

# 人工智能基础 Lab2

PB20000180 刘良宇

## Part 1

---

### 1.1 贝叶斯

#### 实验原理

首先统计每个标签的出现概率作为先验概率和当某个标签出现时，每个像素是黑色还是白色的概率作为条件概率。

在对新的手写数字图片分类时，根据贝叶斯公式计算每个标签的后验概率，并选择具有最大后验概率的标签作为预测结果。

```
# fit the model with training data
def fit(self, pixels, labels):
    """
    pixels: (n_samples, n_pixels, )
    labels: (n_samples, )
    """
    self.labels_prior = np.bincount(labels).astype(np.float64)
    for i in range(self.n_pixels):
        np.add.at(self.pixels_cond_label[i], (pixels[:, i], labels), 1)
    self.pixels_cond_label /= self.pixels_cond_label.sum(axis=1, keepdims=True)

# predict the labels for new data
def predict(self, pixels):
    """
    pixels: (n_samples, n_pixels, )
    return labels: (n_samples, )
    """
    n_samples = len(pixels)
    labels = np.zeros(n_samples)
    for i in range(n_samples):
        label_prob = self.labels_prior.copy()
        for j in range(self.n_pixels):
            label_prob *= self.pixels_cond_label[j][pixels[i][j]]
        labels[i] = np.argmax(label_prob)
    return labels
```

- `fit` 操作分别统计先验概率和条件概率：
  - 先验概率直接 `bincount` 即可；
  - 条件概率需要进行归一化以求得。
- `predict` 操作计算后验概率即可。这里不严格要求是后验概率，因为只需要选取其中最大的即可。

## 实验结果

```
> python Bayesian-network.py
test score: 0.843800
```

## 1.2 K-means

### 实验原理

根据设定的簇数对像素进行聚类即可。距离计算可以采用范数（即像素点在三维空间中的距离）。

```
# Assign each point to the closest center
def assign_points(self, centers, points):
    """
    centers: (n_clusters, n_dims,)
    points: (n_samples, n_dims,)
    return labels: (n_samples, )
    """
    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):
        distances = np.linalg.norm(points[i] - centers, axis=1)
        labels[i] = np.argmin(distances)

    return labels
```

首先补全分配点的函数。需要把每个像素点分配到某个具体的簇，计算像素点离哪个簇中心最近即可。

```
# Update the centers based on the new assignment of points
def update_centers(self, centers, points):
    """
    centers: (n_clusters, n_dims,)
    labels: (n_samples, )
    points: (n_samples, n_dims,)
    return centers: (n_clusters, n_dims,)
    """
    # TODO: Update the centers based on the new assignment of points
    clusters = self.assign_points(centers, points)
    new_centers = np.zeros_like(centers)
    for k in range(self.k):
        new_centers[k] = points[clusters == k].mean(axis=0)

    return new_centers
```

随后更新中心。计算均值即可。

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

```

for _ in range(self.max_iter):
    new_centers = self.update_centers(centers, points)
    if np.allclose(new_centers, centers):
        break
    centers = new_centers
return centers

```

核心 `fit` 功能函数，每个迭代更新聚类中心。这里增加了收敛时 `break` 的功能。

```



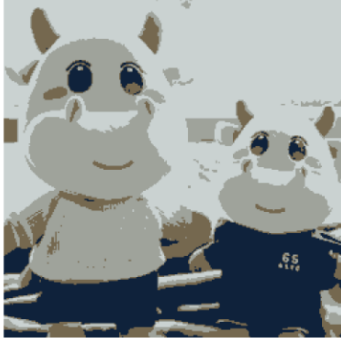



def compress(self, img):
    """
    img: (width, height, 3)
    return compressed img: (width, height, 3)
    """

    # 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)
    labels = self.assign_points(centers, points)
    for i in range(len(points)):
        points[i] = centers[int(labels[i])]
    return points.reshape(img.shape)

```

`compress` 函数，实现了图片压缩，求出聚类中心后，把每个像素点分配。

## 实验结果

Origin image	k=2	k=4
		
k=8	k=16	k=32
		

上面的图片都是完全迭代收敛后的结果，可以看出效果还是不错的。

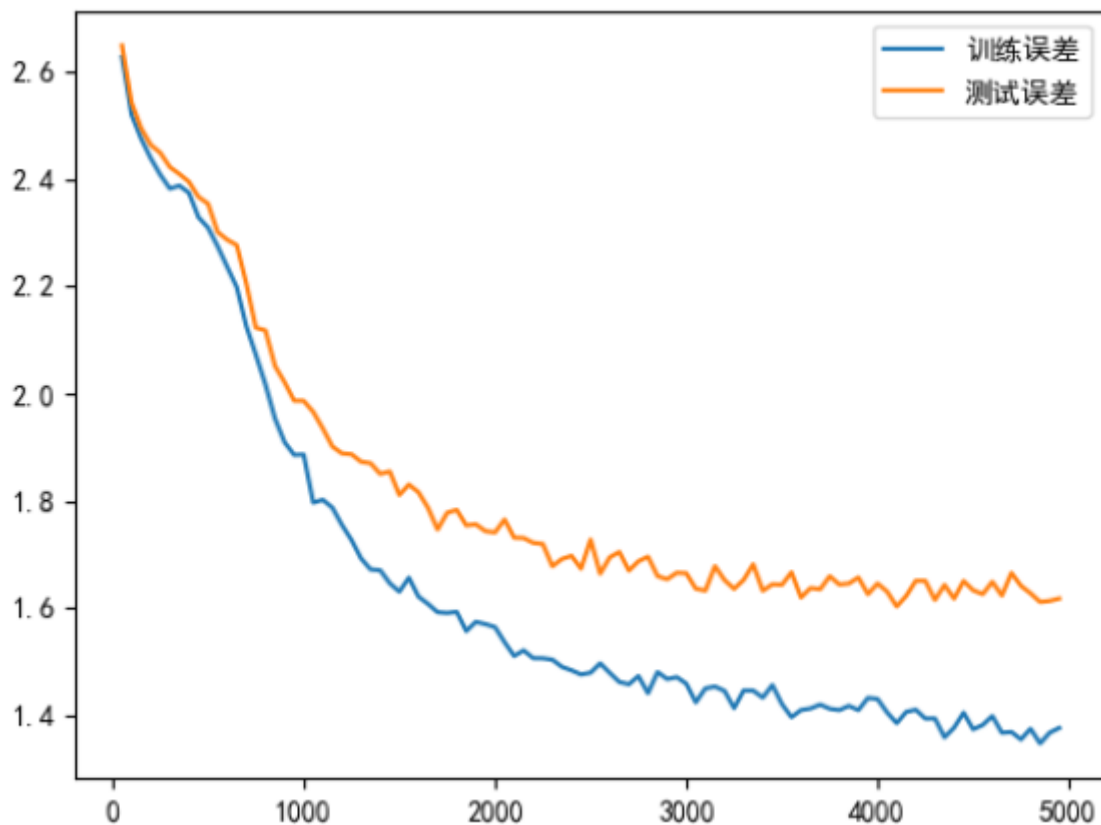
## Part2

首先给出最后结果，输入 `I have a dream that one day`，输出：

```
step 4850: train loss 1.3488, val loss 1.6122
step 4900: train loss 1.3689, val loss 1.6142
step 4950: train loss 1.3777, val loss 1.6186
I have a dream that one days, would me,
That's love me did for no coul strayw:
Twickelly against against and sometime,
Might and alter their care;
Are theit blest England from the summer;
Then art names coct: amilttle and Earfon,
You are voices, and like him wash to depate,
The prince the will therehor he pray fair
In bestred the forty. An oftend your slaughter
That you, this rusties of and hath been the ogived;
Look bloody in hearts, with art! good to her, sword;
Pay it is that swands, that he day thanks.

LEONTES:
Kill
```

误差：



## 代码原理

分词比较简单：

```
class char_tokenizer:
    """
    a very simple char-based tokenizer. the tokenizer turns a string into a list of
    integers.
    """

    def __init__(self, corpus: List[str]):
        self.corpus = corpus
        # create a dictionary that maps each character to a unique integer
        self.char_to_int = {char: i for i, char in enumerate(corpus)}

    def encode(self, string: str):
        # convert a string into a list of integers and return
        return [self.char_to_int[char] for char in string]

    def decode(self, codes: List[int]):
        # convert a list of integers into a string and return
        return "".join(self.corpus[code] for code in codes)
```

Head :

```
class Head(nn.Module):
    """single head of self-attention"""

    def __init__(self, head_size):
        super().__init__()
        self.Key = nn.Linear(head_size, head_size)
        self.Query = nn.Linear(head_size, head_size)
        self.Value = nn.Linear(head_size, head_size)
        self.register_buffer("tril", torch.tril(torch.ones(block_size, block_size)))

    def forward(self, inputs):
        keys = self.Key(inputs)
        queries = self.Query(inputs)
        values = self.Value(inputs)

        attn_scores = torch.matmul(queries, keys.transpose(-2, -1)) / torch.sqrt(
            torch.tensor(inputs.shape[-1], dtype=torch.float)
        )
        attn_probs = F.softmax(
            attn_scores.masked_fill(
                self.tril[: attn_scores.size(1), : attn_scores.size(2)] == 0,
                float("-inf"),
            ),
            dim=-1,
        )
        out = torch.matmul(attn_probs, values)

        return out
```

根据要求创建 `Key`，`Query` 和 `Value` 并计算即可。

MultiHeadAttention 和 FeedForward 都比较简单:

```
class MultiHeadAttention(nn.Module):
    def __init__(self, n_heads, head_size):
        super().__init__()
        # TODO: implement heads and projection
        self.heads = nn.ModuleList([Head(head_size) for _ in range(n_heads)])
        self.projection = nn.Linear(n_heads * head_size, head_size)
        # End of your code

    def forward(self, inputs):
        # TODO: implement the forward function of the multi-head attention
        heads_out = [head(inputs) for head in self.heads]
        out = torch.cat(heads_out, dim=-1)
        return self.projection(out)

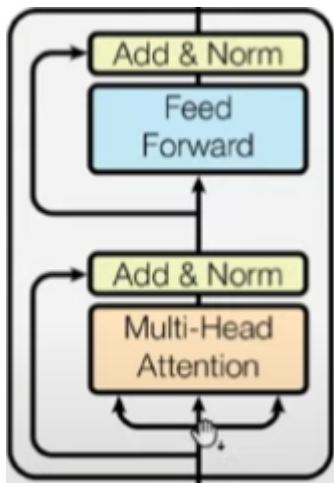
class FeedForward(nn.Module):
    def __init__(self, n_embd):
        super().__init__()
        # TODO: implement the feed-forward network
        self.net = nn.Sequential(
            nn.Linear(n_embd, 4 * n_embd),
            nn.ReLU(),
            nn.Linear(4 * n_embd, n_embd),
        )
        # End of your code

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

Block 实现 transformer block:

```
class Block(nn.Module):
    def __init__(self, n_embd, n_heads):
        super().__init__()
        self.attention = MultiHeadAttention(n_heads, n_embd)
        self.norm1 = nn.LayerNorm(n_embd)
        self.norm2 = nn.LayerNorm(n_embd)
        self.ff = FeedForward(n_embd)

    def forward(self, inputs):
        attention_out = self.attention(inputs)
        attention_out = self.norm1(attention_out + inputs)
        ff_out = self.ff(attention_out)
        out = self.norm2(ff_out + attention_out)
        return out
```



最后就可以实现 decoder 部分了。

```
class Transformer(nn.Module):
    def __init__(self):
        super().__init__()
        self.embedding = nn.Embedding(n_vocab, n_embd)
        self.blocks = nn.ModuleList([Block(n_embd, n_heads) for _ in
range(n_layers)])
        self.norm = nn.LayerNorm(n_embd)
        self.linear = nn.Linear(n_embd, n_vocab)

    def forward(self, inputs, labels=None):
        embedding = self.embedding(inputs)
        out = self.norm(embedding)
        for block in self.blocks:
            out = block(out)
        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

    def generate(self, inputs, max_new_tokens):
        for _ in range(max_new_tokens):
            logits, _ = self.forward(inputs[:, -block_size:])
            predicted_tokens = torch.multinomial(
                F.softmax(logits[:, -1], dim=-1), num_samples=1
            )
            inputs = torch.cat((inputs, predicted_tokens), dim=1)
        return inputs
```

这里训练时暂时不需要 `softmax`，推理时使用即可。

还有一个值得一提的是使用 `multinomial` 使得输出更真实，重复少。