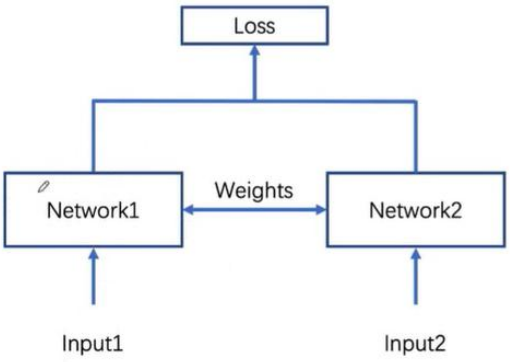
预训练模型在摘要任务的改进

一、siamese network structuce

1. 孪生网络结构

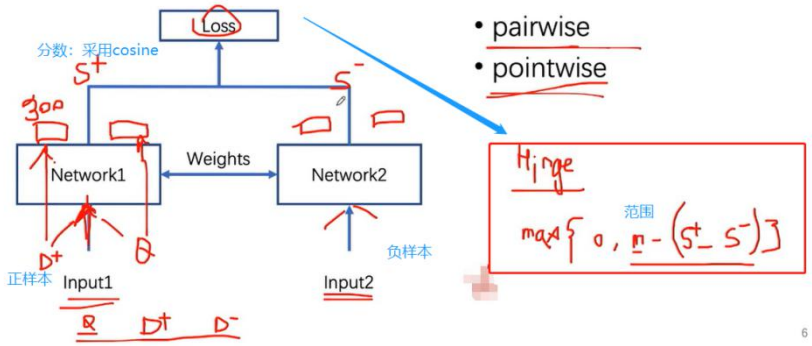
孪生网络原本是用来解决文本匹配问题，network1 与 netwprk2 采用同一个网络，从而实现参数共享。



- 语义相似度分析
- QA中的q和a匹配
- question pair比对

模型细节：

(1) Pairwise



输入：input1=(Q,D+),input2=(Q,D-),其中 Q 为查询，D+为答案正样本，D-为答案负样本

任务：input1 输入 network1 网络得到 Q 与 D+的分数 s+（采用 cosine 计算），将 input2 输入到 network2 网络中得到 Q 与 D-的分数 s-,再将 s+、s-求 loss,loss 采用 hinge_loss。

(2) Pointwise

描述：简单的分类问题

方式一：

输入：input1=sent1,input2=sent2,label=1/0

任务：预测 sent1 与 sent2 是否相似

方式二：

输入：input1=(Q,D+),input2=(Q,D-),label=1/0

任务：将 Q 与 D+进行特征融合再输入到 network1 中，将 Q 与 D-进行特征融

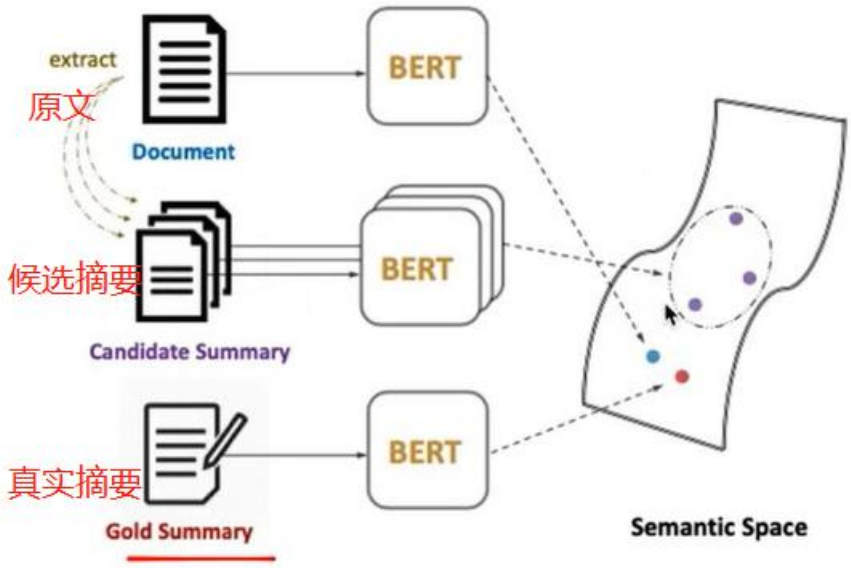
合再输入到 network2 中，最后进行分类。

2. 文本匹配应用

常见的匹配一般有关键词匹配和语义匹配。在检索系统或者 QA 问答中，一般先对数据集进行关键词匹配/倒排索引+BM25 来召回，然后通过语义匹配来进行精排。

3. MATCHSUM

根据对 matchsum 的大致理解，原文与候选摘要的语义空间的距离应该很近，而我们只要选出候选摘要中与真实摘要、原文最近的一条即可，则摘要问题就变成检索问题，具体参考论文《Extractive Summarization as Text Matching》



候选摘要是如何选取的？

答：将原文按句子划分，采用 bertsum(或其他模型)给每一个句子打分，然后进行排序，选择前 m 个句子，紧接着从 m 个句子中选出 k 个句子（顺序按原文的顺序），共有 c(m,k)中组合，这些组合就是候选集。

3.1 Matchsum 的具体流程

①构建候选摘要

②bert 进行文本匹配打分（采用 siamese-bert）

loss1 还是孪生网络的 pairwise 方式求解，f 为模型，loss1 越近越好（候选集不是一个一个往里扔，而是采用矩阵方式），最终得到的是[batch_size,num(候选集数)]维度的分数 score；loss2 则考虑候选集之间的差距尽可能拉开,将 score 放到 loss2 公式中；最终 loss 为 loss1+loss2。

Siamese-BERT architecture

1）一个用来考虑候选摘要和原文档相似度：

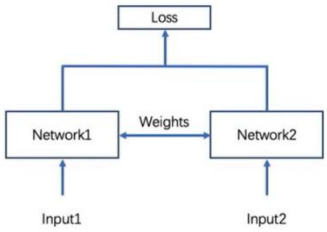
孪生网络loss　原文　候选摘要　真实摘要

$$\mathcal{L}_1 = \max(0, f(D, C) - f(D, C^*) + \gamma_1)$$

2）另一个考虑候选摘要之间的差异，认为排名靠前的候选答案应该比靠后的答案得分高：

新增loss:保证第一名与第二名的差距拉大

$$\mathcal{L}_2 = \max(0, f(D, C_j) - f(D, C_i) + (j - i) * \gamma_2) \quad (i < j)$$



3.2 matchsum 代码

Siamese-BERT architecture

```
# get document embedding
input_mask = ~(text_id == pad_id)
out = self.encoder(text_id, attention_mask=input_mask)[0] # last layer
doc_emb = out[:, 0, :]
assert doc_emb.size() == (batch_size, self.hidden_size) # [batch_size, hidden_size]

# get summary embedding
input_mask = ~(summary_id == pad_id)
out = self.encoder(summary_id, attention_mask=input_mask)[0] # last layer
summary_emb = out[:, 0, :]
assert summary_emb.size() == (batch_size, self.hidden_size) # [batch_size, hidden_size]

# get summary score
summary_score = torch.cosine_similarity(summary_emb, doc_emb, dim=-1)

# get candidate embedding
candidate_num = candidate_id.size(1)
candidate_id = candidate_id.view(-1, candidate_id.size(-1))
input_mask = ~(candidate_id == pad_id)
out = self.encoder(candidate_id, attention_mask=input_mask)[0]
candidate_emb = out[:, 0, :].view(batch_size, candidate_num, self.hidden_size)  # [batch_size, candidate_num, hidden_size]
assert candidate_emb.size() == (batch_size, candidate_num, self.hidden_size)

# get candidate score
doc_emb = doc_emb.unsqueeze(1).expand_as(candidate_emb)
score = torch.cosine_similarity(candidate_emb, doc_emb, dim=-1) # [batch_size, candidate_num]
assert score.size() == (batch_size, candidate_num)
```

https://github.com/maszhongming/MatchSum

3.3 摘要的两大方式

(1) sentence-level extractor（句子级别摘要）

描述：

类似与 BertSum 的方式，给每一个句子打分，取前 k 个句子组合就是预测摘要。

公式：

$$g^{sen}(C) = \frac{1}{|C|} \sum_{s \in C} R(s, C^*)$$

将原文中划分的每一个句子 s 和真实摘要进行打分，打分采用 R_1、R_2 或者 R_L 评估方式，最后归一化，得到分数，这里体现的是 句子与真实摘要的相似。

(2) Summary-level extractor（摘要级别摘要）

描述：

不是一下子就取 k 个句子，而是先取 m 个，再从里面取 k 个组合，生成候选集，从候选集中选出最好的一条。

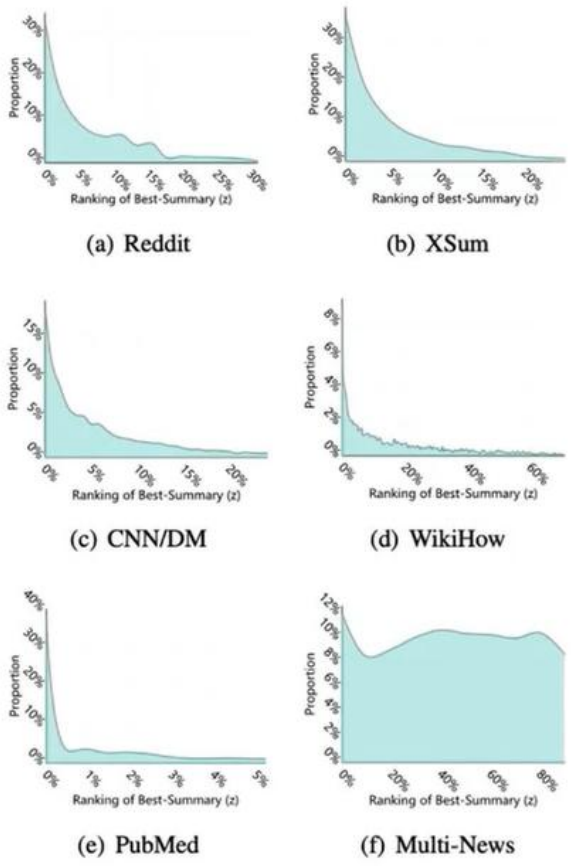
公式：

$$\mathrm{g}^{sum}(C) = \mathrm{R}(C, C^*)$$

与 sentence-level 不同，这里有提前筛选出候选摘要集，采用候选摘要集 C 与真实摘要进行打分(采用矩阵方式)，这里体现的是 每一个候选摘要与真实摘要的相似。（效果就很明显了）

(3) 两大方式的比较

每一次都选取分数最后的句子拼接在一起，整体分数还是最高的？答案明显是不一定,有人做过实验证明，从下图 6 个数据集的验证统计，最好的摘要往往不是 sentece-level 的摘要。然而这些摘要也被称为珍珠摘要。



Pearl-summary(珍珠摘要)，指的是摘要级得分高但句子级得分低的摘要。

二、模型实现过程的 trick

2.1 对抗训练

- FGM

NLP 领域的对抗训练通过对 embedding 添加干扰使模型犯错，作为一种正则化手段来提高模型的泛化能力。

FGM 代码实现：

```
import torch
class FGM():
    def __init__(self, model):
        self.model = model
        self.backup = {}

    def attack(self, epsilon=1., emb_name='emb.'):
        # emb_name这个参数要换成你模型中embedding的参数名
        for name, param in self.model.named_parameters():
            if param.requires_grad and emb_name in name:
                self.backup[name] = param.data.clone()
                norm = torch.norm(param.grad)
                if norm != 0 and not torch.isnan(norm):
                    r_at = epsilon * param.grad / norm
                    param.data.add_(r_at)

    def restore(self, emb_name='emb.'):
        # emb_name这个参数要换成你模型中embedding的参数名
        for name, param in self.model.named_parameters():
            if param.requires_grad and emb_name in name:
                assert name in self.backup
                param.data = self.backup[name]
        self.backup = {}
```

使用代码：

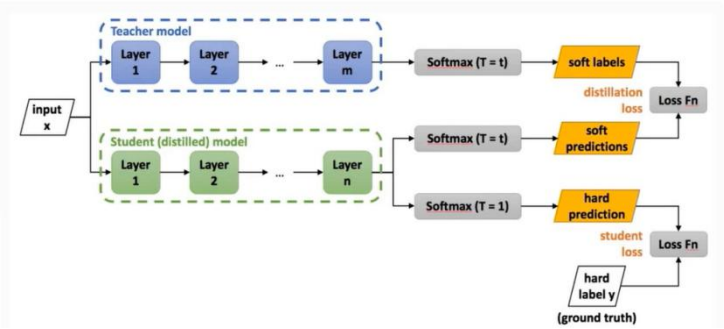
```
# 初始化
fgm = FGM(model)
for batch_input, batch_label in data:
    # 正常训练
    loss = model(batch_input, batch_label)
    loss.backward() # 反向传播，得到正常的grad
    # 对抗训练
    fgm.attack() # 在embedding上添加对抗扰动
    loss_adv = model(batch_input, batch_label)
    loss_adv.backward() # 反向传播，并在正常的grad基础上，累加对抗训练的梯度
    fgm.restore() # 恢复embedding参数
    # 梯度下降，更新参数
    optimizer.step()
    model.zero_grad()
```

- PGD

  在球面的范围内进行干扰添加。

2.2 知识蒸馏

现在的 bert 等预训练效果很好，但是很难在线上使用，因为推理耗时很大，无法满足上线需求，因此需要模型压缩与优化。



知识蒸馏思想：①训练一个老师模型（bert 等模型），将训练数据先传入老师模型，经过 softmax(T=t)得到一个预测概率，称之为软标签。②将训练数据输入一个学生模型（线上用）中，经过 softmax(T=t)得到一个预测概率，称之为软预测。③将软预测与软标签求损失 loss1。④同时，将训练数据输入学生模型，经过 softmax(T=1)得到 hard_prediction，此时得到的损失 loss2。⑤将 loss1 与 loss2 加权求和，共同反向传播，更新参数。

softmax 中的 T 参数有什么意义？
答：

特殊蒸馏：从模型角度蒸馏（对某些层结构）、从数据角度蒸馏（标注数据不足，将数据输入到老师模型，得到伪标签数据）

模型优化：
①例如，训练 12 层的 bert,只取最重要的 k 层结构来进行推理，提升速度。
②量化方式：原本是 float32，可以改成 int 后 float16
③剪枝方式：例如 albert，有针对性的删除一些结构,修改模型结构

2.3 模型优化：
①例如，训练 12 层的 bert,只取最重要的 k 层结构来进行推理，提升速度。
②量化方式：原本是 float32，可以改成 int 后 float16
③剪枝方式：例如 albert，有针对性的删除一些结构

2.3 推理加速：
①op 融合（算子合并，底层 c++处理）
②FT16 加速：精度方式（tensorRT）

三、领域迁移的影响

3.1 DAPT（领域自适应训练）
描述：拿到 bert 预训练模型之后，假设要做一个医疗领域的任务，这时可以找一些医疗领域的数据（没标注，数据多）来进行再次预训练。
结论：当前任务领域的数据 与 预训练模型的原始数据 差异性越大，效果提升越明显；无限的喂当前领域数据进行预训练，效果不一定越好。

3.2 TAPT（任务自适应训练）
描述：拿到 bert 预训练模型之后，假设要做一个医疗领域的任务，这时可以下游任务的数据（有标注，不需要用到 label，数据少）来进行再次预训练。
结论：使用 TAPT，效果也会有提升，但没 DAPT 效果，因此可以 TAPT+DAPT 一起用，效果还会有提升。

四、其他介绍

4.1 做生成式摘要如何用预训练模型

①参数 github 的 GPT2-Summary 与 GPT2-chitchat 来进行 funetune 使用

4.2 百度的 ernie-gan 可以学习一下