CSCI 6962/4140: Homework 5

Assigned Thursday November 7 2024. Due by 11:59pm Thursday November 21 2023.

Create a Jupyter notebook for this assignment, and use Python 3. Write documented, readable and clear code (e.g. use reasonable variable names). Submit this notebook making sure the answers to each question are legible and clearly labeled. You will be graded primarily based on the solutions and answers already present in the notebook, but it must also be runnable to reproduce your results.

Do you remember the initial days of CAPTCHAs[1], when they consisted of images of sequences of letters and characters that you needed to convert to text? Nowadays CAPTCHAs have become torturous interactive and responsive multi-page affairs that tax your spatial and conceptual reasoning in addition to your eyesight. This is because the first generation of CAPTCHAs were exceedingly vulnerable to basic ML attacks. In this homework, you will somewhat prove this by using a CNN+RNN architecture for an OCR task very similar to CAPTCHA solving.

Overview The model you build will take as input an image of variable width W and fixed height 32 as input. It will then use a convolutional architecture to convert this image to a feature tensor of size $64 \times 8 \times T$; here 64 is the number of channels and $T = W/4$. Each slice in the 3rd mode should be viewed as containing information on a single character, so we convert this tensor into a $512 \times T$ matrix where each column corresponds to a single character. The sequence of T feature vectors are then fed through a (deep) bidirectional RNN to convert them to T softmax vectors, each of which represents the probability distribution over the possible output characters. Figure 1 gives an overview of the process.

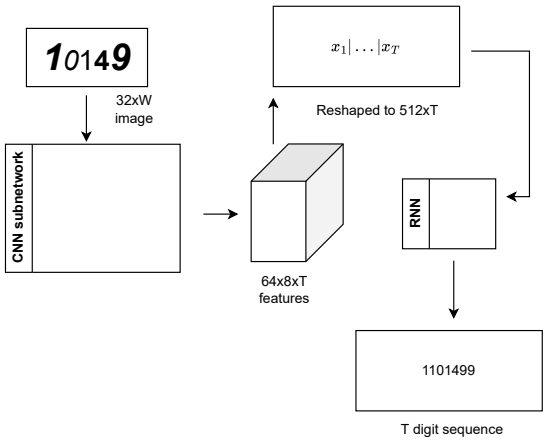

Figure 1: Overview of the architecture of your network.

Of course, the input image may contain a sequence shorter than length T, as in the case of '10149', but the output sequence is always of length T. Further, if for example the first 1 and the 9 in the input are particularly wide, the network may detect these digits in multiple consecutive positions. This is an example of one possible alignment issue between the ground truth sequence and the output sequence: multiple output sequences can correspond to the same input sequences. This and other alignment issues are standard in sequence modeling, and to handle them, we use the Connectionist Temporal Classification (CTC) loss.

To better understand the CTC loss, read the Weights and Biases article on text recognition using a CRNN-CTC network.

NB, this will take time and compute power. Setup the environment on your laptop and get the model running properly there. You will train the model for 300 epochs, so you can infer

[1]Completely Automated Public Turing tests to tell Computers and Humans Apart

the runtime by seeing how long one epoch takes. If the code runs too slow locally, try using GPUs through Google Colab.

0. [0 points] Install the necessary packages. If you are not already using conda, I highly recommend using it with a virtual environment to prevent these changes from impacting your system-wide installation.

   torchvision Install torchvision. You will use it for data augmentation, specifically to translate, shear, and rotate the input images to simulate Captchas.

   trdg Install the Text Recognition Data Generator (trdg). This will be used to generate the training data, consisting of pairs of images of short sequences of numbers, which will serve as input to our OCR model, and the corresponding sequences, which will serve as the labels.

   Pillow You will use Pillow to read the input images.

   CTCDecoder You will train the OCR network using the CTC loss. Pytorch has a built-in CTCLoss function, but the CTCDecoder function in torchaudio did not work for me. Accordingly, install the CTCDecoder from Harald Scheidl's github by following the instructions given there.

   Helper functions Download ocr_batch_functions.py[2] and keep in the same directory as your Jupyter notebook; this will be used to generate our training and validation data using trdg. Similarly, download ocr_visualization.py and keep in the same directory as your Jupyter notebook; this will be used to visualize the performance of your OCR model.

1. [50 points] Write a class `OCR` that implements CNN layers followed by an RNN to convert images of digits into the actual sequence of digits. The class should implement the following functions:

   (CNNBlock) A convenience function `CNNBlock(c, k)` that returns a `Sequential` CNN subnetwork that comprises `LazyConv2d` → `BatchNorm2d` → `ReLU()`, where the convolution creates c output channels, uses $k \times k$ filters, and zero-pads the input channels so that the output channels have the same size as in the input (read the help for `LazyConv2d` to see how to accomplish such padding).

   (Constructor) A constructor that takes the following arguments, in the order they are presented, top to bottom:

`numChars`	the number of characters that the network must learn to recognize
`outputSize`	the size of the features that the convolutional subnetwork will pass into the RNN
`hiddenSize`	the dimensionality of RNN hidden state vectors
`numLayers`	the number of layers in the RNN.

   The constructor initializes:

   * A `Sequential` network, `CNNSubnet`, that comprises `CNNBlock(8,3)` → `CNNBlock(16, 3)` → `MaxPool2d(2)` →`CNNBlock(32, 3)` → `CNNBlock(64, 3)` → `MaxPool2d(2)`.
   * A linear layer, `DenseLayer`, with appropriate input size to convert the flattened output from the CNN subnetwork into a vector of size `outputSize`.
   * A bidirectional LSTM, `Decoder`, with `numLayer` layers that takes inputs of size `outputSize` and uses `hiddenSize`-dimensional hidden state vectors,

[2] A modified version of Avi Kutvonen's code.

* A linear layer, `LogitLayer`, that takes as input a concatenated hidden state from the final LSTM layer (represented as a vector of size $2 \times$ `hiddenSize`) and returns a logits vector of size `numChars` + 1.

(Forward) The forward function takes as input a minibatch `x` of size $(N, C = 3, H = 32, W)$ where N is the minibatch size, C is the number of input channels, and H and W are respectively the height and width of the input images. It returns as output `p` of size $(T, N,$ `numChars` $+ 1)$ where $T = W/4$ is the number of characters in the decoded message. For each allowable value of t and i, the `numChars` $+ 1$ dimensional tensor $P[t, i]$ contains the logits of the probabilities of the tth output token being the corresponding characters. There are `numChars` + 1 characters because we include the special symbol used by the CTCDecoder.

* Apply `CNNSubnet` to `x`. The result has shape $(N, 64, 8, T)$.
* Flatten `x` to have shape $(N, 512, T)$.
* Swap dimensions using `torch.transpose` so that `x` has shape $(N, T, 512)$.
* Apply `DenseLayer` to `x`, followed by a ReLU nonlinearity. Assign the result back to `x`. The result has shape $(N, T,$ `outputSize`$)$.
* Apply `Decoder` to `x` to get the output hidden states and store the results in a tensor `o`; this tensor has shape $(N, T, 2 \times$ `hiddenSize`$)$.
* Apply `LogitLayer` to `o` and store the results in a tensor `p`; this tensor has shape $(N, T,$ `numChars` $+ 1)$.
* Swap dimensions of `p` so that it has shape $(T, N,$ `numChars` $+ 1)$.
* Apply `torch.nn.functional.log_softmax` to the appropriate dimension of `p` and return the result.

2. [30 points] Train the OCR model with loss function `torch.nn.CTCLoss`, using `reduction="sum"`.

   a. Generate the custom training and validation DataLoaders that use TRDG, using the following code.

```
from trdg.generators import GeneratorFromRandom
from torchvision.transforms import v2
import ocr_batch_functions

epochSize = 10 # the number of minibatches per epoch
batchSize = 100
height = 32

allChars= string.digits
trdgGenerator = GeneratorFromRandom(use_symbols=False, use_letters=False,
                                    background_type=1)

charsToLabelsDict = dict((char, ind+1) for ind, char in enumerate(allChars))
labelsToCharsDict = dict((ind+1, char) for ind, char in enumerate(allChars))

transforms = v2.Compose([
    v2.PILToTensor(),  # Convert PIL image to tensor
    v2.RandomAffine(degrees=2.0,translate=(0.04,0.15),shear=4.0),
    v2.ToDtype(torch.float32),  # Normalize expects float input
    v2.Normalize(mean=[255*0.485, 255*0.456, 255*0.406],
                 std=[255*0.229, 255*0.224, 255*0.225]),
]) # generates slightly perturbed OCR images to simulate Captchas

dgParams = {"epochSize": epochSize,
```

```
                "batchSize": batchSize,
                "height": height,
                "transform": transforms,
                "char_to_lbl_dict": charsToLabelsDict}

trainGenerator = ocr_batch_functions.OCR_generator(trdgGenerator, **dgParams)
valGenerator = ocr_batch_functions.OCR_generator(trdgGenerator, **dgParams,
                            isValidation=True)
```

Of course, you should move the package imports up to the top of the notebook; they are included inline here for clarity.

Enumerating `trainGenerator` returns `X, targets, targetLengths, inputLengths`, while enumerating `valGenerator` returns `images, X, targets, targetLengths, inputLengths`. These outputs are described in Table 1.

`images`	the sequence of raw images generated by trdg in this minibatch
`X`	the tensor formed by converting the images to tensors; has shape (N, H, W)
`targets`	the tensor containing the character sequences corresponding to each image; this has shape $(N, \ell_{\max})$, where $\ell_{\max}$ is the length of the longest character sequence in the minibatch; the other character sequences are padded to that length.
`inputLengths`	this tensor contains the true length of each character sequence in the minibatch, and has shape N.

Table 1: Outputs of the train and validation data generators.

b. Use `valGenerator` to create 10 example input images, and display them in a 10 row array.

c. Create `model`, an instance of the `OCR` class. Take an appropriate `numChars`, `outputSize = 128`, `hiddenSize = 64`, and `numLayers = 4` (for final training; during development and testing, you may want to reduce these values to make the training quicker).

d. Create an ADAM optimizer for training `model`, using initial global learning rate `lr=5e-3`. Control the stepsize of the optimizer by using a learning rate scheduler to reduce the global learning rate by 1/2 after every 50 optimization steps; use `torch.optim.lr_scheduler.StepLR` to accomplish this.

e. Write standard training functions `train_epoch(dataGenerator, model, optimizer, reportInterval=1)` and `validate(dataGenerator, model)`. The training function should print out the current minibatch loss every `reportInterval` minibatches. The validation function should accumulate the CTCLoss over the entire validation set, and print and return this accumulated loss divided by the number of minibatches in that epoch.

   Inside of these training functions, you must check when an epoch has ended and break out of the loop enumerating over the DataLoaders, as they do not automatically end themselves at the end of an epoch. You can do so by checking `batchnum == len(dataGenerator)`, where `batchnum` is the index of the current minibatch.

f. Set `epochs=300` (for final training; during development and testing you should use a small number of epochs). Train the model using the following code to periodically report its performance and to save the best performing model.

```
import ocr_visualization
losses = [] # store the validation losses during training
```

```
bestIndex = 0
bestLoss = np.inf

for t in range(epochs):
    print(f"Epoch {t+1}, lr={scheduler.get_last_lr()}\n-----------------------")
    model.train()
    train_epoch(trainGenerator, model, optimizer)
    model.eval()
    valLoss = validate(valGenerator, model)
    losses.append(valLoss)
    if valLoss < bestLoss:
        bestIndex = t
        bestLoss = valLoss

    ocr_visualization.visualize_text(model, valGenerator, allChars, \\
                labelsToCharsDict, device) # helper function
    scheduler.step()

model.eval()
print("Done Training!")
```

3. [20 points] Visualize the performance of the trained model.

    a. In an appropriately labeled graph, plot the average validation loss (the quantity returned by the validation function) as a function of the number of epochs of training. For more meaningful visualization, only show results on the last 150 training epochs.

    b. Visualize the output of the best model on 10 example inputs from the validation generator, using `ocr_visualization.visualize_image`.