中国人民大学
RENMIN UNIVERSITY OF CHINA

课 程 论 文

course thesis

论文题目：基于机器学习的倒立摆研究

（英 文）：**Research on CartPole Based on Machine Learning**

作　　者：郑子浩 2017202117

指导教师：胡鹤

2019　年　12　月　27　日

目录

【摘要】

本实验主要研究对象为 open AI gym 内的倒立摆（CartPole），研究方法主要包括机器学习的基本方法。实验过程采用了 24 种机器学习方法，包括回归分析、分类与聚类、集成学习和随机森林、人工神经网络、深度学习、强化学习等类别，最后发现标准线性回归、多项式核、DQN 处于明显优势。而进一步研究后，我们又得出：因为原始空间线性可分性显著，所以回归分析和 SVM 普遍很好；而损失函数过弱或过强，导致套索回归和 SVM 回归效果不佳；集成学习、深度学习受数据开销影响，TensorFlow 与自适应提升法可以很好地优化它们；DQN 是综合指标最好的方法。

【关键词】机器学习、深度学习、神经网络、倒立摆

【正文】

一、理论基础

机器学习的定义：就是使计算机在没有特定程序的情况下拥有学习的能力。

机器学习系统的类型主要有：

1.是否需要人工监督进行训练；

2.是否能递增式学习；

3.是基于比较新数据与旧数据的相似度来学习还是基于数据的分布模式来学习。

这些类型不是互相排斥的，他们可以结合使用，比如一个垃圾邮件过滤器，它是在线递增学习也是监督学习。

根据训练过程中的监督类型可以分为四大类：监督学习（supervised learning），非监督学习（unsupervised learning），半监督学习（semisupervised learning）和强化学习（reinforcement learning）。

对于监督学习来说，喂给算法的训练数据里应该包括每条数据的标签即该条数据的期望结果。

非监督学习，跟监督学习相反，训练数据里没有标签。很多降维算法都是非监督算法。

半监督学习，训练数据有一部分是有标签的，大多数时候只有一小部分有标签，大部分没有标签。大部分的半监督学习是非监督学习和监督学习的结合。

强化学习非常不同，他能通过观察环境选择行为，并且将获得的反馈（正向反馈 or 负向反馈）返回给系统，在这个循环过程中，他自己能学习到最合适的策略（能获得最好的正向反馈的策略）。

根据是否是递增式的从一串输入数据中不断学习来分类，可以分为批量学习（batch learning）和在线学习（online learning）两类。

批量学习必须一次性用所有数据来训练，比较费时也比较占用资源。

在线学习可以把数据分成多个小部分来训练，这些小部分数据通常我们叫 mini-batches；学习率（learning rate）决定了学习系统应该以怎样的速度去适应新数据，如果学习率很高，那么学习系统将很快的去适应新数据并很快忘记旧数据，相反如果学习率很低，那么学习系统会有很大惯性，它的学习速度会比较慢，但是它有个好处是它对于新数据中的噪音也不会那么敏感。

根据机器学习系统如何泛化来分类，可分为基于实例的学习（instance-based learning）和基于模型的学习（model-based learning）两类。

基于实例的学习指的是系统把所有的实例记住，通过计算新的例子与所记住的旧的例子的相似性来泛化预测新的例子的标签。

基于模型的学习是通过一系列的例子来构建一个模型，然后用这个模型对新数据做预

测。

机器学习的主要的挑战是两点：糟糕的算法，或者糟糕的数据。

从数据角度来看主要有：训练数据不够充足；训练数据不具有代表性；数据质量差；不相关的特征。

从算法角度来看就是两点：训练过拟合；训练欠拟合。

二、实验过程与结果

2.1 基本思路

总共使用了 24 种方法，具体如下：

类别	方法
回归分析	标准线性回归（Linear Regression） 多项式回归（Polynomial Regression） 正则线性模型： 　岭回归（Ridge Regression） 　套索回归（Lasso Regression） 　弹性网络（Elastic Net） 逻辑回归（Logistic Regression） Softmax 回归（Softmax Regression）
分类与聚类	线性 SVM 分类（Linear SVM Classification） 非线性 SVM 分类 　原始 SVM 　多项式核（Polynomial Kernel） 　高斯 RBF 核函数（Gaussian RBF Kernel） SVM 回归（SVM regression）
集成学习和随机森林	bagging 和 pasting（Bagging and Pasting） 随机森林（Random Forests) 提升法 　自适应提升（Adaptive Boosting） 　梯度提升（Gradient Boosting）
人工神经网络	使用 API（Using the Estimator API） Plain TensorFlow（Training a DNN Using Plain TensorFlow） dense（Using dense() instead of neuron_layer()）
深度学习	BN（Batch Normalization） Adam Optimizer（Faster Optimizers：Adam） L1 和 l2 正则（Avoiding Overfitting Through Regularization）
Policy Gradient	策略梯度（Policy Gradient）
DQN	DQN

2.2 数据采集

训练集来自每个得分达到 200 分的回合的每个步骤，它包含的变量有：
动作（action）：左移（0）；右移（1）
状态变量（state variables）：

x ：小车在轨道上的位置（position of the cart on the track）

θ：杆子与竖直方向的夹角（angle of the pole with the vertical）

- $\dot{x}$：小车速度（cart velocity）
- $\dot{\theta}$：角度变化率（rate of change of the angle）

原始数据集大约 2000 条。

2.3 模型构建

2.3.1 回归分析

2.3.1.1 标准线性回归(Linear Regression)

线性回归模型预测

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

使用向量化的形式可改写为

$$\hat{y} = h_\theta(X) = \theta^T \cdot X$$

线性回归模型损失函数（cost function）

$$MSE(X, h_\theta) = \frac{1}{m}(\theta^T \cdot X^{(i)} - y^{(i)})^2$$

通过求偏导，求极值点，可得到

$$\hat{\theta} = (X^T \cdot X)^{-1} \cdot X^T \cdot Y$$

$\hat{\theta}$是使得损失函数最小的θ

Y 是目标值向量，包含 $y^{(1)}$ 到 $y^{(m)}$

2.3.1.2 多项式回归（Polynomial Regression）

用线性回归模型拟合非线性数据，增加每个属性的次方作为新的属性。这被称作多项式回归。将 n 个特征扩充为$\frac{(n+d)!}{d!n!}$个特征。 其实就是 n 元 d 次完全多项式的项数。

2.3.1.3 岭回归（Ridge Regression）

$$J(\theta) = MSE(\theta) + \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$$

对损失函数增加正则项，降低模型复杂度，降低过拟合的风险。其中 α 是个超参数。

需要说明的是，训练模型时增加正则项，但是用测试集评估模型时，就不能使用正则项了。模型训练和模型评估使用不同的损失函数是很常见的。除了正则化，另一个原因是训练时的损失函数需要便于求导，但是在测试集评估时就要与目标值尽可能接近。例如，逻辑回归在训练时使用 log loss 作为损失函数，但是评估时使用精度/召回率。θ_0 不进行正则化。

训练之前，统一数据不同属性的取值范围很重要（比如使用 StandardScaler），绝大多数的正则化模型都需要这么做。

岭回归解析解：

$$\hat{\theta} = (X^T \cdot X + \alpha A)^{-1} \cdot X^T \cdot Y$$

其中，A 是个近似的 n×n 的单位矩阵，只是左上角是个 0。其实正则化带来了一个额外的好处，那就是矩阵$(X^T \cdot X + \alpha A)$一定可逆，从而解析解一定存在。

2.3.1.4 套索回归（Lasso Regression）

损失函数：

$$J(\theta) = MSE(\theta) + \alpha \sum_{i=1}^{n} |\theta_i|$$

Lasso 回归倾向于去除掉最不重要的特征（也就是将其权重设置为 0）。

2.3.1.5 弹性网络（Elastic Net）

Elastic Net 介于 Ridge 回归和 Lasso 回归之间，损失函数如下：

$$J(\theta) = MSE(\theta) + \gamma\alpha \sum_{i=1}^{n} |\theta_i| + \frac{1-\gamma}{2} \alpha \sum_{i=1}^{n} \theta_i^2$$

2.3.1.6 逻辑回归（Logistic Regression）

和线性回归类似，逻辑回归也是计算输入特征的加权和（再加上偏置项），但与前者直接输出计算结果不同，后者返回计算结果的逻辑（logistic）。

逻辑回归模型评估概率（向量化形式）：

$$\hat{p} = h_\theta(X) = \sigma(\theta^T \cdot X)$$

logistic,也被称作 logit，记做 σ(•)，是一个 sigmoid 函数（比如 S 形函数）。

$$\sigma(t) = \frac{1}{1+e^{-t}}$$

$$\hat{y} = \begin{cases} 0 \; if \; \hat{p} < 0.5 \\ 1 \; if \; \hat{p} \geq 0.5 \end{cases}$$

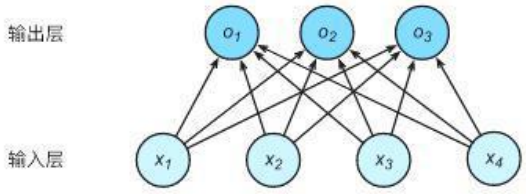
逻辑回归损失函数（log loss）：

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\{y^{(i)}log(\hat{p}^{(i)}) + (1-y^{(i)})log(1-\hat{p}^{(i)})\}$$

2.3.1.7Softmax 回归（Softmax Regression）

与线性回归的不同在于 Softmax 回归的输出单元从一个变成了多个，同时引入 Softmax 运算使得输出更加适合离散值的预测和训练；

Softmax 回归模型：



与线性回归相同，都是将输入特征与权重做线性叠加，，其输出层也是一个全连接层。

与线性回归的最大不同在于：Softmax 回归的输出值个数等于标签中的类别数。

2.3.2 分类与聚类

2.3.2.1 线性 SVM 分类（Linear SVM Classification）

SVM 的解决问题的思路是找到离超平面的最近点，通过其约束条件求出最优解。

支持向量满足函数：

$$\omega^T x + b = \pm 1$$

支持向量点到超平面的距离：

$$\frac{\left|y(\omega^T x + b)\right|}{\|\omega\|_2} = \frac{1}{\|\omega\|_2}$$

我们解题的思路是：让所有分类的点各自在支持向量的两边，同时要求尽量使得支持向量远离超平面，优化问题可以用数学公式可以表示如下：

$$\max_{\omega,b}\frac{1}{\|\omega\|_2}$$

$$s.t: y^{(i)}(\omega^T * x^{(i)} + b) \geq 1, i=1,2,...,m$$

以上优化问题可以转化为：

可以转化为求损失函数 J(w）的最小值，如下表示：

$$J(\omega) = \frac{1}{2}\|\omega\|_2^2$$

$$s.t: y^{(i)}(\omega^T * x^{(i)} + b) \geq 1, i=1,2,...,m$$

以上问题可以用 KKT 条件求解

$$L(\omega, b, \beta) = \frac{1}{2}\|\omega\|_2^2 + \sum_{i=1}^{m}\beta_i[1 - y^{(i)}(\omega^T * x^{(i)} + b)], \beta \geq 0$$

2.3.2.2 原始 SVM

线性 SVM 分类器是有效的，但是现实中不是所有数据集都可以线性分离的。处理非线性数据集的方法之一是添加更多特征。初步尝试搭建一条流水线：一个 Polynomical Features 转换器，接着一个 StandardScaler，然后是 linear SVC.

2.3.2.3 多项式核（Polynomial Kernel）

有些数据集本身就不是线性的，一个解决方案就是增加特征，比如多项式特征。增加多项式特征很简单，但是次数太低无法拟合复杂函数，次数太高又会增加大量的特征。幸运的是，SVM 可以使用一种被称作核技巧（kernel trick）的数学方法。它和增加很多多项式特征的表现一样，但实际上有没有增加特征。

2.3.2.4 高斯 RBF 核函数（Gaussian RBF Kernel）

与多项式核代替直接增加多项式特征相似，我们也可以使用高斯 RBF 核代替直接增加相似度特征。

2.3.2.5 SVM 回归（SVM regression）

与分类问题求得类别间的最大间隔不同，SVM 回归的目的是使得间隔里面包含最多的样本点。

2.3.3 集成学习和随机森林

如果集成一系列分类器的预测结果，也将会得到由于单个预测期的预测结果。一组预测期称为一个集合（ensemble），因此这一技术被称为集成学习（Ensemble Learning）。集成学习算法称作集成方法（Ensemble method）。

例如，可以基于训练集的不同随机子集，训练一组决策树分类器。做预测是，首先拿到每一个决策树的预测结果，得票数最多的一个类别作为最终结果，这就是随机森林。

此外，通常还可以在项目的最后使用集成方法。比如已经创建了几个不错的分类器，可以将其集成为一个更优秀的分类器。

2.3.3.1bagging 和 pasting（Bagging and Pasting）

获得不同分类器的方法是，训练算法虽然是相同的，但训练数据确是从训练集中随机选取的不同子集。如果子集的选取是有放回采样（sampling with replacement。replace 在这里是复位、归还的意思，不是代替的意思。），这一方法称为 bagging（bootstrap aggregating 的简称。在统计学上，有放回采样称为 bootstrapping）。如果是无放回采样（sampling without replacement），则称之为 pasting。

2.3.3.2 随机森林（Random Forests）

随机森林是一系列决策树的集成，一般使用 bagging 方法。

基于训练集的不同随机子集，训练一组决策树分类器。做预测是，首先拿到每一个决策树的预测结果，得票数最多的一个类别作为最终结果，这就是随机森林。

2.3.3.3 自适应提升（Adaptive Boosting）

Boosting (最初被称作 hypothesis boosting)是指所有可以联合一系类弱学习器使其成为强学习器的方法。其基本思想是循环地训练预测器，每一次都尝试更新其预测。现在有很多 Boosting 方法，最有名的是 AdaBoost（Adaptive Boosting 的简称）和 Gradient Boosting。再此选用 AdaBoost。

2.3.3.4 梯度提升（Gradient Boosting）

与 AdaBoost 类似，Gradient Boosting 也是不停地增加预测器。不同的是，Gradient Boosting 新增加的预测器，回去拟合其前任的残差（residual errors）。该算法处理回归任务表现很好，被称作 Gradient Tree Boosting 或者 Gradient Boosted Regression Trees (GBRT，梯度提升决策树)。

2.3.4 人工神经网络

2.3.4.1 使用 API（Using the Estimator API）

与 TensorFlow 一起训练 MLP 最简单的方法是使用高级 API TF.Learn，这与 sklearn 的 API 非常相似。 DNN Classifier 可以很容易训练具有任意数量隐层的深度神经网络，而 softmax 输出层输出估计的类概率.训练两个隐藏层的 DNN（一个具有 300 个神经 元，另一个具有 100 个神经元）和一个具有 10 个神经元的 SOFTMax 输出层进行分类：

2.3.4.2 Plain TensorFlow（Training a DNN Using Plain TensorFlow）

使用低级被 python API 训练 DNN。

2.3.4.3 Using dense() instead of neuron_layer()

最好使用 tf.layers.dense（），因为 contrib 模块中的任何内容可能会更改或删除。

dense（）函数与 fully_connected（）函数几乎相同，除了一些细微的差别：

几个参数被重命名：scope 变为名称，activation_fn 变为激活（同样_fn 后缀从其他参数（如 normalizer_fn）中删除），weights_initializer 成为 kernel_initializer 等。默认激活现在是无，而不是 tf.nn.relu。

2.3.5 深度学习

2.3.5.1 BN（Batch Normalization）

Sergey Ioffe 和 Christian Szegedy 在其 2015 年的论文中提出了一种被称作 Batch Normalization (BN)的解决方案。比梯度消失（爆炸）更一般的问题是，由于前层参数的变化，造成后层输入数据分布变化的问题（他们称为 Internal Covariate Shift 问题）。

该技术就是在每一层应用激活函数之前，增加一些操作。首先对数据进行简单的 zero-centering 和 normalizing（其实就是转换成均值为 0，标准差为 1 的数据），然后使用两个参数对数据进行 scaling 和 shifting（缩放和平移）操作。换句话说，这两步就是让模型学习每层输入最优的 scale（规模）和均值。

2.3.5.2 Adam Optimizer（Faster Optimizers：Adam）

训练一个非常大的深度神经网络可能会非常缓慢。 到目前为止，我们已经看到了四种加速训练的方法（并且达到更好的解决方案）：对连接权重应用良好的初始化策略，使用良好的激活函数，使用批量规范化以及重用预训练网络的部分。 另一个巨大的速度提升来自使用比普通渐变下降优化器更快的优化器。目前最流行的优化器有：动量优化，Nesterov 加速梯度，AdaGrad，RMSProp，最后是 Adam 优化。

而实际上几乎总是应该使用 Adam_optimization，所以如果不关心它是如何工作的，只需使用 Adam Optimizer 替换您的 Gradient Descent Optimizer。 只需要这么小的改动，训练通常会快几倍。

2.3.5.3 l1 和 l2 正则（Avoiding Overfitting Through Regularization）

深度神经网络通常具有数以万计的参数，有时甚至是数百万。有了这么多的参数，网络拥有难以置信的自由度，可以适应各种复杂的数据集。但是这个很大的灵活性也意味着它很容易过拟合训练集。通过神经网络正则化技术，实现 tensorflow 避免过拟合，l1 和 l2 正则化、drop out、最大范数正则化和数据增强。

可以使用 l1 和 l2 正则化约束一个神经网络的连接权重（但通常不是它的偏置）。使用 TensorFlow 做到这一点的一种方法是简单地将适当的正则化项添加到损失函数中。

2.3.6 策略梯度（Policy Gradient）

1、首先，让神经网络策略玩几次游戏，并在每一步计算梯度，这使得智能体更可能选择行为，但不应用这些梯度。

2、运行几次后，计算每个动作的得分（使用前面段落中描述的方法）。

3、如果一个动作的分数是正的，这意味着动作是好的，可应用较早计算的梯度，以便将来有更大的的概率选择这个动作。但是，如果分数是负的，这意味着动作是坏的，要应用负梯度来使得这个动作在将来采取的可能性更低。我们的方法就是简单地将每个梯度向量乘以相应的动作得分。

4、最后，计算所有得到的梯度向量的平均值，并使用它来执行梯度下降步骤。

2.3.7 DQN

DQN 与 Qleanring 类似都是基于值迭代的算法，但是在普通的 Q-learning 中，当状态和动作空间是离散且维数不高时可使用 Q-Table 储存每个状态动作对的 Q 值，而当状态和动作空间是高维连续时，使用 Q-Table 不动作空间和状态太大十分困难。

所以在此处可以把 Q-table 更新转化为一函数拟合问题，通过拟合一个函数 function 来代替 Q-table 产生 Q 值，使得相近的状态得到相近的输出动作。因此我们可以想到深度神经网络对复杂特征的提取有很好效果，所以可以将 Deep Learning 与 Reinforcement Learning 结合。这就成为了 DQN。

2.4 结果输出

24 中方法结果如下：

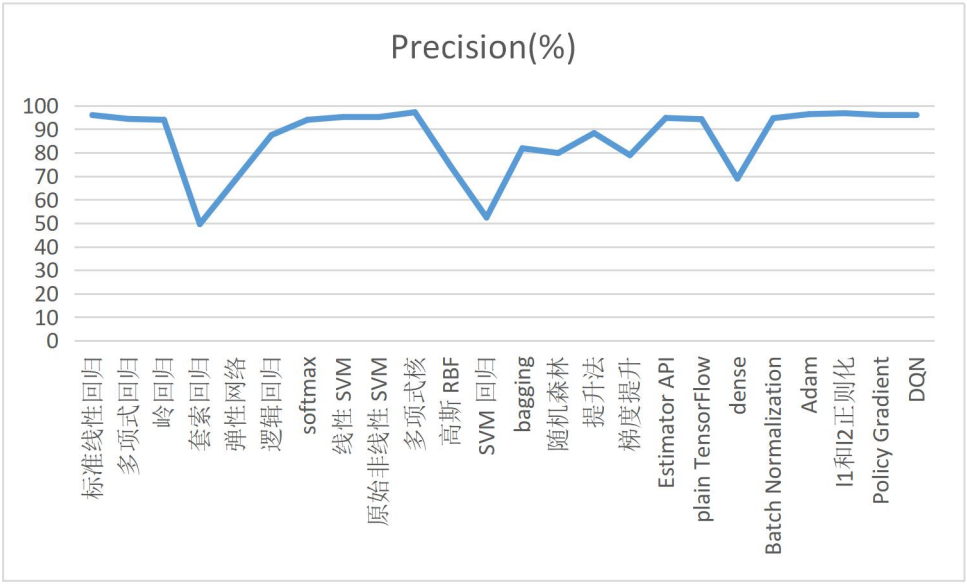
其中 time 表示该方法下，积分达到 200 分所需时间；

Reward 表示该方法下，一个回合结束时所用时间；

Precision 表示该方法的准确度。

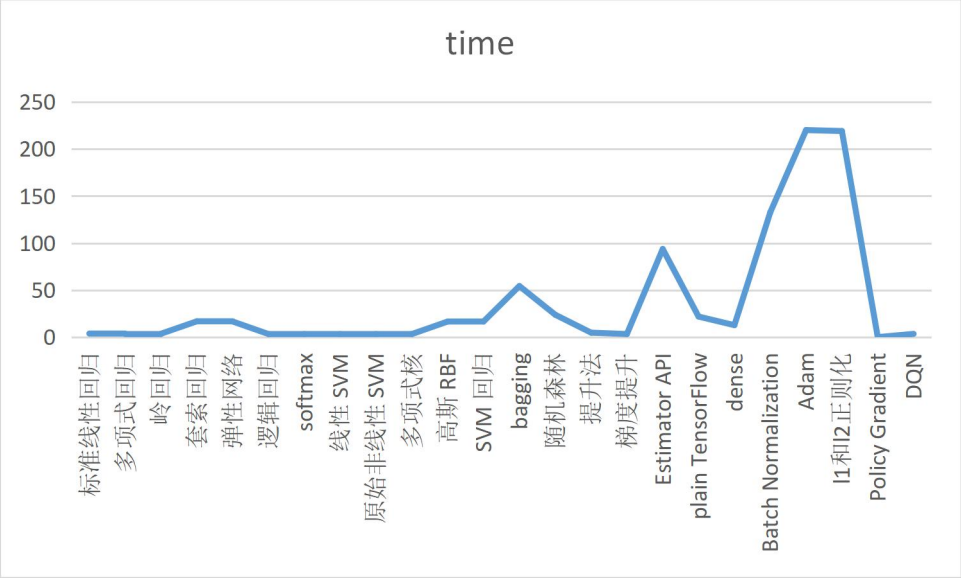
方法	Precision(%)	reward	time
标准线性回归	95.97	200	3.88
多项式回归	94.35	200	3.36
岭回归	93.95	200	3.71
套索回归	49.6	9	16.9
弹性网络	68.55	176	16.69
逻辑回归	87.5	200	3.35
softmax	93.95	200	3.38
线性 SVM	95.16	200	3.35
原始非线性 SVM	95.16	200	3.37
多项式核	97.18	200	3.36
高斯 RBF	74.19	87	16.68
SVM 回归	52.42	32	16.69
bagging	81.85	200	54.36
随机森林	79.84	200	23.89
提升法	88.31	200	4.85
梯度提升	78.93	200	3.38
Estimator API	94.75	200	93.66
plain TensorFlow	94.23	200	21.92
dense	68.95	200	12.8
Batch Normalization	94.66	18	132.56
Adam	96.37	10	219.96
l1 和 l2 正则化	96.77	10	218.96
Policy Gradient	96	114	0.12
DQN	96	200	3.55

24 种方法准确度如下：



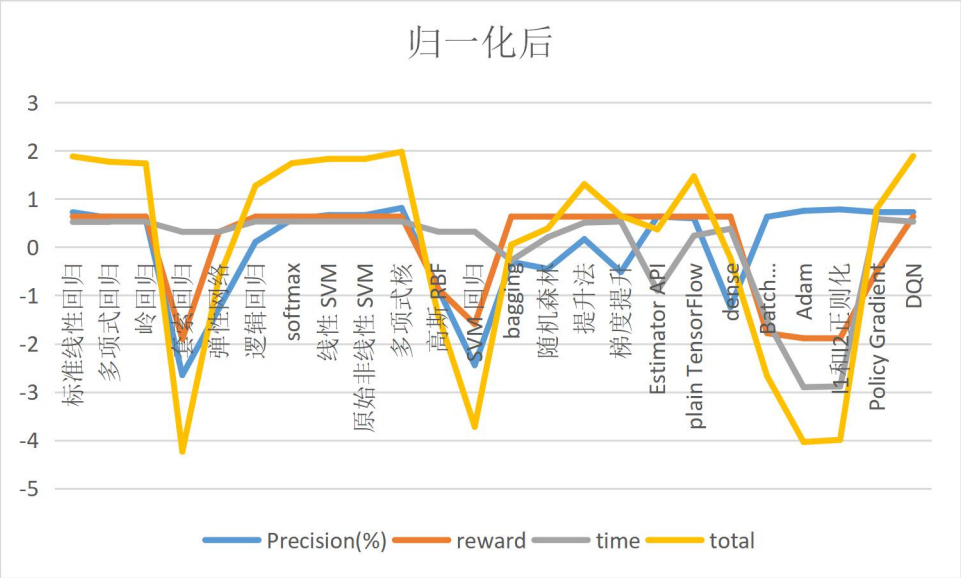
24 种方法得分情况如下：

24 种方法时间如下：



24 种方法应该综合起来，因为三种指标量纲不同，所以使用方差和均值实现标准化，其次时间越长反而性能降低，所以时间指数取反，归一化和调整后情况如下：

其中 total（黄线）代表的是三个指标的综合评分。



综合来看多项式核 SVM 是性能最好的，DQN、线性回归稍稍逊色。

三、拓展研究

在 24 种方法中，有几个方法明显处于劣势，分别是：套索回归、SVM 回归、Adam、l1 和 l2 正则化。

有几个方法处于明显的优势：标准线性回归、多项式核、DQN。

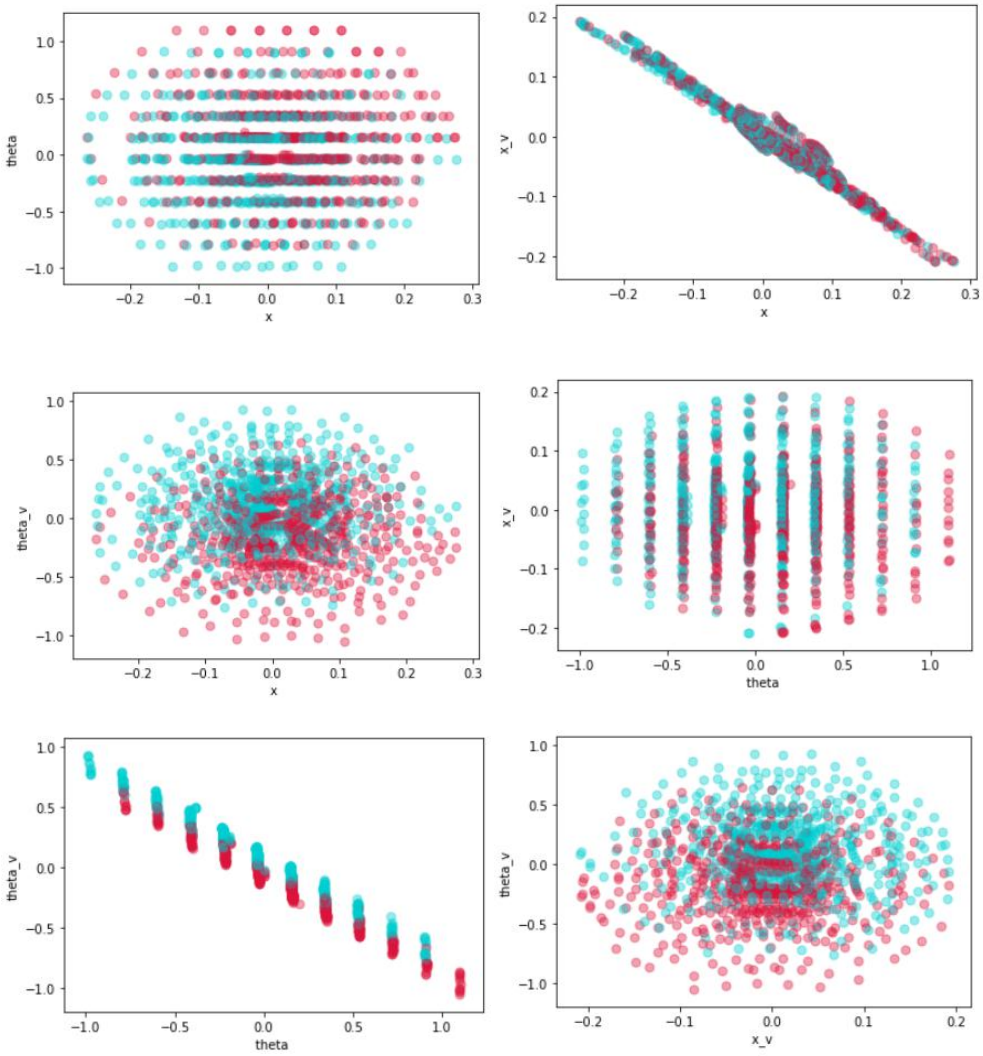
3.1 回归分析和 SVM 普遍很好

标准线性回归、多项式核、DQN 处于明显优势，其次，回归分析和运用核函数的 SVM

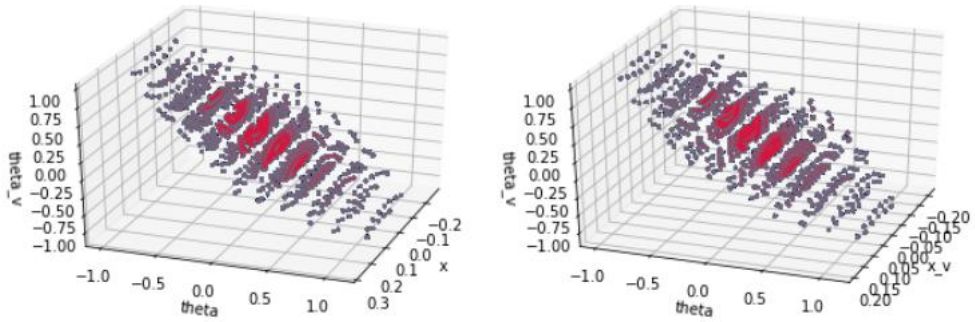
水平相当，且都效果很好。而套索回归、SVM 回归在非深度学习的机器学习方法中较为差，那么造成此现象的原因是什么呢？

我们知道核函数的目的是，低维空间线性不可分的模式通过非线性映射到高维特征空间则可能实现线性可分。由此我们猜测原始空间本身就线性可分。

原始空间的性值两两组合的投影如下：



很明显的看出 theta_v,theta 平面的投影是线性可分的。



三维下 x,theta_v,theta 和 x_v,theta_v,theta 上可分。

这就导致了其实只需要线性回归、甚至是线性 SVM 就可以达到很好的效果了。

3.2 套索回归和 SVM 回归效果不佳

套索回归、SVM 回归在非深度学习的机器学习方法中较为差，从归一化后的综合指标图中我们可以看出，这两种方法的 precision、award、time 指数都很低，所以我们猜测这是因为这两种方法的准确度过低导致整体性能偏低。

我们知道 lasso 估计是基于岭回归估计的。

岭回归估计的定义为：

$$\hat{\beta}^{ridge} = \arg\min_{\beta}\{\sum_{i=1}^{N}(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda\sum_{j=1}^{p}\beta_j^2\}$$

Lasso 估计的定义为：

$$\hat{\beta}^{ridge} = \arg\min_{\beta}\{\sum_{i=1}^{N}(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda\sum_{j=1}^{p}|\beta_j|\}$$

在建立模型之初，为了尽量减小因缺少重要自变量而出现的模型偏差，通常会选择尽可能多的自变量。然而，建模过程需要寻找对因变量最具有强解释力的自变量集合，也就是通过自变量选择(指标选择、字段选择)来提高模型的解释性和预测精度。指标选择在统计建模过程中是极其重要的问题。Lasso 算法则是一种能够实现指标集合精简的估计方法。

Lasso 回归与岭回归有一点不同，它在惩罚部分使用的是绝对值，而不是平方值。这导致惩罚（即用以约束估计的绝对值之和）值使一些参数估计结果等于零。使用的惩罚值越大，估计值会越趋近于零。这将导致我们要从给定的 n 个变量之外选择变量。

Lasso 回归有一定的局限性，譬如：

在 Lasso 回归求解路径中，对于 N×P 的设计矩阵来说，最多只能选出 min(N,p)个变量。当 p>N 的时候，最多只能选出 N 个预测变量．因此，对于 p~ N 的情况，Lasso 方法不能够很好的选出真实的模型。

如果预测变量具有群组效应，则用 Lasso 回归时，只能选出其中的一个预测变量。

对于通常的 N>P 的情形，如果预测变量中存在很强的共线性，Lasso 的预测表现受控于岭回归。

首先我们实验的情况是 N>P，而岭回归在这次实验中表现也不佳，Lasso 的预测表现受控于岭回归，所以 lasso 回归也表现不佳。而实际上原始变量具有很强的共线性（共线性概念指两个或者更多的自变量高度相关）。

另外我们尝试输出 lasso 回归的模型，得出最终模型的四个变量的系数，发现系数基本为零，由此可以看出预测变量具有群组效应（即变量之间存在高相关性），这就进一步证明了 lasso 回归并不适合本实验环境。

```
print('系数',lasso_reg.coef_)
```

```
系数 [-0. -0.  0.  0.]
测试集准确率：49.596774%
```

而 SVM 回归之所以相对于其他的 SVM 模型差的原因在于其损失函数。

我们知道回归模型的目标是让训练集中的每个样本点(x_i, y_i)，尽量拟合到一个线性模型$y_i = w^T x_i + b$上。对于一般的回归模型，我们是用均方误差作为损失函数的，但 SVM 不

是这样定义损失函数的。

SVM 回归算法采用$\varepsilon - insensitive$误差函数，该误差函数定义为，如果预测值$\hat{y}_i$与真实值$y_i$之间的差值$|\hat{y}_i - y_i| \leq \epsilon$，则不产生损失，否则，损失代价为$|\hat{y}_i - y_i| - \epsilon$。

总结一下，SVM 回归模型的损失函数度量为：

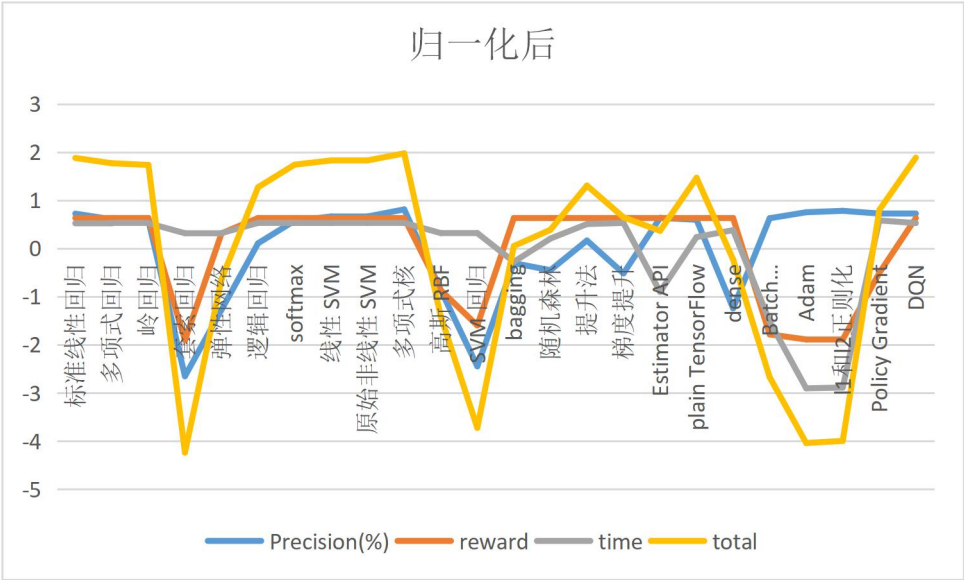
$$\hat{y}_i = w^T x_i + b$$

$$E_\epsilon(\hat{y}_i - y_i) = \begin{cases} 0, & if\ |\hat{y}_i - y_i| < \epsilon; \\ |\hat{y}_i - y_i| - \epsilon, & otherwise \end{cases}$$

该损失函数的性质是忽略低于阈值的差值，这样的方法对于数据样本庞大或者误差过大的数据集可以起到很好的收敛效果；但是对于本实验预测介于 0 和 1 两个值之间，其和真实值的差值过小，并不适合本方法。而且 SVM 回归使用的是残差而非均方误差，均方误差也是避免了过小误差的影响，而且是的误差函数在微积分中可微。

3.3 集成学习、深度学习的局限性与优化

本实验很好地体现了深度学习相对于传统机器学习的局限性。



我们可以看出集成学习、深度学习的 reward（橙色线）其实得分都很高，意味着深度学习的确可以实现很好地预测，而导致综合指数（黄色线）并不显著高于传统机器学习的在于它们的时间开销（灰色线）过低，而且 precision（蓝色线）也表现一般。

虽然深度学习中，神经网络天生具备拟合任意复杂函数的特点，可以做非常复杂的非线性映射。但本实验提供的数据集中非线性性质并不明显，虽然最终模型预测效果良好，但是得不偿失。

其次，神经网络强大的假设空间，使得神经网络极易陷入局部最优，导致模型泛化能力很差，使得模型过拟合见到的数据分布，并不能很好的预测未知的数据。

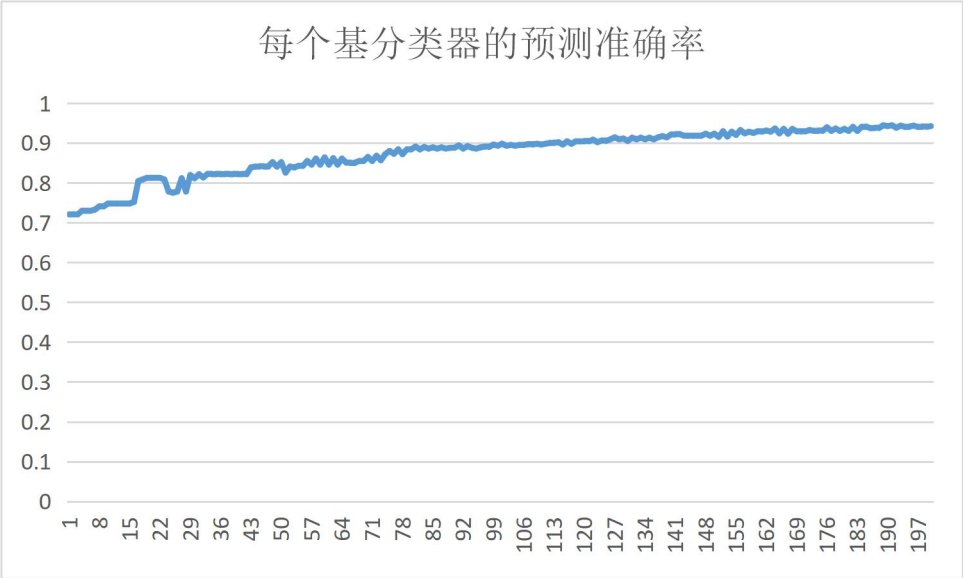
第三，神经网络参数量巨大，网络过深，会使得收敛很慢。

和其他方法相比，Tensor Flow 与自适应提升法起到了很好的优化效果。

自适应提升法属于集成学习，它基于这样一种思想：对于一个复杂任务来说，将多个专家的判断进行适当的综合所得出的判断，要比其中任何一个专家单独的判断好。

对于本实验的二分类问题而言，给定一个训练样本集，求比较粗糙的分类规则（弱分类器）要比求 精确的分类规则（强分类器）容易得多。提升方法就是从弱学习算法出发，反复学习，得到一系列弱分类器（又称为基本分类器），然后组合这些弱分类器，构成一个强分类器。本实验使用的提升方法都是改变训练数据的概率分布（训练数据的权值分布），针对不同的训练数据分布调用弱学习算法学习一系列弱分类器。

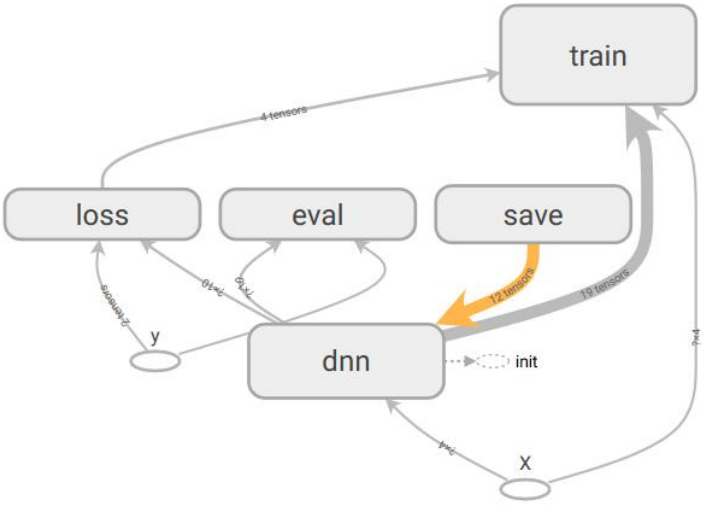
本实验自适应方法的基分类器工有 200 个，每个基分类器的预测准确率如下：

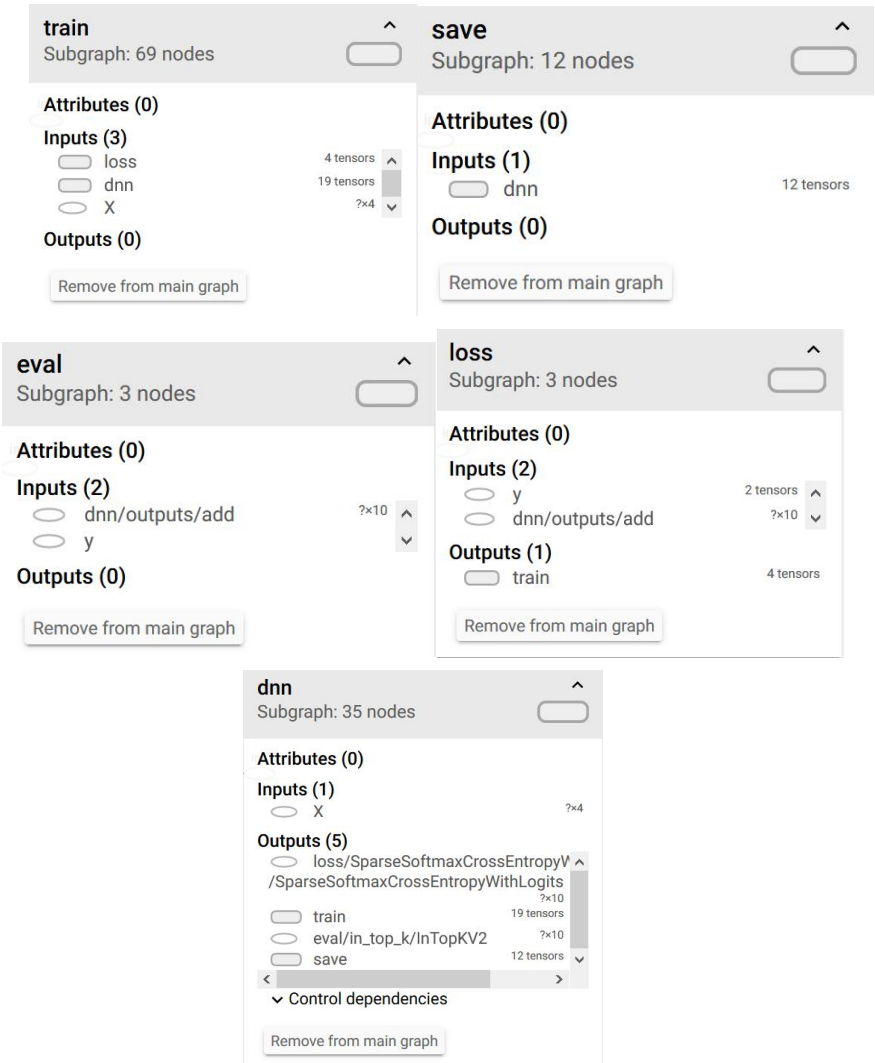


200 个分类器呈现递增的准确度，通过综合所有“专家”意见，避免了其他集成方法带来的武断性。

TensorFlow 是一个采用数据流图（data flow graphs），用于数值计算的开源软件库。与 HTML 类似，Tensorflow 是用于表示某种类型的计算抽象(称为“计算图”)的框架。当我们用 Python 操作 Tensorflow 时，我们用 Python 代码做的第一件事是组装计算图。之后我们的第二个任务就是与它进行交互(使用 Tensorflow 的“会话”)。

利用 tensor board 将模型可视化，结果如下：





3.4DQN 建模过程介绍

上文中我们知道标准线性回归、多项式核、DQN 处于明显的优势地位。

现在我们具体介绍一下 DQN。

```
class DQN():
  # DQN Agent
  def __init__(self, env): #初始化

  def create_Q_network(self): #创建Q网络

  def create_training_method(self): #创建训练方法

  def perceive(self,state,action,reward,next_state,done): #感知存储信息

  def train_Q_network(self): #训练网络

  def egreedy_action(self,state): #输出带随机的动作

  def action(self,state): #输出动作
```

主要只需要以上几个函数。上面已经注释得很清楚，这里不再加以解释。

我们知道，我们的 DQN 一个很重要的功能就是要能存储数据，然后在训练的时候 minibatch 出来。所以，我们需要构造一个存储机制。这里使用 deque 来实现。

```
self.replay_buffer = deque()
```

初始化：

这里要注意一点就是 egreedy 的 epsilon 是不断变小的，也就是随机性不断变小。怎么理解呢？就是一开始需要更多的探索，所以动作偏随机，慢慢的我们需要动作能够有效，因此减少随机。

```
def __init__(self, env):
  # init experience replay
  self.replay_buffer = deque()
  # init some parameters
  self.time_step = 0
  self.epsilon = INITIAL_EPSILON
  self.state_dim = env.observation_space.shape[0]
  self.action_dim = env.action_space.n

  self.create_Q_network()
  self.create_training_method()

  # Init session
  self.session = tf.InteractiveSession()
  self.session.run(tf.initialize_all_variables())
```

创建 Q 网络：

我们这里创建最基本的 MLP，中间层设置为 20：

```
def create_Q_network(self):
  # network weights
  W1 = self.weight_variable([self.state_dim,20])
  b1 = self.bias_variable([20])
  W2 = self.weight_variable([20,self.action_dim])
  b2 = self.bias_variable([self.action_dim])
  # input layer
  self.state_input = tf.placeholder("float",[None,self.state_dim])
  # hidden layers
  h_layer = tf.nn.relu(tf.matmul(self.state_input,W1) + b1)
  # Q Value layer
  self.Q_value = tf.matmul(h_layer,W2) + b2

def weight_variable(self,shape):
  initial = tf.truncated_normal(shape)
  return tf.Variable(initial)

def bias_variable(self,shape):
  initial = tf.constant(0.01, shape = shape)
  return tf.Variable(initial)
```

只有一个隐层，然后使用 relu 非线性单元。要注意的是我们 state 输入的格式，因为使用 minibatch，所以格式是[None,state_dim]

编写 perceive 函数：

```
def perceive(self,state,action,reward,next_state,done):
  one_hot_action = np.zeros(self.action_dim)
  one_hot_action[action] = 1
   self.replay_buffer.append((state,one_hot_action,reward,next_state,done))
  if len(self.replay_buffer) > REPLAY_SIZE:
    self.replay_buffer.popleft()

  if len(self.replay_buffer) > BATCH_SIZE:
    self.train_Q_network()
```

这里需要注意的一点就是动作格式的转换。我们在神经网络中使用的是 one hot key 的形式，而在 OpenAI Gym 中则使用单值。什么意思呢？比如我们输出动作是 1，那么对应的 one hot 形式就是[0,1]，如果输出动作是 0，那么 one hot 形式就是[1,0]。这样做的目的是为了之后更好的进行计算。

在 perceive 中一个最主要的事情就是存储。然后根据情况进行 train。这里我们要求只要存储的数据大于 Batch 的大小就开始训练。

编写 action 输出函数：

```
def egreedy_action(self,state):
    Q_value = self.Q_value.eval(feed_dict = {
      self.state_input:[state]
      })[0]
    if random.random() <= self.epsilon:
      return random.randint(0,self.action_dim - 1)
    else:
      return np.argmax(Q_value)

    self.epsilon -= (INITIAL_EPSILON - FINAL_EPSILON)/10000

def action(self,state):
    return np.argmax(self.Q_value.eval(feed_dict = {
      self.state_input:[state]
      })[0])
```

区别之前已经说过，一个是根据情况输出随机动作，一个是根据神经网络输出。由于神经网络输出的是每一个动作的 Q 值，因此我们选择最大的那个 Q 值对应的动作输出。

编写 training method 函数：

```
def create_training_method(self):
  self.action_input = tf.placeholder("float",[None,self.action_dim]) # one hot presentation
  self.y_input = tf.placeholder("float",[None])
  Q_action = tf.reduce_sum(tf.mul(self.Q_value,self.action_input),reduction_indices = 1)
  self.cost = tf.reduce_mean(tf.square(self.y_input - Q_action))
  self.optimizer = tf.train.AdamOptimizer(0.0001).minimize(self.cost)
```

这里的 y_input 就是 target Q 值。我们这里采用 Adam 优化器，其实随便选择一个必然 SGD，RMS Prop 都是可以的。可能比较不好理解的就是 Q 值的计算。这里记住动作输入是 one hot key 的形式，因此将 Q_value 和 action_input 向量相乘得到的就是这个动作对应的 Q_value。然后用 reduce_sum 将数据维度压成一维。

编写 training 函数：

```
def train_Q_network(self):
    self.time_step += 1
    # Step 1: obtain random minibatch from replay memory
    minibatch = random.sample(self.replay_buffer,BATCH_SIZE)
    state_batch = [data[0] for data in minibatch]
    action_batch = [data[1] for data in minibatch]
    reward_batch = [data[2] for data in minibatch]
    next_state_batch = [data[3] for data in minibatch]

    # Step 2: calculate y
    y_batch = []
    Q_value_batch = self.Q_value.eval(feed_dict={self.state_input:next_state_batch})
    for i in range(0,BATCH_SIZE):
      done = minibatch[i][4]
      if done:
        y_batch.append(reward_batch[i])
      else :
        y_batch.append(reward_batch[i] + GAMMA * np.max(Q_value_batch[i]))

    self.optimizer.run(feed_dict={
      self.y_input:y_batch,
      self.action_input:action_batch,
      self.state_input:state_batch
      })
```

首先就是进行 mini batch 的工作，然后根据 batch 计算 y_batch。最后就是用 optimizer 进行优化。

【参考文献】

[1]CSDN.OpenAI Gym 经典控制环境介绍——CartPole（倒立摆）

[ER/OL].https：//blog.csdn.net/qq_32892383/article/details/89576003,2019-04-28

[2]CSDN.OpenAI 教程度

[ER/OL].https://blog.csdn.net/cs123951/article/details/71171260/,2017-05-04

[3]CSDN.OpenAI Gym 学习

[ER/OL].https://blog.csdn.net/u012692537/article/details/79418566,2018-03-01