یادگیری عمیق

تمرین پنجم

استاد درس: دکتر محمدی

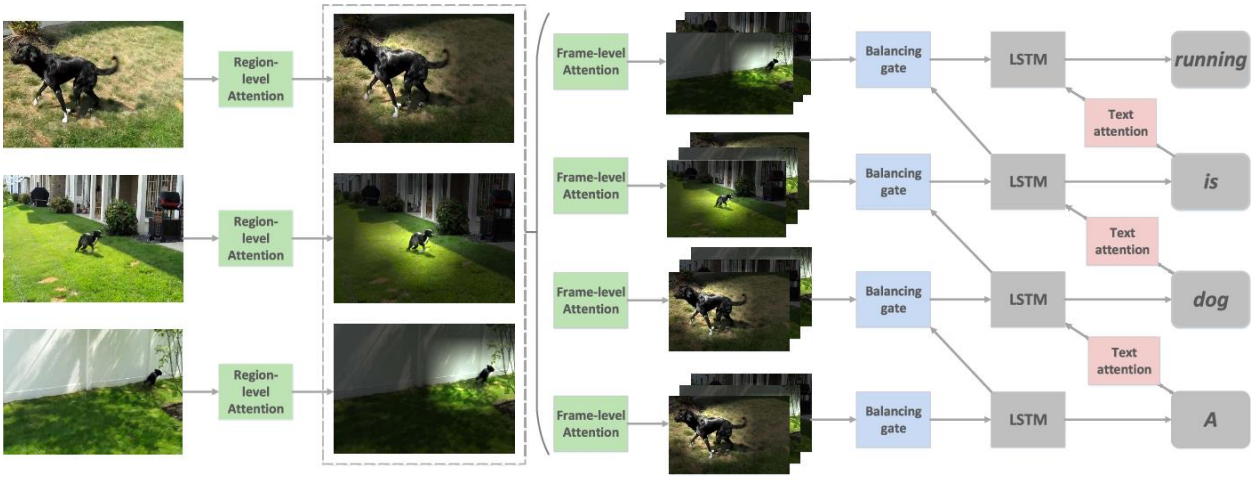
مهسا موفق بهروزی

بهار ۱۴۰۲

سوال یک)

الف) تکنیک video captioning، تکنیکی‌ست که اطلاعات تصویری (visual) و متنی را برای توصیف محتوای یک ویدئو ترکیب می‌کند. CAM-RNN مدلی است که برای video captioning پیشنهاد شده است که مکانیزم توجه مشترک (CAM) را با شبکه عصبی بازگشتی (RNN) ترکیب می‌کند. CAM برای encode کردن مرتبط‌ترین ویژگی‌های بصری و متنی با کپشن استفاده می‌شود و RNN به عنوان decoder برای تولید کپشن کلمه به کلمه به کار می‌رود.

معماری CAM-RNN در شکل زیر نمایش داده شده است:



فرآیند تولید کپشن ویدیو:

- تولید کپشن ویدئو با RNN

از LSTM به عنوان تولید کننده‌ی کپشن استفاده شده است. حالت پنهان (hidden state) در LSTM بر اساس ورودی فعلی و حالت نهان قبلی محاسبه می‌شود. احتمال پیش‌بینی هر کلمه بر اساس ویژگی‌های بصری، متنی و حالت نهان محاسبه شده و یک گیت متعادل کننده میزان تاثیر ویژگی بصری را تنظیم می‌کند.

- استخراج ویژگی بصری توسعه یافته توجه (Attention Extended Visual Feature Extraction)

ماژول توجه بصری در این رویکرد دارای دو لایه است: توجه در سطح منطقه (region) و توجه در سطح فریم. توجه در سطح منطقه بر روی مناطق برجسته در هر فریم متمرکز می‌شود تا ویژگی‌های فریم را استخراج کند. ویژگی‌های فریم با محاسبه مجموع وزنی (weighted sum) ویژگی‌های منطقه به دست می‌آیند، که در آن وزن توجه (attention weight) هر منطقه بر اساس ویژگی منطقه آن و وزن توجه منطقه مربوطه در فریم قبلی تعیین می‌شود. توجه در سطح فریم با توجه به زیرمجموعه‌ای از ویژگی‌های فریم که بیشترین ارتباط را با کپشن ویدیو دارند، ویژگی بصری ویدیو را encode می‌کند. وزن توجه هر فریم بر اساس امتیاز مربوط به آن محاسبه می‌شود، که با استفاده از ویژگی‌های کدگذاری شده قبلی به دست می‌آید و برای تعیین ویژگی بصری در هر مرحله زمانی نرمال می‌شود. این رویکرد تداخل ویژگی‌های منطقه نامربوط را کاهش می‌دهد.

- استخراج ویژگی متنی توسعه یافته توجه (Attention Extended Text Feature Extraction)

هدف ماژول توجه متن در این رویکرد محاسبه ویژگی متن E_t در هر مرحله زمانی است. این ماژول بر روی عباراتی که توسط کلماتی که قبلاً تولید شده‌اند برای توجه به مرتبط‌ترین اطلاعات متنی در طول تولید کلمه عمل می‌کند. ویژگی‌های عبارت، بر اساس کلمات تولید شده قبلی با در نظر گرفتن عبارات unigram، bigram و trigram محاسبه می‌شوند. ویژگی متنی E_t با محاسبه مجموع وزنی از ویژگی‌های عبارت به دست می‌آید، که در آن وزن توجه هر عبارت با بردار ویژگی آن و اطلاعات تاریخی ذخیره شده در حالت پنهان تولید کننده کپشن، با استفاده از توابع خاص و پارامترهای آموزشی تعیین می‌شود.

- گیت متعادل‌کننده (Balancing Gate)

گیت تعادل بخشی از رویکرد پیشنهادی برای زیرنویس ویدیویی است که به تعادل تأثیر ویژگی‌های بصری و متن کمک می‌کند و تاثیر ویژگی‌های بصری را هنگام تولید کلمات غیر بصری کاهش می‌دهد. گیت متعادل کننده توسط یک تابع سیگموئید محاسبه می‌شود که ضریب ویژگی بصری را در محدوده $[۰,۱]$ نشان می دهد که مقادیر نزدیک به صفر، مربوط به کلمات غیر بصری و نزدیک به یک مربوط به کلمات بصری‌ست.

ب) از این رویکرد می‌توان برای تولید کپشن برای تصاویر، به نحوی که یک تصویر را یک ویدئو با یک فریم در نظر گرفت، استفاده کرد.

سوال دو)

گیت reset در GRU وظیفه‌ی تعیین اینکه چه مقدار از حالت نهان قبلی باید فراموش یا Reset شود را برعهده دارد. با حذف گیت reset، مدل توانایی reset یا حذف اطلاعات از حالت پنهان قبلی را از دست می‌دهد. که ممکن است چالش‌های زیر را داشته باشد:

- وابستگی‌های بلند مدت (Long-term dependencies)

گیت reset به مدل GRU کمک می‌کند تا وابستگی‌های بلندمدت را ثبت کند و به آن اجازه می‌دهد تصمیم بگیرد کدام بخش از حالت پنهان قبلی را فراموش کند. با حذف آن، مدل ممکن است در حفظ اطلاعات در توالی‌های طولانی مشکل داشته باشد.

- Overfitting

گیت reset به مدل کمک می‌کند تا مقدار اطلاعات منتقل شده را در حالت پنهان تنظیم کند. با حذف آن، حالت پنهان ممکن است اطلاعات غیرضروری یا نویزی را از مراحل قبلی حفظ کند، که به طور بالقوه می‌تواند منجر به overfitting شود.

- Gradient explosion/vanishing

گیت‌های بازگشتی برای کاهش مشکل گرادیان (ناپدید شدن یا انفجار) طراحی شده‌اند و گیت reset یکی از اجزایی است که به کنترل جریان گرادیان‌ها در طول زمان کمک می‌کند و حذف آن ممکن است مدل را نسبت به این مسائل مربوط به گرادیان مستعدتر کند.

ب) برای تغییر معادلات GRU به طوری که خروجی در مرحله زمانی t فقط به یک ورودی خاص در مرحله زمانی t_prime (که t_prime < t) بستگی داشته باشد، می‌توان گیت ریست را تا مرحله زمانی t_prime، صفر و بعد از آن ۱ تنظیم کرد. همچنین، می‌توان گیت آپدیت را در مرحله زمانی t_prime، یک و در تمام مراحل زمانی دیگر ۰ تنظیم کرد.

سوال سه)

تعداد عملیات در softmax:

اگر N کلاس داشته باشیم، N عمل exp، یک عمل جمع، N عمل تقسیم

تعداد عملیات در sigmoid:

یک عمل exp، یک عمل جمع، یک عمل تقسیم

تعداد کلاس‌ها = ۲۰

در softmax: ۲۰ exp + ۱ جمع + ۲۰ تقسیم

در sigmoid: ۱ exp + ۱ جمع + ۱ تقسیم

پس ۱۹ exp و ۱۹ تقسیم کاهش یافته است.

ب) در RNN، هر دو لایه‌ی دنس و امبدینگ برای جنرالیزیشن استفاده می‌شود. لایه دنس یک لایه fully connected است که پارامترهای بسیار زیادی دارد و توانایی یادگیری مسائل غیرخطی را داراست و برای داده‌های numericalکاربرد دارد. لایه امبدینگ برای داده‌های categorical استفاده شده و ورودی را به یک فضای حالت با ابعاد کمتر map می‌کند به نحوی که داده‌های مشابه به یکدیگر نزدیک‌ترند.

هر دو، هم توانایی یادگیری و هم جنرالیزیشن را دارند اما در کاربرد متفاوت‌اند. مثلا در nlp استفاده از امبدینگ بهتر است؛ چون کلماتی داریم که توالی آن‌ها یکسری rule مشخص دارد. (اینطور نیست که با یک احتمالی فعل قبل از فاعل باشد و با یک احتمال دیگر فعل بعد از فاعل؛ همیشه فعل بعد از فاعل خواهد بود). در مواردی که time series می‌باشد ولی مثل زبان نیستند مثلا پیش‌بینی آب و هوا، نمی‌توان به صورت شرطی بررسی کرد و با یک احتمالی ممکن است چندین اتفاق بیفتد؛ بنابراین استفاده از دنس بهتر است. (مثلا اگر روز سوم نسبت به روز دوم دما بالاتر بود، امروز باران می‌بارد یا نه)

سوال چهار) الف، ب، ج)

```
# write dataloader
from torchtext.vocab import build_vocab_from_iterator
tokenizer = get_tokenizer("basic_english")
from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer
porter = PorterStemmer()
import nltk
nltk.download('stopwords')
stop_words = stopwords.words('english')
tokenToIndex = {"<PAD>": 0, "<SOS>": 1, "<EOS>": 2, "<UNK>": 3}
indexToToken = {0: "<PAD>", 1: "<SOS>", 2: "<EOS>", 3: "<UNK>"}
class MyDataset(Dataset):
  def __init__(self, img_dir, annotation_dir, transformer,threshold = 5):
    super(MyDataset, self).__init__()

    self.transformer = transformer;

    captions = open(annotation_dir)
    global tokenToIndex,indexToToken,stop_words,porter


    captions = [line.strip().split(",") for line in captions]
    captions = self.captions = captions[1:]
    # image name = captions[0]    image caption = captions[1]
    frequencies = {}
    for caption in captions:
      tokens = tokenizer(caption[1])
      for token in tokens:
        token = porter.stem(token)
        if token in frequencies.keys():
          frequencies[token] += 1
        else:
          frequencies[token] = 1
```

```
    vocabCount = 4

    for token in frequencies.keys():
      if token.isalpha() and token not in stop_words:
        tokenToIndex[token] = vocabCount
        indexToToken[vocabCount] = token
        vocabCount += 1


      # if frequencies[token] >= threshold:
      #     tokenToIndex[token] = vocabCount
      #     indexToToken[vocabCount] = token
      #     vocabCount += 1
    print(vocabCount)
    self.vocabCount= vocabCount
    ##########################################
    # your code here, you can add new parameter in constructor
    ##########################################
```

در این بخش ابتدا فایل caption.txt را می‌خوانیم و در هر خط اسم عکس و کپشن مورد نظر را که با , جدا شده‌اند از هم جدا می‌کنیم. سپس با یک لوپ بر روی تمام کپشن‌ها خط به خط کپشن‌ها را بررسی کرده و ابتدا هر کپشن را توکنایز می‌کنیم سپس هر توکن را ریشه‌یابی می‌کنیم تا کلمات هم‌ریشه چند بار به عنوان کلمات مختلف در دیتاست ما وجود نداشته باشند. سپس ریشه‌های یونیک را در آبجکت frequencies ذخیره می‌کنیم. اگر کلمه‌ای مجدد در کپشن‌های دیگر هم تکرار شود تعداد تکرار آن کپشن را در آبجکت ذکر شده یک واحد افزایش می‌دهیم. سپس یک لوپ بر روی تمام آیتم‌های موجودی در آبجکت frequencies می‌نویسیم و به شرط آن‌که کلمه‌ی ریشه‌یابی شده جزوی از استاپ‌وردهای انگلیسی نباشد اطلاعات را به ۲ شکل یکبار به شکل هش‌مپی از کلمات به ایندکس و یک‌بار به صورت هش‌مپی از ایندکس‌ها به کلمات ذخیره می‌کنیم. لازم به ذکر است که از قبل نیز ۴ تا کلمه‌ی ساختگی برای شروع و پایان و توکن ناشناخته و توکن پدینگ در نظر گرفته‌ایم و به همین دلیل vocabCount که برای مپ کردن توکن‌ها به اعداد هست از ۴ شروع شده و با هر بار اجرای لوپ یک واحد افزایش می‌یابد.

```
def __len__(self):
  return len(self.captions)
  ##########################################
  #your code here, you should be return size of vocabulary here
  ##########################################


def stringfy(self, array):
  string = ""
  for item in array:
    if item in indexToToken.keys():
      string = string +' ' +indexToToken[item]
    else:
      string = string +' ' + '<UKN>'+item+' '
  return string
def vocab_size(self):
  return self.vocabCount
```

در بخش __len__ تعداد آیتم‌های درون دیتاست که شامل تعداد آیتم آرایه‌ی کپشن می‌باشد را برمی‌گردانیم و یک تابع stringfy هم برای تبدیل آرایه‌ی ایندکس‌ها به یک متن تعریف کرده‌ایم که جلوتر از آن استفاده خواهیم کرد.

```
def __getitem__(self, index):
  imgs = Image.open("/content/flickr8k/images/" + self.captions[index][0]).convert('RGB')
  imgs = imgs.resize((64,64))
  res = [1] #start of string
  for token in tokenizer(self.captions[index][1]):
    if token in tokenToIndex.keys():
      res.append(tokenToIndex[token])
  res.append(2) #end of string
  return self.transformer(imgs) , torch.tensor(
      res
  )
 ##########################################
 #your code here, you should be retrun image and caption of that, please attention caption of image tokenized and cleaned
 ##########################################
```

در این بخش به ازای هر آیتم از دیتاست، عکس مربوط به آن را می‌خوانیم. تصویر را با هدف بیشتر شدن سرعت ترین، ریسایز می‌کنیم و بعد از انجام عمل ترنسفورم و تبدیل به تنسور، ریترن می‌کنیم. در مورد متن‌ها هم هر کپشن را خوانده و توکنایز می‌کنیم و توکن‌هایی از آن که ایندکس مربوط به آن‌ها از قبل محاسبه شده است را انتخاب کرده و آن‌ها را در یک آرایه پشت هم می‌چینیم و نهایتا آن آرایه را ترنسفورم کرده و برمی‌گردانیم.

```
dataset = MyDataset("/content/flickr8k/images", "/content/flickr8k/captions.txt",  transforms.Compose([
        transforms.ToTensor(),
    ]))

class BatchArrange:
    def __init__(self, pad_idx):
        self.pad_idx = pad_idx

    def __call__(self, batch):
        imgs = []
        for item in batch:
          imgs.append(item[0].unsqueeze(0))
        imgs = torch.cat(imgs, dim=0)
        targets = [item[1] for item in batch]
        targets = pad_sequence(targets, batch_first=False, padding_value=self.pad_idx)
        return imgs, targets

#####################################################
#your code here
#####################################################

train_set, val_set, test_set = torch.utils.data.random_split(dataset, [len(dataset)-200, 195,5])
train_loader = torch.utils.data.DataLoader(train_set, shuffle=True,batch_size=5,collate_fn=BatchArrange(pad_idx=0))
val_loader = torch.utils.data.DataLoader(val_set,shuffle=True,batch_size=5,collate_fn=BatchArrange(pad_idx=0))
test_loader = torch.utils.data.DataLoader(test_set,shuffle=True,batch_size=5,collate_fn=BatchArrange(pad_idx=0))
```

در این بخش یک نمونه از کلاس دیتاست‌مان ایجاد می‌کنیم و سپس دیتاها را برای سه بخش ترین و تست و ولیدیشن به نسبت مشخص شده در تصویر جداسازی می‌کنیم و دیتالودرهای مربوط به هرکدام از آن‌ها را می‌سازیم. از طریق collable_fn در ساختن بچ این نکته را در نظر می‌گیریم که لازم است در هر بچ تعداد توکن‌های کپشن‌های مربوط به آن بچ برابر باشد در غیر این‌صورت با دایمنشن‌های مختلف امکان ترین شبکه وجود ندارد. برای این منظور آیتم‌هایی که کپشن آن‌ها توکن‌های کمتری دارد در هر بچ با اضافه کردن توکن پدینگ به حداکثر تعداد توکن آیتم‌های آن بچ خواهد رسید.

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'

num_epochs = 2
model = ImageCaptioning(dataset.vocab_size(),800,512,40) #800,512,40
model = model.to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
criterion = nn.CrossEntropyLoss(ignore_index=0)
from torchsummary import summary
verbose = 0
for epoch in range(num_epochs):
    print("epoch: " + str(epoch))
    model.train()
    for idx, (imgs, captions) in enumerate(tqdm.tqdm(train_loader)):
      optimizer.zero_grad()
      imgs = imgs.to(device)
      captions = captions.to(device)
      outputs = model(imgs, captions[:-1])
      tmp = np.argmax(outputs.cpu().detach().numpy(),2)
      loss = criterion(
          outputs.reshape(-1, outputs.shape[2]), captions.reshape(-1)
          )
      loss.backward(loss)
      optimizer.step()

    model.eval()
    validation_loss = 0
    for idx, (imgs, captions) in enumerate(val_loader):
      imgs = imgs.to(device)
      captions = captions.to(device)
      outputs = model(imgs,captions[:-1])
      loss = criterion(
          outputs.reshape(-1, outputs.shape[2]), captions.reshape(-1)
      )
      validation_loss = validation_loss + loss.item()
      tmp = np.argmax(outputs.cpu().detach().numpy(),2)
      # print(dataset.stringfy(tmp[:,0]))
      # print(dataset.stringfy(captions[:,0].cpu().detach().numpy()))
    print("validation Loss: " + str(validation_loss))
```

در این بخش عملیات ترین خود را با استفاده از لاس فانکشن کراس انتروپی و بهینه ساز آدام انجام می‌دهیم. در هر مرحله‌ی ایپاک مدل را به حالت ترین برده و عکس و کپشن خود، به غیر از آخرین کپشن (که نتیجه‌ای است که شبکه باید آن‌را پیش‌بینی کند) به شبکه می‌دهیم و سپس جواب به دست آمده را argmax می‌گیریم تا ببنیم در هر موقعیت کدام یک از توکن‌های ما احتمال بیشتری در جواب نهایی داشته است. (البته این کار را خود تابع کراس انتروپی برای محاسبه‌ی لاس هم انجام می‌دهد و این محاسبه‌ی ما در اینجا برای استخراج رشته‌ی تولید شده توسط شبکه و دیباگ کردن خودمان است و نه محاسبه‌ی لاس)

درمرحله‌ی بعد، لاس را محاسبه کرده و نتیجه را در شبه بک پروپگیت می‌کنیم. بعد از انجام هر بار ترین یکبار هم شبکه‌ی خود را روی داده‌های ولیدیشن تست می‌کنیم و مطمئن می‌شویم که مقدار لاس ولیدیشن ما در هر ایپاک در حال کمتر شدن باشد.

```
# Test model and bleu-1 and bleu-2


for idx, (imgs, captions) in enumerate(test_loader):
      imgs = imgs.to(device)
      captions = captions.to(device)
      outputs = model(imgs,captions[:-1])
      loss = criterion(
          outputs.reshape(-1, outputs.shape[2]), captions.reshape(-1)
      )
      validation_loss = validation_loss + loss.item()
      tmp = np.argmax(outputs.cpu().detach().numpy(),2)
      print(dataset.stringfy(tmp[:,0]))
      print(dataset.stringfy(captions[:,0].cpu().detach().numpy()))
      y_pred = dataset.stringfy(tmp[:,0])
      y=[]
      for i in range(0,tmp.shape[1]):
        y.append(dataset.stringfy(captions[:,i].cpu().detach().numpy()))


      print(f'bleu 1 for item {i}: {sentence_bleu(y, y_pred)}')
      y_pred = []
      for i in range(0,tmp.shape[1]):
        y_pred.append(dataset.stringfy(tmp[:,1]))
      print(f'bleu 2 for item {i}: {corpus_bleu(y, y_pred)}')
```

در بخش بعد میزان پارامترهای bleu را بر روی داده‌های تست محاسبه می‌کنیم. به ازای هر آیتم‌ها تست ست خود جواب شبکه را بر روی دیتای مورد نظر محاسبه می‌کنیم و در مورد bleu1 جواب به دست آمده و مجموعه کپشن‌های آن بچ و در مورد bleu2 جواب‌های به دست آمده و مجموعه کپشن‌های هر بچ را به توابع آماده‌ی آن داده تا پارامترهای مورد نظر را محاسبه کند.

```
!wget http://nlp.stanford.edu/data/glove.6B.zip
!unzip glove.6B.zip
!ls -lat

vocab,embeddings = [],[]
with open('glove.6B.50d.txt','rt') as fi:
    full_content = fi.read().strip().split('\n')
for i in range(len(full_content)):
    i_word = full_content[i].split(' ')[0]
    i_embeddings = [float(val) for val in full_content[i].split(' ')[1:]]
    vocab.append(i_word)
    embeddings.append(i_embeddings)

import numpy as np
vocab_npa = np.array(vocab)
embs_npa = np.array(embeddings)

vocab_npa = np.insert(vocab_npa, 0, '<PAD>')
vocab_npa = np.insert(vocab_npa, 1, '<SOS>')
vocab_npa = np.insert(vocab_npa, 2, '<EOS>')
vocab_npa = np.insert(vocab_npa, 3, '<UNK>')

pad_emb_npa = np.zeros((1,embs_npa.shape[1]))
unk_emb_npa = np.mean(embs_npa,axis=0,keepdims=True)
embs_npa = np.vstack((pad_emb_npa,unk_emb_npa,embs_npa))
```

در این بخش فایل وزن‌های glove را از آدرس ذکر شده دانلود می‌کنیم و مطابق با توضیحات ذکر شده در سایت مرجع آرایه‌ی vocan و embedings را ایجاد می‌کنیم. در این دیتاست برای تمام ریشه‌ی کلمات وزن‌هایی برای ترین شدن شبکه در نحوه‌ی استفاده از کلمات و ترتیب قرارگیری آنها محاسبه شده است.

با توجه به اینکه کاراکترهایی که ما خودمان به لیست کلمات برای شروع و پایان و پدینگ و ناشناخته اضافه کردیم در لیست کلمات glove وجود ندارد این کلمات هم به لیست کلمات اضافه می‌کنیم.

```
class ImageCaptioning(nn.Module):
  def __init__(self, vocab_size, embed_size, hidden_size, num_layers):
    super(ImageCaptioning, self).__init__()
    self.vocab_size = vocab_size
    self.embed_size = embed_size
    self.hidden_size = hidden_size
    self.num_layers = num_layers
    self.featuresCNN = models.resnet50(pretrained=True)
    for param in self.featuresCNN.parameters():
      param.requires_grad = False
    self.featuresCNN.fc = nn.Linear(self.featuresCNN.fc.in_features, embed_size)
    self.fc = nn.Linear(hidden_size, vocab_size)
    self.relu = nn.ReLU()

    self.embed = nn.Embedding(self.vocab_size, self.embed_size)
    with torch.no_grad():
      self.embed.from_pretrained(torch.from_numpy(embs_npa).float(), freeze=True)

    self.lstm = nn.LSTM(self.embed_size, self.hidden_size, self.num_layers)
    self.linear = nn.Linear(self.hidden_size, self.vocab_size)
  def forward(self, images, captions):
    features = self.featuresCNN(images)
    features = self.relu(features)
    embeddings = self.embed(captions)
    embeddings = torch.cat((features.unsqueeze(0), embeddings), dim=0)
    hiddens, _ = self.lstm(embeddings)
    outputs = self.fc(hiddens)
    return outputs
```

سپس شبکه‌ی قبلی را مجدد باز تعریف می‌کنیم؛ با این تفاوت که این‌بار وزن‌های آماده شده را به عنوان وزن‌های از پیش ترین شده به شبکه embed خود می‌دهیم. و مجدد فرآیند ترین را مانند قبل تکرار می‌کنیم.

قسمت ه – امتیازی:

```
my_resnet = models.resnet50(pretrained=True)
for idx,item in enumerate(my_resnet.parameters()):
  item.requires_grad = False
  if idx >= 158: #make last 3 layer weights updateable
    item.requires_grad = True


class ImageCaptioning(nn.Module):
  def __init__(self, vocab_size, embed_size, hidden_size, num_layers):
    super(ImageCaptioning, self).__init__()
    self.vocab_size = vocab_size
    self.embed_size = embed_size
    self.hidden_size = hidden_size
    self.num_layers = num_layers
    self.featuresCNN = my_resnet#models.resnet50(pretrained=True)
    for param in self.featuresCNN.parameters():
      param.requires_grad = False
    self.featuresCNN.fc = nn.Linear(self.featuresCNN.fc.in_features, embed_size)
    self.fc = nn.Linear(hidden_size, vocab_size)
    self.relu = nn.ReLU()

    self.embed = nn.Embedding(self.vocab_size, self.embed_size)
    with torch.no_grad():
      self.embed.from_pretrained(torch.from_numpy(embs_npa).float(), freeze=True)

    self.lstm = nn.LSTM(self.embed_size, self.hidden_size, self.num_layers)
    self.linear = nn.Linear(self.hidden_size, self.vocab_size)
  def forward(self, images, captions):
    features = self.featuresCNN(images)
    features = self.relu(features)
    embeddings = self.embed(captions)
    embeddings = torch.cat((features.unsqueeze(0), embeddings), dim=0)
    hiddens, _ = self.lstm(embeddings)
    outputs = self.fc(hiddens)
    return outputs
```

در این بخش طبق چیزی که سوال خواسته است شبکه ی resnet از پیش ترین شده ایجاد می‌کنیم. سپس با لوپ بین تمام لایه‌های آن صرفا ۳ لایه‌ی آخر آن‌را جوری تنظیم می‌کنیم که در زمان ترین وزن‌های آن گرادیان را دریافت کرده و آپدیت شوند ولی بقیه‌ی لایه‌های آن به شکل فریز شده باقی می‌ماند. سپس دوباره شبکه‌ی خود را تعریف می‌کنیم با این تفاوت که این‌بار به جای شبکه‌ی resnet اصلی از شبکه‌ی ساخته شده‌ی خودمان استفاده می‌کنیم و سپس مجدد فرآیندهای ترین و ولیدیشن وتست و محاسبه‌ی bleu را تکرار می‌کنیم.

د) علاوه بر اینکه می‌خواهیم کلمات مرتبط با تصویر را داشته باشیم، ترتیب کلمات و جمله‌بندی درست نیز مهم است. برای این مساله باید یک language model ترین کنیم که مشخص کند کلمات با چه ترتیبی پشت هم قرار می‌گیرند. با دیتاستی که داریم می‌توانیم کلمات موجود در تصاویر را ترین کنیم اما برای یادگیری ترتیب کلمات، باید دیتاست عظیم‌تری داشته باشیم که حالت‌های مختلف قرارگیری کلمات در آن وجود داشته باشد.

بنابراین در مقایسه، به نظر می‌رسد در حالت ج، که LSTM از وزن‌های از پیش‌ترین شده استفاده می‌کند، ترتیب کلمات به جملات واقعی نزدیک‌تر است.

و) حدس من این است که لایه امبدینگ باعث گرادیان ونیشینگ شود، امبدینگ از آن‌جایی که ابعاد فضای ورودی را کاهش می‌دهد موجب از دست رفتن اطلاعات خواهد شد. این از دست رفتن اطلاعات (information loss) باعث می‌شود که بازگشت گرادیان در شبکه سخت‌تر شود. بنابراین حالت اول که رزنت را دستکاری نکرده‌ایم و pretrained است و اصلا قرار نیست گرادیانی بگیرد و لایه امبدینگ بعد از آن است، دچار گرادیان ونیشینگ نخواهیم شد. اما در حالتی که رزنت را Trainable کرده‌ایم گرادیان کمتری بازمی‌گردد و بنابراین گرادیان ونیشینگ داریم. در حالتی که رزنت ثابت است و در LSTM وزن لود کرده‌ایم نیز گرادیان ونیشینگ نداریم.

بنابراین در دو حالت اول گرادیان ونیشینگ نداریم اما در حالت آخر داریم چون امبدینگ لایر، عملا گرادایان زیادی به عقب برنمی‌گرداند.

نتایج BLEU در نوت‌بک قابل مشاهده است و مقایسه دقت به شکل زیر است:

دقت در حالت دوم که در LSTM وزن لود می‌کند بیشتر از همه است. بعد حالت سوم که خود رزنت را ترین ابل می‌کند دقت بیشتری دارد و در آخر دقت در حالت اول از بقیه کمتر است.

ز) مزیت اصلی مدل‌های RNN مبتنی بر کاراکتر این است که فضای گسسته‌ای که با آن کار می‌کنیم، بسیار کوچک‌تر است. با در نظر گرفتن همه علائم نگارشی، در حدود ۹۷ کاراکتر انگلیسی رایج داریم؛ در صورتی‌که واژگان می‌تواند هزاران کلمه باشد که به این معنی‌ست که ذخیره‌ی word embedding به حافظه‌ی زیادی نیاز دارد. همچنین گنجاندن word embedding در مدل، پارامترهای بسیار زیادی را به مدل اضافه می‌کند و هزینه‌ی محاسباتی افزایش می‌یابد.

نکته دیگر این است که در RNN مبتنی بر کلمه، کلمات با غلط املایی را به عنوان توکن unknown در نظر می‌گیریم؛ در این‌صورت مدل در تولید متن جدید چندان انعطاف پذیر نیست. از طرفی هم RNN مبتنی بر کاراکتر می‌تواند به طور خود به خود کلمات غیرعادی را با احتمال کمی تولید کند که می تواند به این معنی باشد که کلماتی با اشتباهات تایپی و .. می توانند در متن تولید شده وجود داشته باشند، اما یک مدل جنریتور به خوبی آموزش دیده، معمولاً در یادگیری نحوه املا بسیار خوب عمل می‌کند.