3η εργασία στα Νευρωνικά δίκτυα

Ιωακείμ Ευαγγελινός (4187)

January 24, 2025

1 Εισαγωγή

Κατά την εκπόνηση της παρούσας εργασία, εκπαιδεύονται και συγκρίνονται 4 μοντέλα autoencoder με 2 διαφορετικούς encoders και decoders, το καθένα με διαφορετικές υπέρ-παραμέτρους, στο σύνολο δεδομένων του CIFAR10 [1] για αναδημιουργία εικόνας. Συγκεκριμένα, συγκρίνονται το loss στα σύνολα δεδομένων εκπαίδευσης και δοκιμής και οι χρόνοι εκπαίδευσης. Έπειτα, συγκρίνονται με PCA.

2 Υπόβαθρο

Το CIFAR10 είναι ένα σύνολο δεδομένων 60000 RGB εικόνων με διαστάσεις $32x32$. Το σύνολο δεδομένων χωρίζεται σε σύνολο δεδομένων εκπαίδευση (50000 εικόνες) και δοκιμής (10000 εικόνες). Χρησιμοποιείται εκτενώς για εκπαιδευτικούς σκοπούς με στόχο την σωστή κατηγοροποίηση μιας εικόνας στην σωστή κατηγορία.
Το PyTorch είναι μια πολύ χρήσιμη βιβλιοθήκη της Python για την δημιουργία νευρωνικών δικτύων.
Το Scikit-learn είναι μια πολύ χρήσιμη βιβλιοθήκη της Python για την δημιουργία μοντέλων PCA

3 Προεπεξεργασία δεδομένων

Κατά την φόρτωση των δεδομένων, εφαρμόζω τα transformations μόνο για μετατροπή σε tensor. Στην συνέχεια, μετατρέπω τα δεδομένα σε μορφή που είναι αποδεκτή από το PCA του Scikit-learn για την εκπαίδευση του PCA.

4 Αλγόριθμος

Δημιουργώ 2 encoders και 2 encoders και φτιάχνω 4 autoencoders.

Ο πρώτος encoder έχει αρκετά πολλούς παραμέτρους στο hidden layer και χρησιμοποιεί και convolution layers και pooling layers και linear layers.

```
Encoder(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (dropout_relu): Sequential(
    (0): Dropout(p=0, inplace=False)
    (1): ReLU()
  )
  (conv_stack1): Sequential(
    (0): Conv2d(3, 1500, kernel_size=(2, 2), stride=(1, 1), padding=(1, 1))
    (1): MaxPool2d(kernel_size=2, stride=2, padding=1, dilation=1, ceil_mode=False)
  )
  (conv_stack2): Sequential(
    (0): Conv2d(1500, 500, kernel_size=(2, 2), stride=(1, 1), padding=(1, 1))
    (1): MaxPool2d(kernel_size=2, stride=2, padding=1, dilation=1, ceil_mode=False)
  )
  (conv_stack3): Sequential(
```

```
    (0): Conv2d(500, 64, kernel_size=(2, 2), stride=(1, 1), padding=(1, 1))
    (1): MaxPool2d(kernel_size=2, stride=2, padding=1, dilation=1, ceil_mode=False)
  )
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=2304, out_features=108, bias=True)
  )
)
    def forward(self, x):
        x=self.conv_stack1(x)
        x=self.dropout_relu(x)
        x=self.conv_stack2(x)
        x=self.dropout_relu(x)
        x=self.conv_stack3(x)
        x=self.dropout_relu(x)
        x=self.flatten(x)
        x=self.linear_relu_stack(x)
        x=self.dropout_relu(x)
        return x
```

Ο δεύτερος encoder έχει λογότερους παραμέτρους στο hidden layer και χρησιμοποιεί και μόνο convolution και pooling layers.

```
Encoder2(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (dropout_relu): Sequential(
    (0): Dropout(p=0, inplace=False)
    (1): ReLU()
  )
  (conv_stack1): Sequential(
    (0): Conv2d(3, 7, kernel_size=(2, 2), stride=(1, 1), padding=(1, 1))
    (1): MaxPool2d(kernel_size=2, stride=2, padding=1, dilation=1, ceil_mode=False)
  )
  (conv_stack2): Sequential(
    (0): Conv2d(7, 15, kernel_size=(2, 2), stride=(1, 1), padding=(1, 1))
    (1): MaxPool2d(kernel_size=2, stride=2, padding=1, dilation=1, ceil_mode=False)
  )
  (conv_stack3): Sequential(
    (0): Conv2d(15, 3, kernel_size=(2, 2), stride=(1, 1), padding=(1, 1))
    (1): MaxPool2d(kernel_size=2, stride=2, padding=1, dilation=1, ceil_mode=False)
  )
)
    def forward(self, x):
        x=self.conv_stack1(x)
        x=self.dropout_relu(x)
        x=self.conv_stack2(x)
        x=self.dropout_relu(x)
        x=self.conv_stack3(x)
        x=self.dropout_relu(x)
        x=self.flatten(x)
        #x=self.linear_relu_stack(x)
        #x=self.dropout_relu(x)
        return x
```

Ο πρώτος decoder έχει χρησιμοποιεί και convolution layers και linear layers.

```
Decoder(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (printL): PrintLayer()
  (unpool): MaxUnpool2d(kernel_size=(2, 2), stride=(2, 2), padding=(0, 0))
  (dropout_relu): Sequential(
```

```
    (0): Dropout(p=0, inplace=False)
    (1): ReLU()
  )
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=108, out_features=108, bias=True)
    (1): Unflatten(dim=1, unflattened_size=torch.Size([3, 6, 6]))
  )
  (conv_stack1): Sequential(
    (0): ConvTranspose2d(3, 15, kernel_size=(3, 3), stride=(2, 2), padding=(2, 2))
  )
  (conv_stack2): Sequential(
    (0): ConvTranspose2d(15, 32, kernel_size=(4, 4), stride=(2, 2), padding=(3, 3))
  )
  (conv_stack3): Sequential(
    (0): ConvTranspose2d(32, 3, kernel_size=(6, 6), stride=(2, 2))
  )
  (output_stack): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=18000, out_features=3072, bias=True)
    (2): Sequential(
      (0): Dropout(p=0, inplace=False)
      (1): ReLU()
    )
    (3): Unflatten(dim=1, unflattened_size=torch.Size([3, 32, 32]))
  )
)
```

```
    def forward(self, x):
        x=self.linear_relu_stack(x)
        x=self.dropout_relu(x)
        #print(x.shape, "After linear Decoder")
        x=self.conv_stack1(x)
        x=self.dropout_relu(x)
        #print(x.shape, "After conv1 Decoder")
        x=self.conv_stack2(x)
        x=self.dropout_relu(x)
        #print(x.shape, "After conv2 Decoder")
        x=self.conv_stack3(x)
        x=self.dropout_relu(x)
        #print(x.shape, "After conv3 Decoder")
        #x=self.output_stack(x)
        #print(x.shape, "After output Decoder")
        return x
```

Ο δεύτερος decoder έχει χρησιμοποιεί μόνο convolution layers.

```
Decoder2(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (printL): PrintLayer()
  (unpool): MaxUnpool2d(kernel_size=(2, 2), stride=(2, 2), padding=(0, 0))
  (dropout_relu): Sequential(
    (0): Dropout(p=0, inplace=False)
    (1): ReLU()
  )
  (linear_relu_stack): Sequential(
    (0): Unflatten(dim=1, unflattened_size=torch.Size([3, 6, 6]))
  )
  (conv_stack1): Sequential(
    (0): ConvTranspose2d(3, 15, kernel_size=(3, 3), stride=(2, 2), padding=(2, 2))
  )
```

```
  (conv_stack2): Sequential(
    (0): ConvTranspose2d(15, 32, kernel_size=(4, 4), stride=(2, 2), padding=(3, 3))
  )
  (conv_stack3): Sequential(
    (0): ConvTranspose2d(32, 3, kernel_size=(6, 6), stride=(2, 2))
  )
  (output_stack): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=18000, out_features=3072, bias=True)
    (2): Sequential(
      (0): Dropout(p=0, inplace=False)
      (1): ReLU()
    )
    (3): Unflatten(dim=1, unflattened_size=torch.Size([3, 32, 32]))
  )
)
```

```
    def forward(self, x):
        x=self.linear_relu_stack(x)
        x=self.dropout_relu(x)
        #print(x.shape, "After linear Decoder")
        x=self.conv_stack1(x)
        x=self.dropout_relu(x)
        #print(x.shape, "After conv1 Decoder")
        x=self.conv_stack2(x)
        x=self.dropout_relu(x)
        #print(x.shape, "After conv2 Decoder")
        x=self.conv_stack3(x)
        x=self.dropout_relu(x)
        #print(x.shape, "After conv3 Decoder")
        #x=self.output_stack(x)
        #print(x.shape, "After output Decoder")
        return x
```

Τέλος, έχω τον εξής AutoEncoder:

```
# AutoEncoder
class AutoEncoder(nn.Module):
    def
    ↪  __init__(self,encoder,decoder,bottleneck=3*6*6,dropout_percent=0,modelName="MyModel"):
        super().__init__()
        self.encoder=encoder
        self.decoder=decoder
        self.modelName=modelName


    def forward(self, x):
        bottleneck=self.encoder(x)
        #print(bottleneck.shape, "before decoder")
        #x=bottleneck[0]
        #torch.cuda.empty_cache()
        x=self.decoder(bottleneck)

        return x

```

Η εκπαίδευση γίνεται με τον ίδιο τρόπο με την 1η εργασία με criterion το MSLoss.

```
def save_checkpoint(model,optimizer=None,filename="checkpoint.pt",params={}):
    params["model_state_dict"]=model.state_dict()
    if (optimizer is not None):
```

```
        params["optimizer_state_dict"]=optimizer.state_dict()
    save(params, filename)

def train_model(model, dataloader, criterion=None,optimizer=None, epoches=5,
↪  limit_batches=-1,dataloader_test=None,path="",
↪  logdir="/home/i/ioakeime/Tensorboard_logs"):
    if (criterion is None):
        criterion=nn.MSELoss()
    if (optimizer is None):
        optimizer=optim.Adam(model.parameters(),lr=0.001)
    time_model=0
    #display.clear_output(wait=True)
    writer = SummaryWriter(logdir)
    file=open("/home/i/ioakeime/Models/Autoencoders/"+model.modelName+".txt","w")

    for epoch in range(0,epoches):
        subloss=0
        total_loss=0
        start_time_epoch=time.time()
        start_time = time.time()
        for batch_number,(x_train,y_train) in enumerate(dataloader):
            x_train,y_train=x_train.to(device),y_train.to(device)
            y_pred=model(x_train)

            #img_list=[y_train[0],y_pred[0]]
            #grid = make_grid(img_list)
            #show(grid)
            #print(y_pred.shape,x_train.shape)
            loss=criterion(y_pred,x_train)
            #print(loss)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            #subloss+=loss.item()
            total_loss+=loss.item()
            #if (batch_number%100==0):
                #print(f"Epoch: {epoch}, Batch Number: {batch_number}, Mini loss
                ↪  average: {subloss/100}, Mini training time: {time.time()-start_time}
                ↪  seconds")#, Model: {model}")
                #subloss=0
                #start_time = time.time()
        stop_time=time.time()
        with no_grad():
            total_loss_test=0
            start_time_test=time.time()
            for batch_number,(x_train,y_train) in enumerate(dataloader_test):
                x_train,y_train=x_train.to(device),y_train.to(device)
                y_pred=model(x_train)
                loss=criterion(y_pred,x_train)
                total_loss_test=+loss.item()
            stop_time_test=time.time()
        file.write("Loss_train: {0}, Loss_test: {1}, Time_train: {2}, Time_test: {3},
        ↪  Epoch:
        ↪  {4}\n".format(total_loss,total_loss_test,stop_time-start_time,stop_time_test-start_time_test,ep
        writer.add_scalar('Loss/train of/'+model.modelName, total_loss, epoch+1)
        writer.add_scalar('Loss/test of/'+model.modelName, total_loss_test, epoch+1)
        writer.add_scalar('Time/train of/'+model.modelName, stop_time-start_time,
        ↪  epoch+1)
        writer.add_scalar('Time/test of/'+model.modelName,
        ↪  stop_time_test-start_time_test, epoch+1)
```

```
            save_checkpoint(model,optimizer,filename="/home/i/ioakeime/Models/Autoencoders/checkpoint_"+mod
            #if (limit_batches==batch_number):
            #    break
    file.close()
    writer.close()
    print("Model "+model.modelName+" is trained for "+str(epoches)+" epoches!")
    return model
```

5 Σύγκριση μοντέλων AutoEncoder

Παρακάτω συγκρίνω τα μοντέλα AutoEncoder (με 0% πιθανότητα περιορισμού ενεργοποίησης και batch size = 35) για 100 εποχές, βελτιστοποιητή (Adam με learning rate = 0.001 και Stochastic gradient descent) ως προς τον χρόνο εκπαίδευσης σε δευτερόλεπτα και το MSLoss στα σύνολα εκπαίδευσης και δοκιμής. Από τα αποτελέσματα βλέπουμε ότι encoder 2 και ο decoder 2 (που δεν έχουν linear layers) έχουν καλύτερη απόδοση από τους πρώτους, με το καλύτερο μοντέλο να είναι το encoder2 - decoder2.

Encoder(1/2)	Decoder(1/2)	Training Loss	Testing Loss	Training time	Testing time
1	1	16.023	0.128	3810.9	3.072
1	2	13.827	0.0109	3806.33	3.07
2	1	17.732	0.0147	1044.331	1.563
2	2	16.375	0.1396	971.516	1.604

6 Παραδείγματα σωστής και εσφαλμένης αναδημιουργίας

(Τα μοντέλα μου στο τέλος δεν βγήκαν τόσο αποδοτικά, οπότε και η "σωστή" αναδημιουργία είναι θολή)
Σωστή:

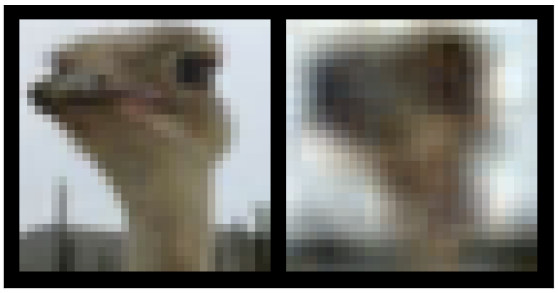

Figure 1: Σωστή αναδημιουργία

Εσφαλμένη:

7 Σύγκριση με PCA

Χρησιμοποίησα την βιβλιοθήκη scikit-learn για την δημιουργία του μοντέλου PCA (με 512 components) με τον εξής κώδικα:

```
pca = PCA(512)
x_proj=pca.fit_transform(x_train)
x_inv_proj=pca.inverse_transform(x_proj)
reconstructed_image=np.reshape(x_inv_proj,(50000,3,32,32))
original=torch.Tensor(np.reshape(np.array(x_train),(50000,3,32,32)))
reconstructed=torch.from_numpy(reconstructed_image)
```

Το MSLoss του pca είναι 0.0026 με training time 14.38 δευτερόλεπτα, με αποτέλεσμα να κυριαρχεί απόλυτα απέναντι στους autoencoder και από άποψη χρόνου εκπαίδευσης και από άποψη απόδοσης.

Συμπερασματικά, θα χρησιμοποιούσα το PCA με έναν ικανοποιητικό αριθμό component (με 32 components έχει χειρότερη απόδοση από τους autoencoder).

References

[1] *Learning Multiple Layers of Features from Tiny Images, Alex Krizhevsky, 2009.* https://www.cs.toronto.edu/ kriz/learning-features-2009-TR.pdf

Figure 2: Εσφαλμένη αναδημιουργία