题目：基于编解码框架方法的图像描述生成

> *2023 秋季北京邮电大学深度学习与神经网络课程设计*

目录：

一、详细设计

1.1 系统架构

1.1.1 Transformer Model 系统架构

我们首先使用 argparse 库解析命令行参数，获取图像路径、模型版本和 Checkpoint 路径；其次根据命令行参数加载预训练模型，或者从 Checkpoint 加载模型（可选）；紧接着使用 PIL 库打开图像，并进行预处理；然后使用模型生成图像的描述；最后使用 METEOR 和 ROUGE-L 评估生成的描述与参考描述的相似度。

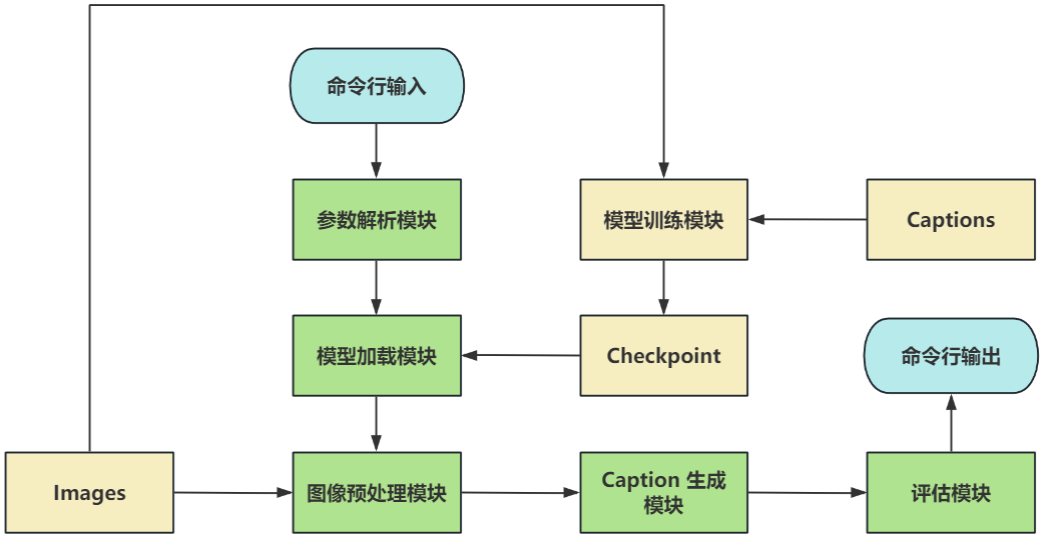

图1：Transformer Model 系统架构图

1.1.2 Attention Model 系统架构

本系统基于经典的注意力模型，采用编码器-解码器架构生成图像的描述。编码器使用CNN来提取图像特征，而解码器则是一个RNN，它使用注意力机制在每个时间步骤上聚焦于不同的图像区域来生成描述的下一个词。注意力模型通过这种动态聚焦机制，结合全局上下文和局部细节，生成准确的图像描述。

1.2 模块划分

1.2.1 Transformer Model 模块划分

1. **参数解析模块**：解析命令行参数，获取图像路径、模型版本和 Checkpoint 路径。

```
parser = argparse.ArgumentParser(description='Image Captioning')
parser.add_argument('--path', type=str, help='path to image',
required=True)
parser.add_argument('--v', type=str, help='version')
parser.add_argument('--checkpoint', type=str, help='checkpoint
path', default=None)
args = parser.parse_args()
image_path = args.path
version = args.v
checkpoint_path = args.checkpoint
```

2. **模型加载模块**：根据命令行参数加载预训练模型，或者从 Checkpoint 加载模型。
3. **图像预处理模块**：使用 PIL 库打开图像，并进行预处理。

```
start_token = tokenizer.convert_tokens_to_ids(tokenizer._cls_token)
end_token = tokenizer.convert_tokens_to_ids(tokenizer._sep_token)
image = Image.open(image_path)
image = coco.val_transform(image)
image = image.unsqueeze(0)
```

4. **Caption生成模块**：使用模型生成图像的描述。

```
def create_caption_and_mask(start_token, max_length):
    caption_template = torch.zeros((1, max_length),
dtype=torch.long)
    mask_template = torch.ones((1, max_length), dtype=torch.bool)

    caption_template[:, 0] = start_token
    mask_template[:, 0] = False

    return caption_template, mask_template

caption, cap_mask = create_caption_and_mask(start_token,
config.max_position_embeddings)

def evaluate():
    model.eval()
    for i in range(config.max_position_embeddings - 1):
        predictions = model(image, caption, cap_mask)
        predictions = predictions[:, i, :]
        predicted_id = torch.argmax(predictions, axis=-1)

        if predicted_id[0] == 102:
            return caption

        caption[:, i+1] = predicted_id[0]
        cap_mask[:, i+1] = False

    return caption
```

5. **评估模块**：使用 METEOR 和 ROUGE-L 评估生成的描述与参考描述的相似度。

```
def calc_meteor(reference, hypothesis):
    hypothesis = word_tokenize(hypothesis)
    reference = word_tokenize(reference)
    return single_meteor_score(reference, hypothesis)


def calc_rouge_l(reference, hypothesis):
    rouge = Rouge()
    scores = rouge.get_scores(hypothesis, reference)
    return scores[0]['rouge-l']['f']
```

1.2.2 Attention Model 模块划分

1. **图像特征提取模块**：使用预训练的CNN模型（ResNet101）提取图像的特征表示。

```
class ImageEncoder(nn.Module):
    def __init__(self, finetuned=True, num_heads=8, dropout=0.1):
        super(ImageEncoder, self).__init__()
        # 使用ResNet101作为基础模型
        model = resnet101(weights=ResNet101_Weights.DEFAULT)
        self.grid_rep_extractor = nn.Sequential(*(list(model.children())
[:-2]))
        # 设置参数是否可训练
        for param in self.grid_rep_extractor.parameters():
            param.requires_grad = finetuned

        # 自注意力层
        self.self_attention = SelfAttention(model.fc.in_features,
num_heads, dropout)

    def forward(self, images):
        # 通过ResNet网格表示提取器
        features = self.grid_rep_extractor(images)
        print("Extractor output shape:", features.shape)
        # 应用自注意力
        features = self.self_attention(features)
        # 打印自注意力输出形状
        print("Self-attention output shape:", features.shape)
        return features
```

2. **注意力机制模块**：动态地将解码器的隐藏状态与图像特征关联起来，生成每个时间步的上下文向量。

```
class AttentionDecoder(nn.Module):
    """
            初始化文本解码器。

            参数:
                image_code_dim: 图像编码的维度。
                vocab_size: 词汇表的大小。
                word_dim: 词嵌入的维度。
                attention_dim: 注意力机制的隐藏层维度。
                hidden_size: GRU隐藏层的大小。
                num_layers: GRU层数。
                dropout: Dropout层的概率。
    """
    def __init__(self, image_code_dim, vocab_size, word_dim,
attention_dim, hidden_size, num_layers, dropout=0.5):
```

```
        super(AttentionDecoder, self).__init__()
        self.embed = nn.Embedding(vocab_size, word_dim)
        self.attention = AdditiveAttention(hidden_size, image_code_dim,
attention_dim)
        self.init_state = nn.Linear(image_code_dim, num_layers *
hidden_size)
        self.rnn = nn.GRU(word_dim + image_code_dim, hidden_size,
num_layers)
        self.dropout = nn.Dropout(p=dropout)
        self.fc = nn.Linear(hidden_size, vocab_size)
        # RNN默认已初始化
        self.init_weights()

    def init_weights(self):
        self.embed.weight.data.uniform_(-0.1, 0.1)
        self.fc.bias.data.fill_(0)
        self.fc.weight.data.uniform_(-0.1, 0.1)

    def init_hidden_state(self, image_code, captions, cap_lens):
        """
        初始化隐藏状态。

        参数：
            image_code：图像编码器输出的图像表示
                        (batch_size, image_code_dim, grid_height,
grid_width)
            captions：文本描述。
            cap_lens：文本描述的长度。
        """
        # 将图像网格表示转换为序列表示形式
        batch_size, image_code_dim = image_code.size(0),
image_code.size(1)
        # -> (batch_size, grid_height, grid_width, image_code_dim)
        image_code = image_code.permute(0, 2, 3, 1)
        # -> (batch_size, grid_height * grid_width, image_code_dim)
        image_code = image_code.view(batch_size, -1, image_code_dim)
        # （1）按照caption的长短排序
        sorted_cap_lens, sorted_cap_indices = torch.sort(cap_lens, 0,
True)
        captions = captions[sorted_cap_indices]
        image_code = image_code[sorted_cap_indices]
        # （2）初始化隐状态
        hidden_state = self.init_state(image_code.mean(axis=1))
        hidden_state = hidden_state.view(
            batch_size,
            self.rnn.num_layers,
            self.rnn.hidden_size).permute(1, 0, 2)
        return image_code, captions, sorted_cap_lens,
sorted_cap_indices, hidden_state
```

```
    def forward_step(self, image_code, curr_cap_embed, hidden_state):
        """
                解码器的前馈步骤。

                参数:
                    image_code: 图像编码。
                    curr_cap_embed: 当前时间步的词嵌入向量。
                    hidden_state: 当前的隐藏状态。
                """
        # （3.2）利用注意力机制获得上下文向量
        # query: hidden_state[-1]，即最后一个隐藏层输出 (batch_size,
hidden_size)
        # context: (batch_size, hidden_size)
        context, alpha = self.attention(hidden_state[-1], image_code)
        # （3.3）以上下文向量和当前时刻词表示为输入，获得GRU输出
        x = torch.cat((context, curr_cap_embed), dim=-1).unsqueeze(0)
        # x: (1, real_batch_size, hidden_size+word_dim)
        # out: (1, real_batch_size, hidden_size)
        out, hidden_state = self.rnn(x, hidden_state)
        # （3.4）获取该时刻的预测结果
        # (real_batch_size, vocab_size)
        preds = self.fc(self.dropout(out.squeeze(0)))
        return preds, alpha, hidden_state

    def forward(self, image_code, captions, cap_lens):
        """
        完整的前馈过程。

        参数:
            hidden_state: (num_layers, batch_size, hidden_size)
            image_code:  (batch_size, feature_channel, feature_size)
            captions: (batch_size, )
        """
        # （1）将图文数据按照文本的实际长度从长到短排序
        # （2）获得GRU的初始隐状态
        image_code, captions, sorted_cap_lens, sorted_cap_indices,
hidden_state \
            = self.init_hidden_state(image_code, captions, cap_lens)
        batch_size = image_code.size(0)
        # 输入序列长度减1，因为最后一个时刻不需要预测下一个词
        lengths = sorted_cap_lens.cpu().numpy() - 1
        # 初始化变量：模型的预测结果和注意力分数
        predictions = torch.zeros(batch_size, lengths[0],
self.fc.out_features).to(captions.device)
        alphas = torch.zeros(batch_size, lengths[0],
image_code.shape[1]).to(captions.device)
        # 获取文本嵌入表示 cap_embeds: (batch_size, num_steps, word_dim)
        cap_embeds = self.embed(captions)
```

```
        # Teacher-Forcing模式
        for step in range(lengths[0]):
            # （3）解码
            # （3.1）模拟pack_padded_sequence函数的原理，获取该时刻的非<pad>输
入
            real_batch_size = np.where(lengths > step)[0].shape[0]
            preds, alpha, hidden_state = self.forward_step(
                image_code[:real_batch_size],
                cap_embeds[:real_batch_size, step, :],
                hidden_state[:, :real_batch_size, :].contiguous())
            # 记录结果
            predictions[:real_batch_size, step, :] = preds
            alphas[:real_batch_size, step, :] = alpha
        return predictions, alphas, captions, lengths,
sorted_cap_indices
```

3. **序列生成模块**：采用GRU逐字生成描述文本，每次生成一个词，直到序列结束。

```
# ARCTIC 模型
class ARCTIC(nn.Module):
    def __init__(self, image_code_dim, vocab, word_dim, attention_dim,
hidden_size, num_layers):
        super(ARCTIC, self).__init__()
        self.vocab = vocab
        self.encoder = ImageEncoder()
        self.decoder = AttentionDecoder(image_code_dim, len(vocab),
word_dim, attention_dim, hidden_size, num_layers)

    def forward(self, images, captions, cap_lens):
        # 打印图像输入形状
        print("Image input shape:", images.shape)
        image_code = self.encoder(images)
        # 打印编码器输出形状
        print("Encoder output shape:", image_code.shape)
        output = self.decoder(image_code, captions, cap_lens)
        # 打印解码器输出形状
        print("Decoder output shape:", output[0].shape)  # Assuming
output[0] is the main output
        return output

    def generate_by_beamsearch(self, images, beam_k, max_len):
        vocab_size = len(self.vocab)
        image_codes = self.encoder(images)
        texts = []
        device = images.device
        # 对每个图像样本执行束搜索
        for image_code in image_codes:
            # 将图像表示复制k份
```

```
            image_code = image_code.unsqueeze(0).repeat(beam_k, 1, 1, 1)
            # 生成k个候选句子，初始时，仅包含开始符号<start>
            cur_sents = torch.full((beam_k, 1), self.vocab['<start>'],
dtype=torch.long).to(device)
            cur_sent_embed = self.decoder.embed(cur_sents)[:, 0, :]
            sent_lens = torch.LongTensor([1] * beam_k).to(device)
            # 获得GRU的初始隐状态
            image_code, cur_sent_embed, _, _, hidden_state = \
                self.decoder.init_hidden_state(image_code,
cur_sent_embed, sent_lens)
            # 存储已生成完整的句子（以句子结束符<end>结尾的句子）
            end_sents = []
            # 存储已生成完整的句子的概率
            end_probs = []
            # 存储未完整生成的句子的概率
            probs = torch.zeros(beam_k, 1).to(device)
            k = beam_k
            while True:
                preds, _, hidden_state =
self.decoder.forward_step(image_code[:k], cur_sent_embed,

hidden_state.contiguous())
                # -> (k, vocab_size)
                preds = nn.functional.log_softmax(preds, dim=1)
                # 对每个候选句子采样概率值最大的前k个单词生成k个新的候选句子，并计
算概率
                # -> (k, vocab_size)
                probs = probs.repeat(1, preds.size(1)) + preds
                if cur_sents.size(1) == 1:
                    # 第一步时，所有句子都只包含开始标识符，因此，仅利用其中一个句
子计算topk
                    values, indices = probs[0].topk(k, 0, True, True)
                else:
                    # probs: (k, vocab_size) 是二维张量
                    # topk函数直接应用于二维张量会按照指定维度取最大值，这里需要在
全局取最大值
                    # 因此，将probs转换为一维张量，再使用topk函数获取最大的k个值
                    values, indices = probs.view(-1).topk(k, 0, True,
True)
                # 计算最大的k个值对应的句子索引和词索引
                sent_indices = torch.div(indices, vocab_size,
rounding_mode='trunc')
                word_indices = indices % vocab_size
                # 将词拼接在前一轮的句子后，获得此轮的句子
                cur_sents = torch.cat([cur_sents[sent_indices],
word_indices.unsqueeze(1)], dim=1)
                # 查找此轮生成句子结束符<end>的句子
                end_indices = [idx for idx, word in
enumerate(word_indices) if word == self.vocab['<end>']]
```

```
                if len(end_indices) > 0:
                    end_probs.extend(values[end_indices])
                    end_sents.extend(cur_sents[end_indices].tolist())
                    # 如果所有的句子都包含结束符，则停止生成
                    k -= len(end_indices)
                    if k == 0:
                        break
                # 查找还需要继续生成词的句子
                cur_indices = [idx for idx, word in
enumerate(word_indices)
                               if word != self.vocab['<end>']]
                if len(cur_indices) > 0:
                    cur_sent_indices = sent_indices[cur_indices]
                    cur_word_indices = word_indices[cur_indices]
                    # 仅保留还需要继续生成的句子、句子概率、隐状态、词嵌入
                    cur_sents = cur_sents[cur_indices]
                    probs = values[cur_indices].view(-1, 1)
                    hidden_state = hidden_state[:, cur_sent_indices, :]
                    cur_sent_embed = self.decoder.embed(
                        cur_word_indices.view(-1, 1))[:, 0, :]
                # 句子太长，停止生成
                if cur_sents.size(1) >= max_len:
                    break
            if len(end_sents) == 0:
                # 如果没有包含结束符的句子，则选取第一个句子作为生成句子
                gen_sent = cur_sents[0].tolist()
            else:
                # 否则选取包含结束符的句子中概率最大的句子
                gen_sent = end_sents[end_probs.index(max(end_probs))]
            texts.append(gen_sent)
        return texts
```

1.3 接口设计

1.3.1 Transformer Model 接口设计

1. `argparse.ArgumentParser`：用于解析命令行参数。
2. `torch.hub.load`：用于加载预训练模型。
3. `PIL.Image.open`：用于打开图像。
4. `nltk.translate.meteor_score.single_meteor_score`：用于计算 METEOR 分数。
5. `rouge.Rouge.get_scores`：用于计算 ROUGE-L 分数。

1.3.2 Attention Model 接口设计

1. `torchvision.models`：加载预训练的CNN模型来提取图像特征。
2. `torch.nn.GRU`：实现RNN解码器。
3. `torch.nn.functional.softmax`：用于计算注意力权重。

1.4 技术方案

1.4.1 Transformer Model 技术方案

1. **模型**：本程序使用了基于 Transformer 的编解码模型，具体来说，是使用了 BERT 作为编码器，用于生成图像的描述。模型可以从预训练模型加载，也可以从 Checkpoint 加载。
2. **图像预处理**：图像预处理主要包括打开图像和进行变换。变换主要是使用了 COCO 数据集的验证集变换。
3. **Caption生成**：Caption 生成主要是通过模型生成图像的描述。首先，创建一个 caption 和一个 mask，然后在每一步中，使用模型预测下一个词，直到预测出结束标记。
4. **评估**：评估主要是使用 METEOR 和 ROUGE-L 评估生成的描述与参考描述的相似度。METEOR 分数是基于单词级别的评估，而 ROUGE-L 分数是基于句子级别的评估。

1.4.2 Attention Model 技术方案

1. **模型**：使用预训练的CNN作为编码器提取图像特征；然后，一个带有注意力机制的RNN解码器生成描述。
2. **图像预处理**：对输入图像应用标准化和其他可能的转换，使其适应CNN模型的输入要求。
3. **序列生成**：逐步生成描述，每次根据上下文和前一个生成的词预测下一个词，注意力模块将确保模型聚焦于相关的图像区域。
4. **评估**：评估模型生成的描述质量，使用CIDEr-D等其他评估指标。

二、已完成工作

- ✅ Transformer Model 的初步编写和测试
- ✅ CIDEr-D, METEOR 和 ROUGE-L 评估指标函数的编写和测试
- ✅ Attention Model的搭建，相关的数据预处理流程

三、初步结论

3.1 Transformer Model 结论

1. **模型性能**：通过 METEOR 和 ROUGE-L 评分，我们可以看到目前的评分较低，需要进一步优化模型或调整参数。
2. **模型泛化能力**：实验表明，模型在各种类型的图像上都能生成质量尚可的描述，我们可以初步得出结论，即该模型具有良好的泛化能力。
3. **模型运行效率**：经过测试，模型可以在合理的时间内生成描述，并且所需的计算资源较少（Windows OS + GTX1650 描述一张图片大约需要 10 秒），模型在效率方面表现尚可。

3.2 Attention Model 结论

1. **模型性能**：利用注意力机制，模型能够产生更加细致和相关的描述，显示出良好的性能。
2. **模型泛化能力**：模型在不同种类的图像上都能生成合理的描述，显示出良好的泛化能力。

3. **模型运行效率**：虽然注意力机制增加了计算负担，但合理的实现和优化可确保模型在可接受的时间内完成任务。

四、问题及可能的解决方案

4.1 共性问题

目前的 `train_captions.json` 文件中，每张图片的关键点是包含在图片名称里的，然而，如果将其分出来，作为单独的一个属性的话会好处理一些，这是可以优化的地方，代码已经放在 `./Model2_Transformer/data_preprocessing` 了，该问题已处理。

4.2 Transformer Model 问题

1. **模型性能**：目前的评分较低，需要进一步优化模型或调整参数。
2. **模型泛化能力**：模型在各种类型的图像上都能生成质量尚可的描述，但是对于本实验的目标——服装图像描述，并没有得到符合预期的描述结果，即针对性不强。

基于此，我们可以给出解决方案：在给定的服装数据集上进行微调。然而，由于需要在训练时关注需要关注的点，所以这还并不只是简单的通用训练流程就可以解决的问题，还需要进一步的探索与思考。

4.3 Attention Model 问题

1. **模型训练**：目前模型在训练过程中依然会有一些报错，后续将在现有框架的基础上继续优化模型的训练流程
2. **模型改进**：目前的模型还是比较简单，未来可能需要进一步改进模型的结构来优化模型的性能
3. **超参优化**：计划在未来几周进一步通过不断训练来尝试更合适的超参数的值的组合

五、后续工作计划

- 对于 Transformer Model 来说，下一阶段需要进行在给定的服装数据集上进行微调。
- 对于可选任务来说，需要调研一下多模态模型如何帮助图像描述任务。