

AI Group Project Report

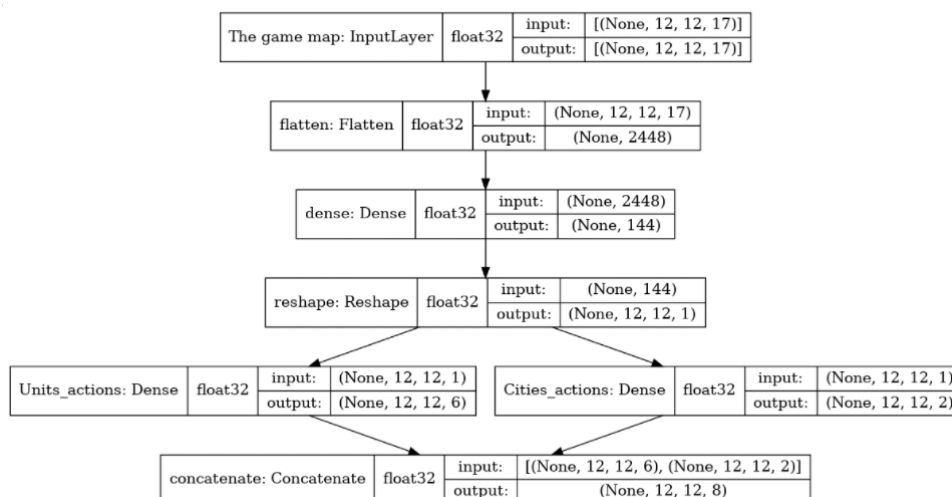
Group18: 魏龙轩 汤志豪 陈泽越 徐薪

一、Baseline比较与选择

1. Baseline1 Deep Q Network(DQN)

DQN是一种融合了神经网络和Q-Learning的强化学习方法。由于随着问题愈发复杂，状态愈发繁多，传统表格形式的强化学习陷入瓶颈，我们无法存储每个状态下每个行为的Q值。所以DQN引入神经网络，通过神经网络生成Q值或是直接输出所有的动作值，然后按照Q-Learning的原则直接选择拥有最大值的动作。

在本问题中，Baseline1使用Keras搭建DQN网络，在输入层将地图的状态信息构造成(12, 12, 17)维度的形式输入，接着如下图层次构造网络以实现由状态做出该游戏中各单位的行为选择和方向



接着做Q-Learning，Q值生成函数如下，模型训练完毕后可以得到一个比较好的结果。

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \underbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)}_{\text{new value (temporal difference target)}}$$

我们小组在学习了baseline1后发觉对初学者而言实操DQN方法难度不小，并且网络结构较为复杂，想较短时间内掌握并进行修改优化成效可能不会特别明显，故与baseline3模仿学习相比我们选择了后者作为研究方向。

2. Baseline2 Working Title Bot

Baseline2基于规则对游戏单位进行任务规划和动作决策。顶层逻辑如下：城市首先根据研究水平和是否可以建立更多的单位做出行动，接着做任务规划，每个任务都有一个目标位置和可选的行动，其受游戏规则制约以及启发函数启发，最后是行动选择，倘若单元已在目标位置则执行目标行动，行动选择依据作者制定的破坏性策略和鼓励棋盘探索。总之，baseline2通过逻辑与算法像人类玩这个游戏一样制定策略，表现也较为不错。

我们小组学习了该baseline后，综合考量在其上做修改的空间不算很多，并且我们小组更偏向于结合课程内容尝试机器学习方向的工作，所以并没有选择baseline2的方向。

3. Baseline3 Imitation Learning

Baseline3选择模仿学习作为解决该游戏问题的方法。模仿学习是指从提供的范例中学习，一般提供人类专家的决策数据，每个决策包含状态和动作序列，将所有状态-动作对抽取出来构造新的集合，之后就可以把状态作为特征，动作作为标签进行分类或回归的学习而得到最优的策略模型。

在该游戏问题中，baseline3将排名第一团队的获胜的对局记录作为模仿的对象，将地图的状态处理为(20, 32, 32)的向量，标签为向北走、向南走、向西走、向东走和建造城市五种。由于输入的地图状态信息类似图像信息，展开为向量可能会丢失空间信息，大量的参数对训练的效率和拟合程度也有不好的影响，故选择卷积神经网络(CNN)进行模仿学习，20个特征首先转化为32个通道和一个卷积层，其后是12层用3×3过滤器的卷积层，跳过连接和批标准化(Batch Normalization)，在整个层中棋盘的尺寸保持不变，用全连接层在单位和城市的坐标上对特征进行预测，可能的行动为上述五个动作，即四个方向行走和建造城市。在数据集处理上，将每轮数据集按测试集10%的比重划分训练集和测试集。模型训练完毕后，即使以较小规模的数据集也能得到一个不错的结果。

我们小组在学习了baseline3后，对模仿学习产生了钻研的兴趣，同时考虑到模仿学习样例代码的可读性较高，比较适合初学者学习实践，以及在baseline3的基础上修改优化的空间很大，可以修改网络的架构、数据集的处理、参数的调教等等。所以我们最终决定选择baseline3模仿学习作为我们的基础以及研究方向，在其上进行修改与优化。

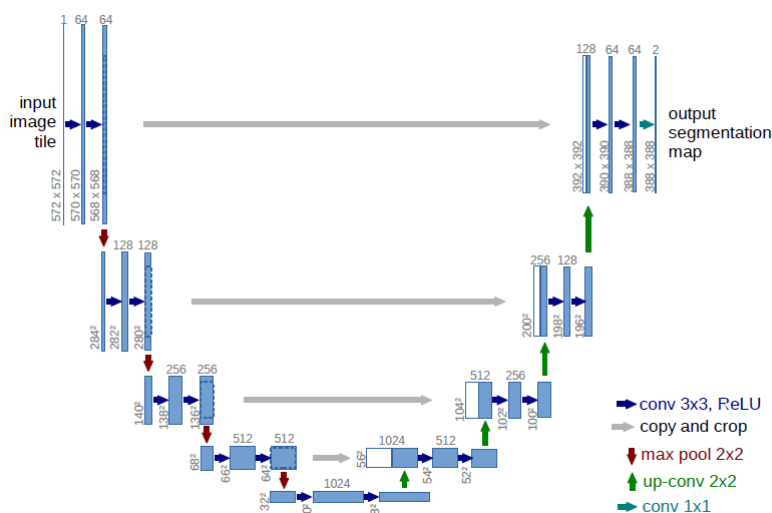
二、模型方法与代码实现

1. U-Net模型实现

通过对baseline3的苦心研究与学习后，我们认为baseline3的神经网络需要进行大幅度的改进。我们便开始在网络上搜寻较为复杂且能力较强的神经网络。在学习部分神经网络的开源代码后，我们尝试使用其中的U-Net进行神经网络的搭建。

该结构在全卷积网络（FCN）上进行修改，使其能够在很少的数据上得到很高的准确率。全卷积神经网络的主要思想是通过连续的层实现压缩网络(contracting network)，pooling层都被替换为上采样层。这些层增大了输出的分辨率。为了定位，来自压缩路径的高分辨率的特征与上采样的输出结合，连续的卷积层就可以学会基于该信息形成更准确的输出。

该结构对FCN很重要的一个改进是，在上采样部分，也有很多特征通道，使得网络可以将情境信息传到更高分辨率的层。那么expansive path或多或少与contracting path保持对称，形成U形结构，如下图所示。每个蓝色框对应一个多通道特征图（map），其中通道数在框顶标，x-y的大小位于框的左下角。白色框表示复制的特征图。箭头表示不同的操作。FCN在上采样时，根据前一池化层h和上采样层的结合实现像素的密集预测。而U-Net也是在上采样部分（扩展路径）结合下采样部分（收缩路径）生成特征向量。



首先我们实现U-Net中的基础卷积单元：卷积层+正则化层+激活函数的双层结构。

```

class DoubleConv(nn.Module):
    """(convolution => [BN] => ReLU) * 2"""

    def __init__(self, in_channels, out_channels, mid_channels=None):
        super().__init__()
        if not mid_channels:
            mid_channels = out_channels
        self.double_conv = nn.Sequential(
            nn.Conv2d(in_channels, mid_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(mid_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(mid_channels, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        return self.double_conv(x)

```

接着，U-Net中的特征提取我们卷积层间加入池化层来实现。

```

class Down(nn.Module):
    """Downscaling with maxpool then double conv"""

    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.maxpool_conv = nn.Sequential(
            nn.MaxPool2d(2),
            DoubleConv(in_channels, out_channels)
        )

    def forward(self, x):
        return self.maxpool_conv(x)

```

然后，在上采样层中我们需要实现U-Net中的转置卷积和长连接。

```

class Up(nn.Module):
    """Upscaling then double conv"""

    def __init__(self, in_channels, out_channels, bilinear=True):
        super().__init__()

        if bilinear:
            self.up = nn.Upsample(scale_factor=2, mode='bilinear',
align_corners=True)
            self.conv = DoubleConv(in_channels, out_channels, in_channels // 2)
        else:
            self.up = nn.ConvTranspose2d(in_channels, in_channels // 2,
kernel_size=2, stride=2)
            self.conv = DoubleConv(in_channels, out_channels)

    def forward(self, x1, x2):
        x1 = self.up(x1)
        # input is CHW
        diffY = x2.size()[2] - x1.size()[2]
        diffX = x2.size()[3] - x1.size()[3]

```

```

x1 = F.pad(x1, [diffx // 2, diffx - diffx // 2,
               diffy // 2, diffy - diffy // 2])
x = torch.cat([x2, x1], dim=1)
return self.conv(x)

```

最后我们设置好各层的参数，完成U-Net的整体构建。

```

class LuxNet(nn.Module):
    def __init__(self, n_channels, n_classes, bilinear=True):
        super(LuxNet, self).__init__()
        self.n_channels = n_channels
        self.n_classes = n_classes
        self.bilinear = bilinear

        self.inc = DoubleConv(n_channels, 64)
        self.down1 = Down(64, 128)
        self.down2 = Down(128, 256)
        self.down3 = Down(256, 512)
        factor = 2 if bilinear else 1
        self.down4 = Down(512, 1024 // factor)
        self.up1 = Up(1024, 512 // factor, bilinear)
        self.up2 = Up(512, 256 // factor, bilinear)
        self.up3 = Up(256, 128 // factor, bilinear)
        self.up4 = Up(128, 64, bilinear)
        self.outc = OutConv(64, n_classes)
        self.head_p = nn.Linear(n_classes, 3, bias=False)

    def forward(self, x):
        x1 = self.inc(x)
        x2 = self.down1(x1)
        x3 = self.down2(x2)
        x4 = self.down3(x3)
        x5 = self.down4(x4)
        x = self.up1(x5, x4)
        x = self.up2(x, x3)
        x = self.up3(x, x2)
        x = self.up4(x, x1)
        h = self.outc(x)
        h_head = (h * x[:, :1]).view(h.size(0), h.size(1), -1).sum(-1)
        p = self.head_p(h_head)
        return p

```

在最后的输出中，为了与原有的action集相匹配，我们添加了降维的操作使得能正确计算交叉熵损失函数。

除了神经网络的改进，我们还增大了训练的数据集。我们在[Simulations Episode Scraper Match Downloader](#)中获取到高分agent的对局记录，并从中精心挑选600场组成我们所用的数据集。

其次，我们还对部分超参数进行了尝试和调整。在对学习率、损失函数、优化函数等的改动与试错后，我们最终动态调整了学习率，使其随着学习过程不断下降。

代码详见[lux-ai-with-imitation-learning.ipynb](#)。Unet相关代码的编写参考了以下文档<https://github.com/milesial/Pytorch-UNet/tree/master/unet>。

2. Unet改进

(1) Map旋转

在上一部分中，我们的Unet网络的输出维度为5。

```
self.head_p = nn.Linear(n_classes, 5, bias=False)
```

这5个维度的输出对应了东南西北四种方向的移动与建造成市（buildcity）动作，其实我们可以将该输出简化以达到更好的训练效果。

由于map的生成是随机的，四个方向上的移动其实本质上相同，我们并不需要一次性计算4个维度，而可以通过旋转map的方式计算1个维度4次。

这里我们将输出改为3维，分别对应原地不动（center）、移动和建造城市（buildcity）。

当计算输出时，先后将map旋转0度、90度、180度、270度，并生成4个3维的张量，一共为12维。

```
policies = [model(states),
             model(torch.rot90(states,1,(2,3))),
             model(torch.rot90(states,2,(2,3))),
             model(torch.rot90(states,3,(2,3)))]
```

为了与6维的actions（c、n、s、w、e、bcity）间计算loss，我们需要将这4个张量整合为一个6维的张量。这一点不难办到，观察它们的对应关系，只需要对4个3维张量中的center和buildcity维度取平均，并将它们的move维度分别对应到东西南北并拼接，即可得到一个6维张量。

```
centers = []
buildcities = []

for p in policies:
    centers.append(p[:,0])
    buildcities.append(p[:,2])

center = (centers[0]+centers[1]+centers[2]+centers[3])*0.25
buildcity = (buildcities[0]+buildcities[1]+buildcities[2]+buildcities[3])*0.25

policy = torch.stack((center,policies[0][:,1],policies[2][:,1],policies[1][:,1],policies[3][:,1],buildcity), 1)
```

通过这样做，我们成功地降低了神经网络的输出维度。

在测试该模型的效果时，我们发现虽然神经网络的输出维度被降低，理论上能够达到更好的训练效果，但由于每次行动需要神经网络输出的次数由1次变为了4次，超时的概率大大增加，经常出现本来已经占据很大优势，却因超时而导致对战失败的情况。

代码详见Unet with Map Rotation.ipynb.

(2) Global提取

在上一部分中，我们的Unet网络的输入维度为20，这是因为make_input函数输出了一个20维的numpy数组。

这里我们的思想是提取出这个数组中一些global的东西，例如研究点数（Research Points）、日夜循环（Day/Night Cycle）和轮数（Turns），从而让该输入数组的特征化更明显，以求达到更好的训练效果。因此，我们将make_input的输出改为一个降维后只有16维的数组b和一个4×4×4的global数组。原20维b数组中的4维赋给了global数组。

```
elif input_identifier == 'rp':
    # Research Points
    team = int(strs[1])
    rp = int(strs[2])
    global_features[(team - obs['player']) % 2, :] = min(rp, 200) / 200
```

```
global_features[2, :] = obs['step'] % 40 / 40
global_features[3, :] = obs['step'] / 360
```

此外，在Unet模型中也要略微改动，即需要将两个数组连接，b数组降维至4维后即可与global数组连接。

```
x1 = self.inc(x)
x2 = self.down1(x1)
x3 = self.down2(x2)
x4 = self.down3(x3)

x = torch.cat((x4, x_features), 1)
```

在测试该模型的效果时，我们发现提取global的效果不太明显，甚至有时反而会降低分数，故而摒弃了这种想法。

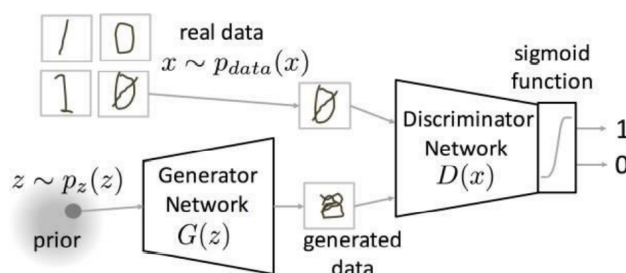
代码详见Unet With Global Extracted.ipynb.

三、其他模型尝试

1. GAN

(1) 模型介绍

GAN包含有两个模型，一个是生成模型 (generative model)，一个是判别模型(discriminative model)。生成模型的任务是生成看起来自然真实的、和原始数据相似的实例。判别模型的任务是判断给定的实例看起来是自然真实的还是人为伪造的。模型的流程图如下所示：



GAN的目标函数为：

$$\min_G \max_D V(D, G)$$

$$V(D, G) = E_{x \sim p_{data}(x)} [\log D(x)] + E_{z \sim p_z(z)} [\log (1 - D(G(z)))]$$

我们小组的思路是通过GAN模型学习第一名的agent的行为思路而单纯地对其进行模仿。换言之，我们并不把第一名的agent的行为看作是标准答案，而是学习它的思考方式，并且期望我们自己的GAN模型生成的agent超过它。

对于输入，我们参考了baseline 3的input，并且新增了action作为input。Generator由两个部分构成：一是分类器，用于得到action；二是特征生成器，其输出是Discriminator的输入。Discriminator根据特征生成器生成的特征进行真伪判别。

但是GAN模型比较难训练，它要求Generator和Discriminator之间需要很好地同步和相当的实力，否则模型可能会出现无法达到纳什平衡从而无法收敛的情况。在实际训练中，我们的模型的确出现了无法收敛的情况，经改进无果后最终放弃了该方案。

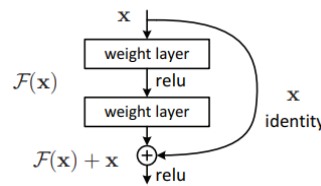
(2) 代码实现

代码详见文件：*Lux_GAN_Model.ipynb*。

2. Resnet

(1) 模型介绍

Resnet模型是为了解决神经网络的准确度随着层数增加不增还减的问题而提出的。究其原因，是由于深层神经网络难以实现恒等变换。于是Resnet创造性地引入了快捷连接分支，将求解恒等变换转化为求解残差映射函数。其基本结构如下所示：



我们小组沿用了Deep Residual Learning for Image Recognition论文中Resnet网络的结构（如下图所示），并在测试阶段取得了不错的效果。但是由于提交阶段的超时问题，该模型并没有提交成功。






layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 8$
conv4_x	14×14	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 36$
conv5_x	7×7	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10 ⁹	3.6×10 ⁹	3.8×10 ⁹	7.6×10 ⁹	11.3×10 ⁹

(2) 代码实现

代码详见文件：*Lux_Resnet_Model.ipynb*。代码参考了pytorch的官方文档和<https://github.com/WZMIAO/MIAO/deep-learning-for-image-processing>。

四、排名情况

本小组最终分数为1403.2，在Leaderboard中排名为128，提交模型为Unet（未使用Map旋转与Global提取）。

128	—	Weilong Ltd.	    	1403.2	17		20d
-----	---	--------------	--	--------	----	---	-----

五、小组分工

魏龙轩：Unet代码编写调试，尝试了Map旋转和Global提取。

汤志豪：U-Net的模型实现，数据集的获取与处理，超参数调整。

陈泽越：baseline分析比较，参与Map旋转改进尝试，代码调试。

徐薪：GAN模型，Resnet模型代码的测试与调试。

实验报告和展示PPT由本组成员共同完成。