人工智能基础第二次实验

PB21020651 武宇星

Part 1

决策树

实验原理

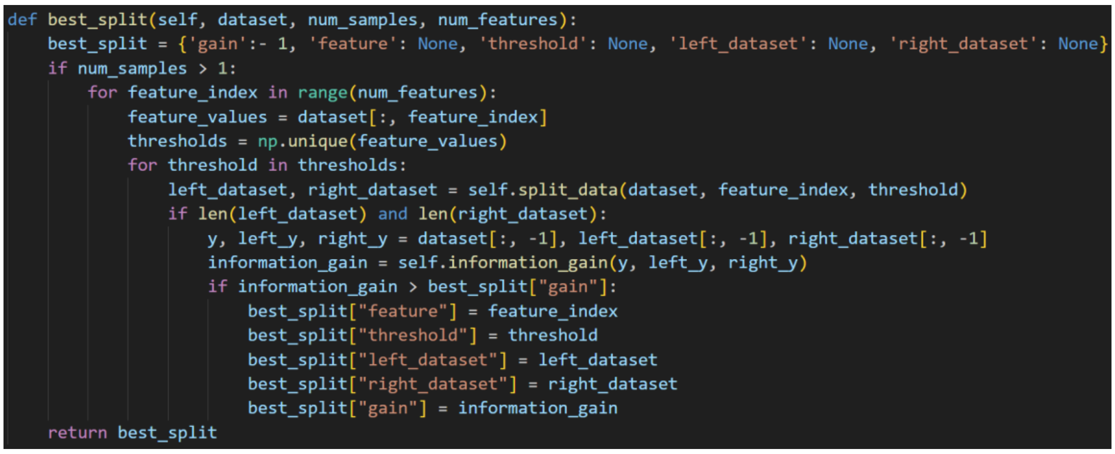
对于决策树，首先需要定义其结点类型，每个非叶结点需要划分，feature 记录该结点用于分割的特征，因为离散特征也已经转化为值，用 threshhold 来进行划分，left，right 指向划分后的左右子结点，value 记录当不可再分即特征用尽或叶结点特征均相同时样本数最多的标签，非叶结点值为 None

```
class Node():
    def __init__(self, feature=None, threshold=None, left=None, right=None, gain=None, value=None):
        self.feature = feature
        self.threshold = threshold
        self.left = left
        self.right = right
        self.gain = gain
        self.value = value
```

在划分时需要信息增益做出决定，根据信息熵和信息增益定义完成即可

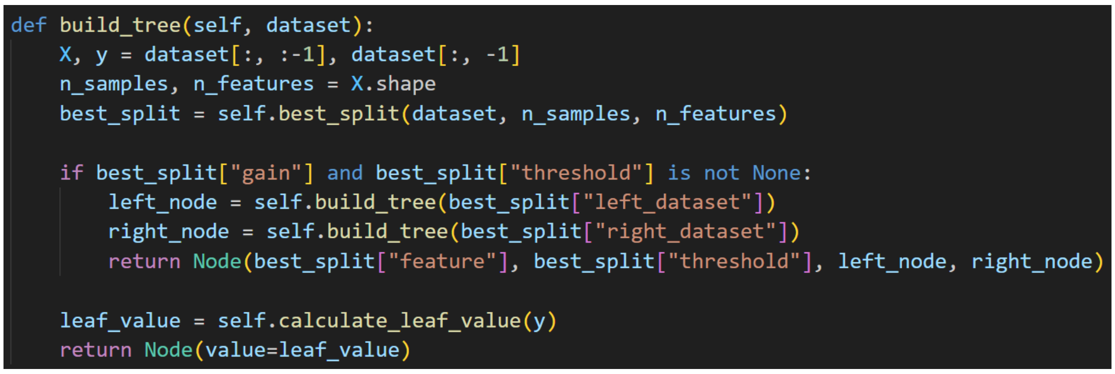
```
def entropy(self, y):
    entropy = 0
    labels = np.unique(y)
    for label in labels:
        label_examples = y[y == label]
        pl = len(label_examples) / len(y)
        entropy += -pl * np.log2(pl)
    return entropy

def information_gain(self, parent, left, right):
    information_gain = 0
    parent_entropy = self.entropy(parent)
    weight_left = len(left) / len(parent)
    weight_right= len(right) / len(parent)
    entropy_left, entropy_right = self.entropy(left), self.entropy(right)
    weighted_entropy = weight_left * entropy_left + weight_right * entropy_right
    information_gain = parent_entropy - weighted_entropy
    return information_gain
```

而在划分时则遍历所有的特征，而域值则遍历每个不同的值，寻找信息增益最大的划分，记录划分的特征，阈值，划分后数据集，信息增益并返回



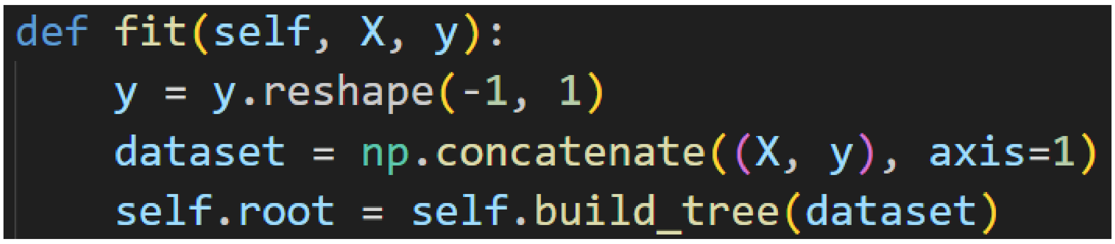
```python
def best_split(self, dataset, num_samples, num_features):
    best_split = {'gain':- 1, 'feature': None, 'threshold': None, 'left_dataset': None, 'right_dataset': None}
    if num_samples > 1:
        for feature_index in range(num_features):
            feature_values = dataset[:, feature_index]
            thresholds = np.unique(feature_values)
            for threshold in thresholds:
                left_dataset, right_dataset = self.split_data(dataset, feature_index, threshold)
                if len(left_dataset) and len(right_dataset):
                    y, left_y, right_y = dataset[:, -1], left_dataset[:, -1], right_dataset[:, -1]
                    information_gain = self.information_gain(y, left_y, right_y)
                    if information_gain > best_split["gain"]:
                        best_split["feature"] = feature_index
                        best_split["threshold"] = threshold
                        best_split["left_dataset"] = left_dataset
                        best_split["right_dataset"] = right_dataset
                        best_split["gain"] = information_gain
    return best_split
```

结合这些可以函数，就可以用来建立决策树了，使用递归算法不断划分构建结点，根据西瓜书算法，仅在特征用尽或所有样本为同一类时返回叶结点，即可通过样本数为 1 或者信息增益为 0 来判断



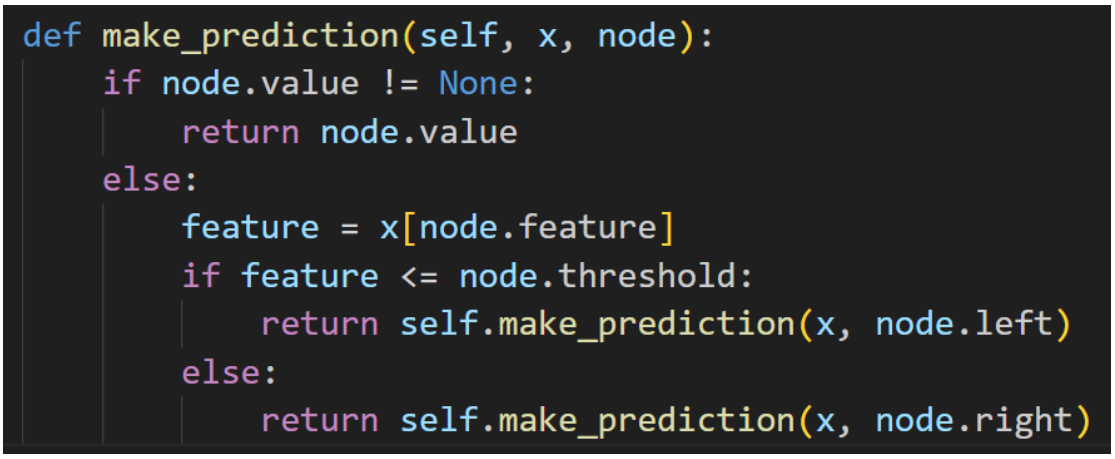
```python
def build_tree(self, dataset):
    X, y = dataset[:, :-1], dataset[:, -1]
    n_samples, n_features = X.shape
    best_split = self.best_split(dataset, n_samples, n_features)

    if best_split["gain"] and best_split["threshold"] is not None:
        left_node = self.build_tree(best_split["left_dataset"])
        right_node = self.build_tree(best_split["right_dataset"])
        return Node(best_split["feature"], best_split["threshold"], left_node, right_node)

    leaf_value = self.calculate_leaf_value(y)
    return Node(value=leaf_value)
```

接口 fit 将特征标签以数据集格式传入调用该函数即可



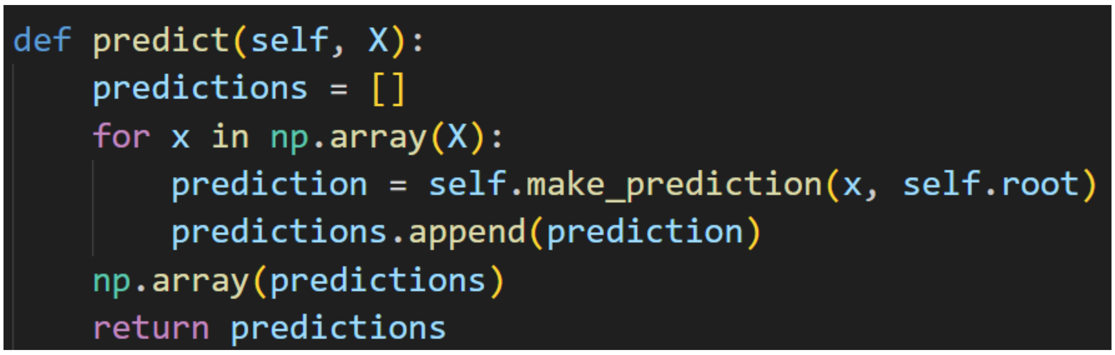
```python
def fit(self, X, y):
    y = y.reshape(-1, 1)
    dataset = np.concatenate((X, y), axis=1)
    self.root = self.build_tree(dataset)
```

而要使用构建好的决策树进行样本预测，同样可以运用递归，根据样本的特征值和当前结点划分阈值比较进入下一结点， 直至到达叶结点获得标签



```python
def make_prediction(self, x, node):
    if node.value != None:
        return node.value
    else:
        feature = x[node.feature]
        if feature <= node.threshold:
            return self.make_prediction(x, node.left)
        else:
            return self.make_prediction(x, node.right)
```

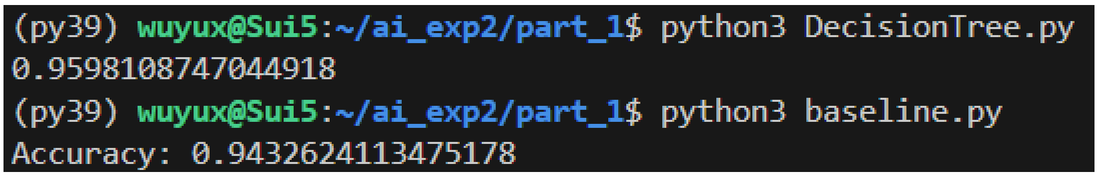
接口 predict 遍历每个待预测样本传入调用该函数即可



```python
def predict(self, X):
    predictions = []
    for x in np.array(X):
        prediction = self.make_prediction(x, self.root)
        predictions.append(prediction)
    np.array(predictions)
    return predictions
```

实验结果

使用自行搭建的决策树和 sklearn 库的决策树准确度如下所示，效果相当

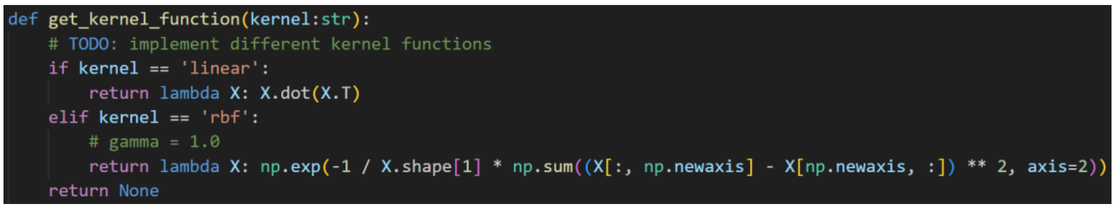


```
(py39) wuyux@Sui5:~/ai_exp2/part_1$ python3 DecisionTree.py
0.9598108747044918
(py39) wuyux@Sui5:~/ai_exp2/part_1$ python3 baseline.py
Accuracy: 0.9432624113475178
```

PCA，Kmeans

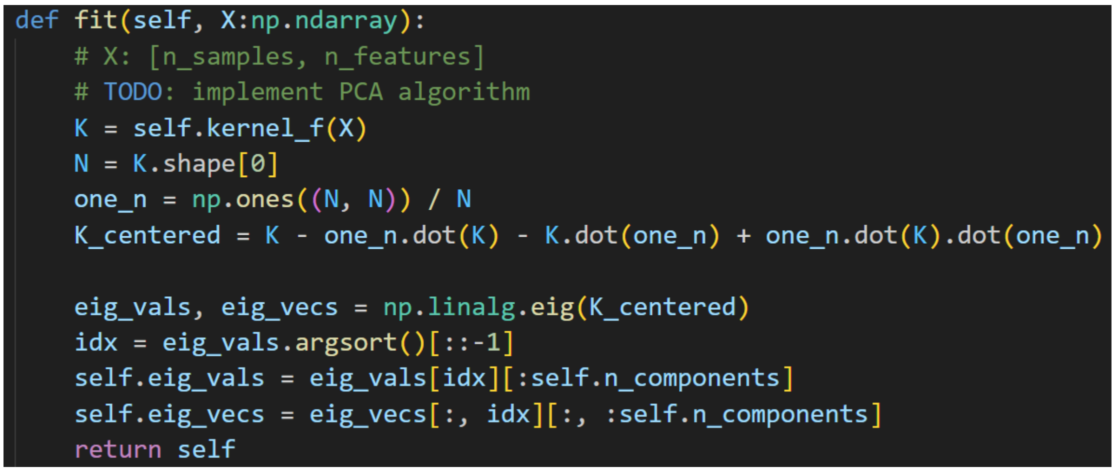
实验原理

Kernel PCA 先将输入空间映射到高维特征空间，再进行主成分分析，对于核函数，若使用 rbf 核，则计算数据点欧式距离后应用高斯函数，取 gamma 为样本数避免数据点聚集



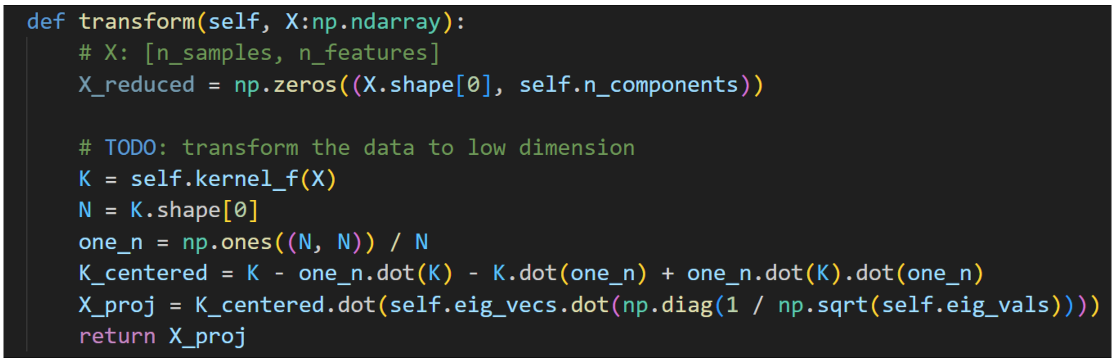
```python
def get_kernel_function(kernel:str):
    # TODO: implement different kernel functions
    if kernel == 'linear':
        return lambda X: X.dot(X.T)
    elif kernel == 'rbf':
        # gamma = 1.0
        return lambda X: np.exp(-1 / X.shape[1] * np.sum((X[:, np.newaxis] - X[np.newaxis, :]) ** 2, axis=2))
    return None
```

Fit 函数根据伪代码将核矩阵中心化，如何分解得到其特征值和特征向量，取特征值最大的 n_components 个特征向量。



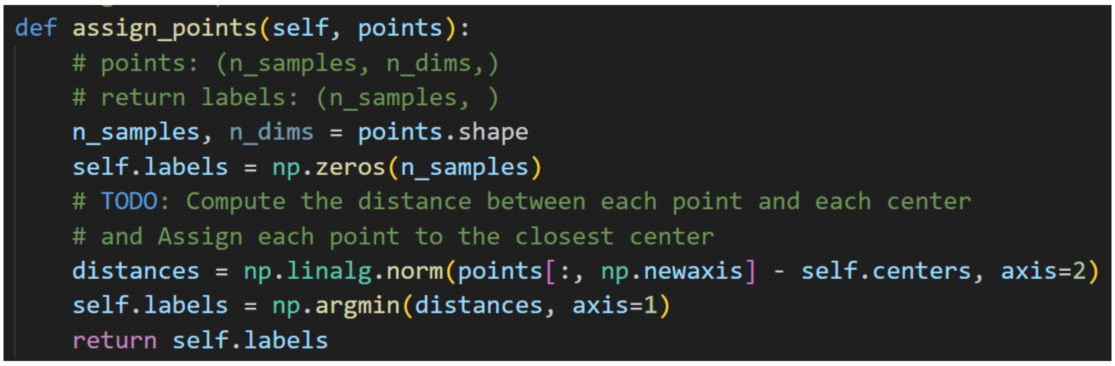
```python
def fit(self, X:np.ndarray):
    # X: [n_samples, n_features]
    # TODO: implement PCA algorithm
    K = self.kernel_f(X)
    N = K.shape[0]
    one_n = np.ones((N, N)) / N
    K_centered = K - one_n.dot(K) - K.dot(one_n) + one_n.dot(K).dot(one_n)

    eig_vals, eig_vecs = np.linalg.eig(K_centered)
    idx = eig_vals.argsort()[::-1]
    self.eig_vals = eig_vals[idx][:self.n_components]
    self.eig_vecs = eig_vecs[:, idx][:, :self.n_components]
    return self
```

Transform 函数则返回样本在特征向量上的投影



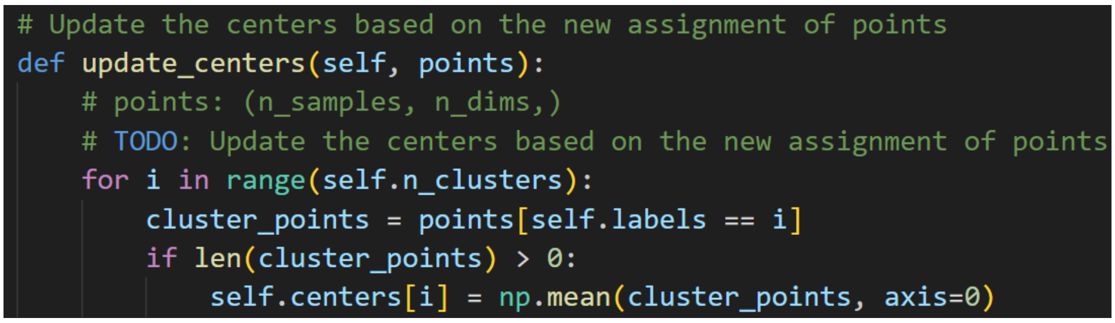
```python
def transform(self, X:np.ndarray):
    # X: [n_samples, n_features]
    X_reduced = np.zeros((X.shape[0], self.n_components))

    # TODO: transform the data to low dimension
    K = self.kernel_f(X)
    N = K.shape[0]
    one_n = np.ones((N, N)) / N
    K_centered = K - one_n.dot(K) - K.dot(one_n) + one_n.dot(K).dot(one_n)
    X_proj = K_centered.dot(self.eig_vecs.dot(np.diag(1 / np.sqrt(self.eig_vals))))
    return X_proj
```

对于 Kmeans，中心已随机初始化后，即每个样本计算到每个簇中心欧式距离，划分到最近的一个簇



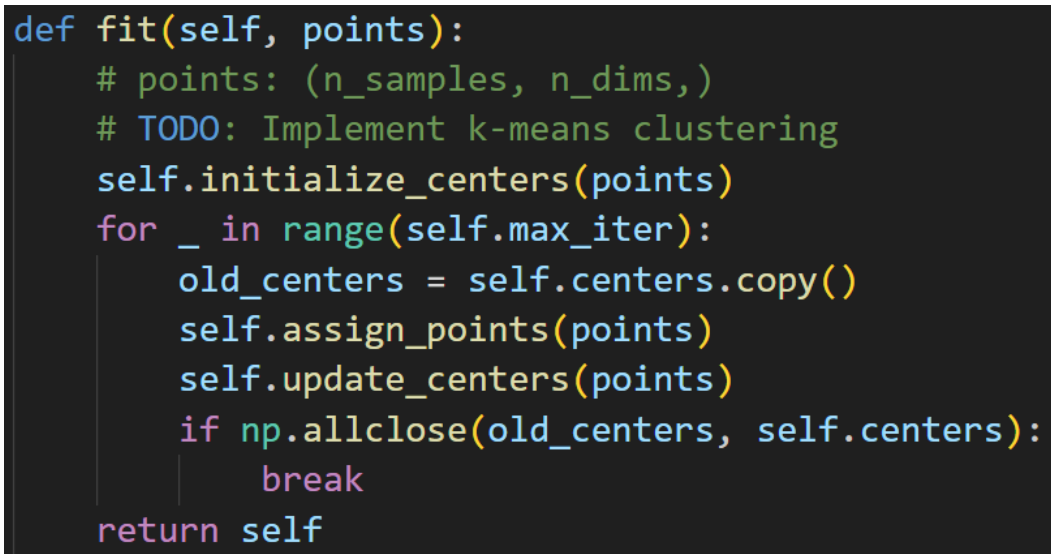
```python
def assign_points(self, points):
    # points: (n_samples, n_dims,)
    # return labels: (n_samples, )
    n_samples, n_dims = points.shape
    self.labels = np.zeros(n_samples)
    # TODO: Compute the distance between each point and each center
    # and Assign each point to the closest center
    distances = np.linalg.norm(points[:, np.newaxis] - self.centers, axis=2)
    self.labels = np.argmin(distances, axis=1)
    return self.labels
```

接着对划分好的簇求新中心，即簇中所有点坐标的平均值



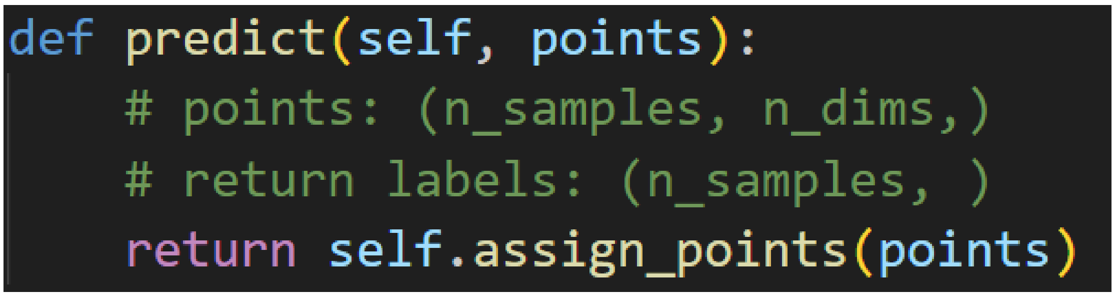
```python
# Update the centers based on the new assignment of points
def update_centers(self, points):
    # points: (n_samples, n_dims,)
    # TODO: Update the centers based on the new assignment of points
    for i in range(self.n_clusters):
        cluster_points = points[self.labels == i]
        if len(cluster_points) > 0:
            self.centers[i] = np.mean(cluster_points, axis=0)
```

Kmeans 即如此反复迭代更新中心，在 fit 函数中迭代至最大迭代次数或新旧中心几乎不变即可



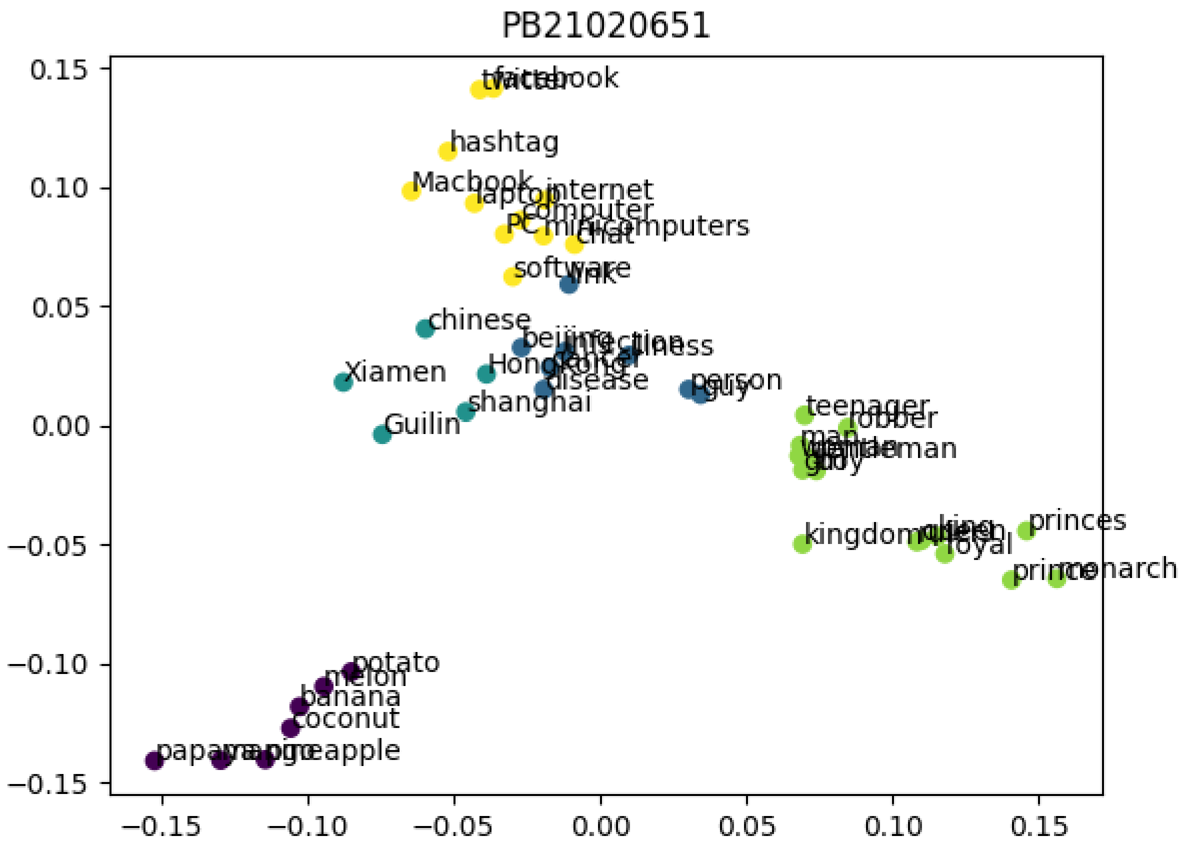
```python
def fit(self, points):
    # points: (n_samples, n_dims,)
    # TODO: Implement k-means clustering
    self.initialize_centers(points)
    for _ in range(self.max_iter):
        old_centers = self.centers.copy()
        self.assign_points(points)
        self.update_centers(points)
        if np.allclose(old_centers, self.centers):
            break
    return self
```

而对于预测只需要找到最近的簇中心，即属于该簇，调用 assignpoint 函数即可



```
def predict(self, points):
    # points: (n_samples, n_dims,)
    # return labels: (n_samples, )
    return self.assign_points(points)
```

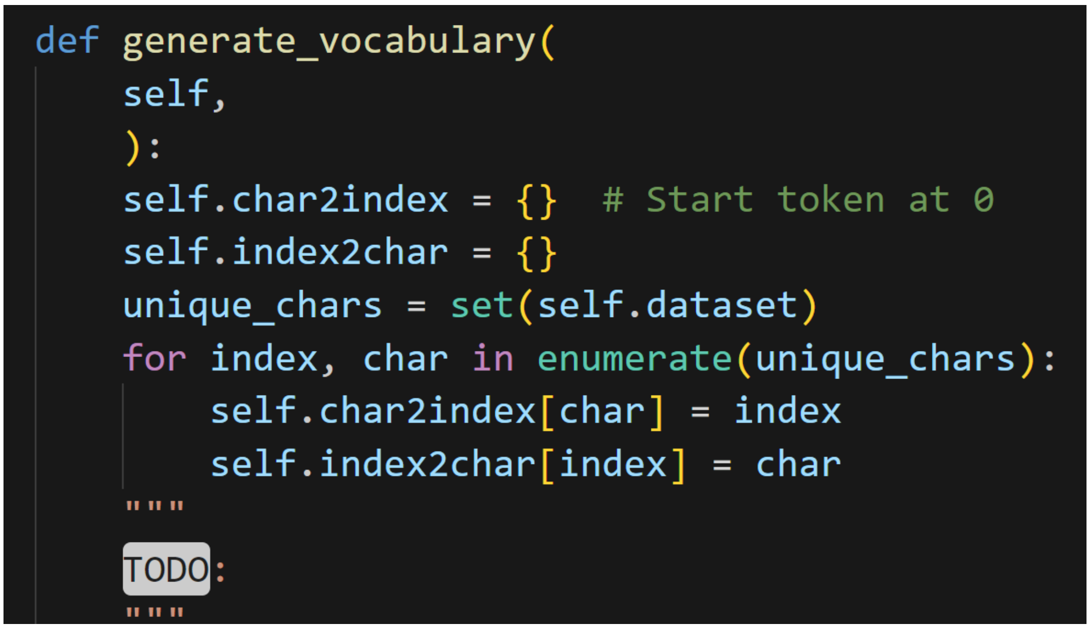
实验结果



Part 2

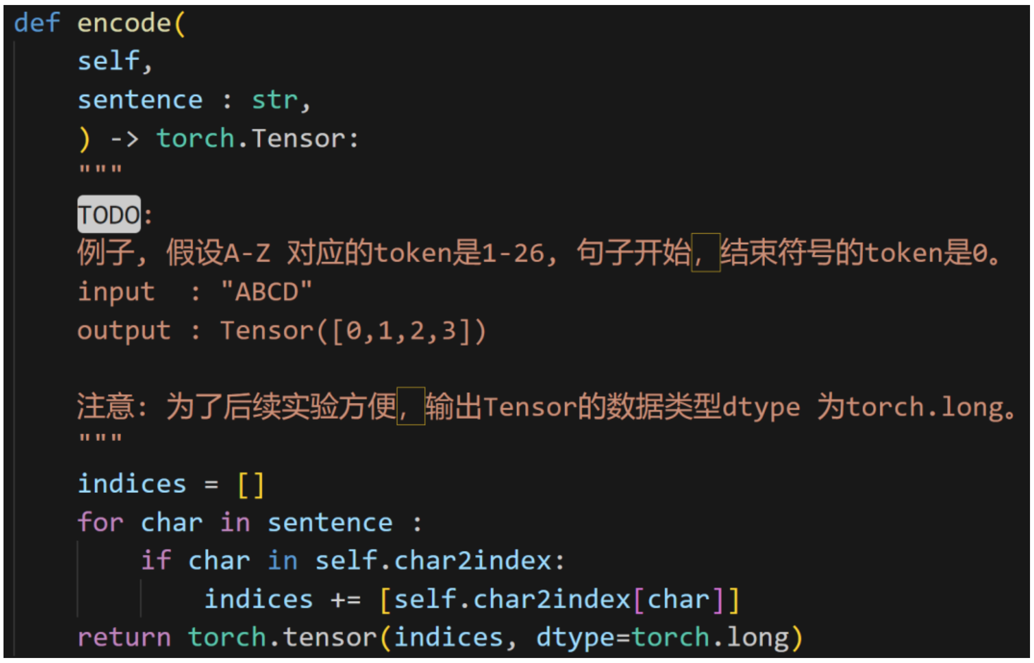
Tokenizer

分词器首先需要构建词表，即遍历数据集，取所有不同字符，将其与数字一一对应存入字典中



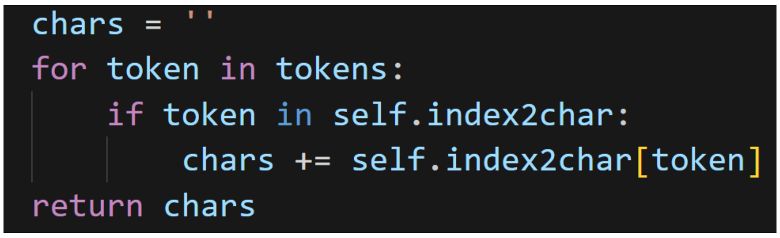
```python
def generate_vocabulary(
        self,
        ):
        self.char2index = {}  # Start token at 0
        self.index2char = {}
        unique_chars = set(self.dataset)
        for index, char in enumerate(unique_chars):
            self.char2index[char] = index
            self.index2char[index] = char
        """
        TODO:
        """
```

编码部分即遍历句子中字符，查字典转化为数字编码



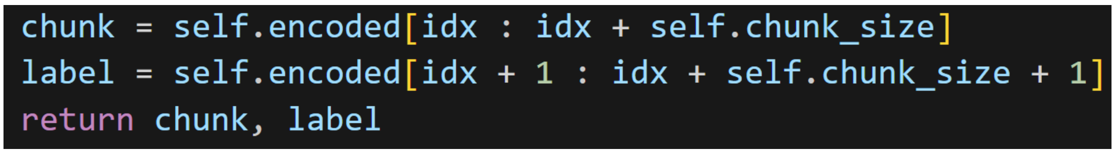
```python
def encode(
        self,
        sentence : str,
        ) -> torch.Tensor:
        """
        TODO:
        例子，假设A-Z 对应的token是1-26，句子开始，结束符号的token是0。
        input  : "ABCD"
        output : Tensor([0,1,2,3])

        注意：为了后续实验方便，输出Tensor的数据类型dtype 为torch.long。
        """
        indices = []
        for char in sentence :
            if char in self.char2index:
                indices += [self.char2index[char]]
        return torch.tensor(indices, dtype=torch.long)
```

解码即将数字编码变回字符



```python
chars = ''
for token in tokens:
    if token in self.index2char:
        chars += self.index2char[token]
return chars
```

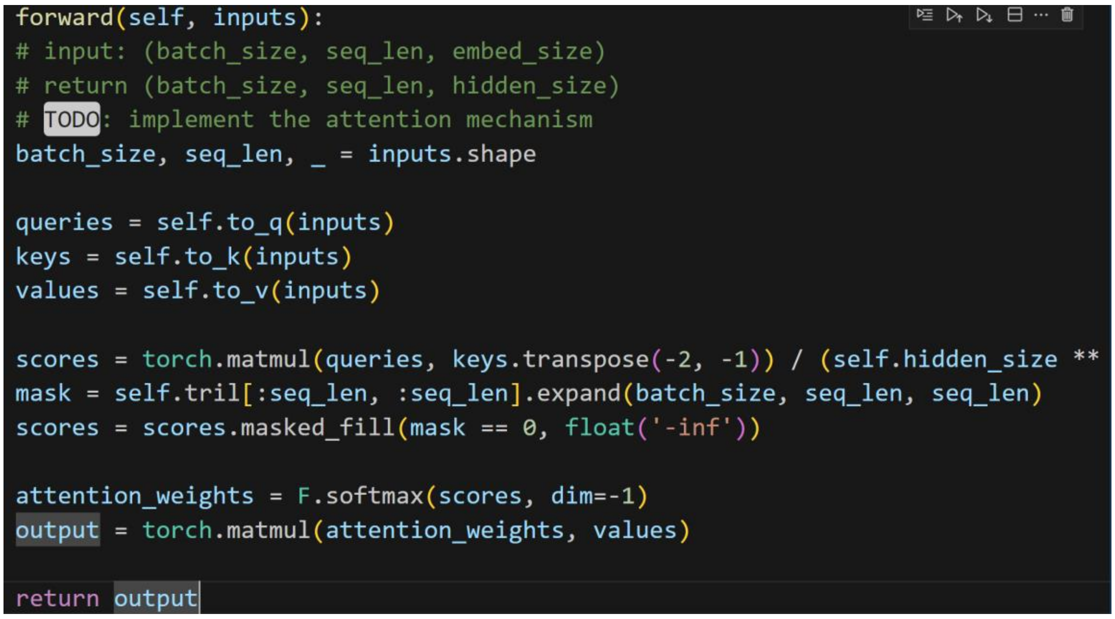
而对于实现 dataset 类，取一个样本特征即对应语块，而向后一位的语块作为标签



```
chunk = self.encoded[idx : idx + self.chunk_size]
label = self.encoded[idx + 1 : idx + self.chunk_size + 1]
return chunk, label
```

Attention

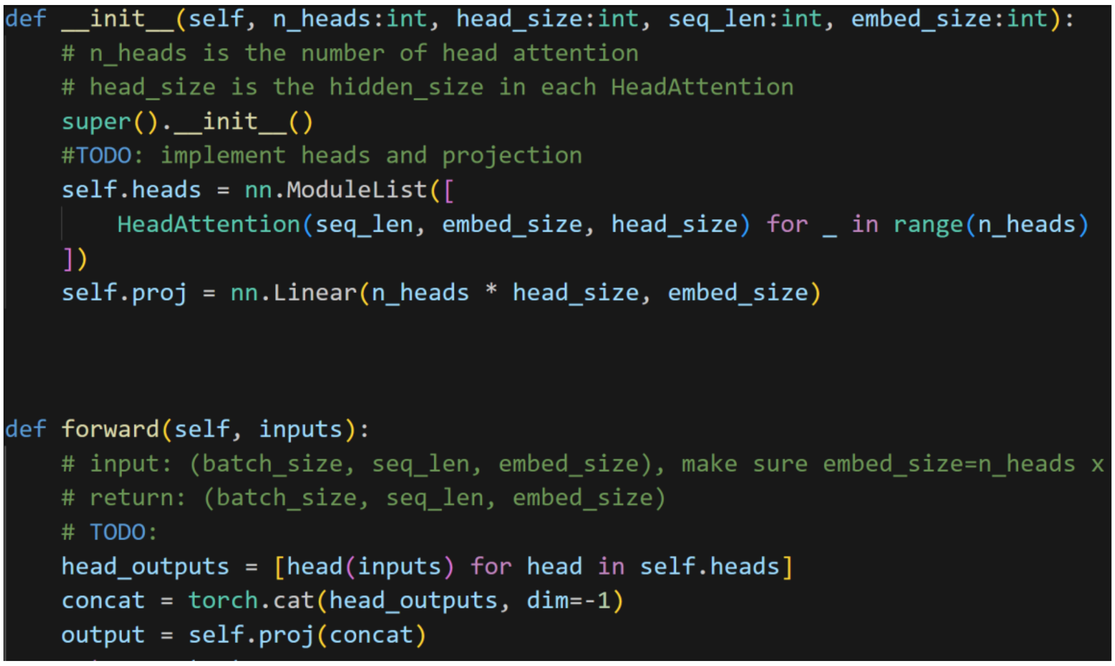
注意力机制即根据公式完成, 对下文信息的掩码以及归一化即参考 attention.ipynb 中 weight 的做法



```
forward(self, inputs):
# input: (batch_size, seq_len, embed_size)
# return (batch_size, seq_len, hidden_size)
# TODO: implement the attention mechanism
batch_size, seq_len, _ = inputs.shape

queries = self.to_q(inputs)
keys = self.to_k(inputs)
values = self.to_v(inputs)

scores = torch.matmul(queries, keys.transpose(-2, -1)) / (self.hidden_size **
mask = self.tril[:seq_len, :seq_len].expand(batch_size, seq_len, seq_len)
scores = scores.masked_fill(mask == 0, float('-inf'))

attention_weights = F.softmax(scores, dim=-1)
output = torch.matmul(attention_weights, values)

return output
```

多头注意力

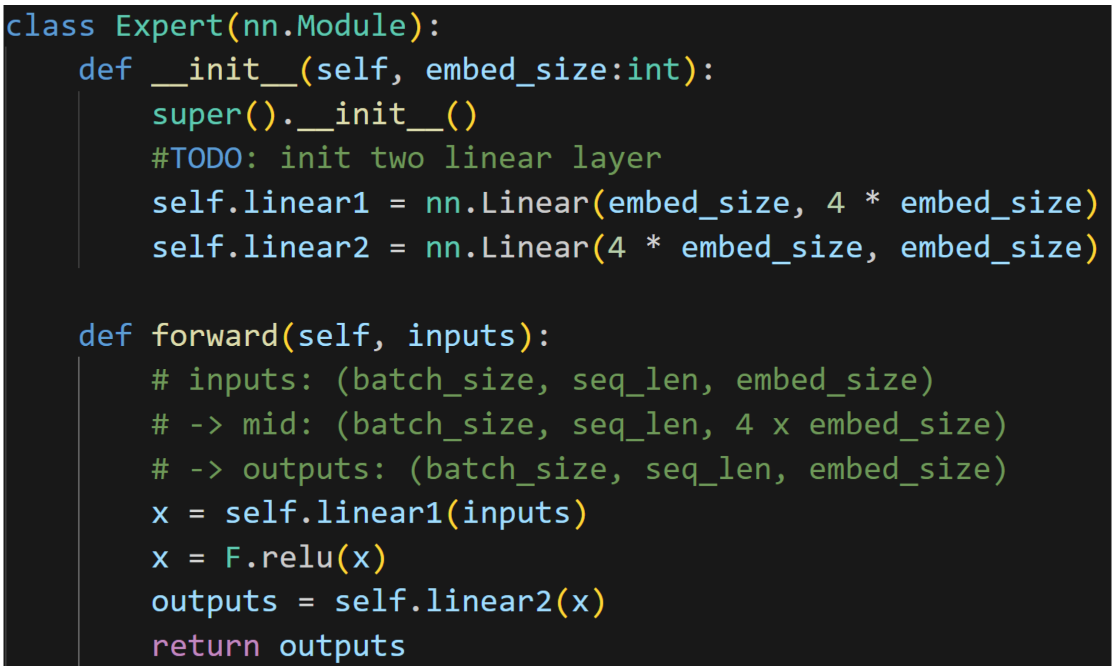
按照要求输入经过不同注意力层，最后拼接结果映射到输出的特征维度



```
def __init__(self, n_heads:int, head_size:int, seq_len:int, embed_size:int):
    # n_heads is the number of head attention
    # head_size is the hidden_size in each HeadAttention
    super().__init__()
    #TODO: implement heads and projection
    self.heads = nn.ModuleList([
        HeadAttention(seq_len, embed_size, head_size) for _ in range(n_heads)
    ])
    self.proj = nn.Linear(n_heads * head_size, embed_size)



def forward(self, inputs):
    # input: (batch_size, seq_len, embed_size), make sure embed_size=n_heads x
    # return: (batch_size, seq_len, embed_size)
    # TODO:
    head_outputs = [head(inputs) for head in self.heads]
    concat = torch.cat(head_outputs, dim=-1)
    output = self.proj(concat)
```

Expert

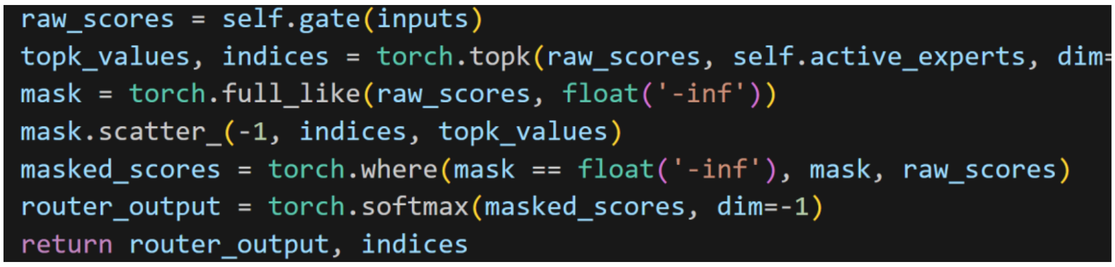
按照要求使用两层全连接层先放大四倍再还原



```python
class Expert(nn.Module):
    def __init__(self, embed_size:int):
        super().__init__()
        #TODO: init two linear layer
        self.linear1 = nn.Linear(embed_size, 4 * embed_size)
        self.linear2 = nn.Linear(4 * embed_size, embed_size)

    def forward(self, inputs):
        # inputs: (batch_size, seq_len, embed_size)
        # -> mid: (batch_size, seq_len, 4 x embed_size)
        # -> outputs: (batch_size, seq_len, embed_size)
        x = self.linear1(inputs)
        x = F.relu(x)
        outputs = self.linear2(x)
        return outputs
```

Topkrouter

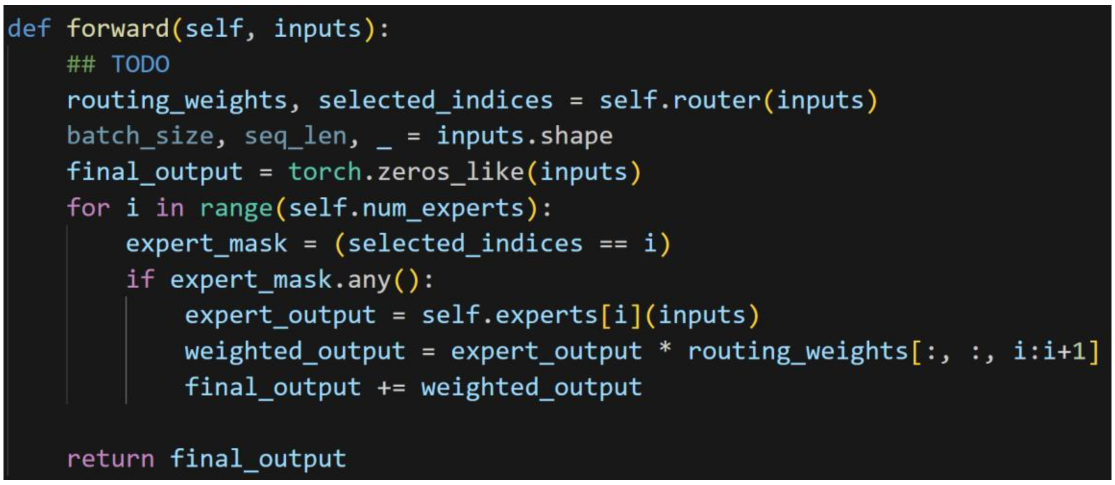
按照教程，self.gate 为全连接层评价专家分数；在选取 top activeexperts 的专家后，为使操作可导，将选取专家的掩码 0 替换为-inf，再取 softmax 函数达到效果，得到教程中的 alpha



```python
raw_scores = self.gate(inputs)
topk_values, indices = torch.topk(raw_scores, self.active_experts, dim=
mask = torch.full_like(raw_scores, float('-inf'))
mask.scatter_(-1, indices, topk_values)
masked_scores = torch.where(mask == float('-inf'), mask, raw_scores)
router_output = torch.softmax(masked_scores, dim=-1)
return router_output, indices
```

MOE

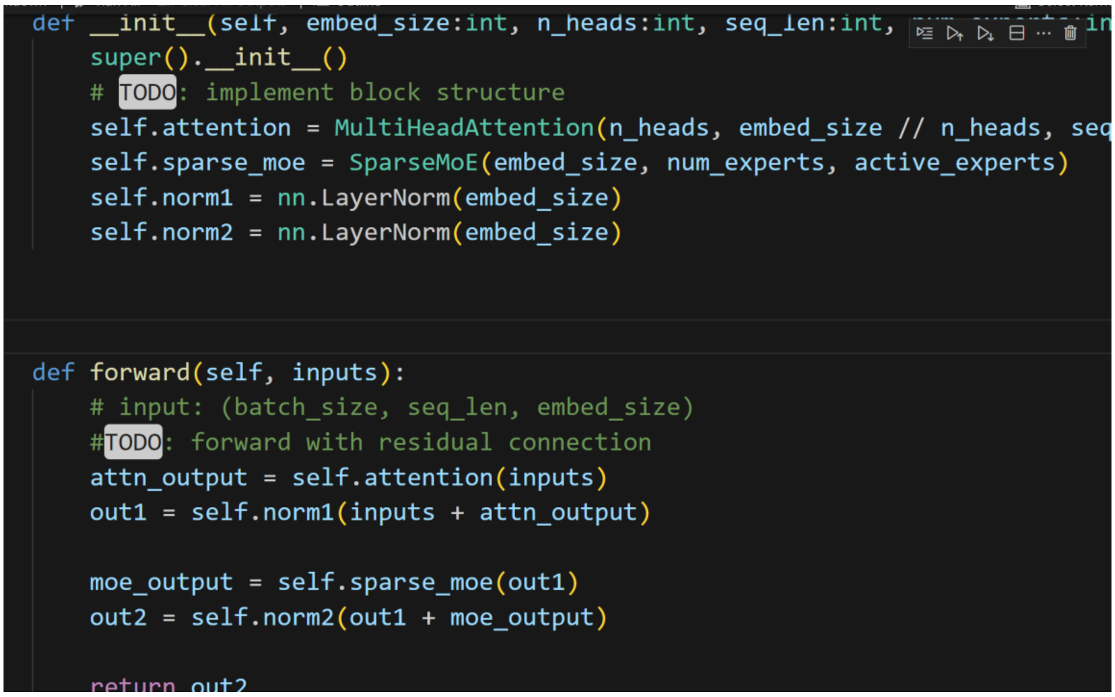
在 moe 中调用专家和选路网络，对于选取的专家利用选路网络计算的 alpha 加权输出所有专家的输出



```python
def forward(self, inputs):
    ## TODO
    routing_weights, selected_indices = self.router(inputs)
    batch_size, seq_len, _ = inputs.shape
    final_output = torch.zeros_like(inputs)
    for i in range(self.num_experts):
        expert_mask = (selected_indices == i)
        if expert_mask.any():
            expert_output = self.experts[i](inputs)
            weighted_output = expert_output * routing_weights[:, :, i:i+1]
            final_output += weighted_output

    return final_output
```

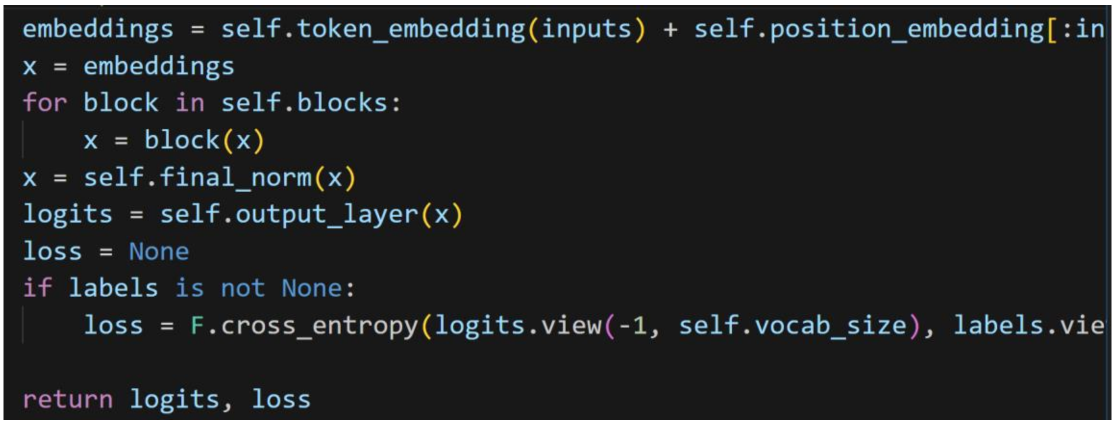
Block

根据注释提示，使用多头注意力层，moe，以及 layer normalization 搭建网络，并且使用残差连接，使得在反向传播过程不容易出现梯度消失



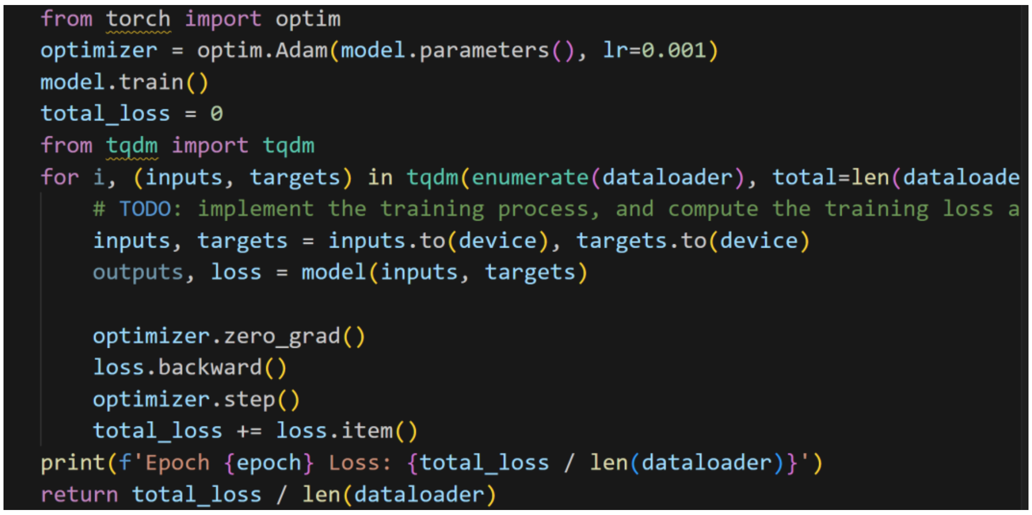
```python
def __init__(self, embed_size:int, n_heads:int, seq_len:int, num_experts:in
    super().__init__()
    # TODO: implement block structure
    self.attention = MultiHeadAttention(n_heads, embed_size // n_heads, seq
    self.sparse_moe = SparseMoE(embed_size, num_experts, active_experts)
    self.norm1 = nn.LayerNorm(embed_size)
    self.norm2 = nn.LayerNorm(embed_size)


def forward(self, inputs):
    # input: (batch_size, seq_len, embed_size)
    #TODO: forward with residual connection
    attn_output = self.attention(inputs)
    out1 = self.norm1(inputs + attn_output)

    moe_output = self.sparse_moe(out1)
    out2 = self.norm2(out1 + moe_output)

    return out2
```

最终的 moetransformer 将输入先进行词嵌入，然后取和其长度相同的位置编码进行嵌入，经过 transformer 基本块，最后映射回语块，并选用交叉熵计算生成语块和实际语块的差异



```python
embeddings = self.token_embedding(inputs) + self.position_embedding[:in
x = embeddings
for block in self.blocks:
    x = block(x)
x = self.final_norm(x)
logits = self.output_layer(x)
loss = None
if labels is not None:
    loss = F.cross_entropy(logits.view(-1, self.vocab_size), labels.vie

return logits, loss
```

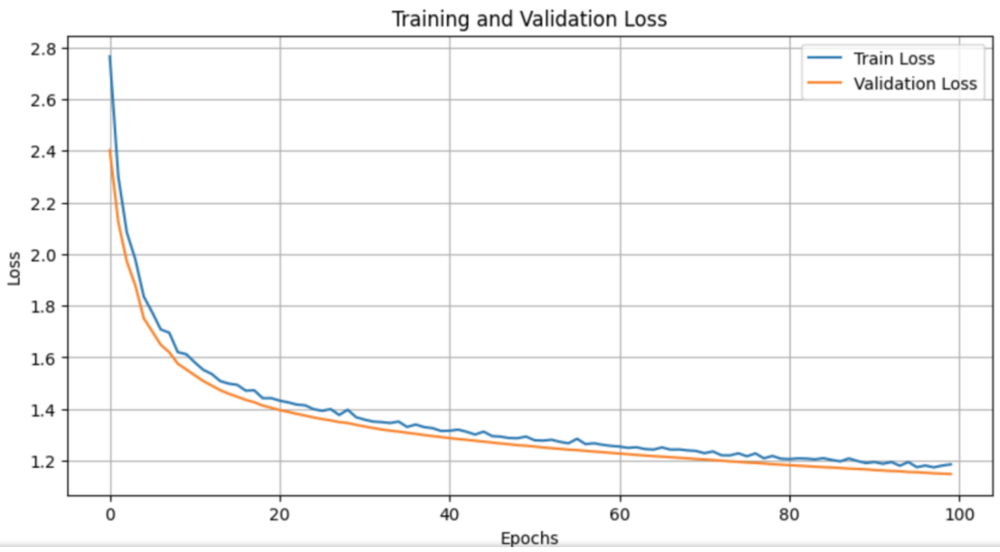
训练过程即选用 Adam 优化器，学习率设为 0.001，完成样本输入，loss 计算和反向传播，而验证过程仅需计算 loss



```python
from torch import optim
optimizer = optim.Adam(model.parameters(), lr=0.001)
model.train()
total_loss = 0
from tqdm import tqdm
for i, (inputs, targets) in tqdm(enumerate(dataloader), total=len(dataloade
    # TODO: implement the training process, and compute the training loss a
    inputs, targets = inputs.to(device), targets.to(device)
    outputs, loss = model(inputs, targets)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    total_loss += loss.item()
print(f'Epoch {epoch} Loss: {total_loss / len(dataloader)}')
return total_loss / len(dataloader)
```

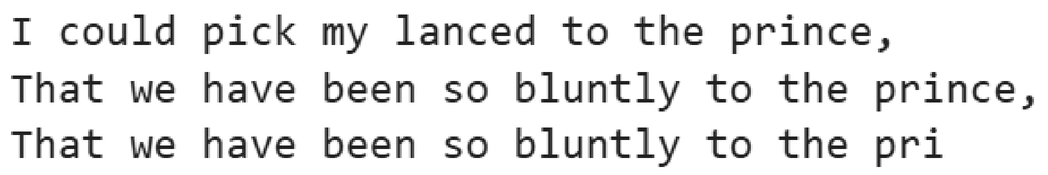
实验结果

以下为训练 100 个 epoch 的结果可以发现模型在训练集和验证集表现相近，没有出现过拟合，并在 100 个 epoch 时已趋于稳定



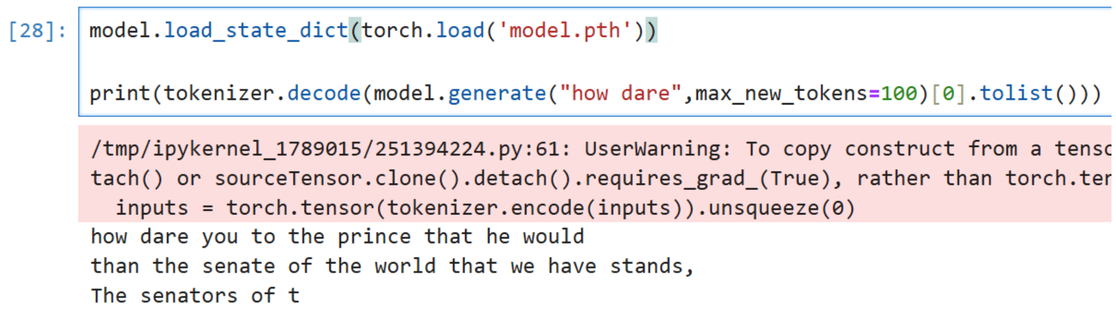
生成测试

对于给定的 I could pick my lance 结果如下，当训练 5 个 epoch 时生成的大多是 the，10 个时大多是 shall be，我认为或许是因为 the 出现频率最高，模型先学习到了分布概率，然后学习到合理应用该词汇，接着 shall 次高于是开始复读 shall，而这里出现 prince 复读也是这一原因，prince 在数据集中出现 200 多次，模型受其频率影响反复生成相关内容。



```
I could pick my lanced to the prince,
That we have been so bluntly to the prince,
That we have been so bluntly to the pri
```

使用 how dare 生成同样出现了 prince



```
[28]: model.load_state_dict(torch.load('model.pth'))

      print(tokenizer.decode(model.generate("how dare",max_new_tokens=100)[0].tolist()))

      /tmp/ipykernel_1789015/251394224.py:61: UserWarning: To copy construct from a tensc
      tach() or sourceTensor.clone().detach().requires_grad_(True), rather than torch.ter
        inputs = torch.tensor(tokenizer.encode(inputs)).unsqueeze(0)
      how dare you to the prince that he would
      than the senate of the world that we have stands,
      The senators of t
```

与之相似对于“：”模型学习到了出现频率最高的是角色开始台词的冒号，故句中含冒号就会生成角色加台词的组合

```python
model.load_state_dict(torch.load('model.pth'))

print(tokenizer.decode(model.generate("You are a saucy boy:",max_new_tokens=100)[0].tolist()))
```

```
/tmp/ipykernel_1789015/251394224.py:61: UserWarning: To copy construct from a tensor, it is rec
tach() or sourceTensor.clone().detach().requires_grad_(True), rather than torch.tensor(sourceTe
  inputs = torch.tensor(tokenizer.encode(inputs)).unsqueeze(0)
You are a saucy boy: and therefore I speak.

KING RICHARD III:
What say you say to the princely sentence
That they shall
```