Term Project

April 28, 2022

1 Recursive Recurrent Nets with Attention Modeling for OCR in the Wild – an Implementation Attempt

- Spencer Cain and William Goble

For our term project we wanted to implement an OCR model and then explore different methods for how the model can be improved. Our model is based off the paper 'Recursive Recurrent Nets with Attention Modeling for OCR in the Wild' by Chen-Yu Lee et al.

1.1 What is OCR?

OCR, or optical character recognition, is the machine learning task of reading text from image data. Versions of this are important in location- and advertisement-based applications in mapping software such as Google Maps or Apple Maps. Further, similar approaches can be used for the recently released text selection from images on iPhones. Another use case of OCR is for the purposes of reading license plates while policing.

1.2 Data Cleaning

We considered the Synth1K, Synth90K, and the IIIT5K Datasets for this project. The Synth1K and Synth90K are datasets that contain generated images of text, the former being of size 1k and the latter being 90k. The IIIT5K Dataset is a 5k word dataset harvested from Google image search. All three of these datasets contained the images and a label, but only the IIIT5K Dataset contained bounding boxes around the characters. We will address why we wanted to use bounding boxes for the characters in our theory section.

When cleaning the data we needed to resize all the images to conform with Lee et al.'s input shape, namely 100 by 32. Since we resized the images, this also meant that we needed to scale the bounding box markers of IIIT5K so it would still bound the right characters. After rescaling, we also gray-scaled the images to provide easier computation for the convolutional neural networks.

1.3 Model Structure

From the paper,

> The network architecture for our Base CNN model is shown in Table A1. It has 8 convolutional layer with 64, 64, 128, 128, 256, 256, 512 and 512 channels, and each convo- lutional layer uses kernel with a 3 x 3 spatial extent. Convo- lutions are performed with stride 1, zero padding, and ReLU activation function. 2 x 2 max pooling follows

the second, fourth, and sixth convolutional layers. The two fully con- nected layers have 4096 units. The input is a resized 32 x 100 gray scale image.

We now provide details for the network structures of the proposed untied recursive CNNs in Table A1. Notice that each of the even number convolutional layer (conv2, conv4, conv6 or conv8) use its own shared weight matrix that has exactly the same input and output dimensional- ity, and so projects feature maps to the same space multi- ple times within one recursive convolutional layer under the same parametric capacity as Base CNN model. For the character-level language modeling, we use RNNs with 1024 hidden units equipped with hyperbolic tangent activation function. Our overall system pipeline is shown in Figure 1.

We apply backpropagation through time (BPTT) algo- rithm to train the models with 256 batch size SGD and 0.5 dropout rate. Initial learning rate is 0.002 and decreased by a factor of 5 as validation errors stop decreasing for 2 epochs. All variants use the same scheme with 30 total epochs determined based on the validation set. We apply gradient clipping at the magnitude of 10, and find it with in place weight decay did not add extra performance gains.

[1]:
```
import torch
from torch import nn
import string

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

[2]:
```
# The energy based attention model placed between the two RNN layers.
class EnergyAttention(nn.Module):
    def __init__(self, I_shape, s_shape):
        super().__init__()

        self.s_shape = s_shape
        self.out_shape = I_shape

        self.s_layer = nn.Linear(self.s_shape, self.out_shape)
        self.tanh = nn.Tanh()
        self.softmax = nn.Softmax(dim=2)

    def forward(self, I, s):
        s_proj = self.s_layer(s)
        result = self.tanh(I + s_proj)  # -> [1, 1, 1024]

        alpha = self.softmax(result)  # -> [1, 1, 1024]
        return alpha * I
```

[3]:
```
# The R^2AM Model.
class BaseModel(nn.Module):
    def __init__(self, eow, recursions=1):
        super().__init__()
```

```
        self.sow_size = eow.size(0)  # SOW = 0, 0, ..., 0, 0
        self.eow = eow  # EOW = 0, 0, ..., 0, 1
        self.recursions = recursions

        self.conv1_0 = nn.Conv2d(1, 64, (3, 3), padding="same")
        self.conv1_t = nn.Conv2d(64, 64, (3, 3), padding="same")

        self.pool1 = nn.MaxPool2d((2, 2), stride=2)

        self.conv2_0 = nn.Conv2d(64, 128, (3, 3), padding="same")
        self.conv2_t = nn.Conv2d(128, 128, (3, 3), padding="same")

        self.pool2 = nn.MaxPool2d((2, 2), stride=2)

        self.conv3_0 = nn.Conv2d(128, 256, (3, 3), padding="same")
        self.conv3_t = nn.Conv2d(256, 256, (3, 3), padding="same")

        self.pool3 = nn.MaxPool2d((2, 2), stride=2)

        self.conv4_0 = nn.Conv2d(256, 512, (3, 3), padding="same")
        self.conv4_t = nn.Conv2d(512, 512, (3, 3), padding="same")

        self.flatten = nn.Flatten()
        self.dense1 = nn.Linear(24576, 4096)
        self.dense2 = nn.Linear(4096, 4096)

        self.rnn_1 = nn.LSTM(input_size=self.sow_size, hidden_size=1024,
                             num_layers=1, batch_first=True,
                             proj_size=self.eow.size(0))

        self.attention = EnergyAttention(4096, self.eow.size(0))

        self.rnn_2 = nn.LSTM(input_size=4096, hidden_size=1024,
                             num_layers=1, batch_first=True,
                             proj_size=self.eow.size(0))

    def forward(self, x):
        x = self.conv1_0(x)
        for _ in range(self.recursions):
            x = self.conv1_t(x)

        x = self.pool1(x)

        x = self.conv2_0(x)
        for _ in range(self.recursions):
            x = self.conv2_t(x)
```

```
        x = self.pool2(x)

        x = self.conv3_0(x)
        for _ in range(self.recursions):
            x = self.conv3_t(x)

        x = self.pool3(x)

        x = self.conv4_0(x)
        for _ in range(self.recursions):
            x = self.conv4_t(x)

        x = self.flatten(x)

        x = self.dense1(x)
        x = self.dense2(x)

        I = torch.unsqueeze(x, dim=1)

        batch_size = x.size(0)
        batched_sow = torch.autograd.Variable(torch.zeros(size=(x.size(0), 1,␣
↪self.sow_size))).to(device)
        h0 = torch.autograd.Variable(torch.zeros(1, batch_size, self.eow.
↪size(0))).to(device)
        c0 = torch.autograd.Variable(torch.zeros(1, batch_size, 1024)).
↪to(device)
        results = torch.autograd.Variable(torch.zeros(batch_size, self.
↪sow_size, 23)).to(device)

        s, (hn_1, cn_1) = self.rnn_1(batched_sow, (h0, c0))
        c_t = self.attention(I, s)
        x, (hn_2, cn_2) = self.rnn_2(c_t, (h0, c0))
        results[:, :, 0] = torch.squeeze(x, dim=1)
        for idx in range(1, 23):
            s, (hn_1, cn_1) = self.rnn_1(x, (hn_1, cn_1))
            c_t = self.attention(I, s)
            x, (hn_2, cn_2) = self.rnn_2(c_t, (hn_2, cn_2))
            results[:, :, idx] = torch.squeeze(x, dim=1)

        return results
```

[4]:
```
eow = torch.zeros(size=(len(string.printable) + 1,))
eow[len(string.printable)] = 1

base_cnn_model = BaseModel(eow=eow).to(device)
```

```
x = torch.rand(1, 1, 32, 100).to(device)
x = base_cnn_model(x)

preds = torch.argmax(x[0].T, dim=1).tolist()  # [0] because its the first item in a batch size of 1
for pred in preds:
    print(string.printable[pred], end='')
```

\\\\\\\\\\\\\\\\\\\\\\\\

[5]:
```
from IIIT5K.dataset import IIIT5KDataset
from torch.utils.data import DataLoader

train_set = IIIT5KDataset(split='train')
val_set = IIIT5KDataset(split='val')

train_loader = DataLoader(train_set, batch_size=256, shuffle=True)
val_loader = DataLoader(val_set, batch_size=256, shuffle=True)
```

[6]:
```
def train_model(model_name: str, num_epochs: int):
    model = BaseModel(eow=eow).to(device)
    print('Total Parameters:', sum(p.numel() for p in model.parameters()))
    print('Trainable Parameters:', sum(p.numel() for p in model.parameters() if p.requires_grad))
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(model.parameters(), lr=.002)
    running_loss = 0.
    for epoch in range(num_epochs):
        for step, (image, label) in enumerate(train_loader):
            image = torch.unsqueeze(image, dim=1).to(device)
            label = torch.stack(label, dim=0).to(device)

            optimizer.zero_grad()

            output = model(image)
            loss = criterion(output, label.T)
            loss.backward()

            torch.nn.utils.clip_grad_norm_(model.parameters(), 10.)

            optimizer.step()

            # Gather data and report
            running_loss += loss.item()
        last_loss = running_loss / step # loss per batch

        acc, total = 0., 0.
```

```
        for o, l in zip(output, label):
            o = torch.argmax(o, dim=1)
            for o_, l_ in zip(o, l):
                if l_ == 100:
                    break
                if o_ == l_:
                    acc += 1.
                total += 1.

        acc = acc / total
        print(f'epoch {epoch+1} -> train_loss: {last_loss:.4f}, train_acc: {acc:
↪.4f}')

        running_loss = 0.

        # Validation
        val_loss = 0.
        val_acc = 0.
        val_total = 0.
        with torch.no_grad():
            for step, (image, label) in enumerate(val_loader):
                image = torch.unsqueeze(image, dim=1).to(device)
                label = torch.stack(label, dim=0).to(device)

                output = model(image)
                loss = criterion(output, label.T)
                val_loss += loss.item()

                for o, l in zip(output, label):
                    o = torch.argmax(o, dim=1)
                    for o_, l_ in zip(o, l):
                        if l_ == 100:
                            break
                        if o_ == l_:
                            val_acc += 1.
                        val_total += 1.
            val_acc = val_acc / val_total
            val_loss = val_loss / step
            print(f'epoch {epoch+1} -> val_loss: {val_loss:.4f}, val_acc:␣
↪{val_acc:.4f}')

    return model
```

[7]:
```
model = train_model(model_name="R2AM", num_epochs=30)
```

```
Total Parameters: 140792256
Trainable Parameters: 140792256
```

```
epoch 1 -> train_loss: 4.9184, train_acc: 0.0000
epoch 1 -> val_loss: 9.2156, val_acc: 0.0060
epoch 2 -> train_loss: 4.9117, train_acc: 0.0000
epoch 2 -> val_loss: 9.2031, val_acc: 0.0109
epoch 3 -> train_loss: 4.9049, train_acc: 0.0336
epoch 3 -> val_loss: 9.1906, val_acc: 0.0030
epoch 4 -> train_loss: 4.8982, train_acc: 0.0000
epoch 4 -> val_loss: 9.1780, val_acc: 0.0131
epoch 5 -> train_loss: 4.8914, train_acc: 0.0085
epoch 5 -> val_loss: 9.1654, val_acc: 0.0200
epoch 6 -> train_loss: 4.8845, train_acc: 0.0131
epoch 6 -> val_loss: 9.1527, val_acc: 0.0070
epoch 7 -> train_loss: 4.8776, train_acc: 0.0088
epoch 7 -> val_loss: 9.1398, val_acc: 0.0000
epoch 8 -> train_loss: 4.8707, train_acc: 0.0128
epoch 8 -> val_loss: 9.1269, val_acc: 0.0026
epoch 9 -> train_loss: 4.8637, train_acc: 0.0000
epoch 9 -> val_loss: 9.1139, val_acc: 0.0146
epoch 10 -> train_loss: 4.8566, train_acc: 0.0064
epoch 10 -> val_loss: 9.1006, val_acc: 0.0039
epoch 11 -> train_loss: 4.8494, train_acc: 0.0000
epoch 11 -> val_loss: 9.0871, val_acc: 0.0063
epoch 12 -> train_loss: 4.8421, train_acc: 0.0159
epoch 12 -> val_loss: 9.0736, val_acc: 0.0032
epoch 13 -> train_loss: 4.8347, train_acc: 0.0088
epoch 13 -> val_loss: 9.0597, val_acc: 0.0066
epoch 14 -> train_loss: 4.8272, train_acc: 0.0000
epoch 14 -> val_loss: 9.0457, val_acc: 0.0117
epoch 15 -> train_loss: 4.8196, train_acc: 0.0000
epoch 15 -> val_loss: 9.0314, val_acc: 0.0075
epoch 16 -> train_loss: 4.8117, train_acc: 0.0214
epoch 16 -> val_loss: 9.0168, val_acc: 0.0089
epoch 17 -> train_loss: 4.8038, train_acc: 0.0294
epoch 17 -> val_loss: 9.0019, val_acc: 0.0031
epoch 18 -> train_loss: 4.7957, train_acc: 0.0096
epoch 18 -> val_loss: 8.9867, val_acc: 0.0078
epoch 19 -> train_loss: 4.7873, train_acc: 0.0000
epoch 19 -> val_loss: 8.9710, val_acc: 0.0067
epoch 20 -> train_loss: 4.7788, train_acc: 0.0161
epoch 20 -> val_loss: 8.9551, val_acc: 0.0031
epoch 21 -> train_loss: 4.7702, train_acc: 0.0099
epoch 21 -> val_loss: 8.9388, val_acc: 0.0000
epoch 22 -> train_loss: 4.7613, train_acc: 0.0057
epoch 22 -> val_loss: 8.9220, val_acc: 0.0059
epoch 23 -> train_loss: 4.7519, train_acc: 0.0117
epoch 23 -> val_loss: 8.9048, val_acc: 0.0000
epoch 24 -> train_loss: 4.7426, train_acc: 0.0198
epoch 24 -> val_loss: 8.8870, val_acc: 0.0134
```

```
epoch 25 -> train_loss: 4.7328, train_acc: 0.0000
epoch 25 -> val_loss: 8.8688, val_acc: 0.0000
epoch 26 -> train_loss: 4.7229, train_acc: 0.0000
epoch 26 -> val_loss: 8.8499, val_acc: 0.0134
epoch 27 -> train_loss: 4.7125, train_acc: 0.0097
epoch 27 -> val_loss: 8.8306, val_acc: 0.0039
epoch 28 -> train_loss: 4.7019, train_acc: 0.0078
epoch 28 -> val_loss: 8.8104, val_acc: 0.0033
epoch 29 -> train_loss: 4.6909, train_acc: 0.0133
epoch 29 -> val_loss: 8.7898, val_acc: 0.0039
epoch 30 -> train_loss: 4.6796, train_acc: 0.0097
epoch 30 -> val_loss: 8.7682, val_acc: 0.0067
```

As shown above, the model structure from the paper and dataset used to generate results in the paper did not work in our implementation. While implementing the model, we ran into some issues with the `torch` API, including the optimizer not updating the weights and the loss not changing throughout training. We explored several different versions of the model to troubleshoot the issues, which reduced our time to implement our novel theories. Unfortunately, we could not get past these issues. Missing from the paper's architecture in comparison to ours is a scheduled learning rate, dropout, and weight decay. However, these missing components do not seem to be the likely cause behind our results.

1.4 Novel Theory

Our confusion when reading the paper by Lee et al. involved how the RNN layers could read letters from the linear layers. Since linear layers are not sequential by nature, we believed that this could be one of the larger difficulties that the model needed to learn from. Our biggest concern was how the RNN could read/predict letters from left to right. Could this solely be an implicitly learned task? To provide a novel improvement to the model, we intended to bridge the gap from non-sequential data to sequential data with bounding boxes. This would require two submodels, rather than an end-to-end implementation. Our theory was to train a CNN-based model to predict the locations of letters in an image with bounding boxes. Then, with a pretrained CNN as the bounding box predictor, we would use the bounding boxes in order from left to right to classify letters and use RNN sequence prediction abilities to smooth out any poor classification predictions made by the linear layers. Had we been able to implement this approach, we believe it would perform more consistantly in a wider variety of application cases than the base model.

1.5 Ethical Consideration

Optical Character Recognition models (OCR) is one method for converting images of either hand-written, typed, or printed text into machine encoded text. This technology is a useful tool for automating tasks that traditionally require a human to examine an image and determine what text say. One such example is the automation of the traffic violation pipeline. Currently if someone were to run through a red light with a traffic camera, or drive through a toll plaza, a photo of their license plate would be taken, processed, and a bill would be sent to the drivers address. A task such as identifying a license plate, and extracting the license plate number, use to require human intervention, but thanks to technology like OCR models, this task can now be fully automated. Another example of a task OCR models could accomplish is reading signs or products that an individual might have in the background during a virtual meeting, such as over Zoom or Microsoft

Teams, and then send targeted ads to the individual to encourage them to buy more, or similar, products. As a quick note, we are not saying any one company is doing this, but it is reasonable to assume this could be done. In both the targeted ads and the license plate reader scenario, as machine learning engineers we are forced to face the question where does data collection end and personal privacy begin?

One important thing to keep in mind when working with OCR models is having transparency with why the model processes a certain type of image text. A lack of transparency regarding OCR model usage is the difference between an automated toll booth system, and a device a police state could use to track the locations of citizens. Like in most fields, transparency is the key to gaining the trust of the users, and encourages the users to buy into the system. When companies are transparent with their intentions of the model, it places the choice back into the hands of the individual, and with their consent can lead to better improvements to the overall system. Additionally, machine learning engineers and companies need to be on gaurd against people trying to abuse their models in order to exploit the users. If the company is transparent about how to protects the users data, this transparency will also help build public trust.