Lab2 实验报告

PB20111689 蓝俊玮

1. 贝叶斯网络

1.1 模型原理以及代码实现

1.1.1 `fit()` 函数

在 `fit()` 函数中，首先计算的是每个真实标签的先验概率。我们通过统计数据集中真实标签的分布来计算每个标签的先验概率，`bincount()` 就是用来统计每个标签的出现的频数，然后除以总数就可以得到先验概率 $P(Y)$：

```
label_counts = np.bincount(labels)
self.labels_prior = label_counts / n_samples
```

然后就是根据这个先验概率的值，我们要计算在给定每个标签的情况下，每个像素显示亮暗状态的概率，即计算该条件概率 $P(F_{i,j}|Y)$。

那么就是首先需要给定每个标签：

```
for label in range(self.n_labels):
    label_pixels = pixels[labels == label]
```

然后从像素数据中筛选出给定标签的样本。在手写数字识别的问题中，我们希望根据像素数据来预测对应的数字标签。为了训练模型和计算条件概率，我们需要将训练数据中的像素数据与其对应的标签进行匹配。因此通过该方式，我们可以从 `pixels` 数组中仅选择具有给定标签的像素数据。这样做的目的是为了在计算条件概率时，仅使用与特定标签相关的像素数据。这样，`label_pixels` 的表示维度就可以用 `(n_label_samples, n_pixels, )` 来表示了，其中 `n_label_samples` 表示为给定 `label` 下的样本数量。

```
for pixel in range(self.n_pixels):
    label_pixel_status_counts = np.bincount(label_pixels[:, pixel],
minlength=self.n_values) + 1
    current_label_counts = label_counts[label] + self.n_values
    self.pixels_cond_label[pixel, :, label] = label_pixel_status_counts /
current_label_counts
```

紧接着遍历每个像素，从给定标签的样本 `label_pixels` 中获取相应的像素值，即像素的亮暗状态。`label_pixels[:, pixel]` 就可以表示选取 `pixel`，然后就可以统计亮暗的出现的频数。依然使用 `np.bincount()` 函数来统计在给定像素上的亮暗的频数。参数 `minlength=self.n_values` 确保了即使某个亮暗状态没有出现在样本中，也会被计入频数中，避免了零概率问题。同时统计给定标签的样本数量 `label_counts[label]`，并且以拉普拉斯修正的方式来平滑计算概率。

$$\hat{P}(Y) = \frac{|D_Y| + 1}{|D| + N}$$

$$\hat{P}(F_{i,j}|Y) = \frac{|D_{Y,F_{i,j}}| + 1}{|D_Y| + N_{F_{i,j}}}$$

其中 D 表示样本数据集，N 表示样本中的类别数量，下标表示给定条件。那么先验概率 $P(Y)$ 平滑后得到的 $\hat{P}(Y)$ 为：

```
label_counts = np.bincount(labels) + 1
self.labels_prior = label_counts / (n_samples + self.n_labels)
```

而给定标签、每个像素的平滑条件概率为 $\hat{P}(F_{i,j}|Y)$ 为：

```
label_pixel_status_counts = np.bincount(label_pixels[:, pixel],
minlength=self.n_values) + 1
current_label_counts = label_counts[label] + self.n_values
self.pixels_cond_label[pixel, :, label] = label_pixel_status_counts /
current_label_counts
```

最后的 `fit()` 函数如下：

```
def fit(self, pixels, labels):
    n_samples = len(labels)
    # TODO: calculate prior probability and conditional probability

    # count each label
    label_counts = np.bincount(labels) + 1
    # calculate label prior probability
    self.labels_prior = label_counts / (n_samples + self.n_labels)

    # calculate conditional probability
    for label in range(self.n_labels):
        # label_pixels: (n_label_samples, n_pixels, )
        label_pixels = pixels[labels == label]
        for pixel in range(self.n_pixels):
            # count each pixel status given label
            label_pixel_status_counts = np.bincount(label_pixels[:, pixel],
minlength=self.n_values) + 1
            # get current label counts
            current_label_counts = label_counts[label] + self.n_values
            # calculate conditional probability
            self.pixels_cond_label[pixel, :, label] = label_pixel_status_counts
/ current_label_counts
```

1.1.2 `predict()` 函数

在 `predict()` 函数中，我们需要利用我们在训练过程中学习到的条件概率来进行预测。在预测过程中，对于每个新的样本，我们需要计算它属于每个可能标签的概率，并选择具有最高概率的标签作为预测结果。那么预测过程就可以表示成如下过程：

$$P(Y|F) = \frac{P(F|Y)P(Y)}{P(F)} = \alpha P(F|Y)P(Y) = \alpha \prod_{i,j} P(F_{i,j}|Y)P(Y)$$

因为我们只需要在最后预测过程中取最大的值作为预测值：

$$Y^* = \arg\max \left(P(Y|F) \right)$$

因此无需考虑归一化的系数 α，只需要计算后面的联合概率，再求最值即可。同时由于概率是个比较小的值，在大的样本数据集下进行连乘操作容易出现下溢出现象。因此对其进行取对数操作：

$$\log\left(P(Y|F)\right) = \log\left(P(Y)\right) + \sum_{i,j}\log\left(P(F_{i,j}|Y)\right)$$

$$Y^* = \arg\max\left(\log\left(P(Y|F)\right)\right)$$

那么对于预测过程 `predict()` 的实现如下：

```
def predict(self, pixels):
    n_samples = len(pixels)
    labels = np.zeros(n_samples)
    # TODO: predict for new data
    for i in range(n_samples):
        prob1 = np.log(self.labels_prior)
        prob2 = np.sum(np.log(self.pixels_cond_label[np.arange(self.n_pixels),
pixels[i]]), axis=0)
        pixel_prob = prob1 + prob2
        labels[i] = np.argmax(pixel_prob)

    return labels
```

其中 `prob1` 和 `prob2` 分别对应上述表达式的一部分。在计算得到每个像素对应的标签概率值 `pixel_prob` 后，选择具有最高概率的标签，使用 `np.argmax()` 函数找到综合概率数组中具有最大值的索引，即具有最高概率的标签。将该索引作为预测的标签存储在 `labels[i]` 中。

1.2 预测结果

程序最后预测的准确率为 84.31%：

```
问题 2  输出  终端  端口  调试控制台
(ai_env) junwei@admin:~/src_exp2_ai2023sp_ustc/part_1/src$ python Bayesian-network.py
test score: 0.843100
(ai_env) junwei@admin:~/src_exp2_ai2023sp_ustc/part_1/src$ echo PB20111689 蓝俊玮
PB20111689 蓝俊玮
(ai_env) junwei@admin:~/src_exp2_ai2023sp_ustc/part_1/src$
```

2. K-mans 聚类算法

2.1 模型原理以及代码实现

2.1.1 `assign_points()` 函数

在 K-means 算法中，我们在更新迭代过程中，首先是需要在固定聚类中心的情况下，计算每个样本数据点与各个聚类中心的距离，然后将每个样本数据点分配给最近的聚类中心。在本次实验中，我们采用的距离度量方式为欧式距离：

$$\text{dist}(\boldsymbol{x}_i, \boldsymbol{x}_j) = ||\boldsymbol{x}_i - \boldsymbol{x}_j||_2 = \sqrt{\sum_{u=1}^{n}|x_{iu} - x_{ju}|^2}$$

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

实现时，首先需要遍历所有的样本数据，计算数据点与每个中心之间的欧氏距离，使用 `np.linalg.norm` 函数实现。这样就得到了一个大小为 `n_clusters` 的数组 `distances`，其中每个元素表示数据点与特定中心之间的距离。参数 `axis=1` 指定沿第二个轴（即数据点与每个中心之间的距离）计算距离。然后，使用 `np.argmin` 函数找到距离数据点最近的中心的索引。该索引对应于数据点分配的簇标签。该标签存储在 `labels` 数组的索引 `i` 处。

2.1.2 `update_centers()` 函数

在 K-means 算法中，我们在更新迭代过程中，将每个样本分配到最近的聚类中心之后，需要在更新后重新计算聚类中心，采用的方法是以每个聚类内所有样本坐标的均值，作为新的聚类中心的坐标。

$$\boldsymbol{\mu}_i' = \frac{1}{C_i} \sum_{\boldsymbol{x} \in C_i} \boldsymbol{x}$$

```
def update_centers(self, centers, labels, points):
    # TODO: Update the centers based on the new assignment of points
    for k in range(self.k):
        cluster_points = points[labels == k]
        centers[k] = cluster_points.mean(axis=0)
    return centers
```

实现时，首先需要遍历每个聚类，并获取属于该聚类的数据点集合，通过布尔索引 `labels == k` 获取当前聚类的数据集 `cluster_points`，然后计算该集合的均值，即新的聚类中心，并将其存储在聚类中心 `centers` 数组中对应的索引 `k` 处。

2.1.3 `fit()` 函数

在 K-means 算法中，在更新迭代的过程中，其一般的过程如下：

- 首先固定聚类中心，计算每个样本数据点与每个聚类中心之间的距离，然后将每个样本数据点分配给距离最近的聚类。
- 在分配完每个数据点的类别后，然后更新聚类中心，以每个聚类类别所有样本数据的均值向量作为该类的聚类中心。
- 当所有的聚类中心都没有发生变化时，算法停止迭代过程。否则就继续进行迭代，直至达到迭代次数要求。

```
def fit(self, points):
    # TODO: Implement k-means clustering
    centers = self.initialize_centers(points)

    for _ in range(self.max_iter):
        labels = self.assign_points(centers, points)
        new_centers = self.update_centers(centers, labels, points)

        # If the centers did not change, stop iterating
        if np.all(centers == new_centers):
            break
        centers = new_centers
    return centers
```

实现时，首先，通过调用 `self.initialize_centers(points)` 初始化聚类中心，将返回的初始中心存储在 `centers` 变量中。然后，使用 `for` 循环执行最大迭代次数 `self.max_iter` 次聚类迭代过程。在每次迭代中，首先调用 `self.assign_points(centers, points)` 方法，将数据点分配给最近的聚类中心，并将返回的标签存储在 `labels` 变量中。

接下来，调用 `self.update_centers(centers, labels, points)` 方法，根据新的数据点分配情况更新聚类中心，并将返回的更新后的中心存储在 `new_centers` 变量中。

然后，通过比较 `centers` 和 `new_centers` 是否完全相等，判断聚类中心是否发生了变化。如果聚类中心没有发生变化，则停止迭代，否则将 `centers` 更新为 `new_centers`。

2.1.4 `compress()` 函数

使用 K-means 算法压缩图像的原理为，将图像上的每一个像素 $(\mathrm{R}, \mathrm{G}, \mathrm{B})$ 视作一个数据点，并对所有点进行聚类。完成聚类之后，将所有数据点的值替换成其聚类中心的值，这样仅需保留几个聚类中心的数据点值以及其他点的类别索引，从而达到压缩图片的目的。

```
def compress(self, img):
    # flatten the image pixels
    points = img.reshape((-1, img.shape[-1]))
    # TODO: fit the points and
    # Replace each pixel value with its nearby center value
    centers = self.fit(points)

    # Assign each point to the closest center
    labels = self.assign_points(centers, points)

    # Replace each pixel value with its nearby center value
    compressed_points = centers[labels.astype(int)]
    compressed_img = compressed_points.reshape(img.shape)

    return compressed_img
```

所以首先我们需要根据图像的像素数据点对其进行聚类操作：调用 `self.fit(points)` 方法对展平的像素点进行聚类，返回最终的聚类中心，并将其存储在 `centers` 变量中。

接下来，使用 `self.assign_points(centers, points)` 方法将每个像素点分配给最近的聚类中心，返回分配的标签，并将其存储在 `labels` 变量中。

然后，通过 `centers[labels.astype(int)]` 取出每个像素点对应的最近聚类中心的值，得到压缩后的像素点集合 `compressed_points`。

最后，通过 `compressed_points.reshape(img.shape)` 将压缩后的像素点重新恢复为与原始图像相同的维度，得到压缩后的图像 `compressed_img`。

2.2 实验结果

3. Transformer

3.1 模型原理以及代码实现

3.1.1 `char_tokenizer`

```
# read the dataset
with open("../data/input.txt", "r", encoding="utf-8") as f:
    text = f.read()
chars = sorted(list(set(text)))
```

发现在本次实验过程中，与常规的 transformer 的分词器实现不太一样，常规 transformer 的分词器是基于一个单词的词根来实现，而在本次实验中是采用基于字符级的分词。`chars = sorted(list(set(text)))` 将语料库分成了不同的字符，并且对其进行频数排序，将出现次数多的字符放到最前面。

那么实现分词器的时候，就可以将每个字符映射到一个索引值：

```
def __init__(self, corpus: List[str]):
    self.corpus = corpus
    self.n_vocab = len(corpus)
    self.char_to_int = {char: i for i, char in enumerate(corpus)}
    self.int_to_char = {i: char for char, i in self.char_to_int.items()}
```

- `__init__(self, corpus: List[str])`：构造函数接收一个字符串列表（`corpus`）作为输入，表示语料库。在构造函数中，它计算了语料库的大小（`n_vocab`）并创建了两个字典（`char_to_int` 和 `int_to_char`），用于将每个字符映射到唯一的整数和将整数映射回字符。

```
def encode(self, string: str):
    return [self.char_to_int[char] for char in string]
```

- `encode(self, string: str)`：`encode`方法将一个输入字符串(`string`)转换为一个整数索引列表，使用在构造函数中创建的`char_to_int`字典将每个字符映射到相应的整数。

```
def decode(self, codes: List[int]):
    return ''.join([self.int_to_char[code] for code in codes])
```

- `decode(self, codes: List[int])`：`decode`方法将一个整数索引列表(`codes`)转换回一个字符串，使用在构造函数中创建的`int_to_char`字典将每个整数映射回相应的字符。

这个分词器的基本思想是将输入的字符串分解为字符，并使用整数索引列表来表示。通过将字符映射到整数，可以在模型中处理和生成文本数据。这样，就可以在模型中使用整数索引列表表示文本数据，而不是直接使用字符串。然后在后续处理中，就可以利用`nn.Embedding`来学习字符嵌入向量了。

3.1.2 `PositionalEncoding`

```
def __init__(self, d_model, max_len):
    super().__init__()
    self.d_model = d_model

    # Create position encoding matrix
    pe = torch.zeros(max_len, d_model)
    position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
    div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-
math.log(10000.0) / d_model))
    pe[:, 0::2] = torch.sin(position * div_term)
    pe[:, 1::2] = torch.cos(position * div_term)
    pe = pe.unsqueeze(0)
    self.register_buffer('pe', pe)
```

- `__init__(self, d_model, max_len)`：在构造函数`__init__`中，我们定义了位置编码的一些参数。`d_model`表示嵌入向量的维度，`max_len`表示输入序列的最大长度。接下来，我们创建了位置编码矩阵`pe`，其形状为`(max_len, d_model)`。位置编码矩阵中的每个元素都表示一个位置和一个维度之间的映射关系。

  为了计算位置编码矩阵的值，我们首先创建了一个表示位置索引的张量`position`，其取值范围是从0到`max_len-1`。然后，我们使用`torch.exp`和`torch.arange`函数计算一个除数项`div_term`，该项用于计算正弦和余弦的周期。接下来，我们分别使用正弦和余弦函数将位置索引与除数项相乘，得到位置编码矩阵中奇数索引和偶数索引位置的值。

  最后，我们通过对位置编码矩阵进行维度扩展，将其形状从`(max_len, d_model)`扩展为`(1, max_len, d_model)`，并将其作为缓冲区注册到模型中，以便在前向传播中使用。

```
def forward(self, x):
    # Add position embeddings to the input
    x = x + self.pe[:, :x.size(1)]
    return x
```

- `forward(self, x)`：在前向传播函数`forward`中，我们将位置编码矩阵的前`x.size(1)`个位置编码加到输入张量`x`上。这里的`x.size(1)`表示输入序列的长度，因为位置编码矩阵的长度是预先定义的`max_len`。通过将位置编码与输入相加，模型可以学习在不同位置上的不同编码，从而捕捉到位置信息。

$$\begin{cases} PE(\text{ pos}, 2i) = \sin\left(\text{ pos }/10000^{2i/d_{\text{model}}}\right) \\ PE(\text{pos}, 2i+1) = \cos\left(\text{ pos }/10000^{2i/d_{\text{model}}}\right) \end{cases}$$

总之，这段代码实现了位置编码的计算逻辑，用于将位置信息嵌入到输入序列中，以帮助 transformer 模型处理序列数据时考虑位置顺序。

3.1.3 `Head`

```
def __init__(self, head_size):
    super().__init__()
    self.Key = nn.Linear(n_embd, head_size)
    self.Query = nn.Linear(n_embd, head_size)
    self.Value = nn.Linear(n_embd, head_size)
    # End of your code
    self.register_buffer("tril", torch.tril(torch.ones(block_size, block_size)))
```

- `__init__(self, head_size)`：构造函数接收一个整数 `head_size` 作为输入，表示注意力头的大小。在构造函数中，它创建了三个线性层（`Key`，`Query` 和 `Value`），每个层将输入的 `n_embd` 维度映射到 `head_size` 维度，并将它们分配给 `self.Key`，`self.Query` 和 `self.Value`。

```
def forward(self, inputs):
    batch_size, time, n_embd = inputs.shape
    # Apply linear transformations to obtain Key, Query, and Value
    keys = self.Key(inputs)
    queries = self.Query(inputs)
    values = self.Value(inputs)

    # Calculate attention scores
    attention_scores = torch.matmul(queries, keys.transpose(-1, -2))
    attention_scores = attention_scores / math.sqrt(keys.size(-1))

    # Apply masking to attention scores
    attention_scores = attention_scores.masked_fill(self.tril[:time, :time] ==
0, float('-inf'))

    # Apply softmax activation to obtain attention weights
    attention_weights = F.softmax(attention_scores, dim=-1)

    # Apply attention weights to values
    out = torch.matmul(attention_weights, values)

    # End of your code
    return out
```

- `forward(self, inputs)`：`forward` 方法实现了注意力头的前向传播。输入（`inputs`）是一个张量，形状为 `(batch, time, n_embd)`，表示输入序列。输出是一个形状为 `(batch, time, head_size)` 的张量，表示应用注意力机制后的输出序列。在这个方法中，它首先对输入应用线性变换，得到 `Key`、`Query` 和 `Value`。然后，它计算注意力分数通过将 Queries 与 Keys 的转置矩阵相乘，并除以 `sqrt(keys.size(-1))` 以进行缩放。接下来，它应用掩码操作将注意力分数的上三角部分设置为负无穷大，以防止模型在生成序列时关注未来的位置。然后，它应用 softmax 激活函数来获得注意力权重。最后，它将注意力权重应用于 Values，得到输出张量。

$$\text{Attention}(q, k, v) = \text{SoftMax}(\frac{qk^T}{\sqrt{d_k}})v$$

这个注意力头的作用是在自注意力机制中计算单个头部的注意力，用于在 transformer 模型中构建多头自注意力层。它将输入序列映射到不同的表示空间，通过计算输入中不同位置之间的相似度来确定每个位置的权重，并将权重应用于输入的不同位置进行加权平均，从而得到输出序列。

3.1.4 `MultiHeadAttention`

```
def __init__(self, n_heads, head_size):
    super().__init__()
    self.n_heads = n_heads
    self.head_size = head_size

    # Create individual heads
    self.heads = nn.ModuleList([Head(head_size) for _ in range(n_heads)])

    # Linear projection layer
    self.projection = nn.Linear(n_heads * head_size, n_embd)
```

- `__init__(self, n_heads, head_size)`：构造函数中创建多个独立的注意力头（`Head`），并将它们存储在 `nn.ModuleList` 中。我们使用 `Head` 类的实例化来创建每个注意力头。创建线性投影层（`projection`），该层将多个注意力头的输出进行线性投影，以得到最终的输出。该投影层的输入维度是 `n_heads * head_size`，输出维度是 `n_embd`。

```
def forward(self, inputs):
    # Split inputs into multiple heads
    head_outputs = [head(inputs) for head in self.heads]

    # Concatenate head outputs along the head dimension
    out = torch.cat(head_outputs, dim=-1)

    # Apply linear projection to obtain the final output
    return self.projection(out)
```

- `forward(self, inputs)`：`forward` 函数用于执行多头注意力的前向传播。它接受输入 `inputs` 作为参数，并执行以下操作：将输入 `inputs` 传递给每个注意力头（`Head`），得到多个注意力头的输出。在注意力头的输出上进行拼接操作，沿着最后一个维度（注意力头维度）进行拼接，以得到形状为 `(batch, time, n_heads * head_size)` 的张量。将拼接后的张量传递给线性投影层，进行线性投影，将其维度转换为 `(batch, time, n_embd)`。返回线性投影层的输出作为多头注意力的最终输出。

$$\text{MultiHead}(Q, K, V) = \text{Concat}\left(\text{head}_1, \ldots, \text{head}_n\right) W_o$$
$$\text{head}_i = \text{Attention}\left(QW_Q^i, KW_K^i, VW_V^i\right)$$

3.1.5 `FeedForward`

```
def __init__(self, n_embd):
    super().__init__()

    self.net = nn.Sequential(
        nn.Linear(n_embd, 4 * n_embd),
        nn.ReLU(),
        nn.Linear(4 * n_embd, n_embd),
    )
```

- `__init__(self, n_embd)`：在构造函数 `__init__` 中，我们接收一个参数 `n_embd`，它表示输入和输出的维度大小。然后，我们执行以下操作：创建一个 `nn.Sequential` 对象，该对象是一个有序的层序列。在序列中添加三个线性层，这些层用于实现前馈神经网络的非线性变换。第一个线性层的输入维度是 `n_embd`，输出维度是 `4 * n_embd`，第二个线性层使用ReLU作为激活函数，没有改变维度，第三个线性层的输入维度是 `4 * n_embd`，输出维度是 `n_embd`。

```
def forward(self, inputs):
    return self.net(inputs)
```

- `forward(self, inputs)`： `forward` 函数将输入 `inputs` 传递给前馈神经网络的序列模型 `self.net`，该序列模型会自动按顺序执行添加的层。最后，函数返回前馈神经网络的输出。

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

这段代码的作用是实现一个简单的前馈神经网络，用于在 transformer 模型的每个位置编码和每个块之间进行非线性变换。这有助于增加模型的表达能力和学习复杂的特征表示。

3.1.6 `Block`

```
def __init__(self, n_embd, n_heads):
    super().__init__()

    # Multi-head attention layer
    self.attention = MultiHeadAttention(n_heads, n_embd // n_heads)

    # Layer normalization layers
    self.norm1 = nn.LayerNorm(n_embd)
    self.norm2 = nn.LayerNorm(n_embd)

    # Feed-forward network
    self.feed_forward = FeedForward(n_embd)
```

- `__init__(self, n_embd, n_heads)`：创建一个多头注意力层 (MultiHeadAttention)，使用 `n_heads` 和 `n_embd // n_heads` 作为参数。这个注意力层将输入进行多头注意力计算。然后创建两个层归一化层 (LayerNorm)，分别用于输入和注意力输出的归一化。最后创建一个前馈神经网络 (FeedForward)，使用 `n_embd` 作为输入和输出维度

```
def forward(self, inputs):
    # Apply layer normalization to the inputs
    norm_inputs = self.norm1(inputs)

    # Perform multi-head attention
    attention_out = self.attention(norm_inputs)

    # Add residual connection and apply layer normalization
    attention_out = norm_inputs + attention_out
    norm_attention_out = self.norm2(attention_out)

    # Apply feed-forward network
    ff_out = self.feed_forward(norm_attention_out)

    # Add residual connection
    out = norm_attention_out + ff_out

    # End of your code
```

```
        return out
```

- `forward(self, inputs)`： `forward` 函数将输入 `inputs` 传递给模块的前向传播。然后，按照 transformer 的原始形式执行以下操作：对输入进行层归一化，得到归一化后的输入 `norm_inputs` 。将归一化后的输入传递给多头注意力层，得到注意力输出 `attention_out` 。将注意力输出与归一化后的输入进行残差连接，并再次进行层归一化，得到归一化后的注意力输出 `norm_attention_out` 。将归一化后的注意力输出传递给前馈神经网络，得到前馈网络的输出 `ff_out` 。将前馈网络的输出与归一化后的注意力输出进行残差连接，得到模块的最终输出 `out` 。

这段代码的作用是实现 transformer 模型中的一个基本模块。该模块由多头注意力层、层归一化层和前馈神经网络组成，其中注意力层用于捕捉输入的全局依赖关系，前馈神经网络用于引入非线性变换。层归一化层用于规范化输入和输出，以稳定模型的训练。通过堆叠多个这样的基本模块，可以构建更复杂的 transformer 模型。

3.1.7 `Transformer`

```
def __init__(self):
    super().__init__()
    # Embedding table
    self.embedding = nn.Embedding(n_vocab, n_embd)
    self.positional_encoding = PositionalEncoding(n_embd, block_size)


    # Stack of transformer blocks
    self.blocks = nn.ModuleList([
        Block(n_embd, n_heads) for _ in range(n_layers)
    ])

    # Layer normalization layer
    self.norm = nn.LayerNorm(n_embd)
    # Linear layer for output projection
    self.linear = nn.Linear(n_embd, n_vocab)
```

- `__init__(self)`：在构造函数 `__init__` 中，我们创建了 transformer 模型的各个组件。让我们来看看每个组件的作用：创建了一个嵌入层 (Embedding)，用于将输入的整数标记转换为密集的嵌入向量。嵌入层的大小为 `(n_vocab, n_embd)` ，其中 `n_vocab` 表示词汇表的大小， `n_embd` 表示嵌入向量的维度。创建了一系列 transformer 模块 (Block) 的堆叠，使用 `n_embd` 和 `n_heads` 作为参数。通过重复调用 `Block` 类的构造函数，我们创建了 `n_layers` 个相同的模块，并将它们存储在 `blocks` 列表中。创建了一个层归一化层 (LayerNorm)，用于对 transformer 模块的输出进行归一化。创建了一个线性层 (Linear)，用于将 transformer 模型的输出投影到与词汇表大小相等的向量空间，以便进行下游任务（例如语言模型的预测）。线性层的大小为 `(n_embd, n_vocab)` ，其中 `n_vocab` 表示词汇表的大小。

```
def forward(self, inputs, labels=None):
    # Embedding inputs
    embedded_inputs = self.embedding(inputs)
    # Apply transformer blocks
    out = embedded_inputs
    for block in self.blocks:
        out = block(out)
    # Apply layer normalization
    out = self.norm(out)

    # Linear projection
```

```
        logits = self.linear(out)
        if labels is None:
            loss = None
        else:
            batch, time, channel = logits.shape
            logits = logits.view(batch * time, channel)
            labels = labels.view(batch * time)
            loss = F.cross_entropy(logits, labels)
        return logits, loss
```

`forward(self, inputs, labels=None)`：在 `forward` 函数中，我们定义了 transformer 模型的前向传播。我们按照以下步骤执行：将输入 `inputs` 传递给嵌入层，将整数标记转换为嵌入向量。将嵌入向量传递给堆叠的 transformer 模块。每个模块的输出将成为下一个模块的输入。对最后一个模块的输出应用层归一化。将归一化后的输出传递给线性层，得到最终的 Logits。如果提供了 `labels` (用于训练模型)，则计算交叉熵损失。

```
def generate(self, inputs, max_new_tokens):
    for _ in range(max_new_tokens):
        # generates new tokens by iteratively sampling from the model's
predicted probability distribution,
        # concatenating the sampled tokens to the input sequence, and returning
the updated sequence.

        # truncate
        inputs = inputs[:, -block_size:]
        logits, _ = self.forward(inputs)
        probabilities = F.softmax(logits[:, -1], dim=-1)
        sampled_token = torch.multinomial(probabilities, num_samples=1)
        inputs = torch.cat([inputs, sampled_token], dim=1)
    # End of your code
    return inputs
```

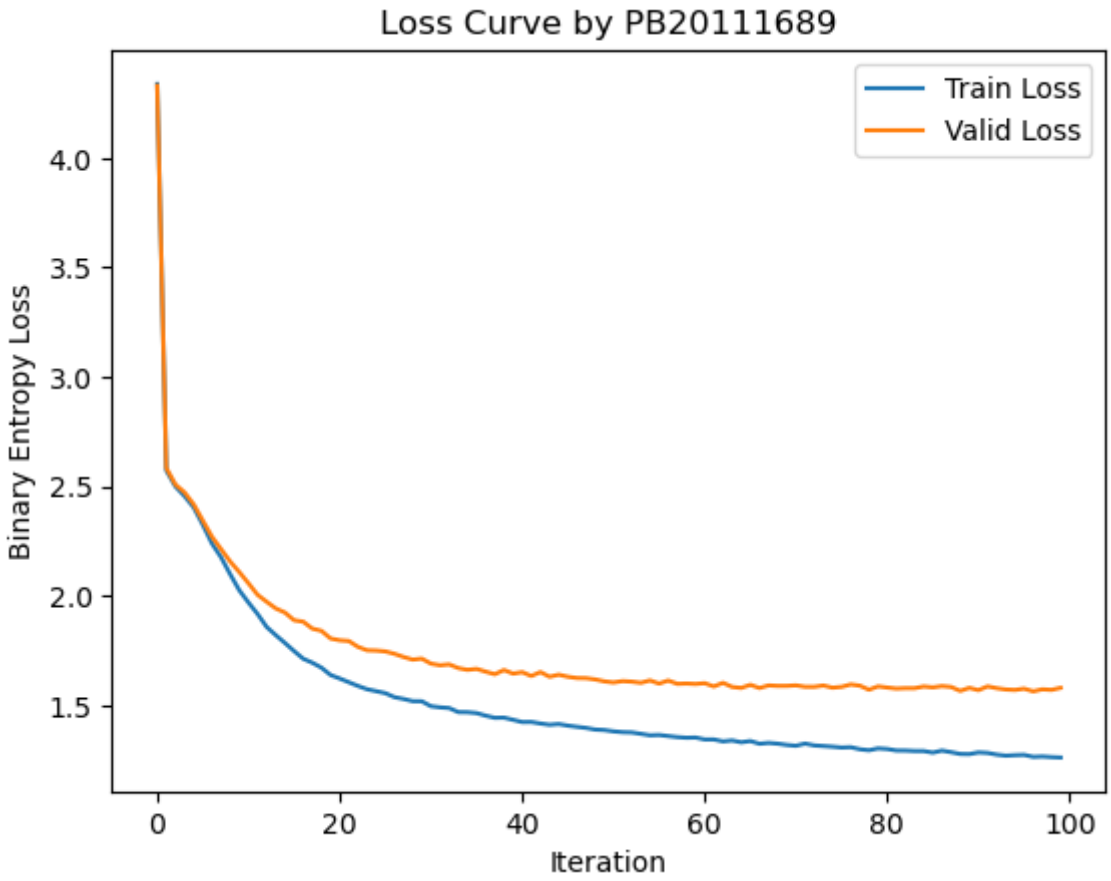
`generate(self, inputs, max_new_tokens)`：在 `generate` 函数中，我们实现了使用 transformer 模型生成新标记的逻辑。给定输入 `inputs` 作为上下文，我们通过迭代地从模型预测的概率分布中采样标记，并将采样的标记拼接到输入序列中，从而生成新标记。我们重复这个过程 `max_new_tokens` 次，并返回生成的标记序列。

总结起来，这段代码定义了一个完整的 transformer 模型，包括嵌入层、多个 transformer 模块、层归一化层和线性层。通过堆叠和连接这些组件，模型可以接受输入并生成相应的输出。

3.2 实验结果

选择参数如下进行训练：

```
batch_size = 32
block_size = 256
max_iters = 5000 # set the number of training iterations as you like
eval_interval = 50
learning_rate = 1e-3
eval_iters = 200
n_embd = 96
n_heads = 8
n_layers = 6
```



首先将字符 `str1` 和 `str2` 进行编码得到 `context`，然后将其传入函数进行预测：

```
# PB20111689 蓝俊玮
str1 = "caixukun I love "
context = torch.tensor([tokenizer.encode(str1)], device=device, dtype=torch.long)
generate(model, context=context, max_new_tokens=128)
```

✓ 1.6s

```
caixukun I love him Edward?

VOLUMNIA:
I hope, that we do
Than this ance doth have I on here with thee!
This an other Earl shut the king blunty,
```

```
# PB20111689 蓝俊玮
str2 = "Hefei is a great "
context = torch.tensor([tokenizer.encode(str2)], device=device, dtype=torch.long)
generate(model, context=context, max_new_tokens=128)
```

[18] ✓ 2.2s

```
Hefei is a great think,
Such by that, to foreign out the poor drink.
The indeed-golder of thy lost house is king,
And that shall bear this this b
```