has 128 neurons. This indicates that the highest learned feature vector from each side of SigNet has a dimension equal to 128, so where is the other network?

> *Since the weights are constrained to be identical for both networks, we use one model and feed it two images in succession. After that, we calculate the loss value using both the images and then backpropagate. This saves a lot of memory and also computational efficiency.*

Loss Function:

For this task, we will use Contrastive Loss, which learns embeddings in which two similar points have a low Euclidean distance and two dissimilar points have a large Euclidean distance, In Pytorch the implementation of Contrastive Loss will be as follows,

```
class ContrastiveLoss(torch.nn.Module):
    """
    Contrastive loss function.
    Based on:
    """

    def __init__(self, margin=1.0):
        super(ContrastiveLoss, self).__init__()
        self.margin = margin

    def forward(self, x0, x1, y):
        # euclidian distance
        diff = x0 - x1
        dist_sq = torch.sum(torch.pow(diff, 2), 1)
        dist = torch.sqrt(dist_sq)

        mdist = self.margin - dist
        dist = torch.clamp(mdist, min=0.0)
        loss = y * dist_sq + (1 - y) * torch.pow(dist, 2)
        loss = torch.sum(loss) / 2.0 / x0.size()[0]
```

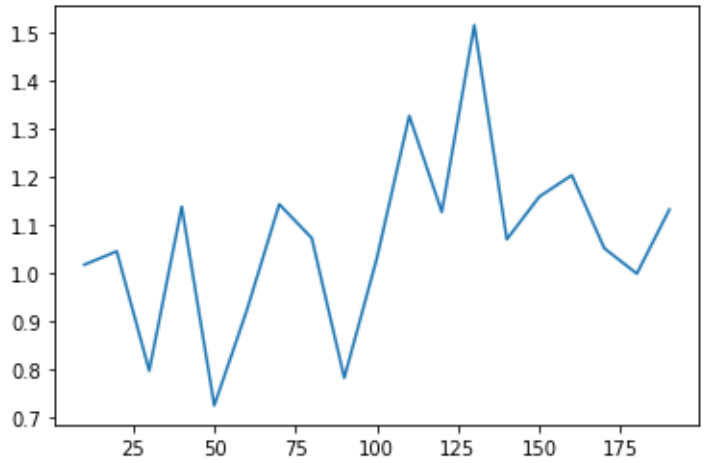
Training the Network:

The training process of a Siamese network is as follows:

- Initialize the network, loss function, and Optimizer(we will be using Adam for this project)
- Pass the first image of the image pair through the network.
- Pass the second image of the image pair through the network.
- Calculate the loss using the outputs from the first and second images.
- Back propagate the loss to calculate the gradients of our model.
- Update the weights using an optimizer
- Save the model

```
# Declare Siamese Network
net = SiameseNetwork().cuda()
# Decalre Loss Function
criterion = ContrastiveLoss()
# Declare Optimizer
optimizer = th.optim.Adam(net.parameters(), lr=1e-3, weight_decay=0.0005)
#train the model
def train():
    loss=[]
    counter=[]
    iteration_number = 0
    for epoch in range(1,config.epochs):
        for i, data in enumerate(train_dataloader,0):
            img0, img1 , label = data
            img0, img1 , label = img0.cuda(), img1.cuda() , label.cuda()
            optimizer.zero_grad()
            output1,output2 = net(img0,img1)
            loss_contrastive = criterion(output1,output2,label)
            loss_contrastive.backward()
            optimizer.step()
        print("Epoch {}\n Current loss {}\n".format(epoch,loss_contrastive.it
        iteration_number += 10
        counter.append(iteration_number)
        loss.append(loss_contrastive.item())
    show_plot(counter, loss)
    return net
#set the device to cuda
device = torch.device('cuda' if th.cuda.is_available() else 'cpu')
model = train()
```

The model was trained for 20 epochs on google colab for an hour, the graph of the loss over time is shown below.



Graph of loss over time

Testing the model:

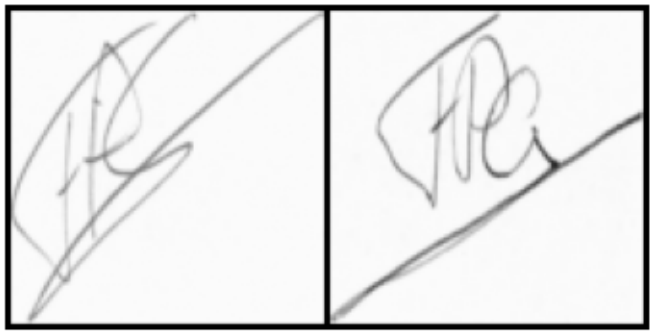
Now let's test our signature verification system on the test dataset,

- Load the test dataset using DataLoader class from Pytorch
- Pass the image pairs and the labels
- Find the euclidean distance between the images
- Based on the euclidean distance print the output

```
# Load the test dataset
test_dataset = SiameseDataset(training_csv=testing_csv,training_dir=testing_d
                                        transform=transforms.Compose([transfo
                                                                      transfo
                                                                      ])
                                       )
test_dataloader = DataLoader(test_dataset,num_workers=6,batch_size=1,shuffle=
#test the network
count=0
for i, data in enumerate(test_dataloader,0):
  x0, x1 , label = data
  concat = torch.cat((x0,x1),0)
  output1,output2 = model(x0.to(device),x1.to(device))

  eucledian_distance = F.pairwise_distance(output1, output2)

  if label==torch.FloatTensor([[0]]):
    label="Original Pair Of Signature"
  else:
    label="Forged Pair Of Signature"

  imshow(torchvision.utils.make_grid(concat))
  print("Predicted Eucledian Distance:-",eucledian_distance.item())
  print("Actual Label:-",label)
  count=count+1
  if count ==10:
```


The predictions were as follows,

Predicted Eucledian Distance:- 0.5056638121604919
Actual Label:- Forged Pair Of Signature



Predicted Eucledian Distance:- 1.1049458980560303
Actual Label:- Forged Pair Of Signature

Predicted Eucledian Distance:- 1.0919311046600342
Actual Label:- Original Pair Of Signature

Predicted Eucledian Distance:- 0.6714734435081482
Actual Label:- Original Pair Of Signature

Conclusion:

In this article, we discussed how Siamese networks are different from normal deep learning networks and implemented a Signature verification system using Siamese networks, you can find the entire code here

References:

https://hackernoon.com/one-shot-learning-with-siamese-networks-in-pytorch-8ddaab10340e

SigNet: Convolutional Siamese Network for Writer Independent Offline Signature Verification

Offline signature verification is one of the most challenging tasks in biometrics and document forensics. Unlike other...

arxiv.org

https://innovationincubator.com/siamese-neural-network-with-pytorch-code-example/

Facial Similarity with Siamese Networks in PyTorch

Implementing Face similarity with Siamese Networks in PyTorch

hackernoon.com

https://neptune.ai/blog/content-based-image-retrieval-with-siamese-networks