CS/DS 541: Deep Learning

Homework 4

Tom Arnold & Nate Hindman

Due: 5:59pm ET Thursday October 9

This problem can be done in teams of up 2 students.

1 Window Type Classification [20 points]

In this problem, the goal is to classify window images into one of five categories: "New Awning Window", "New Bay Window", "New Fixed Window", "New Horizontal Sliding Window", and "New Hung Window".

The training and test datasets can be accessed via the following links: * https://canvas.wpi.edu/files/7719816/download?download_frd=1 and * https://canvas.wpi.edu/files/7719811/download?download_frd=1.

To help you get started, a demo code are available at: https://colab.research.google.com/drive/1fG0f6LiPnv7a4nDWNPtVyo1Y0Xh5vp71?usp=sharing.

Answer

```
# import packages
import pandas as pd
import numpy as np
import torch
import json
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

```python
# connect to google drive
from google.colab import drive
drive.mount('/content/drive/')

# Step 1

# NEW TOM + NATE BLOCK
import torch.nn as nn

class betterCNN(nn.Module):
    def __init__(self, num_classes=5):
        super(betterCNN, self).__init__()

        # convolutional block
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(64)
        self.conv2 = nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1)
        self.bn2 = nn.BatchNorm2d(64)

        # convolutional block
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
        self.bn3 = nn.BatchNorm2d(128)
        self.conv4 = nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1)
        self.bn4 = nn.BatchNorm2d(128)

        # convolutional block
        self.conv5 = nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1)
        self.bn5 = nn.BatchNorm2d(256)
        self.conv6 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1)
        self.bn6 = nn.BatchNorm2d(256)

        # pooling and dropout
        self.pool = nn.MaxPool2d(2, 2)
        self.dropout = nn.Dropout(0.2)

        # fully connected layers
        self.fc1 = nn.Linear(256 * 8 * 8, 512)
        self.fc2 = nn.Linear(512, num_classes)

    def forward(self, x):
        # 1 block
        x = self.conv1(x)
```

```
        x = self.bn1(x)
        x = torch.relu(x)
        x = self.conv2(x)
        x = self.bn2(x)
        x = torch.relu(x)
        x = self.pool(x)

        # 2 block
        x = self.conv3(x)
        x = self.bn3(x)
        x = torch.relu(x)
        x = self.conv4(x)
        x = self.bn4(x)
        x = torch.relu(x)
        x = self.pool(x)

        # 3 block
        x = self.conv5(x)
        x = self.bn5(x)
        x = torch.relu(x)
        x = self.conv6(x)
        x = self.bn6(x)
        x = torch.relu(x)
        x = self.pool(x)

        # flatten
        x = x.view(-1, 256 * 8 * 8)

        # fully connected layers
        x = self.fc1(x)
        x = torch.relu(x)
        x = self.dropout(x)
        x = self.fc2(x)

        return x

model = betterCNN(num_classes=5)
model = model.to(device)
print(model)

# Step 2
```

```
# NEW TOM + NATE BLOCK

from torchvision import datasets, transforms
from torch.utils.data import DataLoader, random_split

# transform with augmentation
train_transform = transforms.Compose([
    transforms.Resize((64, 64)),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomRotation(degrees=15),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2,
 ↪  hue=0.1),
    transforms.RandomAffine(degrees=0, translate=(0.1, 0.1)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# val transform (no augmentation)
val_transform = transforms.Compose([
    transforms.Resize((64, 64)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

trainset_dir = '/content/drive/MyDrive/train_set'

full_dataset = datasets.ImageFolder(root=trainset_dir, transform=None)

total_size = len(full_dataset)
indices = list(range(total_size))
np.random.seed(42)
np.random.shuffle(indices)

train_size = int(0.85 * total_size)
val_size = total_size - train_size

train_indices = indices[0:train_size]
val_indices = indices[train_size:total_size]

train_dataset = datasets.ImageFolder(root=trainset_dir,
 ↪  transform=train_transform)
train_subset = torch.utils.data.Subset(train_dataset, train_indices)
```

```
val_dataset = datasets.ImageFolder(root=trainset_dir,
 ↪  transform=val_transform)
val_subset = torch.utils.data.Subset(val_dataset, val_indices)

train_loader = DataLoader(train_subset, batch_size=64, shuffle=True)
val_loader = DataLoader(val_subset, batch_size=64, shuffle=False)


# Step 3

# NEW TOM + NATE BLOCK

import torch.optim as optim
criterion = nn.CrossEntropyLoss(label_smoothing=0.09) # added label smoothing
optimizer = optim.Adam(model.parameters(), lr=0.002, weight_decay=0.00001) #
 ↪  added weight decay
# added learning rate scheduler
scheduler = optim.lr_scheduler.ReduceLROnPlateau(
    optimizer,
    mode='max',
    factor=0.5,
    patience=5)

# Step 4

# NEW TOM + NATE BLOCK

def train(model, train_loader, val_loader, criterion, optimizer, scheduler,
 ↪  num_epochs=30):
    best_val_acc = 0.0
    patience_counter = 0
    early_stop_patience = 10

    train_losses = []
    val_accuracies = []

    for epoch in range(num_epochs):
        # train
        model.train()
        running_loss = 0.0
        train_correct = 0
```

```
        train_total = 0

        for inputs, labels in train_loader:
            inputs = inputs.to(device)
            labels = labels.to(device)

            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss = running_loss + loss.item()

            predictions = torch.max(outputs, 1)[1]
            train_total = train_total + labels.size(0)
            train_correct = train_correct + (predictions ==
↪   labels).sum().item()

        avg_train_loss = running_loss / len(train_loader)
        train_acc = 100.0 * train_correct / train_total
        train_losses.append(avg_train_loss)

        # val
        model.eval()
        val_correct = 0
        val_total = 0

        with torch.no_grad():
            for inputs, labels in val_loader:
                inputs = inputs.to(device)
                labels = labels.to(device)
                outputs = model(inputs)
                predictions = torch.max(outputs, 1)[1]
                val_total = val_total + labels.size(0)
                val_correct = val_correct + (predictions ==
↪   labels).sum().item()

        val_acc = 100.0 * val_correct / val_total
        val_accuracies.append(val_acc)

        print(f"Epoch {epoch + 1}/{num_epochs}")
```

```
        print(f" -> Train Loss: {avg_train_loss:.4f}, Train Acc:
         ↪  {train_acc:.2f}%")
        print(f" -> Val Acc: {val_acc:.2f}%")

        # new scheduler step
        scheduler.step(val_acc)

        # save state added
        if val_acc >= best_val_acc:
            best_val_acc = val_acc
            torch.save(model.state_dict(), 'best_model.pth')
            print(f"----> Best model saved ---> Val Acc: {val_acc:.2f}%")
            patience_counter = 0
        else:
            patience_counter = patience_counter + 1

        # early stopping
        if patience_counter >= early_stop_patience:
            break

        print()

    return train_losses, val_accuracies

train_losses, val_accuracies = train(
    model,
    train_loader,
    val_loader,
    criterion,
    optimizer,
    scheduler,
    num_epochs=100
)


# Step 5

# NEW TOM + NATE BLOCK

def evaluate(model, val_loader):
    model.eval()
    correct = 0
```

```
    total = 0

    with torch.no_grad():
        for inputs, labels in val_loader:
            inputs = inputs.to(device)
            labels = labels.to(device)
            outputs = model(inputs)
            predictions = torch.max(outputs, 1)[1]
            total = total + labels.size(0)
            correct = correct + (predictions == labels).sum().item()

    accuracy = 100.0 * correct / total
    print(f'Val Accuracy: {accuracy:.2f}%')
    return accuracy

model.load_state_dict(torch.load('best_model.pth'))
evaluate(model, val_loader)

Please don't make any change after this line. The only parameters you may
 ↪  modify are those within the "test_transform" function.

import os
from PIL import Image
class CustomImageDataset(torch.utils.data.Dataset):
    def __init__(self, folder_path, transform=None):
        self.folder_path = folder_path
        self.transform = transform
        self.image_paths = [os.path.join(folder_path, filename) for filename
         ↪  in os.listdir(folder_path) if filename.endswith(('png', 'jpg',
         ↪  'jpeg'))]

    def __len__(self):
        return len(self.image_paths)

    def filename2index(self, filename):
        return os.path.basename(filename).replace('.jpg', '')

    def __getitem__(self, idx):
        img_path = self.image_paths[idx]
        img = Image.open(img_path).convert('RGB')
        if self.transform:
            img = self.transform(img)
```

```python
        return img, self.filename2index(img_path)

test_folder = '/content/drive/MyDrive/test_set'
test_transform = transforms.Compose([
    transforms.Resize((64, 64)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
test_dataset = CustomImageDataset(test_folder, transform=test_transform)
test_loader = DataLoader(test_dataset, batch_size=1, shuffle=False)

class_to_idx = train_dataset.class_to_idx
idx_to_class = {v: k for k, v in class_to_idx.items()}

# Make predictions
def evaluate_model(model, test_loader, idx_to_class):
    all_predictions = {}
    with torch.no_grad():
        for inputs, index in test_loader:
            inputs = inputs.to(device)

            outputs = model(inputs)
            _, predicted = torch.max(outputs, 1)
            predicted_class = predicted.item()
            predicted_class_name = idx_to_class[predicted_class]
            all_predictions[index[0]] = predicted_class_name

    return all_predictions

predictions = evaluate_model(model, test_loader, idx_to_class)
with open('predictions.json', 'w') as json_file:
    json.dump(predictions, json_file, indent=4)

print("Evaluation completed and predictions saved.")

# you may need to install thop when you first run this code
!pip install thop

# Compute FLOPs using thop
import thop
input_tensor = test_dataset[0][0].unsqueeze(0).to(device) # must have exact
↪ same size of the data input (batch, channel, height, width) and be on the
↪ same device as the model
```

```python
flops, params = thop.profile(model, inputs=(input_tensor,))
print(f"FLOPs: {flops}")
print(f"Number of Parameters: {params}")
flops_and_params = {
    "FLOPs": flops,
    "Parameters": params
}

output_json_path = 'flops_and_params.json'

with open(output_json_path, 'w') as json_file:
    json.dump(flops_and_params, json_file, indent=4)

print(f"FLOPs and parameters have been saved to {output_json_path}")
```

```
Val Accuracy: 69.94%
```

```
69.94328922495274
```

---

2 Comparing Vanilla RNN with Variants in Sequence Modeling [20 points]

You will implement and train three different neural networks for sequence modeling: a Vanilla RNN (a simple RNN with shared weights), and two variants of a NN with a similar architecture to the Vanilla RNN but which do not share weights.

You will compare their performance on a sequence prediction task and analyze the differences between them.

Prediction task: this is a many-to-one regression task, i.e., a sequence of inputs is used for predicting a single output. In particular, the i-th input sequence $\mathcal{X}^{(i)} = (x_1^{(i)}, ..., x_{l_i}^{(i)})$ has length l_i, where $x_j^{(i)} \in \mathbb{R}^{10}$, the i-th output is a scalar $y^{(i)} \in \mathbb{R}$.

Dataset: You will use a synthetic dataset containing sequences of variable lengths stored in the zip file homework5_question2_data.zip. Each sequence consists of input features and corresponding target values. The sequences are generated such that they represent a time-dependent process. Note that l_i may be different than l_j for $i \neq j$. So the (pickled) numpy object X is actually a list of sequences.

Tasks:

1. (4 points) Implement a Vanilla RNN: Implement a Vanilla RNN architecture (needless to say, weights are shared across time steps). A pytorch starter code is provided in homework4_starter.py. Important: You are not allowed to use an RNN layer implementation from any library.

2. (4 points) Implement a NN with Sequences Truncated to the Same Length: Implement a NN where sequences are truncated to have the same length before training. In other words, if the shortest sequence in the dataset has length L, all sequences should be truncated to length L before training.

3. (4 points) Implement a NN with Sequences Padded to the Same Length: Implement another variant of NN where sequences are padded to have the same length before training. Use appropriate padding techniques to ensure that all sequences have the same length, and implement a mechanism to ignore the padding when computing loss and predictions.

4. Train and Compare the Models:

   (a) (1 point) Train all three models (Vanilla RNN, Truncated NN, Padded NN) on the provided dataset.
   (b) (1 point) Use a suitable loss function for sequence prediction tasks, such as mean squared error (MSE) or cross-entropy.
   (c) (1 point) Train each model for a fixed number of epochs or until convergence.
   (d) (1 point) Monitor and record performance metrics, such as training loss, on a validation set during training.

5. Evaluate and Compare the Models:

   (a) (1 point) Evaluate the trained models on a separate test dataset.
   (b) (2 point) Compare the performance of the three models in terms of MSE, convergence speed, and overfitting tendencies.
   (c) (1 point) Analyze the results and discuss the advantages and disadvantages of each approach in terms of modeling sequences with varying lengths.

Additional Information: You can choose the specific hyperparameters for your models, such as the number of hidden units, learning rate, batch size, and sequence length. Feel free to use any deep learning framework or library you are comfortable with, and provide clear code documentation. Note: Be sure to clearly explain your implementation, provide code comments, and present your results in a well-organized manner in the report.

Answer

```
# connect to google drive
from google.colab import drive
drive.mount('/content/drive/')

path = '/content/drive/MyDrive/'

import torch
import torch.nn as nn
import torch.optim as optim
from torch.nn.utils.rnn import pad_sequence
import numpy as np
import random
import os
os.chdir(path)


# load training and test data
def loadData():
    X_train = np.load('X_train.npy',allow_pickle=True)
    y_train = np.load('y_train.npy',allow_pickle=True)
    X_test = np.load('X_test.npy',allow_pickle=True)
    y_test = np.load('y_test.npy',allow_pickle=True)

    X_train = [torch.Tensor(x) for x in X_train]  # List of Tensors
 ↪  (SEQ_LEN[i],INPUT_DIM) i=0..NUM_SAMPLES-1
    X_test = [torch.Tensor(x) for x in X_test]  # List of Tensors
 ↪  (SEQ_LEN[i],INPUT_DIM)
    y_train = torch.Tensor(y_train) # (NUM_SAMPLES,1)
    y_test = torch.Tensor(y_test) # (NUM_SAMPLES,1)

    return X_train, X_test, y_train, y_test


# Define a Vanilla RNN layer by hand
class RNNLayer(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(RNNLayer, self).__init__()
        self.hidden_size = hidden_size
        self.input_size = input_size
        self.W_xh = nn.Parameter(torch.randn(input_size, hidden_size) * 0.01)
        self.W_hh = nn.Parameter(torch.randn(hidden_size, hidden_size) *
         ↪  0.01)
```

```
        self.activation = torch.tanh

    def forward(self, x, hidden):
        hidden = self.activation(x @ self.W_xh + hidden @ self.W_hh)
        return hidden

# Define a sequence prediction model using the Vanilla RNN
class SequenceModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SequenceModel, self).__init__()
        self.hidden_size = hidden_size
        self.rnn = RNNLayer(input_size, hidden_size)
        self.linear = nn.Linear(hidden_size, output_size)

    def forward(self, input_seq, seq_lengths):
        batch_size = len(input_seq)
        last_hidden = torch.zeros(batch_size, self.hidden_size,
 ↪  device=device)

        for b in range(batch_size):
            hidden = torch.zeros(self.hidden_size, device=device)

            seq_length =  seq_lengths[b]

            for t in range(seq_length):
                hidden = self.rnn(input_seq[b][t], hidden)

            # Store the last hidden state in the output tensor
            last_hidden[b] = hidden

        output = self.linear(last_hidden)
        return output

# Define a sequence prediction model for fixed length sequences, BUT NO
 ↪  SHARED WEIGHTS
class SequenceModelFixedLen(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, seq_len):
        super(SequenceModelFixedLen, self).__init__()
        self.hidden_size = hidden_size
        self.seq_len = seq_len
        self.rnn_layers = nn.ModuleList([RNNLayer(input_size, hidden_size)
         ↪  for _ in range(seq_len)])
```

```
        self.linear = nn.Linear(hidden_size, output_size)

    def forward(self, input_seq, seq_lengths):
        batch_size = len(input_seq)
        last_hidden = torch.zeros(batch_size, self.hidden_size,
 ↪  device=device)

        for b in range(batch_size):
            hidden = torch.zeros(self.hidden_size, device=device).to(device)

            seq_length = min(self.seq_len, seq_lengths[b])
            for t in range(seq_length):
                hidden = self.rnn_layers[t](input_seq[b][t], hidden)

            # Store the last hidden state in the output tensor
            last_hidden[b] = hidden

        output = self.linear(last_hidden)
        return output



class PaddedModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, seq_len_max):
        super(PaddedModel, self).__init__()
        self.hidden_size = hidden_size
        self.seq_len_max = seq_len_max
        self.rnn_layers = nn.ModuleList([RNNLayer(input_size, hidden_size)
         ↪  for _ in range(seq_len_max)])
        self.linear = nn.Linear(hidden_size, output_size)

    def forward(self, padded_batch, lengths):
        B, T, _ = padded_batch.shape
        device = padded_batch.device

        hidden = [torch.zeros(self.hidden_size, device=device) for _ in
 ↪  range(B)]

        for t in range(T):
            for b in range(B):
                if t < lengths[b]:
```

```
                    hidden[b] = self.rnn_layers[t](padded_batch[b, t],
 ↪  hidden[b])

        last_hidden = torch.stack(hidden, dim=0)
        return self.linear(last_hidden)



# Define hyperparameters and other settings
input_size = 10  # Replace with the actual dimension of your input features
hidden_size = 64
output_size = 1
num_epochs = 10
learning_rate = 0.001
batch_size = 32


# load data
X_train, X_test, y_train, y_test = loadData()
device = y_train.device

# Create the model using min length input
seq_lengths = [seq.shape[0] for seq in X_train]


all_indices = np.arange(len(X_train))
np.random.shuffle(all_indices)

train_cutoff = int(0.8 * len(all_indices))
train_indices = all_indices[:train_cutoff]
val_indices   = all_indices[train_cutoff:]


X_train_split = []
for i in train_indices:
    X_train_split.append(X_train[i])
y_train_split = y_train[train_indices]


X_val_split = []
for i in val_indices:
    X_val_split.append(X_train[i])
```

```
y_val_split = y_train[val_indices]


# Training loop
def train(model, num_epochs, lr, batch_size, X_train, y_train, seq_lengths):
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=lr)
    print("training!")
    for epoch in range(num_epochs):

        print("epoch ", epoch)

        for i in range(0, len(X_train), batch_size):
            inputs = X_train[i:i+batch_size]
            targets = y_train[i:i+batch_size]
            lengths = seq_lengths[i:i+batch_size]

            #GPU related stuff to ensure it picks the right device
            inputs  = [x.to(device) for x in inputs]
            targets = targets.to(device)

            optimizer.zero_grad()
            outputs = model(inputs, lengths)
            loss = criterion(outputs, targets)
            loss.backward()
            optimizer.step()
        MSE_val = mse_padded(model, X_val_split, y_val_split)
        print("MSE ", MSE_val)
        print(loss)
    return model

def train_padded(model, num_epochs, lr, batch_size, X_train, y_train):
    model.train()
    optimizer = optim.Adam(model.parameters(), lr=lr)
    criterion = nn.MSELoss()

    print("training padded!")

    for epoch in range(num_epochs):
        print("epoch ",epoch)
        for i in range(0, len(X_train), batch_size):
            batch = X_train[i:i+batch_size]
```

```
            targets = y_train[i:i+batch_size].to(device)

            lengths = [len(s) for s in batch]
            padded = pad_sequence(batch, batch_first=True).to(device)
            optimizer.zero_grad()
            outputs = model(padded, lengths)
            loss = criterion(outputs, targets)
            loss.backward()
            optimizer.step()
        MSE_val = mse_padded(model, X_val_split, y_val_split)
        print("Padded MSE ", MSE_val)
        print(loss.item())


def mse(model, inputs, y):
    model.eval()
    crit = nn.MSELoss()
    preds = []
    bs = 64
    lengths = []

    for x in inputs:
        lengths.append(len(x))

    for i in range(0, len(inputs), bs):

        batch = []
        for x in inputs[i:i+bs]:
            batch.append(x.to(device))

        lens  = lengths[i:i+bs]
        preds.append(model(batch, lens))

    preds = torch.cat(preds, dim=0)

    return crit(preds, y.to(device)).item()



def mse_padded(model, inputs, y):
    model.eval()
    crit = nn.MSELoss()
```

```
    preds = []
    bs = 64

    lengths = []
    for x in inputs:
        lengths.append(len(x))

    for i in range(0, len(inputs), bs):
        batch = []
        for x in inputs[i:i+bs]:
            batch.append(x.to(device))

        lens = lengths[i:i+bs]
        padded = pad_sequence(batch, batch_first=True)

        preds.append(model(padded, lens))

    preds = torch.cat(preds, dim=0)
    return crit(preds, y.to(device)).item()



# initialize and train Vanilla RNN
if __name__ == "__main__":

    X_train, X_test, y_train, y_test = loadData()

    if torch.cuda.is_available():
        device = torch.device("cuda") # pick my gpu
        print("cuda selected!")
    else:
        device = torch.device("cpu")
        print("cpu selected. no visible gpu")



    seq_lengths_tr  = [len(x) for x in X_train_split]
    seq_lengths_val = [len(x) for x in X_val_split]

    print("Vanilla RNN . . . . .")
    vanilla = SequenceModel(input_size, hidden_size, output_size).to(device)
    train_vanilla_RNN =train(vanilla, num_epochs, learning_rate, batch_size,
 ↪  X_train, y_train, seq_lengths)
```

```
print ("fixed length truncated model....")


Lmin = min(seq_lengths)
X_train_trunc = []

for x in X_train:
     truncated_seq = x[:Lmin]
     X_train_trunc.append(truncated_seq)

seq_lengths_trunc = [Lmin] * len(X_train_trunc)


trunc = SequenceModelFixedLen(input_size, hidden_size, output_size,
↪ seq_len=Lmin).to(device)
Train_trunc = train(trunc, num_epochs, learning_rate, batch_size,
↪ X_train_trunc, y_train, seq_lengths_trunc)



print("padded model ....")
Lmax = max(seq_lengths)
padded_model = PaddedModel(input_size, hidden_size, output_size,
↪ seq_len_max=Lmax).to(device)
train_padded(padded_model, num_epochs, learning_rate, batch_size,
↪ X_train, y_train)


print("testing each")
vanilla_test = mse(vanilla, X_test, y_test)

trunc_test = []

for x in X_test:
    truncated_seq = x[:Lmin]
    trunc_test.append(truncated_seq)

test_trunc   = mse(trunc, trunc_test, y_test)

padded_test  = mse_padded(padded_model, X_test, y_test)
```

```
    print("vanilla test!!  ", vanilla_test , "truncated test!! " , test_trunc , "Padded Test!! ", padded_test)

print("vanilla test!!  ", vanilla_test , "truncated test!! " , test_trunc , "Padded Test!! ", padded_test)


#############################################################################################

# initialize and train Sequential NN fixing #timesteps to the minimum sequence length

# initialize and train Sequential NN fixing #timesteps to the maximum sequence length
# NOTE: it is OK to use torch.nn.utils.rnn.pad_sequence; make sure to set parameter batch_first correctly
```

```
cpu selected. no visible gpu
Vanilla RNN . . . . .
training!
epoch  0
MSE  0.019275978207588196
tensor(0.0193, grad_fn=<MseLossBackward0>)
epoch  1
MSE  0.012702079489827156
tensor(0.0139, grad_fn=<MseLossBackward0>)
epoch  2
MSE  0.008365921676158905
tensor(0.0088, grad_fn=<MseLossBackward0>)
epoch  3
MSE  0.005394916981458664
tensor(0.0056, grad_fn=<MseLossBackward0>)
epoch  4
MSE  0.0033081411384046078
tensor(0.0036, grad_fn=<MseLossBackward0>)
epoch  5
MSE  0.0019416179275140166
tensor(0.0023, grad_fn=<MseLossBackward0>)
epoch  6
MSE  0.0011280523613095284
```

```
tensor(0.0014, grad_fn=<MseLossBackward0>)
epoch  7
MSE  0.0006615080637857318
tensor(0.0009, grad_fn=<MseLossBackward0>)
epoch  8
MSE  0.00040001189336180687
tensor(0.0006, grad_fn=<MseLossBackward0>)
epoch  9
MSE  0.0002588455390650779
tensor(0.0004, grad_fn=<MseLossBackward0>)
fixed length truncated model....
training!
epoch  0
MSE  0.01090302038937807
tensor(0.0119, grad_fn=<MseLossBackward0>)
epoch  1
MSE  0.009235186502337456
tensor(0.0101, grad_fn=<MseLossBackward0>)
epoch  2
MSE  0.008770066313445568
tensor(0.0096, grad_fn=<MseLossBackward0>)
epoch  3
MSE  0.008530529215931892
tensor(0.0086, grad_fn=<MseLossBackward0>)
epoch  4
MSE  0.008626209571957588
tensor(0.0081, grad_fn=<MseLossBackward0>)
epoch  5
MSE  0.008597632870078087
tensor(0.0083, grad_fn=<MseLossBackward0>)
epoch  6
MSE  0.008535130880773067
tensor(0.0087, grad_fn=<MseLossBackward0>)
epoch  7
MSE  0.008532710373401642
tensor(0.0086, grad_fn=<MseLossBackward0>)
epoch  8
MSE  0.00852106511592865
tensor(0.0084, grad_fn=<MseLossBackward0>)
epoch  9
MSE  0.008514193817973137
tensor(0.0084, grad_fn=<MseLossBackward0>)
padded model ....
```

```
training padded!
epoch  0
Padded MSE  0.023514097556471825
0.034565091133117676
epoch  1
Padded MSE  0.012983511202037334
0.017391806468367577
epoch  2
Padded MSE  0.008208412677049637
0.01027042418718338
epoch  3
Padded MSE  0.005899517796933651
0.006686745211482048
epoch  4
Padded MSE  0.004288738127797842
0.0052238949574530125
epoch  5
Padded MSE  0.003118603490293026
0.0042826831340789795
epoch  6
Padded MSE  0.002316394355148077
0.0033069662749767303
epoch  7
Padded MSE  0.0016895529115572572
0.0025147662963718176
epoch  8
Padded MSE  0.0012454017996788025
0.0019126953557133675
epoch  9
Padded MSE  0.0009358171373605728
0.001503477804362774
testing each
vanilla test!!   0.00036904800799675286 truncated test!!
0.009197115898132324 Padded Test!!  0.00724533898755908
```

Q5(b)

Vanilla RNN: best test MSE 0.000369; 25x better than truncated (0.00920), 20x better than padded (0.00725). Fastest convergence—MSE $<$ 0.001 by epoch 6. Truncated plateaued = 0.0085, no further gain. Padded intermediate—trained to = 0.001. No overfitting; validation loss tracked training loss.

Q5(c)

Vanilla RNN best, shared weights, true sequence lengths are likley the reason. Truncated worst, cuts data beyond min length, loses long-range info. Padded moderate, keeps data but zero-padding adds noise. Per-timestep RNNs handle uneven inputs. Truncated/padded models larger—separate weights per timestep, harder to train with limited data.

---

3 Fine-tune a DistilBERT model [20 points + 2 bonus points]

In this project, you will first train a classification head using a pre-trained DistilBERT model on a dataset of social media tweets to classify tweets as containing medical information or not. You are provided with a dataset of social media tweets, where each tweet is labeled as either containing medical information (class 1) or not containing medical information (class 0).

The preprocessing of the dataset, by tokenizing the tweets and converting them into a format suitable for DistilBERT, is already provided in the starter code: https://colab.research.google.com/drive/17syAcTav5Wtq-n_Rs3P1cQ10szIjLlQS?usp=sharing

Useful documentation for this question can be found here: https://huggingface.co/docs/transformers/index https://huggingface.co/docs/transformers/training https://huggingface.co/docs/transformers/tasks/sequence_classification

You will need to make the following changes to the existing code:

Q1: (2 points) Add 4+ relevant arguments to the parser. Hint: Check how args is used within load_pretrained_and_finetune and think about which other arguments should be added. Note: while argparse is designed to read arguments from the command line, it is currently adapted to work with a jupyter notebook by passing arguments to parser.parse_args as a list.

Q2: (2 points) Split the code in train, val and eval (test) sets stratified by classes.

Q3: (2 points) Convert sets to HuggingFace Dataset and tokenize using function tokenize_batch.

Q4: (6 points) Implement grid search for at least one hyperparameter by training a classification head for pre-trained DistilBERT model on the training set and evaluate its performance on the validation set. You can use DistilBertForSequenceClassification.from_pretrained from the transformers library to load the pre-trained model. Note: DO NOT train the entire model, only the classification head. You can initially freeze all parameters except the classification head by setting requires_grad=False for all parameters in the base model.

Q5: (6 points) Run the final training on train+val with best hyperparameters.

Q6: (2 points) Based on the item above, discuss the performance of the model, any challenges faced during fine-tuning, and potential improvements that can be made to further improve accuracy.

Q7: (BONUS: 2 points) Apply a principled change to your code in order to achieve F1-macro > 0.50. Explain what you did and why you did it.

Answer

---

4 NOT PART OF HW4: Tensor Shapes in a Transformer Layer [0 points]

The figura above shows the transformer layer. The input size of the transformer layer is [10, 90, 20] (where 10 represents batch size, 90 represents sequence length, and 20 represents hidden size). We consider 5 attention heads in this attention layer. The shape of the (combined) projection matrices is $H \times H$, which are used to project the input data to Q (query), K (key), V (value). Please compute the size of the each output, including: 1. The shape of Q 2. The shape of K 3. The shape of V 4. The shape of Q for each head 5. The shape of K for each head 6. The shape of V for each head 7. The shape of the attention map (output of softmax) 8. The shape of Dropout-1's output 9. The shape of Output 10. The shape of Dropout-2's output 11. Total number of the parameters in this transformer layer

For this homework, you are encouraged to experiment with various models and training strategies. To achieve a full score, your model must achieve at least 60% accuracy on the test set.

Submission

Submit one PDF file that includes your notes for the theoretical problems (scanned or typed) and screenshots of your code for the programming problems. All material in the submitted PDF must be presented in a clear and readable format.