به نام خدا

گزارش فاز دوم پروژه NLP

تسک image captioning

شقایق مبشر

ملیکا نوبختیان

غزل زمانی نژاد

تمامی کدها و داده ها در ریپازیتوری زیر موجود است:

https://github.com/ghzamani/ImageCaptioning

همچنین بدلیل حجم زیاد مدل ها موفق به قرار دادن آنها در گیت نشدیم. لینک مدل هر بخش در گوگل درایو در قسمت زیر آورده شده است:

بخش اول:

https://drive.google.com/drive/folders/1p_OcBVl4iezeJWX4itrxnOhTjgmFjzrl?usp=sharing

بخش دوم:

https://drive.google.com/drive/folders/14x6yTGip6YAZ_7Y295KCB-wUX7_bmhHj?usp=sharing

بخش سوم:

https://drive.google.com/drive/folders/1xS1dmIh0ggp8Zo7VjUobavJNKsYBKRqI?usp=sharing

بخش اول

Installs & Imports

ابتدا کتابخانه های مورد نیاز را نصب و import می کنیم. همچنین درایو گوگل را mount می کنیم تا بتوانیم دیتاست را از آن بخوانیم و مدل را در آن ذخیره کنیم.

Preprocess images and captions

در این قسمت باید روی کپشن و تصاویر پیش پردازش انجام دهیم. در ابتدا یک کلاس Vocabulary برای توکن های منحصر به فرد می سازیم. این کلاس شامل دو دیکشنری است که در یکی key خود کلمه و value ایندکس آن است و دیگری بالعکس (key ایندکس و value خود کلمه). متد add_word برای ساختن دیکشنری ها به کار می رود. در متد __call__ اگر یک کلمه در دیکشنری موجود نباشد به جای آن توکن <unk> قرار می دهیم.

```
class Vocabulary(object):
    def __init__(self):
        self.word2idx = {}
        self.idx2word = {}
        self.idx = 0

    def add_word(self, word):
        if not word in self.word2idx:
```

```
            self.word2idx[word] = self.idx
            self.idx2word[self.idx] = word
            self.idx += 1

    def __call__(self, word):
        if not word in self.word2idx:
            return self.word2idx['<unk>']
        return self.word2idx[word]

    def __len__(self):
        return len(self.word2idx)
```

در متد build_vocab، ابتدا فایل json دیتاست را می خوانیم. سپس به دلیل مشکل RAM و crash شدن کولب، به ناچار از بخشی از دیتاست استفاده کردیم. یک نمونه از کلاس Vocabulary تشکیل دادیم و با iterate کردن روی داده ها آنها را tokenize کردیم و به دیکشنری ها اضافه کردیم.

```
def build_vocab(path, count):
    with open(path, 'r') as f:
        dataset_json = json.load(f)

    dataset_json = dataset_json[:count]

    vocab = Vocabulary()
    vocab.add_word('<pad>') # 0
    vocab.add_word('<start>') # 1
    vocab.add_word('<end>') # 2
    vocab.add_word('<unk>') # 3

    for sample in dataset_json:
        caption = sample['caption']
        tokens = nltk.tokenize.word_tokenize(caption)
        for t in tokens:
            vocab.add_word(t)

    return vocab
```

از متد بالا استفاده کرده و برای 500 نمونه از دیتاست، واژگان منحصر به فرد را تشکیل می دهیم و در قالب یک فایل pickle آن را ذخیره می کنیم.

```
caption_path = "dataset/cleaned_dataset.json"
vocab_path = "gdrive/MyDrive/vocab.pkl"
vocab = build_vocab(caption_path, 500)
with open(vocab_path, 'wb') as f:
    pickle.dump(vocab, f)
```

از متد resize_image برای هم اندازه کردن تصاویر استفاده می شود. ابتدا تصویر کراپ شده و سپس به 224*224 تغییر اندازه می دهد.

```
def resize_image(image):
    width, height = image.size
    if width > height:
        left = (width - height) / 2
        right = width - left
        top = 0
        bottom = height
    else:
        top = (height - width) / 2
        bottom = height - top
        left = 0
        right = width
    image = image.crop((left, top, right, bottom))
    image = image.resize([224, 224], Image.ANTIALIAS)
    return image
```

با استفاده از متد preprocess_image آدرس تصاویر موجود در دایرکتوری را ذخیره می کنیم و بر روی یک به یک آنها متد resize_image را صدا می زنیم و خروجی را در درایو گوگل ذخیره می کنیم تا دفعات بعد نیاز به انجام تمام نباشد.

```
def prepocess_image(caption_path):
    with open(caption_path, 'r') as f:
        dataset_json = json.load(f)
    dataset_len = len(dataset_json)
    dataset_ids = [*range(dataset_len)]
    image_paths = ['dataset/images/{0:09d}.jpg'.format(id) for id in datas
et_ids]
    resized_paths = ['gdrive/MyDrive/resized/{0:09d}.jpg'.format(id) for i
d in dataset_ids]

    resized_folder = 'gdrive/MyDrive/resized'
    if not os.path.exists(resized_folder):
        os.makedirs(resized_folder)

        counter = 0
        for img_path, resized_path in zip(image_paths, resized_paths):
            with Image.open(img_path) as image:
                image = resize_image(image)
                image.save(resized_path)
            if(counter%1000 == 0):
```

```
            print("finished", counter)
        counter += 1
```

DataLoader:

در پایتورچ از آن برای تنظیم دیتاست به صورت دلخواه و iterable استفاده می شود. در اینجا شامل کلمات منحصر به فرد، آدرس جایی که تصاویر در آن ذخیره شده، دیتاست و transform می شود. در متد __get__item، تصویر با ایندکس خواسته شده را باز می کنیم و بر روی آن transform را اعمال می کنیم. سپس کپشن را tokenize می کنیم و با استفاده از vocab کلمات آن را به ایندکس آن واژه تبدیل می کنیم. به ابتدا و انتهای کپشن به ترتیب توکن های <start> و <end> می افزاییم. آن را به تنسور تبدیل می کنیم، به همراه تصویر return می کنیم.

```
class DataLoader(data.Dataset):
    def __init__(self, root, dataset, vocab, transform=None):

        self.root = root
        self.dataset = dataset
        self.vocab = vocab
        self.transform = transform

    def __getitem__(self, index):
        vocab = self.vocab
        caption = self.dataset[index]['caption']
        path = '{0:09d}.jpg'.format(index)
        image = Image.open(os.path.join(self.root, path)).convert('RGB')
        if self.transform is not None:
            image = self.transform(image)

        tokens = nltk.tokenize.word_tokenize(str(caption))
        caption = []
        caption.append(vocab('<start>'))
        caption.extend([vocab(token) for token in tokens])
        caption.append(vocab('<end>'))
        target = torch.Tensor(caption)
        return image, target

    def __len__(self):
        return len(self.dataset)
```

اگر طول ورودی متغیر باشد، از متد collate_fn استفاده می کنیم.

```
def collate_fn(data):
    data.sort(key=lambda  x: len(x[1]), reverse=True)
    images, captions = zip(*data)

    images = torch.stack(images, 0)
```

```
    lengths = [len(cap) for cap in captions]
    targets = torch.zeros(len(captions), max(lengths)).long()
    for i, cap in enumerate(captions):
        end = lengths[i]
        targets[i, :end] = cap[:end]
    return images, targets, lengths
```

متد get_loader برای خواندن دیتاست (با تعداد سطرهای مورد نظر) و ساختن شی dataloader به کار می رود.
در خروجی dataloader را که با سایز batch دلخواه ساخته شده برمی گرداند.

```
def get_loader(vocab, batch_size, count):
    root = 'gdrive/MyDrive/resized'
    with open('dataset/cleaned_dataset.json', 'r') as f:
        dataset_json = json.load(f)
    dataset_json = dataset_json[:count]

    # resnet transformation/normalization
    transform = transforms.Compose([
        transforms.RandomCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize((0.485, 0.456, 0.406),
                             (0.229, 0.224, 0.225))])

    dataset = DataLoader(root=root, dataset=dataset_json, vocab=vocab, tra
nsform=transform)

    data_loader = torch.utils.data.DataLoader(dataset=dataset,
                                              batch_size=batch_size,
                                              shuffle=False,
                                              num_workers=1,
                                              collate_fn=collate_fn)
    return data_loader
```

مقادیر هایپرپارامترها را تنظیم می کنیم و dataloaderها را می سازیم.

```
PAD = 0
START = 1
END = 2
UNK = 3

grad_clip = 5.
num_epochs = 1
batch_size = 32
```

```
decoder_lr = 0.0004

# Load vocabulary wrapper
with open('gdrive/MyDrive/vocab.pkl', 'rb') as f:
    vocab = pickle.load(f)
print(len(vocab))

# load data
train_loader = get_loader(vocab, batch_size, 500)
test_loader = get_loader(vocab, 5, 20)
```

:Build Model

در این بخش ابتدا باید با استفاده از encoder ویژگی های تصاویر را استخراج کنیم. سپس از مدل BERT برای کپشن ها استفاده می کنیم و در نهایت از یک decoder بهره می بریم که با بردار embedding تصویر و متن، خروجی را تولید می کند. این decoder برای عملکرد بهتر از attention نیز استفاده می کند.

مدل و توکنایزر BERT را استفاده می کنیم. لایه های نخستین BERT را فریز می کنیم (برای اینکار مقدار requires_grad را false قرار می دهیم) اما لایه های انتهایی آن را آموزش می دهیم.

```
# Load pre-trained tokenizer (vocabulary)
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

# Load pre-trained model (weights)
model = BertModel.from_pretrained('bert-base-uncased')

for name, param in list(model.named_parameters())[:-79]:
    param.requires_grad = False
```

برای استخراج ویژگی تصاویر از مدل از پیش آموخته RESNET بهره بردیم. دو لایه آخر که برای تسک طبقه بندی تصاویر آموزش دیده بودند را کنار می گذاریم و به جای آنها از یک لایه pooling استفاده می کنیم. Fine tuning بر روی مدل RESNET انجام نمی دهیم.

```
# enocder
class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        resnet = models.resnet101(pretrained=True)
        self.resnet = nn.Sequential(*list(resnet.children())[:-2])
        self.adaptive_pool = nn.AdaptiveAvgPool2d((14, 14))

    def forward(self, images):
        out = self.adaptive_pool(self.resnet(images))
        # batch_size, img size, imgs size, 2048
```

```
        out = out.permute(0, 2, 3, 1)
        return out
```

در ابتدا از لایه های attention استفاده می کنیم زیرا می توانند در معماری encoder-decoder عملکرد مدل را بهبود بخشند زیرا در هر قدم به بخش خاصی توجه بیشتری می کند. برای overfit نشدن مدل از لایه dropout استفاده می کنیم. از یک لایه LSTM استفاده می کنیم که اندازه ورودی آن، مجموع اندازه embedding تصویر و خروجی BERT است. لایه های hidden،cell state و forget را نیز اضافه می کنیم.

```
# decoder
class Decoder(nn.Module):

    def __init__(self, vocab_size):
        super(Decoder, self).__init__()
        self.encoder_dim = 2048
        self.attention_dim = 512
        self.embed_dim = 768

        self.decoder_dim = 512
        self.vocab_size = vocab_size
        self.dropout = 0.5

        # soft attention
        self.enc_att = nn.Linear(2048, 512)
        self.dec_att = nn.Linear(512, 512)
        self.att = nn.Linear(512, 1)
        self.relu = nn.ReLU()
        self.softmax = nn.Softmax(dim=1)

        # decoder layers
        self.dropout = nn.Dropout(p=self.dropout)
        self.decode_step = nn.LSTMCell(self.embed_dim + self.encoder_dim,
self.decoder_dim, bias=True)
        self.h_lin = nn.Linear(self.encoder_dim, self.decoder_dim)
        self.c_lin = nn.Linear(self.encoder_dim, self.decoder_dim)
        self.f_beta = nn.Linear(self.decoder_dim, self.encoder_dim)
        self.sigmoid = nn.Sigmoid()
        self.fc = nn.Linear(self.decoder_dim, self.vocab_size)

        # init variables
        self.fc.bias.data.fill_(0)
        self.fc.weight.data.uniform_(-0.1, 0.1)
```

در تابع forward، اندازه طولانی ترین کپشن را محاسبه می کنیم. سپس کپشن ها را در صورت نیاز pad می کنیم تا طول آنها یکسان شود. آنها را tokenize می کنیم و سپس از BERT عبور می دهیم. خروجی لایه 12 (لایه آخر) را نگه می داریم. با جمع کردن بردارهای خروجی BERT که به همان کلمه اصلی تعلق دارند، embeddingها را detokenize می کنیم. پس از انجام این مراحل به embedding کپشن ها می رسیم. برای ساختن batch_size امبدینگ ها را stack می کنیم.
بعد از روی خروجی انکودر hidden stateهای دیکودر را با لایه linear، initialize می کنیم. باقی مراحل به ساختار مدل مربوط است.

در اولین بار اجرا مدل را می سازیم، اما اگر از قبل موجود باشد آن را load می کنیم.

```
# initialize a new model
# decoder = Decoder(vocab_size=len(vocab)).to(device)
# decoder_optimizer = torch.optim.Adam(params=decoder.parameters(),lr=deco
der_lr)

# encoder = Encoder().to(device)

# load model
encoder = Encoder().to(device)
encoder_checkpoint = torch.load('./gdrive/MyDrive/checkpoints/encoder')
encoder.load_state_dict(encoder_checkpoint['model_state_dict'])

decoder = Decoder(vocab_size=len(vocab)).to(device)
decoder_optimizer = torch.optim.Adam(params=decoder.parameters(),lr=decode
r_lr)
decoder_checkpoint = torch.load('./gdrive/MyDrive/checkpoints/decoder')
decoder.load_state_dict(decoder_checkpoint['model_state_dict'])
decoder_optimizer.load_state_dict(decoder_checkpoint['optimizer_state_dict
'])
```

برای تابع ضرر از cross-entropy استفاده می کنیم.

```
criterion = nn.CrossEntropyLoss().to(device)
```

همچنین یک کلاس loss_obj برای نگه داشتن مقدار میانگین خطاها تعریف می کنیم.

```
class loss_obj(object):
    def __init__(self):
        self.avg = 0.
        self.sum = 0.
        self.count = 0.

    def update(self, val, n=1):
        self.sum += val * n
        self.count += n
        self.avg = self.sum / self.count
```

Train

حالا به مرحله trainمی‌رسیم و مدل خود را آموزش می‌دهیم. لازم است ابتدا تصاویری که در هر batch از دیتاست به دست می‌آوریم را به encoder بدهیم. سپس ویژگی‌های به دست آمده از encoder به همراه caption تصویر و طول آن را به decoder بدهیم تا نتیجه نهایی را به دست آوریم. خروجی‌های train در اینجا scores, caps_sorted, decode_lengths, alphas خواهند بود. در قدم بعد scores را به pack_padded_sequence خواهیم داد تا با pack کردن label هایی که داریم و با داشتن خروجی RNN که داریم به طور مستقیم loss را حساب کنیم. targets هم ورودی دوم خواهد بود که pack خواهد شد و در مرحله بعد برای محاسبه lossمورد استفاده قرار خواهد گرفت. سپس از این loss برای آموزش مدل استفاده خواهیم کرد.

سپس پارامترهایی ازoptimizer که دارای گرادیان هستند در یک بازه مشخص قرار می‌دهیم و در نهایت optimizer را اجرا کرده و مقدار loss را آپدیت می‌کنیم تا به مقدار نهایی آن batch برسیم. هم چنین هر epoch 5 نیز مقدار نرخ یادگیری را تنظیم خواهیم کرد و مدل‌های decoder و encoder را در checkpoint ذخیره خواهیم کرد. در پایان آموزش نیز مقدار میانگین کلی loss را خواهیم داشت.

```
Batch 0/16 loss:tensor(7.0080)
Batch 0/16 loss:tensor(7.0080)
saving model...
model saved
Batch 1/16 loss:tensor(6.9140)
Batch 2/16 loss:tensor(6.9299)
Batch 3/16 loss:tensor(6.9744)
Batch 4/16 loss:tensor(7.0085)
Batch 5/16 loss:tensor(7.0224)
```

Generate Captions

تابع print_sample

ایندکس پیش‌بینی‌ها و کپشن‌های حقیقی (references) را به کلمات خود تبدیل می‌کنیم. از بین تصاویر تصویر kام را برمیداریم. در انتها این تصویر و دو کپشن حقیقی و پیش‌بینی را پرینت می‌کنیم.

تابع validate

این متد برای تست کردن test_loader است که همانند train ابتدا ورودی‌های گرفته شده از test_loader را به GPU یا CPU (در device مشخص شده) برده و سپس آن‌ها را به مدل می‌دهد تا پیش‌بینی مدل و scoreهای آن را به دست آورد. چون

این دو pad شده اند تا به یک اندازه مشخص درآیند، به کمک pack_padded_sequence به تعداد گام‌های زمانی درست خود بازمیگردند.

همانند متد train خطا را به کمک متد criterion محاسبه کرده و losses را آپدیت می‌کنیم.

سپس برای هر کدام از نمونه‌های حقیقی درون batch با یک حلقه for کپشن‌هایش جدا کرده، از توکن‌های PAD و START و END پاک می‌کنیم و آن را به رفرنس ها اضافه می‌کنیم.

همین کار را برای فرض مدل یا پیش‌بینی آن تکرار می‌کنیم: برای به دست آوردن یک توکن برای هر گام زمانی از torch.max در بعد ۲ استفاده کرده‌ایم پس ماکزیمم امتیاز را در یک گام زمانی به عنوان کلمه انتخاب شده برمی‌گرداند. از این برای پیدا کردن ایندکس پیش‌بینی‌ها از decode_lengths کمک می‌گیریم. پس از حذف توکن‌های PAD، START و END آن‌ها را در پیش‌بینی‌ها نگه می‌داریم.

در آخر با توجه به اضافه کردن کپشن حقیقی و پیش‌بینی‌ها خود تصویر را به یک لیست اضافه می کند.

برای ارزیابی داده‌های به دست آمده از bleu استفاده می‌کنیم و با وزن‌های مختلف آن را چاپ می‌کنیم.

خروجی:

```
Validation loss: tensor(6.8279)
BLEU: 2.2822147006616005e-155
BLEU-1: 0.18904109589041093
BLEU-2: 0.00289855072463768
BLEU-3: 2.2250738585072626e-308
BLEU-4: 2.2250738585072626e-308
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Hypotheses: a a the the the the the the the the the the the the the the the the the the the the the the the the the . . . . the the
References: a man stands on a rock pillar in the middle of an underground cave a large ray of light casts down on him from an opening in the earth above .
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Hypotheses: a a a a a the the the the the the the the the the the the the the the the the the the the the the
References: a hiker with a blanket draped over their shoulders looks out over a bay surrounded by steep cliffs and mountain peaks .
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Hypotheses: a a a the the the the . . . . . .
References: a badass camper stands in front of a tent ready to haul ass shop trees and take nature on .
```

```
Hypotheses: a a a the the the the the the the the the the the . . .
References: footsteps leave imprints in a sand dune as a person takes a journey across the horizon .
```

کپشن تولید شده خروجی نامطلوبی بود. زیرا مدل نتوانست روی تعداد زیادی داده آموزش ببیند.

بخش دوم

دریافت دیتاست فاز قبل

برای گرفتن دیتاست فاز قبل، می‌توان از روش دانلود فایل زیپ آن که در گوگل درایو share کرده‌ایم استفاده کنید:

```
#To download the dataset from shared link (hardly ever works...)
!curl -L -
o 'NLP_dataset.zip' 'https://drive.google.com/u/0/uc?id=1_UqE0UPcIEeotujyY5Do
im7k02hkD2ix&export=download&confirm=t'
# import gdown
# gdown.download(
```

```
#     f"https://drive.google.com/uc?export=download&confirm=pbef&id=1_UqE0UPc
IEeotujyY5Doim7k02hkD2ix",
#     "NLP_dataset.zip"
# )
!mkdir dataset
!unzip NLP_dataset.zip -d dataset
```

این روش ممکن است به دلیل دانلود چند نفر (حتی در حد دو یا سه دانلود) از کار بیفتد و برای مدتی Access Denied دهد و سپس دوباره شروع به کار کند. به همین دلیل ما به جای استفاده از روش ذکر شده، این فایل share شده را به درایو خود به صورت shortcut اضافه کرده‌ایم و آن را با mount کردن گوگل درایو از درایو برداشته و درون فولدری به نام dataset از حالت زیپ آن را استخراج می‌کنیم:

```
#To get the dataset from drive

# mount Google Drive
from google.colab import drive
drive.mount('/content/gdrive')

!mkdir dataset
!unzip gdrive/MyDrive/NLP_dataset.zip -d dataset
```

پیش‌پردازش داده‌ها

```
with open("dataset/cleaned_dataset.json", 'r') as f:
    dataset_json = json.load(f)

dataset_len = len(dataset_json)
dataset_ids = [*range(dataset_len)]

#shuffle dataset
random.Random(3).shuffle(dataset_ids)

captions = [f"<start> {dataset_json[id]['caption']} <end>" for id in dataset_ids]
image_paths = ['dataset/images/{0:09d}.jpg'.format(id) for id in dataset_ids]

#take 80% as train set, and 20% as test
train_length = int(dataset_len * .8)
test_length = dataset_len - train_length
```

ابتدا cleaned_dataset.json را که یک لیست شامل داده‌های cleaned از فاز قبل است باز می‌کنیم. از این فایل می‌خواهیم تمام کپشن‌ها را برداریم. هر کپشن درون یک آبجکت با کلید "caption" است. همچنین آبجکت‌ها به ترتیب id آن نمونه به لیست درون فایل json اضافه شده اند پس هر اینکس همان id داده مورد نظر می‌شود. پس با تولید لیستی از range ۰ تا طول دیتاست این id ها را میسازیم و به کمک تابع random.shuffle (که با Random(num) یک seed خاص به آن داده شده و با ران‌های متفاوت باز هم یک دنباله خاص و یکسانی از اعداد تصادفی را ایجاد می‌کند) آن را به هم می‌ریزیم.

در ادامه هر کپشن را با ترتیب id به هم ریخته شده با دو توکن <start> و <end> استخراج کرده و در Captions ذخیره می‌کنیم.
همچنین مسیر تک تک تصاویر مربوط به دیتاست به هم ریخته شده را در image_paths ذخیره می‌کنیم. تمام تصاویر دارای نام فایل با ۹ رقم هستند به طوری که از چپ برای رسیدن به ۹ رقم، با صفر پر می‌شوند. به همین دلیل نامشان با {0:09d}.jpg فرمت می‌شود.

با توجه به صورت سوال باید ۸۰ درصد طول دیتاست مختص آموزش و بقیه برای تست استفاده شوند. به همین ترتیب train_length و test_length نیز به دست می‌آیند.

پیش‌پردازش تصاویر

ما تمامی تصاویر را یک بار به یک مدل inception v3 می‌دهیم و featureهای حاصل از آن را در فایل‌های numpy جدا ذخیره کرده و به جای خود تصاویر در مدل استفاده می‌کنیم.

```
def load_image(image_path):
  img = tf.io.read_file(image_path)
  img = tf.io.decode_jpeg(img, channels=3)
  img = tf.keras.layers.Resizing(299, 299)(img)
  img = tf.keras.applications.inception_v3.preprocess_input(img)
  return img, image_path
```

تابع load_image ابتدا مسیر یک تصویر را می‌گیرد، آن را خوانده و decode کرده سپس برای ورودی دادن به inception v3 آن را ریسایز می‌کند تا به اندازه ورودی inception دربیاید. سپس آن را به متد preprocess_input از inception_v3 میدهیم که یک تنسور پیش‌پردازش شده را از تصویر برمیگرداند که ورودی inception خواهد بود.

```
def create_inceptionv3_model():
  inception_model = tf.keras.applications.InceptionV3(include_top=False,
                                                       weights='imagenet')
  return tf.keras.Model(inception_model.input, inception_model.layers[-
1].output)
```

از keras.applications مدل InceptionV3 را initialize می‌کنیم. Include_top = False است زیرا نیازی به لایه‌های آخر کلاسیفایرش نداریم. همچنین می‌خواهیم از وزن‌های از پیش آموزش داده شده بر روی دیتاست imagenet استفاده کند. سپس یک مدل با ورودی برابر با ورودی مدل inception و خروجی برابر با خروجی لایه آخر inception_model می‌سازیم و این مدل را برمی‌گردانیم.

```
#get image features from inceptionv3 model's last layer
img_dataset = tf.data.Dataset.from_tensor_slices(image_paths)
img_dataset = img_dataset.map(
  load_image, num_parallel_calls=tf.data.AUTOTUNE).batch(32)
```

یک دیتاست از مسیرهای تصاویر می‌سازیم که سپس به کمک تابع map، آن‌ها را به خروجی‌های load_image تبدیل می‌کنیم که یک تنسور پیش پردازش شده برای inception v3 است.

سپس مدل inception را ساخته و برای دیتاست تصاویر featureها را تولید می‌کنیم. تک تک این featureها را در مسیر تصویر با فرمت .npy ذخیره می‌کنیم.

پیش‌پردازش متن

```
def get_longest_caption_length(captions):
```

```
  return max([len(caption.split()) for caption in captions])

output_sequence_length = get_longest_caption_length(captions)
train_captions = captions[:train_length]
```

در ابتدا بیشترین تعداد توکن‌های کپشن‌ها را با تابع get_longest_caption_length می‌یابیم (برابر با ۷۵ است). همچنین کپشن‌های بخش آموزش را از Train_lengthتای اول برمی‌داریم.

```
# create a tokenizer and get vocabulary from train captions using TextVect
orization
tokenizer = tf.keras.layers.TextVectorization(
  standardize=None, # we already preprocessed captions in the last phase
  max_tokens=5000,
  output_sequence_length=output_sequence_length)

caption_dataset = tf.data.Dataset.from_tensor_slices(train_captions)
tokenizer.adapt(caption_dataset)
```

یک توکنایزر با TextVectorization می‌سازیم. چون در فاز قبلی کپشن‌ها را clean کرده ایم و همه با اسپیس از هم جدا می‌شوند نیازی به استاندارد شدن ندارند. همچنین تعداد توکن‌ها را ۵۰۰۰ انتخاب کرده و طول دنباله را output_sequence_length می‌گذاریم. یک دیتاست از کپشن‌های آموزش ساخته و توکنایز را با آن adapt می‌کنیم تا بتواند ۵۰۰۰ توکن با بیشترین فرکانس تکرار را پیدا کند. (چون مدل تنها داده‌های آموزش را میشناسد از کپشن‌های آموزش استفاده می‌کنیم و کپشن‌های تست برای مدل مجهولند)

```
#get word2index and index2word for mapping the words and indices
word_to_index = tf.keras.layers.StringLookup(
    mask_token="",
    vocabulary=tokenizer.get_vocabulary())
index_to_word = tf.keras.layers.StringLookup(
    mask_token="",
    vocabulary=tokenizer.get_vocabulary(),
    invert=True)
```

ساخت دیتاست

ابتدا سه هایپرپارامتر را مشخص کرده‌ایم:

```
BATCH_SIZE = 64
embedding_dim = 512
EPOCHS = 40
```

در ادامه ابتدا مسیر تصاویر مربوط به آموزش را جدا کرده و یک دیتاست از آن همراه با کپشن‌هایش می‌سازیم. این دو باید تبدیل به Feature تصاویر و ایندکس‌های توکن‌های کپشن شوند، پس یک تابع تعریف کرده و دیتاست را به آن map می‌کنیم:

```
#image path and caption in string form will be turned to image features an
d a list of token indices for caption
def create_dataset_data(image_path, caption):
  cap_vec = tokenizer(caption)
```

```
  img_vec = np.load(image_path.decode('utf-8')+'.npy')
  return img_vec, cap_vec
```

این متد به کمک توکنایزر caption را تبدیل به ایندکس های توکن‌ها می‌کند. همچنین از مسیر image_path فایل npy. را لود کرده تا featureهای تصاویر به دست آید.

در پایان این دیتاست را با BATCH_SIZE = 64 دسته بندی کرده و با prefetch عمل گرفتن batchهای بعدی را سریع‌تر می‌کنیم. Tf.data.AUTOTUNE پارامتری که به آن اعمال شده را به صورت اتوماتیک ست کرده و تنظیم می‌کند.

مدل کلی

مدل کلی در کلاس ImageCaptioningModel ایجاد شده است. این کلاس طرز محاسبه loss و accuracy و همچنین متدهای آموزش و آزمون را برای یک batch داراست. متدهای تعریف شده درون این کلاس را به ترتیب بررسی می‌کنیم:

```
def __init__(
        self, encoder, decoder, units
    ):
    super().__init__()
    self.units = units
    self.encoder = encoder
    self.decoder = decoder
    self.loss_tracker = keras.metrics.Mean(name="loss")
    self.acc_tracker = keras.metrics.Mean(name="accuracy")
```

این مدل از یک انکودر و یک دیکودر تشکیل شده است که در ادامه توضیح داده می‌شود. اندازه units همان اندازه hidden states برای LSTMهای استفاده شده درون دیکودر است.

همچنین یک loss_tracker و accuracy_tracker داریم که خطاها و دقت‌های داده شده به آن‌ها را میانگین می‌گیرند.

آموزش و تست مدل برای هر batch

```
def train_step(self, batch_data):
    batch_img, batch_seq = batch_data
    batch_loss = 0
    batch_acc = []

    batch_seq_size = batch_seq.get_shape().as_list()[0]
    hidden = tf.zeros((batch_seq_size, self.units))
    decoder_input = tf.expand_dims([word_to_index('<start>')] * batch_seq_
size, 1)
    with tf.GradientTape() as tape:
      encoder_out = self.encoder(batch_img, training=True)
      for i in range(1, batch_seq.shape[1]):
        predictions, hidden = self.decoder(decoder_input, hidden, encoder_
out)
```

```
            mask = tf.math.not_equal(batch_seq[:,i], 0)
            loss = self.calculate_loss(batch_seq[:,i], predictions, mask)
            batch_loss += loss
            acc = self.calculate_accuracy(batch_seq[:,i], predictions, mask)
            if acc != -1:
              batch_acc.append(acc)

            # using teacher forcing
            decoder_input = tf.expand_dims(batch_seq[:, i], 1)

        total_loss = (batch_loss / int(batch_seq.shape[1]))
        total_acc = tf.reduce_mean(batch_acc)
        trainable_variables = self.encoder.trainable_variables + self.decoder.
trainable_variables
        gradients = tape.gradient(batch_loss, trainable_variables)
        self.optimizer.apply_gradients(zip(gradients, trainable_variables))

        self.loss_tracker.update_state(total_loss)
        self.acc_tracker.update_state(total_acc)

        return {"loss": self.loss_tracker.result(), "acc": self.acc_tracker.re
sult()}
```

برای آموزش یک گام با یک batch از داده، از متد train_step استفاده می‌کنیم. Batch_data دارای فیچرهای تصاویر (batch_img) و کپشن‌ها به صورت لیستی از indexهای توکن‌های آن (batch_seq) است.

سپس hidden_state های lstm دوم که برای اتنشن قرار است استفاده شود را initialize می‌کنیم (در ابتدا همه صفر هستند).

اولین timestep برای این batch ورودیش کلمه <start> است به همین دلیل index آن را به تعداد batch_size تکرار می‌کنیم. در ادامه برای گرفتن خروجی انکودر که تنها یک بار اجرا می‌شود از self.encoder استفاده می‌کنیم.

سپس به تعداد timestepهای ممکن برای تولید یک کپشن (که در shape[1] از batch_seq مشخص شده) باید از decoder استفاده کرده و خطا و دقت را محاسبه کنیم. پس از گرفتن خروجی دیکودر که دارای hidden stateهای جدید و پیش‌بینی لایه آخر دیکودر است، برای محاسبه خطا و دقت mask را تشکیل می‌دهیم که برای توکن غیر صفر کپشن حقیقی در آن گام زمانی ۱ و برای دیگر توکن‌ها ۰ برمیگرداند. پس از محاسبه خطا و دقت که در ادامه توضیح داده می‌شود، باید ورودی گام زمانی بعدی را آماده کنیم که میشود index توکن این گام زمانی از داده حقیقی. (به دلیل استفاده از teacher forcing ورودی گام زمانی t برای آموزش برابر با توکن داده حقیقی گام زمانی t-1 است.)

پس از محاسبه تمام گام‌های زمانی میانگین loss و accuracyرا حساب کرده و loss_tracker و acc_tracker را با آن‌ها آپدیت می‌کنیم. همچنین به کمک loss به دست آمده back propagation را برای تمام متغیرهای قابل یادگیری انکودر و دیکودر اجرا می‌کنیم.

```
def test_step(self, batch_data):
    batch_img, batch_seq = batch_data
```

```
    batch_loss = 0
    batch_acc = []

    encoder_out = self.encoder(batch_img)
    batch_seq_size = batch_seq.get_shape().as_list()[0]
    hidden = tf.zeros((batch_seq_size, self.units))
    decoder_input = tf.expand_dims([word_to_index('<start>')] * batch_seq_
size, 1)

    for i in range(output_sequence_length):
      predictions, hidden= self.decoder(decoder_input, hidden, encoder_out
)
      predicted_id = tf.random.categorical(predictions, 1)[:,0].numpy()
      mask = tf.math.not_equal(batch_seq[:,i], 0)
      loss = self.calculate_loss(batch_seq[:,i], predictions, mask)
      batch_loss += loss
      acc = self.calculate_accuracy(batch_seq[:,i], predictions, mask)
      if acc != -1:
        batch_acc.append(acc)

      decoder_input = tf.expand_dims(predicted_id, 1)

    total_loss = (batch_loss / int(batch_seq.shape[1]))
    total_acc = tf.reduce_mean(batch_acc)
    self.loss_tracker.update_state(total_loss)
    self.acc_tracker.update_state(total_acc)

    return {"loss": self.loss_tracker.result(), "acc": self.acc_tracker.re
sult()}
```

برای داده‌های تست هم کاری مشابه آموزش انجام می‌دهیم فقط به دلیل اینکه مرحله تست است نیازی به نگه داشتن gradientها و اعمال آن‌ها برای back propagation ندارد. همچنین دیگر نمی‌توان از کپشن‌های حقیقی به عنوان ورودی دیکودر استفاده کرد و باید پیش بینی گام زمانی قبلی را به عنوان ورودی گام زمانی بعد به کار برد. دیگر قسمت‌ها تفاوتی با متد train_step ندارند.

نحوه محاسبه loss و accuracy

```
def calculate_loss(self, y_true, y_pred, mask):
    loss = self.loss(y_true, y_pred)
    mask = tf.cast(mask, dtype=loss.dtype)
    loss *= mask
    mask_reduced = tf.reduce_sum(mask)
    if mask_reduced == 0:
      return 0
    return tf.reduce_sum(loss) / mask_reduced
```

برای محاسبه loss، ابتدا باید به کمک داده‌های حقیقی و پیش‌بینی مدل و متد lossای که برای این مدل انتخاب شده است، loss تمام ایندکس‌های ممکن را به دست می‌آورد. سپس maskای که قبلا ایجاد کرده ایم را در این loss ضرب می‌کنیم تا تنها loss ایندکس‌های موردنظرمان (که داده‌های حقیقی دارند) را نگه داریم. در صورتی که جملات همه در این گام زمانی به پایان خود رسیده باشند، تمام mask ۰ خواهد بود چون هیچ کدام از توکن‌های درون vocab انتخاب نشده است. در این حالت ما loss نداریم (زیرا قبل از آن جمله با <end> به پایان رسیده است.). میتوانستیم برای این خانه‌های خالی نیز یک توکن درنظر بگیریم.

محاسبه دقت نیز مشابه loss است. برای محاسبه دقت بین داده‌های پیش‌بینی و داده‌های حقیقی، بزرگترین احتمال بین تمام توکن‌های هر نمونه از پیش‌بینی را انتخاب کرده و آن را با داده‌های حقیقی به کمک تابع tf.equal مقایسه می‌کنیم. در صورتی که هر کدام از توکن‌های پیش‌بینی شده درست باشند، این تابع برای آن نمونه ۱ را برمیگرداند. سپس دوباره از mask استفاده می‌کنیم. در صورتی که mask برای هر کدام از نمونه‌ها صفر شود دیگر دقت برای آن یک نمونه داده نباید محاسبه شود به همین دلیل جمع تمام دقت‌ها را تقسیم بر جمع mask می‌کنیم تا میانگین دقت برای نمونه‌هایی که هنوز پایان نیافته‌اند محاسبه شود. در صورتی که تمام داده‌ها برای mask صفر شود ۱- برمیگردانیم تا در تابع اصلی استفاده کننده از دقت، این دقت در نظر گرفته نشود.

ساختار مدل انکودر-دیکودر

انکودر برای ما ویژگی‌های تصویر را استخراج خواهد کرد. تصاویر در ابتدا به یک مدل InceptionV3 داده شده‌اند و خروجی آن به عنوان ویژگی تصویر شناخته می‌شود. برای این کار ما کلاس ImageEncoder را داریم که شامل یک لایه dense با ابعاد خروجی به اندازه embedding_dim است که به صورت زیر پیاده‌سازی و تعریف می‌شود:

```
class ImageEncoder(keras.Model):
  def __init__(self, embedding_dim):
    super(ImageEncoder, self).__init__()
    self.out_fc = tf.keras.layers.Dense(embedding_dim,activation='relu')

  def call(self, x):
    x = self.out_fc(x)
    return x

Image_encoder = ImageEncoder(embedding_dim)
```

مرحله بعد تعریف Decoder است. برای ساختن Decoder باید تعداد vocabulary و هم چنین embedding_dim را داشته باشیم. Decoder در این قسمت ساختار ساده‌ای دارد که شامل یک لایه embedding، یک LSTM و یک لایه dense نهایی است. تعریف کلاس decoder را در زیر می‌بینید:

```
class Decoder(tf.keras.Model):
  def __init__(self, vocab_size, embedding_dim):
    super(Decoder, self).__init__()
    self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
    self.lstm2 = tf.keras.layers.LSTM(512,
                                      return_sequences=True,
                                      return_state=True)
```

```
    self.out_fc = tf.keras.layers.Dense(vocab_size)

  def call(self, x, hidden, features):
    x = self.embedding(x)  # (batch_size, 1, embedding_dim)
    x, new_hidden, _= self.lstm2(x)
    x = tf.reshape(x, (-1, x.shape[2]))
    x = self.out_fc(x)
    return x, new_hidden
```

پس از انجام مراحل ساخت دیتاست و آماده‌سازی مدل و انکودرها و دیکودرها نوبت به آموزش مدل می‌رسد. ابتدا لازم است یک سری از موارد را تعریف کنیم. تابع loss که برای آموزش مدل در نظر گرفته‌ایم SparseCategoricalCrossEntropy خواهد بود و به عنوان optimizer نیز از Adam استفاده می‌کنیم. learning rate را هم برابر 0.01 قرار می‌دهیم:

```
# Define the loss function
cross_entropy = keras.losses.SparseCategoricalCrossentropy(
    from_logits=True, reduction="none"
)
optimizer = keras.optimizers.Adam(learning_rate=0.001)
```

برای اینکه بتوانیم پیشرفت مدل را ثبت کنیم و هر زمان که خواستیم train را دوباره از نقطه قبلی ادامه دهیم از checkpoint استفاده می‌کنیم. در یک مسیر که خودمان مشخص می‌کنیم checkpoint ها به طور منظم ذخیره خواهند شد و در صورتی که بخواهیم دوباره به آن دسترسی داشته باشیم، با چک کردن مسیری که checkpoint ها را در آن ذخیره کردیم آخرین آن‌ها را بازیابی خواهیم کرد:

```
checkpoint_path = "/content/drive/MyDrive/checking"
ckpt = tf.train.Checkpoint(encoder=Image_encoder,
                           decoder=decoder,
                           optimizer=optimizer)
ckpt_manager = tf.train.CheckpointManager(ckpt, checkpoint_path, max_to_ke
ep=5)

epoch_start = 0
if ckpt_manager.latest_checkpoint:
  epoch_start = int(ckpt_manager.latest_checkpoint.split('-')[-1])
  # restoring the latest checkpoint in checkpoint_path
  ckpt.restore(ckpt_manager.latest_checkpoint)
```

در نهایت مدل خود را تعریف و compile می‌کنیم تا برای آموزش آماده شود:

```
model = ImageCaptioningModel(Image_encoder, decoder,512)
model.compile(optimizer=optimizer, loss=cross_entropy)
```

در هر epoch از آموزش با استفاده از data_loader یکی یکی batch ها را لود خواهیم کرد و آموزش را با استفاده از آن‌ها انجام خواهیم داد. هم چنین هر 50 epoch نتایج مدل را نشان خواهیم داد. در پایان مقادیر نهایی loss و accuracy مدل را خواهیم داشت:

```
for epoch in range(epoch_start, EPOCHS):
    total_loss = 0
    total_acc = 0
    iter = 1

    for (batch, (img_tensor, target)) in enumerate(train_dataset):
        loss_acc = model.train_step((img_tensor, target))
        total_loss += loss_acc["loss"]
        total_acc += loss_acc["acc"]
        iter+=1

        if batch % 50 == 0:
            print(f'Epoch {epoch+1} Batch {batch} Loss {loss_acc["loss"]:.
4f} Acc {loss_acc["acc"]:.4f}')

    ckpt_manager.save()

    total_loss = total_loss / iter
    total_acc = total_acc / iter
    print(f'Epoch {epoch+1} Loss {total_loss} Accuracy {total_acc}')
```

نتایج این مدل ساده روی داده‌های train در epoch های پایانی به صورت زیر خواهد بود:

```
Epoch 35 Loss 2.1688790321350098 Accuracy 0.25871020555496216
Epoch 36 Batch 0 Loss 2.1817 Acc 0.2604
Epoch 36 Batch 50 Loss 2.1803 Acc 0.2605
Epoch 36 Batch 100 Loss 2.1795 Acc 0.2605
Epoch 36 Batch 150 Loss 2.1797 Acc 0.2605
Epoch 36 Loss 2.1666524410247803 Accuracy 0.2588813006877899
Epoch 37 Batch 0 Loss 2.1796 Acc 0.2606
Epoch 37 Batch 50 Loss 2.1783 Acc 0.2607
Epoch 37 Batch 100 Loss 2.1777 Acc 0.2607
Epoch 37 Batch 150 Loss 2.1780 Acc 0.2607
Epoch 37 Loss 2.1647326946258545 Accuracy 0.25903043150901794
Epoch 38 Batch 0 Loss 2.1779 Acc 0.2607
Epoch 38 Batch 50 Loss 2.1769 Acc 0.2608
Epoch 38 Batch 100 Loss 2.1764 Acc 0.2608
Epoch 38 Batch 150 Loss 2.1770 Acc 0.2607
Epoch 38 Loss 2.163414716720581 Accuracy 0.25912120938301086
Epoch 39 Batch 0 Loss 2.1769 Acc 0.2607
Epoch 39 Batch 50 Loss 2.1760 Acc 0.2608
Epoch 39 Batch 100 Loss 2.1759 Acc 0.2607
Epoch 39 Batch 150 Loss 2.1767 Acc 0.2606
Epoch 39 Loss 2.1627447605133057 Accuracy 0.2590548098087311
Epoch 40 Batch 0 Loss 2.1767 Acc 0.2606
Epoch 40 Batch 50 Loss 2.1763 Acc 0.2606
Epoch 40 Batch 100 Loss 2.1764 Acc 0.2605
Epoch 40 Batch 150 Loss 2.1776 Acc 0.2604
Epoch 40 Loss 2.163170099258423 Accuracy 0.25889238715171814
```

همان طور که در این قسمت کوتاه از نتایج نیز مشاهده می‌کنید، مقدار acuuracy مدل پس از مدتی بهبود چندانی پیدا نکرده‌است و تقریبا ثابت مانده‌است و حتی می‌توان گفت کمی هم رو به کاهش رفته است. در نهایت پس از اجرای 40 epoch به مقدار accuracy برابر با 0.258 رسیدیم.

نتایج روی داده‌های تست نیز به صورت زیر است که نشان می‌دهد کاهش زیادی در مقدار accuracy داشته ایم و این مقدار به 4 درصد رسیده‌است!

```
Batch 0 Loss 5.7848 Acc 0.0469
Loss 4.842346668243408 Accuracy 0.040326740592718124
```

بخش سوم

توضیح مدل

کد تمام قسمت‌ها مانند بخش دو است و تنها encoder و decoder متفاوت هستند. Image_encoder از نظر ساختاری مانند قبل است فقط به صورت Sequential پیاده سازی شده است:

```
Image_encoder = tf.keras.models.Sequential()
Image_encoder.add(tf.keras.Input(shape=(None, None, 2048)))
Image_encoder.add(tf.keras.layers.Dense(embedding_dim,activation='relu'))
```

اما دیکودر تفاوت‌های بسیاری دارد که در ادامه مشاهده می‌کنید:

دیکودر دارای یک لایه اتنشن از نوع AdditiveAttention است که به صورت زیر پیاده سازی شده:

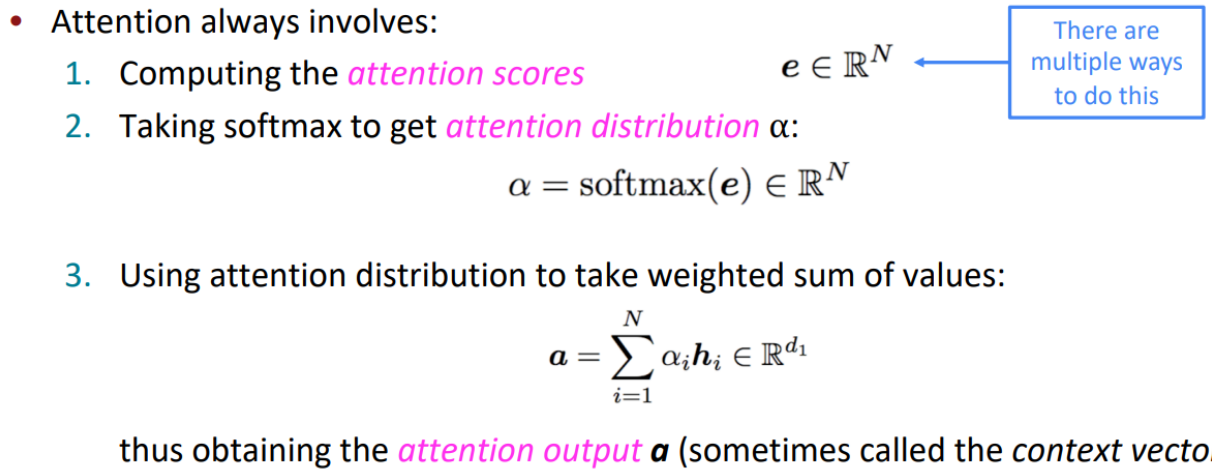
```
class AdditiveAttention(tf.keras.Model):
  def __init__(self, units):
    super(AdditiveAttention, self).__init__()
    self.W1 = tf.keras.layers.Dense(units)
    self.W2 = tf.keras.layers.Dense(units)
    self.V = tf.keras.layers.Dense(1)

  def call(self, features, hidden):
    hidden_with_time_axis = tf.expand_dims(hidden, 1)
    score = self.V(tf.nn.tanh(self.W1(features) +
                                        self.W2(hidden_with_time_axis)))
    normalized_score = tf.nn.softmax(score, axis=1)
    context_vector = normalized_score * features
    context_vector = tf.reduce_sum(context_vector, axis=1)

    return context_vector
```

رابطه به دست آورنده attention score در Additive attention به صورت زیر است:

$$\boldsymbol{e}_i = \boldsymbol{v}^T \tanh(\boldsymbol{W}_1 \boldsymbol{h}_i + \boldsymbol{W}_2 \boldsymbol{s}) \in \mathbb{R}$$

در این مدل ما فرض کرده‌ایم features که همان ویژگی‌های استخراج شده تصاویر از دیکودر هستند همان query و hidden states همان Values هستند.

ما قصد داشتیم تا از score به دست آمده روی ویژگی‌های تصاویر یا همان خروجی‌های دیکودر استفاده کنیم پس بعد از نرمالسازی score توسط تابع softmax، آن‌ها را در features ضرب کرده و مجموعشان را محاسبه می‌کنیم. وکتور به دست آمده context_vector خواهد بود:



دیکودر علاوه بر اتنشن دارای لایه‌های embedding، دو lstm، دو Dense و یک BatchNormalization بینشان است:

```
class DecoderWithAttention(tf.keras.Model):
  def __init__(self, vocab_size, embedding_dim):
    super(DecoderWithAttention, self).__init__()
    self.attention = AdditiveAttention(512)
    self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
    self.lstm = tf.keras.layers.LSTM(512,
                                     return_sequences=True,
                                     return_state=True)
    self.lstm2 = tf.keras.layers.LSTM(512,
                                     return_sequences=True,
                                     return_state=True)
    self.fc = tf.keras.layers.Dense(400)
    self.norm = tf.keras.layers.BatchNormalization()
    self.out_fc = tf.keras.layers.Dense(vocab_size)
```

دلیل return_state = true بودن lstm اول این است که lstm دوم از hidden stateهایش استفاده می‌کند. دلیل return_state = true بودن lstm دوم نیز به کارگیری hidden stateهای آن برای اتنشن گام بعدی است. در هر دو حالت return_ sequences نیز True است تا خروجی لایه‌ها نیز گرفته شود.

لایه Embedding اندازه vocabulary و همچنین embedding dim را میگیرد که نشان‌دهنده اندازه embedding یا خروجی همین لایه است. این لایه برای هر ایندکس یک embedding featuresای در نظر گرفته که براساس معنا و مفهوم کلمه، این embedding هنگام آموزش تنظیم می‌شود.

لایه fc برای پردازش بیشتر و پیچیده‌تر شدن مدل است. پس از این لایه یک BatchNormalization قرار دارد. نرمال کردن باعث می‌شود تمامی مقادیر در بازه خاصی قرار گرفته و برای لایه‌های بعدی ورودی‌های بهتری هستند. Out_fc لایه آخر مدل است و به تعداد vocab_size واحد دارد تا بتواند برای همه توکن‌ها مقدار تولید کند. هر چه این مقدار بالاتر باشد احتمال درست بودن آن توکن از نظر مدل بیشتر است.

```
def call(self, x, hidden, features):
    x = self.embedding(x)
    context_vector = self.attention(features, hidden)
    x = tf.concat([tf.expand_dims(context_vector, 1), x], axis=-1)
    x, _, _= self.lstm(x)
    output, new_hidden, _= self.lstm2(x)
    x = self.fc(output)
    x = tf.reshape(x, (-1, x.shape[2]))
    x = self.norm(x)
    x = self.out_fc(x)
    return x, new_hidden
```

در ابتدا ورودی دیکودر را به لایه embedding میدهیم تا embeddingهایش را به دست آوریم.به طور موازی از لایه اتنشن استفاده می‌کنیم و با values = hidden و query = features (توجه کنید hidden همان hidden state گام قبلی lstm دوم است) اتنشن زده و context_vector را پیدا می‌کنیم. خروجی این دو لایه را concat کرده و کنار هم قراره داده و به lstm می‌دهیم. سپس اوت پوت این لایه را به lstm2 می‌دهیم و hidden statesهای جدید و خروجی آن را می‌گیریم. در پایان این خروجی را از fc گذرانده و برای اینکه به دو بعد تبدیل شود از reshape استفاده میکنیم. (بعد اول دارای batch_size * max_length مقدار و بعد دوم هم اندازه hidden_size خواهد بود.) سپس آن را با استفاده از لایه out_fc به اندازه (batch_size * max_length, vocab_size) تبدیل خواهیم کرد.

بررسی نتایج

این مدل با learning rate = 0.001، Adam optimizer و آموزش 40 epoch به نتایج زیر می‌رسد:

...

```
Epoch 38 Batch 0 Loss 0.8678 Acc 0.6993
Epoch 38 Batch 50 Loss 0.8595 Acc 0.7016
Epoch 38 Batch 100 Loss 0.8555 Acc 0.7027
Epoch 38 Batch 150 Loss 0.8517 Acc 0.7039
Epoch 38 Loss 0.852394163608551 Accuracy 0.6976258754730225
Epoch 39 Batch 0 Loss 0.8507 Acc 0.7041
Epoch 39 Batch 50 Loss 0.8453 Acc 0.7053
Epoch 39 Batch 100 Loss 0.8420 Acc 0.7062
Epoch 39 Batch 150 Loss 0.8397 Acc 0.7071
Epoch 39 Loss 0.8384430408477783 Accuracy 0.7013099789619446
Epoch 40 Batch 0 Loss 0.8391 Acc 0.7072
Epoch 40 Batch 50 Loss 0.8351 Acc 0.7078
Epoch 40 Batch 100 Loss 0.8326 Acc 0.7083
Epoch 40 Batch 150 Loss 0.8289 Acc 0.7096
Epoch 40 Loss 0.8283293843269348 Accuracy 0.7037059664726257
```

با توجه به این که دیتاست ما استاندارد نیست و همچنین اندازه کوچکی داشت (تنها 12601 داده) این دقت خوب است. البته چون داده‌ها زیاد نیستند خطر overfitting را بالا می‌برد که با توجه به نتایج تست این اتفاق رخ نداده است:

```
Loss 0.9147889614105225 Accuracy 0.680516242980957
```

همان طور که مشاهده می‌کنید loss و accuracy داده‌های تست نزدیک به آموزش است پس overfit نشده است. البته این مدل هنوز جای آموزش دارد اما به دلیل سنگین بودن آموزش، آموزش بیشتر از ۴۰ epoch بسیار طول می‌کشید و امکان پذیر نبود.

توجه کنید که در این عناوین تعداد زیادی توکن خالی (برای pad شدن تمام عناوین به یک اندازه ثابت) وجود دارد که در هنگام محاسبه دقت و خطا در نظر گرفته نمی‌شود چون عنوان به پایان رسیده است.

دقت قابل قبول نشان‌دهنده ظرفیت کافی شبکه برای نگهداری اطلاعات مربوط به عنوان‌گذاری تصویر است. همچنین استفاده از لایه attention، LSTM به جای RNN ساده یا GRU و استفاده از batch_normalization بین لایه‌های fully connected به قدرت و پیچیدگی شبکه کمک کرده‌اند.اتنشن کمک می‌کند تا با استفاده hidden stateهای قبل مدل بداند به کدام قسمت تصویر توجه کند.

مقایسه بخش دو و سه

با توجه به اینکه مدل بخش سه بسیار پیچیده تر از بخش دو است و همچنین اتنشن دارد، هم در آموزش هم در تست دقت بسیار بالاتری دارد. (در آموزش 70% در مقایسه با 25.8% و در تست 68% در مقایسه با 4%) البته به دلیل داشتن تعداد لایه‌ها و متغیرهای بیشتر، سرعت پایینتری دارد.

مدل بخش دو underfit کرده است زیرا ظرفیت لازم برای یادگیری این دیتاست را ندارد، و این مشکل در مدل بخش ۳ حل شده است. در مدل بخش سه از لایه هایی مانند اتنشن استفاده شده که به بهبود عملکرد آن کمک می‌کنند.

منابع

- https://github.com/ajamjoom/Image-Captions
- https://keras.io/examples/vision/image_captioning/
- https://www.tensorflow.org/tutorials/text/image_captioning
- https://towardsdatascience.com/attention-and-its-different-forms-7fc3674d14dc