به نام خدا

درس یادگیری عمیق

تمرین پنجم

غزل زمانی نژاد

401722244

1. رویکردهای معمولی این تسک بر اساس شبکه عصبی بازگشتی هستند و در آن کپشن به صورت کلمه به کلمه تولید می شود و پیش بینی هر کلمه بر اساس محتوای تصویری و کلمات تولید شده قبلی صورت می گیرد. اما بر اساس تحقیقات، در ویدئو محتوای اضافی و نامربوط قابل توجهی وجود دارد که ممکن است باعث تداخل در تولید کپشن درست شود. ایده اصلی این مقاله ارائه یک مدل برای تولید کپشن برای ویدیو بر اساس مکانیزم توجه است. ۴ بخش کلی این معماری عبارتند از:

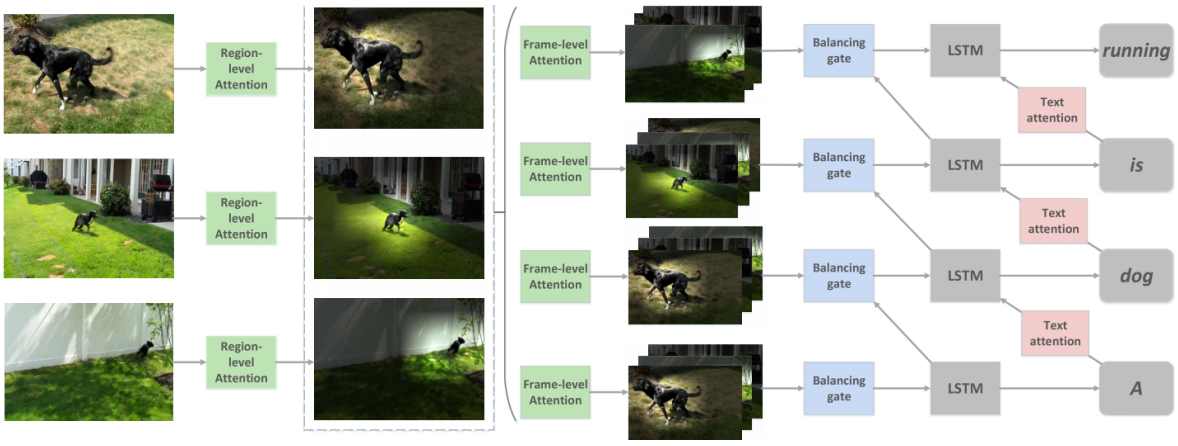
Visual attention module: این ماژول برای استخراج ویژگی های تصویری به کار می رود. شامل دو لایه است و می تواند بر روی مرتبط ترین ویژگی های تصویری تمرکز کند: لایه اول یاد می گیرد که روی برجسته ترین مناطق در هر فریم تمرکز کند در حالیکه لایه دوم تلاش می کند به مرتبط ترین فریم ها توجه داشته باشد. این ویژگی ها در قالب یک بردار با طول ثابت انکود می شوند.

Text attention module: در طول generation این ماژول بر روی کلمات تولید شده قبلی عمل می کند و به طور انتخابی بر روی مرتبط ترین عبارت برای تولید کلمه فعلی تمرکز می کند.

Caption generator: از بردار انکود شده ویژگی های تصویر و ویژگی های متن به همراه اطلاعاتی که توسط LSTM ذخیره شده برای تولید کپشن به صورت کلمه به کلمه استفاده می شود.

Balancing gate: از این گیت برای تنظیم تاثیر ویژگی های تصویری و ویژگی های متن در فرآیند تولید کپشن استفاده شده است.

در تصویر زیر معماری این مدل به خوبی مشاهده می شود:



از ایده مطرح شده در این مقاله می توان برای تولید کپشن برای تصاویر نیز استفاده کرد. با این تفاوت که در تصاویر دیگر بحث چندین فریم مطرح نیست و در ماژول visual attention می توانیم از لایه اول (که به نقاط برجسته هر فریم اشاره دارد) برای آموزش آن استفاده کنیم.

2. الف) گیت آپدیت به مدل کمک می‌کند تا تعیین کند چه مقدار از اطلاعات گذشته باید به آینده منتقل شود. و گیت ریست به مدل کند که چه اطلاعاتی را فراموش کند. اگر گیت ریست وجود نداشته باشد فرمول ها بدین شکل تغییر می کنند:

$$\tilde{\mathbf{h}}_t = \tanh(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{hx}\mathbf{x}_t + \mathbf{b}_h)$$

$$\mathbf{h}_t = \mathbf{u}_t . \tilde{\mathbf{h}}_t + (1 - \mathbf{u}_t) . \mathbf{h}_{t-1}$$

$$\mathbf{u}_t = \sigma(\mathbf{W}_{uh}\mathbf{h}_{t-1} + \mathbf{W}_{ux}\mathbf{x}_t + \mathbf{b}_u)$$

در صورت حذف این گیت شبکه تنها می تواند تصمیم بگیرد چه مقدار از اطلاعات گذشته را در استیت فعلی استفاده کند و در فراموش کردن اطلاعات قدیمی موفق نخواهد بود. مثلا در تسک تشخیص احساسات، فرض کنید فردی در ابتدای کامنت های خود در مورد یک کتاب ذکر کرده باشد: این یک کتاب فانتزی است که ... را به تصویر می کشد. و پس از چند پاراگراف، در جملات آخر گفته باشد: اما من از این کتاب به دلیل ذکر جزئیات لذت نبردم. در اینجا جمله آخر برای تشخیص حس فرد کافی است و گیت ریست می تواند در پاک کردن اطلاعات گذشته که مفید نبودند کمک کند و در صورت نبود آن کار شبکه برای تعیین کلاس نهایی سخت تر خواهد شد.

ب) مطابق فرمول های GRU برای اینکه hidden state تنها به x مرحله قبل بستگی داشته باشد باید مقدار گیت ریست ۰ و آپدیت ۱ باشد. و برای اینکه تنها به hهای مرحله قبل بستگی داشته باشد، برخلاف حالت قبل باید مقدار گیت ریست ۱ و آپدیت ۰ شود.

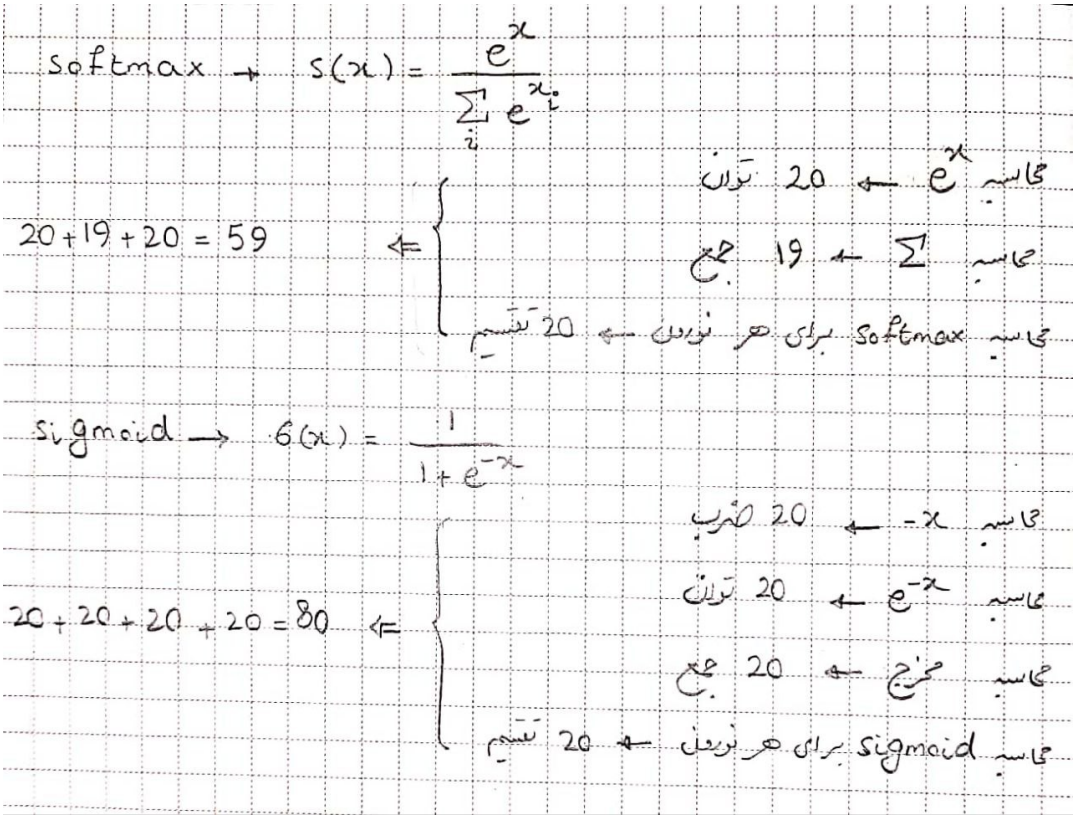
$$\tilde{\mathbf{h}}_t = \tanh(\mathbf{W}_{hh}(\mathbf{r}_t . \mathbf{h}_{t-1}) + \mathbf{W}_{hx}\mathbf{x}_t + \mathbf{b}_h)$$

$$\mathbf{h}_t = \mathbf{u}_t . \tilde{\mathbf{h}}_t + (1 - \mathbf{u}_t) . \mathbf{h}_{t-1}$$

$$\mathbf{u}_t = \sigma(\mathbf{W}_{uh}\mathbf{h}_{t-1} + \mathbf{W}_{ux}\mathbf{x}_t + \mathbf{b}_u)$$

$$\mathbf{r}_t = \sigma(\mathbf{W}_{rh}\mathbf{h}_{t-1} + \mathbf{W}_{rx}\mathbf{x}_t + \mathbf{b}_r)$$

3. الف) تعداد عملیات ها به نحوه پیاده سازی دو تابع مربوط است. در اینجا فرض میکنیم در پیاده سازی softmax، مقدار e**x برای هر یک از نورون ها یک بار محاسبه و ذخیره می شود. همینطور مقدار مخرج (مجموع e**x ها) تنها یک بار حساب می شود.



تعداد ۲۱ عملیات اضافه می شود.

ب) با اضافه کردن لایه dense، ظرفیت یادگیری دو مدل حدودا شبیه به هم خواهد شد. با این تفاوت که در لایه dense می توانیم بایاس و همچنین تابع فعال سازی داشته باشیم.

مزایای لایه embedding نسبت به لایه dense: این لایه می تواند به خوبی ارتباط بین کلمات را آموزش ببیند و بر این اساس یک ماتریس از امبدینگ کلمات بسازد اما به عنوان ورودی لایه dense باید بردارهای one-hot کلمات را بدهیم که در آنها اطلاعات خاصی از کلمات وجود ندارد و نمی تواند نمایانگر فاصله معنایی باشد (فاصله میان هر دو کلمه ۱ است). همچنین در انجام محاسبات لایه dense، نیاز است که ضرب ماتریسی انجام شود اما در لایه embedding، بردار وزن ها را به عنوان یک lookup

table در نظر می گیرد که سطر n ام آن بردار امبدینگ کلمه n در مجموعه کلمات است و همین کار باعث سرعت بخشیدن در فرایند آموزش می شود. بنابراین استفاده از لایه embedding به کاهش اندازه ورودی و کاهش پیچیدگی محاسبات کمک خواهد کرد.

4. الف) ابتدا یک کلاس از نوع Dataset می سازیم و در constructor از مسیر عکس ها و کپشن ها آنها را می خوانیم. داده ها را split کرده و پیش پردازش های مربوطه را با تابع preprocess_caption انجام می دهیم. " را از ابتدا و انتهای کپشن حذف می کنیم و تمامی کلمات را به lower case تبدیل می کنیم. از دیکشنری img_caption_dict برای نگهداری هر ۵ کپشن یک تصویر استفاده می کنیم. همچنین مجموعه کلمات را با تابع build_vocab می سازیم و در دیکشنری مربوطه نگه می داریم. در دیکشنری vocab کلید، توکن و مقدار، ایندکس است. دیکشنری idx_to_vocab برعکس دیکشنری قبلی است یعنی کلید،ایندکس و مقدار، توکن است.

```python
PAD_VALUE = 0

class MyDataset(Dataset):
    def __init__(self, img_dir, annotation_dir, transform=None, tokenizer="spacy"):
        super(MyDataset, self).__init__()
        ########################################
        # your code here, you can add new parameter in constructor
        ########################################
        self.img_dir = img_dir
        self.transform = transform
        self.tokenizer = get_tokenizer(tokenizer)

        with open(annotation_dir) as f:
            self.img_caption_pair = f.readlines()
        self.img_caption_pair = self.img_caption_pair[1:]
        # each element of list is a tuple: (img_name, caption)
        self.img_caption_pair = [self.split_img_caption(line) for line in self.img_caption_pair]
        # dict[img] = [cap1, cap2, cap3, cap4, cap5]
        self.img_caption_dict = {}
        self.build_img_cap_dict()

        self.vocab = {"<PAD>": PAD_VALUE, "<SOS>": 1, "<EOS>": 2, "<UNK>": 3}

        self.build_vocab()
        self.idx_to_vocab = {self.vocab[k]:k for k in self.vocab.keys()}
```

```
    def preprocess_caption(self, caption):
        # remove " from the beginning and the end of a caption
        if caption.startswith('"'):
            caption = caption[1:]
        if caption.endswith('"'):
            caption = caption[:-1]

        # convert characters to lower case
        caption = caption.lower()
        return caption


    def split_img_caption(self, l):
        img, *cap = l.strip().split(',')
        # if the caption itself has comma, join the other parts of it
        cap = ','.join(cap)
        cap = self.preprocess_caption(cap)

        return (img, cap)


    def build_img_cap_dict(self):
        for img, cap in self.img_caption_pair:
            if img in self.img_caption_dict.keys():
                self.img_caption_dict[img].append(cap)
            else: self.img_caption_dict[img] = [cap]


    def build_vocab(self):
        idx = 4
        for _, c in self.img_caption_pair:
            tokens = self.tokenizer(c)
            for t in tokens:
                if t not in self.vocab.keys():
                    self.vocab[t] = idx
                    idx += 1


    def build_vocab_from_words(self, words):
        self.vocab = {}
        c = 0
        for w in words:
            self.vocab[w] = c
            c += 1

        # index_to_vocab dictionary should be updated
        self.idx_to_vocab = {self.vocab[k]:k for k in self.vocab.keys()}
```

در تابع get_item ایندکس را به عنوان ورودی دریافت کرده و عکس مربوطه را می خوانیم و transform را روی آن اعمال می کنیم. سپس کپشن را توکنایز می کنیم. توکن های <EOS> و <SOS> را به ابتدا و انتهای کپشن اضافه می کنیم و اگر کلمه ای داخل vocab قرار نداشته باشد به جای آن توکن <UNK> می گذاریم. در نهایت بردار تصویر و بردار ایندکس کلمات کپشن را برمی گردانیم.

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

        # return len(self.vocab)
        return len(self.img_caption_pair)

    def __getitem__(self, index):
        ########################################
        #your code here, you should be retrun image and caption of th
        ########################################

        # read image
        img_name, caption = self.img_caption_pair[index]
        img = cv2.imread(os.path.join(self.img_dir, img_name))
        if self.transform:
            img = self.transform(Image.fromarray(img))

        # tokenize caption and get its index
        tokenized_caption = self.tokenizer(caption)

        indices = [self.vocab["<SOS>"]]
        for t in tokenized_caption:
            if t in self.vocab.keys():
                idx = self.vocab[t]
            else: idx = self.vocab["<UNK>"]
            indices.append(idx)

        indices.append(self.vocab["<EOS>"])
        return img, indices
```

برای اینکه داده ها را در قالب batch به عنوان ورودی مدل بدهیم، باید طول کپشن ها یکسان باشد.

برای این کار از تابع padify برای پد کردن استفاده می کنیم. همچنین با استفاده از transform تصاویر را

به ۲۲۴* ۲۲۴ (اندازه ورودی رزنت) resize و سپس نرمالیزه می کنیم.

```python
def padify(batch):
    images = []
    captions = []

    for i, c in batch:
        images.append(i)
        captions.append(torch.tensor(c))

    padded_caps = pad_sequence(captions, batch_first=False, padding_value=PAD_VALUE)
    return (torch.stack(images),
            padded_caps)


transform = transforms.Compose(
        [
            transforms.Resize((224, 224)),
            transforms.ToTensor(),
            transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
        ]
    )
```

دیتاست را می سازیم و از آن برای ساختن data loader آموزش و ارزیابی استفاده می کنیم. از batch size = 64 و نسبت ۰.۸ برای داده آموزشی و ۰.۲ برای داده ارزیابی استفاده می کنیم.

```
dataset = MyDataset("/content/flickr8k/images", "/content/flickr8k/captions.txt", transform=transform)

################################################
#your code here
################################################
batch_size = 64

train_split = 0.8
train_size = int(train_split * len(dataset))
val_size = len(dataset) - train_size

train_set, val_set = random_split(dataset, [train_size, val_size])

train_loader = torch.utils.data.DataLoader(train_set, batch_size=batch_size, shuffle=True, collate_fn=padify)
val_loader = torch.utils.data.DataLoader(val_set, batch_size=batch_size, shuffle=False, collate_fn=padify)
```

```
/usr/local/lib/python3.10/dist-packages/torchtext/data/utils.py:105: UserWarning: Spacy model "en" could not be l
  warnings.warn(
```

```
print(len(dataset))
print(len(dataset.vocab))
print(len(train_loader.dataset))
print(len(val_loader.dataset))
```

```
40455
8508
32364
8091
```

ب) در مدل دو تغییر اعمال می کنیم: افزودن padding_idx به لایه امبدینگ و افزودن تابع predict_caption برای پیش بینی داده جدید (ورودی این تابع تنها تصویر خواهد بود از برای پیش بینی کپشن، از کلمات قبلی که خود پیش بینی کرده استفاده می کند. یعنی x پیش بینی شده در این مرحله به عنوان ورودی lstm در مرحله بعد مورد استفاده قرار می گیرد.

```python
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
        # backbone for feature exctract
        self.featuresCNN = models.resnet50(pretrained=True)
        # convert features to feature vector
        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)
        #activation function
        self.relu = nn.ReLU()
        #RNN
        # I added padding_idx
        self.embed = nn.Embedding(self.vocab_size, self.embed_size, padding_idx=PAD_VALUE)
        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

    def predict_caption(self, image, idx_to_vocab, maxlength=40):
        result_caption = []

        with torch.no_grad():
            x = self.featuresCNN(image)
            x = self.relu(x).unsqueeze(0)
            states = None

            for _ in range(maxlength):
                hiddens, states = self.lstm(x, states)
                output = self.fc(hiddens.squeeze(0))
                predicted = output.argmax(1)

                result_caption.append(predicted.item())
                x = self.embed(predicted).unsqueeze(0)

                if idx_to_vocab[predicted.item()] == "<EOS>":
                    break
        return [idx_to_vocab[i] for i in result_caption]
```

هایپرپارامترها را به صورت زیر تنظیم می کنیم:

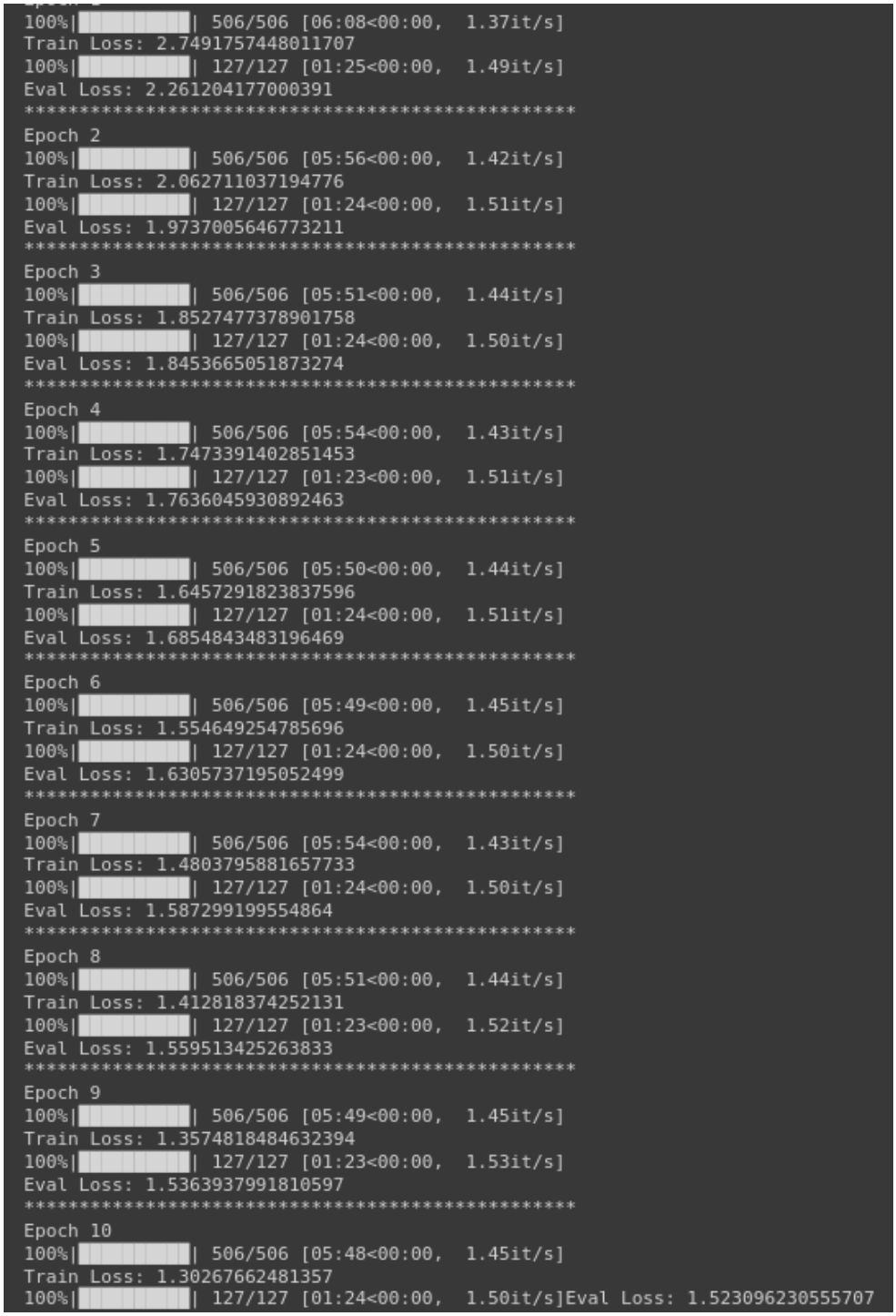
```python
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print("device:", device)

num_epochs = 10
vocab_size = len(dataset.vocab)
embed_size = 256
hidden_size = 256
num_layers = 3
learning_rate = 1e-3
```

```
device: cuda
```

سپس مدل را در ۱۰ ایپاک با بهینه ساز Adam و تابع ضرر cross entropy آموزش می دهیم.

```python
model = ImageCaptioning(vocab_size, embed_size, hidden_size, num_layers)
model.to(device)
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
criterion = nn.CrossEntropyLoss()

for epoch in range(num_epochs):
    print(f"Epoch {epoch+1}")

    # train loop
    running_loss = 0
    model.train()
    # CHANGE to train_loader
    for idx, (imgs, captions) in enumerate(tqdm(train_loader)):
        ###################################
        #your code here, complete train loop
        ###################################
        imgs = imgs.to(device)
        captions = captions.to(device)

        pred = model(imgs, captions[:-1])
        optimizer.zero_grad()
        loss = criterion(pred.reshape(-1, pred.shape[2]), captions.reshape(-1))

        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    running_loss /= len(train_loader)
    print("Train Loss:", running_loss)

    # evaluation loop
    eval_loss = 0
    model.eval()
    with torch.no_grad():
        for idx, (imgs, captions) in enumerate(tqdm(val_loader)):
        ###################################
        # your code here, complete validation loop
        ###################################
            imgs = imgs.to(device)
            captions = captions.to(device)

            pred = model(imgs, captions[:-1])
            loss = criterion(pred.reshape(-1, pred.shape[2]), captions.reshape(-1))
            eval_loss += loss.item()
    eval_loss /= len(val_loader)
    print("Eval Loss:", eval_loss)
```

خروجی به صورت زیر است:



مقدار loss در حین آموزش کاهش یافته و به مقدار ۱.۳۰ برای داده آموزشی و ۱.۵۲ برای داده ارزیابی رسیده است.

پس از پایان آموزش از تابع corpus_blue برای محاسبه امتیاز BLEU استفاده می کنیم. وزن های آن را به گونه ای تنظیم می کنیم که برای unigram و bigram محاسبات را انجام دهد.

تابع predict_whole_data را پیاده سازی می کنیم تا برای همه تصاویر موجود در دیتاست پیش بینی انجام دهد و سپس امتیاز BLEU را با یک پیش بینی و ۵ رفرنس حساب می کنیم. توکن های <EOS> و <SOS> را از کپشن حذف می کنیم.

```python
def compute_bleu(list_of_references, hypotheses):
    weights = [
        (1., 0, 0, 0),
        (1./2., 1./2., 0, 0)
    ]

    return corpus_bleu(list_of_references, hypotheses, weights=weights)
```

```python
def predict_whole_data(dataset, model):
    references = []
    hypotheses = []

    # compute bleu on whole data
    for img_name, caps in dataset.img_caption_dict.items():

        # read image
        img = cv2.imread(os.path.join(dataset.img_dir, img_name))
        if transform:
            img = transform(Image.fromarray(img))

        img = img.to(device)
        predicted_caption = model.predict_caption(img.unsqueeze(0), dataset.idx_to_vocab)
        if '<SOS>' in predicted_caption:
            predicted_caption.remove('<SOS>')
        if '<EOS>' in predicted_caption:
            predicted_caption.remove('<EOS>')

        # print(predicted_caption)
        hypotheses.append(predicted_caption)

        tokenized_captions = [dataset.tokenizer(cap) for cap in caps]
        # print(tokenized_captions)
        references.append(tokenized_captions)

    return references, hypotheses
```

امتیاز BLEU برای این بخش به صورت زیر است:

BLEU-1: 0.5575184961055694

BLEU-2: 0.3455166215514869

ج) ابتدا Glove را دانلود می کنیم. از وزن ها با ابعاد ۲۰۰ استفاده می کنیم. فایل مربوطه را می خوانیم و کلمات آن را داخل لیست vocab و وزن ها را داخل لیست embeddings ذخیره می کنیم. سپس این دو لیست را به آرایه numpy تبدیل می کنیم. توکن های pad, sos, eos, unk را به مجموعه کلمات و امبدینگ اضافه می کنیم.

```
vocab, embeddings = [],[]
# use embedding_size 200 from glove
with open('glove.6B.200d.txt','rt') as f:
    full_content = f.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)
```

```
vocab_np = np.array(vocab)
embeds_np = np.array(embeddings)

#insert '<PAD>' and '<UNK>' tokens at start of vocab_np.
vocab_np = np.insert(vocab_np, 0, '<PAD>')
vocab_np = np.insert(vocab_np, 1, '<SOS>')
vocab_np = np.insert(vocab_np, 2, '<EOS>')
vocab_np = np.insert(vocab_np, 3, '<UNK>')
# print(vocab_np[:10])

pad_emb_np = np.zeros((1, embeds_np.shape[1]))   #embedding for '<PAD>' token.
sos_emb_np = np.random.rand(1, embeds_np.shape[1])   #embedding for '<SOS>' token.
eos_emb_np = np.random.rand(1, embeds_np.shape[1])   #embedding for '<EOS>' token.
unk_emb_np = np.mean(embeds_np, axis=0, keepdims=True)    #embedding for '<UNK>' token

#insert embeddings for pad, sos, eos, unk tokens at top of embeds_np.
embeds_np = np.vstack((pad_emb_np, sos_emb_np, eos_emb_np, unk_emb_np, embeds_np))
print(embeds_np.shape)
```

```
(400004, 200)
```

دیتاست جدید و loaderهای جدید را تشکیل می دهیم.

```python
transform = transforms.Compose(
        [
            transforms.Resize((224, 224)),
            transforms.ToTensor(),
            transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
        ]
    )

dataset_glove = MyDataset("/content/flickr8k/images", "/content/flickr8k/captions.txt", transform=transform)

batch_size = 64

train_split = 0.8
train_size = int(train_split * len(dataset_glove))
val_size = len(dataset_glove) - train_size

train_set2, val_set2 = random_split(dataset_glove, [train_size, val_size])

train_loader2 = torch.utils.data.DataLoader(train_set2, batch_size=batch_size, shuffle=True, collate_fn=padify)
val_loader2 = torch.utils.data.DataLoader(val_set2, batch_size=batch_size, shuffle=False, collate_fn=padify)
```

برای اینکه به مشکل پر شدن RAM کولب بر نخوریم و بتوانیم آموزش را انجام دهیم، از کل ۴۰۰هزار کلمه GLOVE استفاده نمی کنیم، بلکه از مجموعه کلماتی که در دیتای ما و GLOVE مشترک است استفاده می کنیم. برای این کار باید اشتراکات را پیدا کنیم و reduced_vocab و reduced_embeds را بسازیم. سپس مجموعه وکب را آپدیت کنیم.

```python
# a boolean array of same shape as vocab_np
mask = np.isin(vocab_np, np.array(list(dataset_glove.vocab.keys())))
print("mask:", mask)
# indices of words which are common in both (glove_vocab and dataset_vocab)
intersection = np.argwhere(mask)
intersection = np.squeeze(intersection, 1)
print("intersection", intersection)

reduced_vocab = vocab_np[intersection]
reduced_embeds = embeds_np[intersection]
print(reduced_vocab[:20])

# update dataset vocabulary
dataset_glove.build_vocab_from_words(reduced_vocab)
```

```
mask: [ True  True  True ... False False False]
intersection [     0      1      2 ... 396597 398567 399875]
['<PAD>' '<SOS>' '<EOS>' '<UNK>' 'the' ',' '.' 'of' 'to' 'and' 'in' 'a'
 '"' "'s" 'for' '-' 'that' 'on' 'is' 'was']
```

```python
print(len(dataset_glove.vocab))
print(reduced_vocab.shape)
```

```
7846
(7846,)
```

در تعریف مدل جدید، از همان مدل قبل ارث بری می کنیم با این تفاوت که برای لایه امبدینگ از وزن های pretrain استفاده می کنیم.

```python
class ImageCaptioningPretrained(ImageCaptioning):
    def __init__(self, vocab_size, embed_size, hidden_size, num_layers, embeddings):
        super(ImageCaptioningPretrained, self).__init__(vocab_size, embed_size, hidden_size, num_layers)

        # different from the previous part
        self.embed = nn.Embedding.from_pretrained(torch.from_numpy(embeddings).float(), padding_idx=PAD_VALUE)
```

سپس یک مدل instantiate می کنیم و آن را در ۱۰ ایپاک آموزش می دهیم. خروجی آموزش به صورت زیر است:

```
100%|██████████| 506/506 [06:02<00:00,  1.39it/s]
Train Loss: 2.8903727321756687
100%|██████████| 127/127 [01:24<00:00,  1.49it/s]
Eval Loss: 2.3872270415148398
**************************************************
Epoch 2
100%|██████████| 506/506 [05:52<00:00,  1.44it/s]
Train Loss: 2.160399492314682
100%|██████████| 127/127 [01:24<00:00,  1.51it/s]
Eval Loss: 2.0730672567848147
**************************************************
Epoch 3
100%|██████████| 506/506 [05:48<00:00,  1.45it/s]
Train Loss: 1.9235348213802685
100%|██████████| 127/127 [01:24<00:00,  1.50it/s]
Eval Loss: 1.8981473286320845
**************************************************
Epoch 4
100%|██████████| 506/506 [05:50<00:00,  1.44it/s]
Train Loss: 1.7973678368824744
100%|██████████| 127/127 [01:33<00:00,  1.36it/s]
Eval Loss: 1.8083393113819632
**************************************************
Epoch 5
100%|██████████| 506/506 [06:00<00:00,  1.40it/s]
Train Loss: 1.7051508106261846
100%|██████████| 127/127 [01:25<00:00,  1.49it/s]
Eval Loss: 1.7323606661924227
**************************************************
Epoch 6
100%|██████████| 506/506 [05:49<00:00,  1.45it/s]
Train Loss: 1.6367170751801592
100%|██████████| 127/127 [01:23<00:00,  1.51it/s]
Eval Loss: 1.6731525922384787
**************************************************
Epoch 7
100%|██████████| 506/506 [05:48<00:00,  1.45it/s]
Train Loss: 1.5663855751983733
100%|██████████| 127/127 [01:24<00:00,  1.50it/s]
Eval Loss: 1.6271703308961523
**************************************************
Epoch 8
100%|██████████| 506/506 [05:47<00:00,  1.45it/s]
Train Loss: 1.5141459580937864
100%|██████████| 127/127 [01:24<00:00,  1.50it/s]
Eval Loss: 1.5910822904016089
**************************************************
Epoch 9
100%|██████████| 506/506 [05:51<00:00,  1.44it/s]
Train Loss: 1.4750279853702062
100%|██████████| 127/127 [01:24<00:00,  1.51it/s]
Eval Loss: 1.5633375776095653
**************************************************
Epoch 10
100%|██████████| 506/506 [05:50<00:00,  1.44it/s]
Train Loss: 1.4285901988683483
100%|██████████| 127/127 [01:23<00:00,  1.51it/s]Eval Loss: 1.5406883438741128
```

مقدار loss در حین آموزش کاهش یافته و به مقدار ۱.۴۲ برای داده آموزشی و ۱.۵۴ برای داده ارزیابی رسیده است.

امتیاز BLEU برای این بخش به صورت زیر است:

BLEU-1: 0.5240019886572881

BLEU-2: 0.31634241107845434

د) در بخش دوم از وزن های pretrained برای لایه امبدینگ استفاده کردیم اما امتیاز BLUE تغییر چندانی نداشت و حتی کمتر شد. البته در بخش اول از ۲۵۶ = embed_size ولی در بخش دوم از اندازه ۲۰۰ استفاده کردیم و این ممکن است مقایسه را کمی دشوار کند. زمانی که از وزن های pretrained استفاده می کنیم:

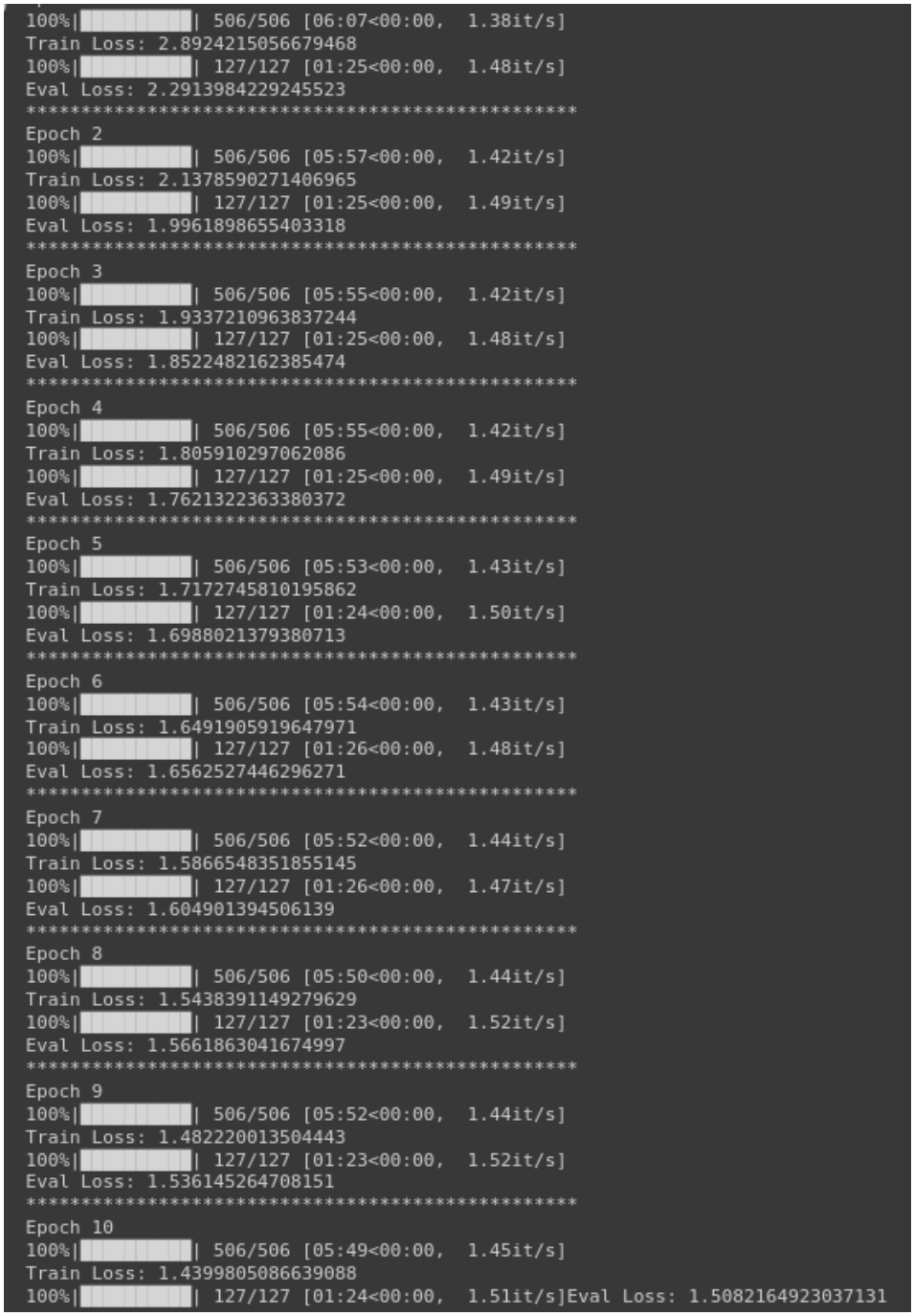
۱. نقطه شروع آموزش بهتر خواهد بود. در حالتی که از وزن های رندوم برای آموزش استفاده می کنیم ممکن است در بعضی مینیمم های محلی گیر کنیم و وزن ها به درستی آموزش نبینند.

۲. در صورت استفاده از وزن های pretrained، می توانیم با تعداد اپاک کمتری آن را برای دیتاست خودمان fine-tune کنیم اما در حالت رندوم به تعداد زیادی اپاک برای آموزش بردارهای امبدینگ نیاز داریم و این کار فرآیند آموزش را کند می کند.

ه) یک کلاس جدید تعریف می کنیم و از کلاس ImageCaptioning ارث بری می کنیم. مقدار require_grad تمامی پارامترها به جز تعدادی از لایه ها را false می کنیم. و دوباره لایه امبدینگ را با وزن های از پیش آموخته GLOVE لود می کنیم.

```python
class ImageCaptioningFinetuneResnet(ImageCaptioning):
    def __init__(self, vocab_size, embed_size, hidden_size, num_layers, embeddings):
        super(ImageCaptioningFinetuneResnet, self).__init__(vocab_size, embed_size, hidden_size, num_layers)

        for name, param in self.featuresCNN.named_parameters():
            if 'layer4.2' in name or 'layer4.1' in name:
                continue
            param.requires_grad = False
            # print(name)

        self.featuresCNN.fc = nn.Linear(self.featuresCNN.fc.in_features, embed_size)

        self.embed = nn.Embedding.from_pretrained(torch.from_numpy(embeddings).float(), padding_idx=PAD_VALUE)

```

یک مدل جدید می سازیم و آن را در ۱۰ ایپاک آموزش می دهیم. نتیجه آموزش به صورت زیر است:



مقدار loss در حین آموزش کاهش یافته و به مقدار ۱.۴۴ برای داده آموزشی و ۱.۵۰ برای داده ارزیابی رسیده است.

امتیاز BLEU برای این بخش به صورت زیر است:

BLEU-1: 0.5225900808585162

BLEU-2: 0.317421936782111O6

مقدار امتیاز BLEU در بخش ه و ج تقریبا برابر است. Fine-tune کردن این لایه تاثیر چندانی در استخراج ویژگی های تصویر نداشته. البته مقدار loss روی داده ارزیابی کمتر شده اما در BLEU اختلافی مشاهده نمی شود. باید ذکر کنیم که BLEU در محاسبات تنها به n-gram ها توجه دارد و تعداد شباهت ها را محاسبه می کند. بهتر است از semantic metrics (مثل SPICE) برای بررسی کپشن پیش بینی شده و کپشن های رفرنس استفاده کنیم تا بتواند از لحاظ معنایی مقایسه را انجام دهد.

و) در صورتی که gradient vanishing داشته باشیم، یعنی مقدار گرادیان ها بسیار کوچک شده و در نتیجه پارامترها نمی توانند به خوبی آپدیت شوند و احتمالا loss به خوبی کاهش نمی یابد اما در هر ۳ قسمت کاهش loss مشاهده می شود و احتمالا دچار gradient vanishing نشده اند. البته برای بررسی بیشتر می توانیم از kernel weights distribution استفاده کنیم. اگر وزن ها کم کم به صفر میل کنند، gradient vanishing رخ داده است.

ز) مزایای استفاده از امبدینگ در سطح کاراکتر:

اندازه ماتریس امبدینگ به طور چشمگیری کاهش می یابد. در هر زبان تنها تعدادی کاراکتر داریم در صورتی که تعداد کلمات آن ممکن است هزار-میلیون باشد. در نتیجه به حافظه بسیار کمتری برای ذخیره ماتریس امبدینگ در سطح کاراکتر نیاز داریم.

تعداد کاراکترهای یک زبان ثابت است اما ممکن است با گذر زمان کلمات جدیدی به vocabulary یک زبان اضافه شود و در این صورت ماتریس امبدینگ مرتبا آپدیت و سایز آن بیشتر می شود. اما در سطح کاراکتر می توانیم کلمات جدید تولید کنیم.

معایب استفاده از امبدینگ در سطح کاراکتر:

اگر به خوبی آموزش نبیند، ممکن است دنباله ای کاراکترها را پشت هم بچیند و کلمات بی معنی تولید کند.

میان بردارهای کاراکترها فاصله معنایی وجود ندارد در صورتی که در سطح کلمه، فاصله بین بردارها می تواند اطلاعات خوبی به ما بدهد.

طول دنباله ورودی می تواند بسیار زیاد شود و این باعث کاهش سرعت محاسبات می شود.

منابع

https://towardsdatascience.com/understanding-gru-networks-2ef37df6c9be

https://medium.com/logivan/neural-network-embedding-and-dense-layers-whats-the-difference-fa177c6d0304

https://plainenglish.io/blog/understanding-collate-fn-in-pytorch-f9d1742647d3

https://medium.com/mlearning-ai/load-pre-trained-glove-embeddings-in-torch-nn-embedding-layer-in-under-2-minutes-f5af8f57416a

https://ai.stackexchange.com/questions/18132/how-to-detect-vanishing-gradients

https://www.lighttag.io/blog/character-level-NLP/

https://stats.stackexchange.com/questions/216000/advantage-of-character-based-language-models-over-word-based