

机器学习工程师毕业项目报告

人工智能 udacity

1 问题描述



机动车驾驶员在行驶的过程中时常会做出一些不符合安全驾驶的行为，例如：接打电话，发信息，和后排乘客谈话等等，这些将影响驾驶员的驾驶行为。本项目通过车内摄像头对车内驾驶员进行拍摄并通过**深度神经网络**处理每一帧的图片来确定驾驶员的状态，确定驾驶员当前是否正在安全驾驶，帮助驾驶员提高驾驶安全性。

2 项目背景

本项目是一个计算机视觉的多分类任务，属于有监督学习。具体是为了解决在机动车驾驶过程中，通过摄像头拍摄的图片确定驾驶员的行为状态是否符合安全驾驶。由于摄像头居中拍摄驾驶员捕获的图像位于捕获图片的正中央，所以本项目无需一些**全卷积网络**算法进行处理。通过深度神经网络，例如：**VGG**，**SSD**，**GoogLeNet**以及**ResNet**等成熟的网络对已经截获的样本做训练并确定当前的状态。

3 数据或输入

本项目的数据均来自udacity学城，[百度云链接](#)。

下载压缩包之后，解压分别有一个csv文件，两个存储图片的目录。

1. *driver_imgs_list.csv*
2. *train*
3. *test*

3.1 **driver_imgs_list.csv**文件内容

此文件类似以下内容，用于表示训练集与测试集的种类。

subject	classname	img
p002	c0	img_44733.jpg
...

- **subject**: 驾驶员的ID
- **classname**: 行为标号
- **img**: 对应的图片文件名称

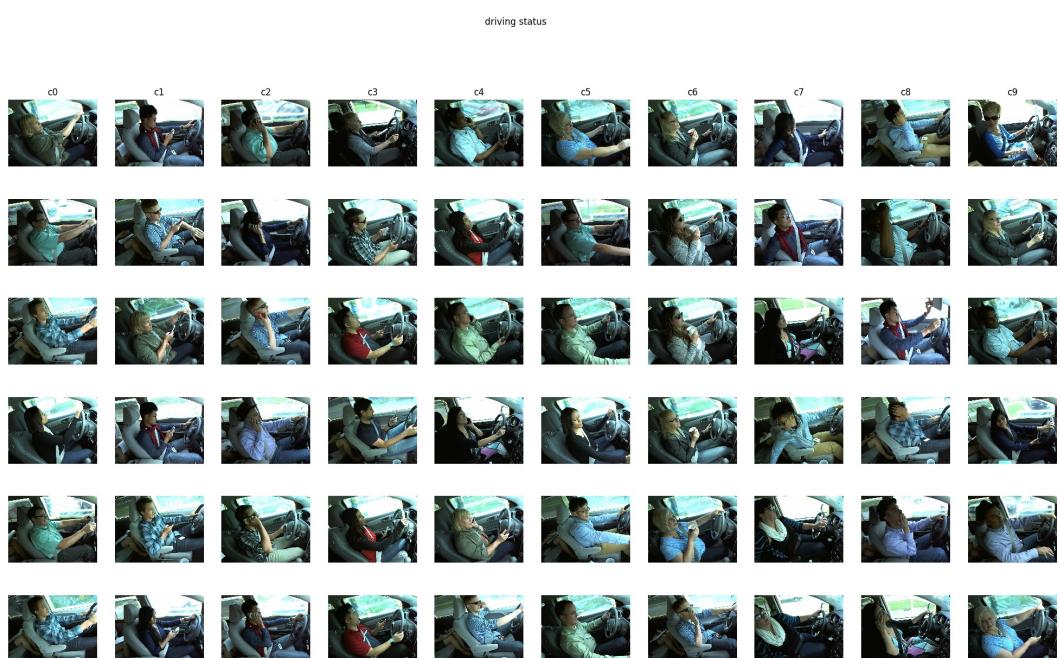
下表列出了当前的所有的行为标号以及对应的意义。

状态列表：

行为标号	说明
c0	安全驾驶

行为标号	说明
c1	右手打字
c2	右手打电话
c3	左手打字
c4	左手打电话
c5	调收音机
c6	喝饮料
c7	拿后面的东西
c8	整理头发和化妆
c9	和其他乘客说话

以下列出了每个状态对应的训练图片样例：

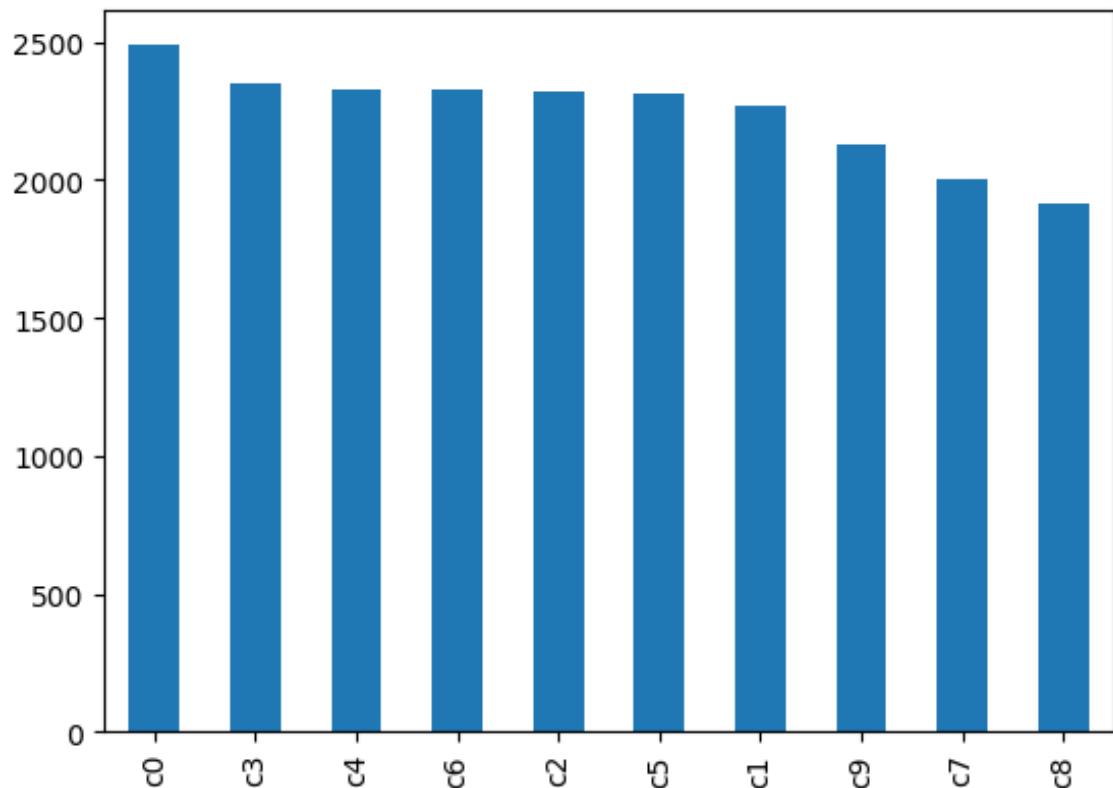


3.2 train目录

- 图片规格：**640 × 480 × 3**

- 样本总数量 : 22424

其中 $c0 - c9$ 十种状态的训练集差不多是按照 $\frac{1}{10}$ 的比例进行分配。其中 $c8$ 最少只有 1911 张图片，其余的图片数量在 2100 – 2300 之间。



通过读取 *driver_imgs_list.csv* 文件并以行为状态来划分，也可以看出每种状态的训练图片的数量都基本相同，可以确定本项目属于一个 **类别平衡的多分类问题**。

这里随机列出了了一些驾驶员的状态信息，每一行是一个驾驶状态：

driving status



3.3 test目录

测试样本具有**79,726**张图片。

由于图像都在中间，所以图像两旁的像素并无实际的以及，所以在读取图片时会从中间进行剪切，将两旁的像素裁剪掉。这里我将图片剪切到(**320 × 480**)

3.4 valid目录

样本集中没有提供验证集的图片，我会挑选两位司机 `['p064', 'p026']` 的所有图片作为验证集。

```
1. # coding:utf-
2.
3. data_dir =
r"D:\\workspace\\udacity\\MLND\\capstone\\distracted_driver_detection\\dev
ilogic_distracted_driver_detection\\"
4.
5. driver_imgs_list_csv = os.path.join(data_dir, "driver_imgs_list.csv")
6. df = pd.read_csv(driver_imgs_list_csv)
7. subjects = list(set(df["subject"]))
8. select = random.sample(range(len(subjects)), 2)
9. valid_subjects = [subjects[select[0]], subjects[select[1]]]
```

4 样本特性分析

通过以上图片，可以得到这样几个信息：

1. 图片都是居中拍摄，并且驾驶员信息占据了图像大部分的位置
2. 图片中几种状态的特征都是以轮廓进行区分
3. 图片中的颜色信息并没有什么用途

从以上分析中，我们可以对图片做这样的处理，对目标图片进行掩码处理，最好有一副掩码图片可以只取得驾驶员当前的状态信息，将其他信息掩去。得到轮廓信息，去掉多余的噪声，在检测时同样应用这样的掩码对要检测的图像进行处理。然后再进入到卷积神经网络中进行判别操作。

5 数据预先处理

这里由于在训练之前需要对图片的颜色数值进行正则化，需要均值与标准差。所以我写了小程序 `data_mean.py` 对训练集进行统计并得到以下数值：

```
1. IMG_MEAN = (0.31633861 0.38164441 0.37510719)
2. IMG_STD = (0.28836174 0.32873901 0.33058995)
```

6 优化器选择

在机器学习中，有非常多的优化器可以选择。下表列出了一些常用的优化器算法以及对应解释。

6.1 梯度下降法(Gradient Descent)

梯度下降法是最基本的一类优化器，目前主要分为三种梯度下降法：标准梯度下降法(**GD**, Gradient Descent)，随机梯度下降法(**SGD**, Stochastic Gradient Descent)及批量梯度下降法(**BGD**, Batch Gradient Descent)。

6.1.1 标准梯度下降(GD)

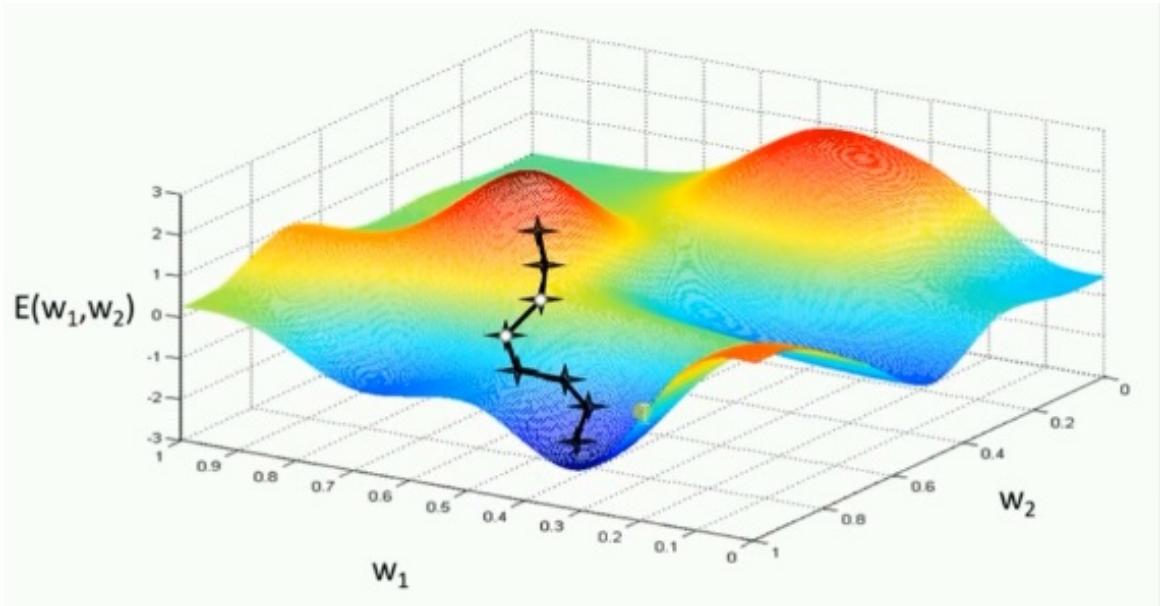
假设要学习训练的模型参数为 \mathbf{W} ，代价函数为 $J(\mathbf{W})$ ，则代价函数关于模型参数的偏导数即相关梯度为 $\Delta J(\mathbf{W})$ ，学习率为 η_t ，则使用梯度下降法更新参数为：

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \eta_t \Delta J(\mathbf{W}_t)$$

其中， \mathbf{W}_t 表示 t 时刻的模型参数。

从表达式来看，模型参数的更新调整，与代价函数关于模型参数的梯度有关，即沿着梯度的方向不断减小模型参数，从而最小化代价函数。

基本策略可以理解为“**在有限视距内寻找最快路径下山**”，因此每走一步，参考当前位置最陡的方向(即**梯度**)进而迈出下一步。可以形象的表示为：



评价：标准梯度下降法主要有**两个缺点**：

- **训练速度慢**：每走一步都要要计算调整下一步的方向，下山的速度变慢。在应用于大型数据集中，每输入一个样本都要更新一次参数，且每次迭代都要遍历所有的样本。会使得训练过程及其缓慢，需要花费很长时间才能得到收敛解。
- **容易陷入局部最优解**：由于是在有限视距内寻找下山的反向。当陷入平坦的洼地，会误以为到达了山地的最低点，从而不会继续往下走。所谓的局部最优解就是鞍点。落入鞍点，梯度为0，使得模型参数不在继续更新。

6.1.2 批量梯度下降法(BGD)

假设**批量训练样本**总数为 n ，每次输入和输出的样本分别为 X^i, Y^i ，模型参数为 W ，代价函数为 $J(W)$ ，每输入一个样本*i*代价函数关于 W 的梯度为

$\Delta J_i(W_t, X^i, Y^i)$ ，学习率为 η_t ，则使用批量梯度下降法更新参数表达式为：

$$W_{t+1} = W_t - \eta_t \sum_{i=1}^n \Delta J_i(W_t, X^i, Y^i)$$

其中， W_t 表示*t*时刻的模型参数。

从表达式来看，模型参数的调整更新与全部输入样本的**代价函数的和**（即批量/全局误差）

有关。即每次权值调整发生在批量样本输入之后，而不是每输入一个样本就更新一次模型参数。这样就会大大加快训练速度。

基本策略可以理解为，在下山之前掌握了附近的地势情况，选择总体平均梯度最小的方向下山。

评价：

- 批量梯度下降法比标准梯度下降法训练时间短，且每次下降的方向都很正确。

6.1.3 随机梯度下降法(**SGD**)

对比批量梯度下降法，假设从一批训练样本 n 中随机选取一个样本 i_s 。模型参数为 W ，代价函数为 $J(W)$ ，梯度为 $\Delta J(W)$ ，学习率为 η_t ，则使用随机梯度下降法更新参数表达式为：

$$W_{t+1} = W_t - \eta_t g_t$$

其中 $g_t = \Delta J_{i_s}(W_t, X^{i_s}, X^{i_s})$, $i_s \in \{1, 2, \dots, n\}$ 表示随机选择一个梯度方向， W_t 表示 t 时刻的模型参数。 $E(g_t) = \Delta J(W_t)$ ，这里虽然引入了随机性和噪声，但期望仍然等于正确的梯度下降。

基本策略可以理解为随机梯度下降像是一个盲人下山，不用每走一步计算一次梯度，但是他总能下到山底，只不过过程会显得扭扭曲曲。

优点：

- 虽然**SGD**需要走很多步的样子，但是对梯度的要求很低（计算梯度快）。而对于引入噪声，大量的理论和实践工作证明，只要噪声不是特别大，**SGD**都能很好地收敛。
- 应用大型数据集时，训练速度很快。比如每次从百万数据样本中，取几百个数据点，算一个**SGD**梯度，更新一下模型参数。相比于标准梯度下降法的遍历全部样本，每输入一个样本更新一次参数，要快得多。

缺点：

- **SGD**在随机选择梯度的同时会引入噪声，使得权值更新的方向不一定正确。此外，**SGD**也没能单独克服局部最优解的问题。

6.2 动量优化法

动量优化方法是在梯度下降法的基础上进行的改变，具有加速梯度下降的作用。一般有标准动量优化方法**Momentum**、**NAG** (Nesterov accelerated gradient) 动量优化方法。

6.2.1 Momentum

使用动量(**Momentum**)的随机梯度下降法(**SGD**)，主要思想是引入一个积攒历史梯度信息动量来加速**SGD**。

从训练集中取一个大小为 n 的小批量 $\{X^1, X^2, \dots, X^n\}$ 样本，对应的真实值分别为 Y^i ，则**Momentum**优化表达式为：

$$\begin{cases} v_t = \alpha v_{t-1} + \eta_t \Delta J(W_t, X^{i_s}, Y^{i_s}) \\ W_{t+1} = W_t - v_t \end{cases}$$

其中， v_t 表示 t 时刻积攒的加速度。 α 表示动力的大小，一般取值为 **0.9** (表示最大速度 **10倍** 于 **SGD**)。 $\Delta J(W_t, X^{i_s}, Y^{i_s})$ 含义见 **SGD** 算法。 W_t 表示 t 时刻的模型参数。

动量主要解决**SGD**的两个问题：一是随机梯度的方法（引入的噪声）；二是 *Hessian* 矩阵病态问题（可以理解为**SGD**在收敛过程中和正确梯度相比来回摆动比较大的问题）。

理解策略为：由于当前权值的改变会受到上一次权值改变的影响，类似于小球向下滚动的时候带上了惯性。这样可以加快小球向下滚动的速度。

6.2.2 NAG

牛顿加速梯度 (**NAG**, Nesterov accelerated gradient) 算法，是**Momentum**动量算法的变种。更新模型参数表达式如下：

$$\begin{cases} v_t = \alpha v_{t-1} + \eta_t \Delta J(W_t - \alpha v_{t-1}) \\ W_{t+1} = W_t - v_t \end{cases}$$

其中， v_t 表示 t 时刻积攒的加速度； α 表示动力的大小； η_t 表示学习率； W_t 表示 t 时刻的模型参数， $\Delta J(W_t - \alpha v_t - 1) \Delta J(W_t - \alpha v_t - 1)$ 表示代价函数关于 W_t 的梯度。

Nesterov 动量梯度的计算在模型参数施加当前速度之后，因此可以理解为往标准动量中添加了一个校正因子。

理解策略：在**Momentum**中小球会盲目地跟从下坡的梯度，容易发生错误。所以需要一个更聪明的小球，能提前知道它要去哪里，还要知道走到坡底的时候速度慢下来而不是又冲上另

一个坡。计算 $W_t - \alpha v_{t-1}$ 可以表示小球下一个位置大概在哪里。从而可以提前知道下一个位置的梯度，然后使用到当前位置来更新参数。

在凸批量梯度的情况下，Nesterov动量将额外误差收敛率从 $O(1/k)$ (k 步后)改进到 $O(1/k^2)$ 。然而，在随机梯度情况下，Nesterov动量对收敛率的作用却不是很大。

6.3 自适应学习率优化算法

自适应学习率优化算法对于机器学习模型的学习率，传统的优化算法要么将学习率设置为常数要么根据训练次数调节学习率。极大忽视了学习率其他变化的可能性。然而，学习率对模型的性能有着显著的影响，因此需要采取一些策略来想办法更新学习率，从而提高训练速度。目前的自适应学习率优化算法主要有：**AdaGrad**算法，**RMSProp**算法，**Adam**算法以及**AdaDelta**算法。

6.3.1 AdaGrad算法

AdaGrad算法，独立地适应所有模型参数的学习率，缩放每个参数反比于其所有梯度历史平均值总和的平方根。具有代价函数最大梯度的参数相应地有个快速下降的学习率，而具有小梯度的参数在学习率上有相对较小的下降。

AdaGrad算法优化策略一般可以表示为：

$$W_{t+1} = W_t - \frac{\eta_0}{\sqrt{\sum_{t'=1}^t (g_{t',i})^2} + \epsilon} \oplus g_{t,i}$$

假定一个多分类问题， i 表示第*i*个分类， t 表示第*t*迭代同时也表示分类*i*累计出现的次数。 η_0 表示初始的学习率取值一般为0.01， ϵ 是一个取值很小的数(一般为 $1e-8$)为了避免分母为0。 W_t 表示*t*时刻即第*t*迭代模型的参数， $g_{t,i} = \Delta J(W_{t,i})$ 表示*t*时刻，指定分类*i*，代价函数*J*(·)关于*W*的梯度。

从表达式可以看出，对出现比较多的类别数据，Adagrad给予越来越小的学习率，而对于比较少的类别数据，会给予较大的学习率。因此**Adagrad适用于数据稀疏或者分布不平衡的数据集。**

Adagrad的主要优势在于不需要人为的调节学习率，它可以自动调节；缺点在于，随着迭代次数增多，学习率会越来越小，最终会趋近于0。

6.3.2 RMSProp算法

RMSProp算法修改了**AdaGrad**的梯度积累为指数加权的移动平均，使得其在非凸设定下效果更好。

RMSProp算法的一般策略可以表示为：

$$\begin{cases} E[g^2]_t = \alpha E[g^2]_{t-1} + (1 - \alpha) g_t^2 \\ W_{t+1} = W_t - \frac{\eta_0}{\sqrt{E[g^2]_t + \epsilon}} \oplus g_t \end{cases}$$

其中， W_t 表示 t 时刻即第 t 迭代模型的参数， $g_t = \Delta J(W_t)$ 表示 t 次迭代代价函数关于 W 的梯度大小， $E[g^2]_t$ 表示前 t 次的梯度平方的均值。 α 表示动力(通常设置为0.9)， η_0 表示全局初始学习率。 ϵ 是一个取值很小的数(一般为 $1e - 8$)为了避免分母为0。

RMSProp借鉴了**Adagrad**的思想，观察表达式，分母为 $\sqrt{E[g^2]_t + \epsilon}$ 。由于取了个加权平均，避免了学习率越来越低的问题，而且能自适应地调节学习率。

RMSProp算法在经验上已经被证明是一种有效且实用的深度神经网络优化算法。目前它是深度学习从业者经常采用的优化方法之一。

6.3.3 AdaDelta算法

AdaGrad算法和**RMSProp**算法都需要指定全局学习率，**AdaDelta**算法结合两种算法每次参数的更新步长即：

$$\Delta W_{AdaGrad,t} = - \frac{\eta_0}{\sqrt{\sum_{t'=1}^t (g_{t',i})^2 + \epsilon}} \oplus g_t$$

$$\Delta W_{RMSProp,t} = - \frac{\eta_0}{\sqrt{E[g^2]_t + \epsilon}} \oplus g_t$$

AdaDelta算法策略可以表示为：

$$\begin{cases} E[g^2]_t = \alpha E[g^2]_{t-1} + (1 - \alpha) g_t^2 \\ \Delta W_t = - \frac{\sqrt{\sum_{i=1}^{t-1} \Delta W_i^2}}{\sqrt{E[g^2]_t + \epsilon}} \\ W_{t+1} = W_t + \Delta W_t \end{cases}$$

其中 W_t 为第 t 次迭代的模型参数， $g_t = \Delta J(W_t)$ 为代价函数关于 W 的梯度。 $E[g^2]_t$ 表示前 t 次的梯度平方的均值。 $\sum_{i=1}^{t-1} \Delta W_i$ 表示前 $t - 1$ 次模型参数每次的更新步长累加权和。

从表达式可以看出，**AdaDelta**不需要设置一个默认的全局学习率。

评价：

- 在模型训练的初期和中期，**AdaDelta**表现很好，加速效果不错，训练速度快。
- 在模型训练的后期，模型会反复地在局部最小值附近抖动。

6.3.4 Adam 算法

首先，**Adam**中动量直接并入了梯度一阶矩（指数加权）的估计。其次，相比于缺少修正因子导致二阶矩估计可能在训练初期具有很高偏置的**RMSProp**，**Adam**包括偏置修正，修正从原点初始化的一阶矩（动量项）和（非中心的）二阶矩估计。

Adam算法策略可以表示为：

$$\begin{cases} m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \\ v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \\ \hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \hat{v}_t = \frac{v_t}{1 - \beta_2^t} \\ W_{t+1} = W_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \end{cases}$$

其中， m_t 和 v_t 分别为一阶动量项和二阶动量项。 β_1, β_2 为动力值大小通常分别取 0.9 和 0.999； m^t, v^t 分别为各自的修正值。 W_t 表示 t 时刻即第 t 迭代模型的参数， $g_t = \Delta J(W_t)$ 表示 t 次迭代代价函数关于 W 的梯度大小； ϵ 是一个取值很小的数（一般为 $1e - 8$ ）为了避免分母为 0。

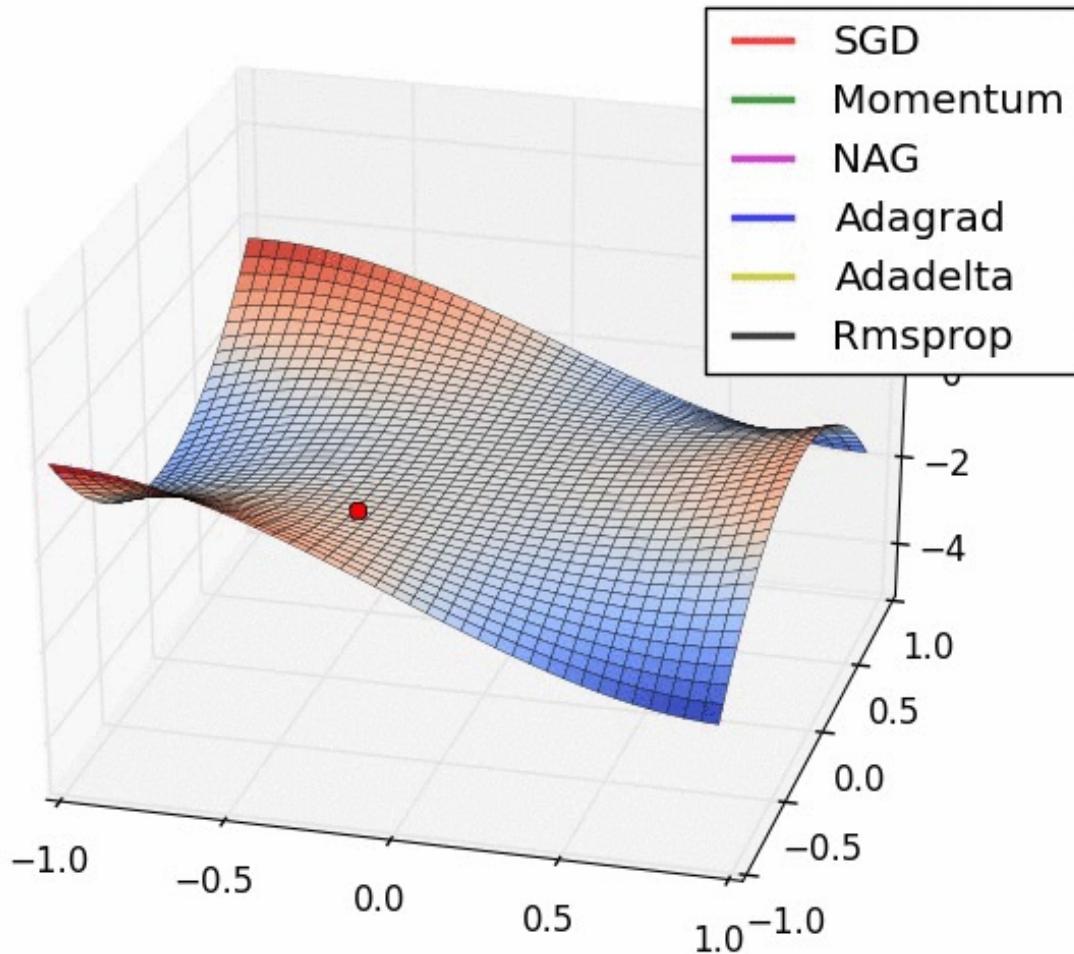
评价：

- **Adam**通常被认为对超参数的选择相当鲁棒，尽管学习率有时需要从建议的默认修改。

6.4 算法对比

下面列出了三种不同问题曲面的状况：

示例一



下降速度：

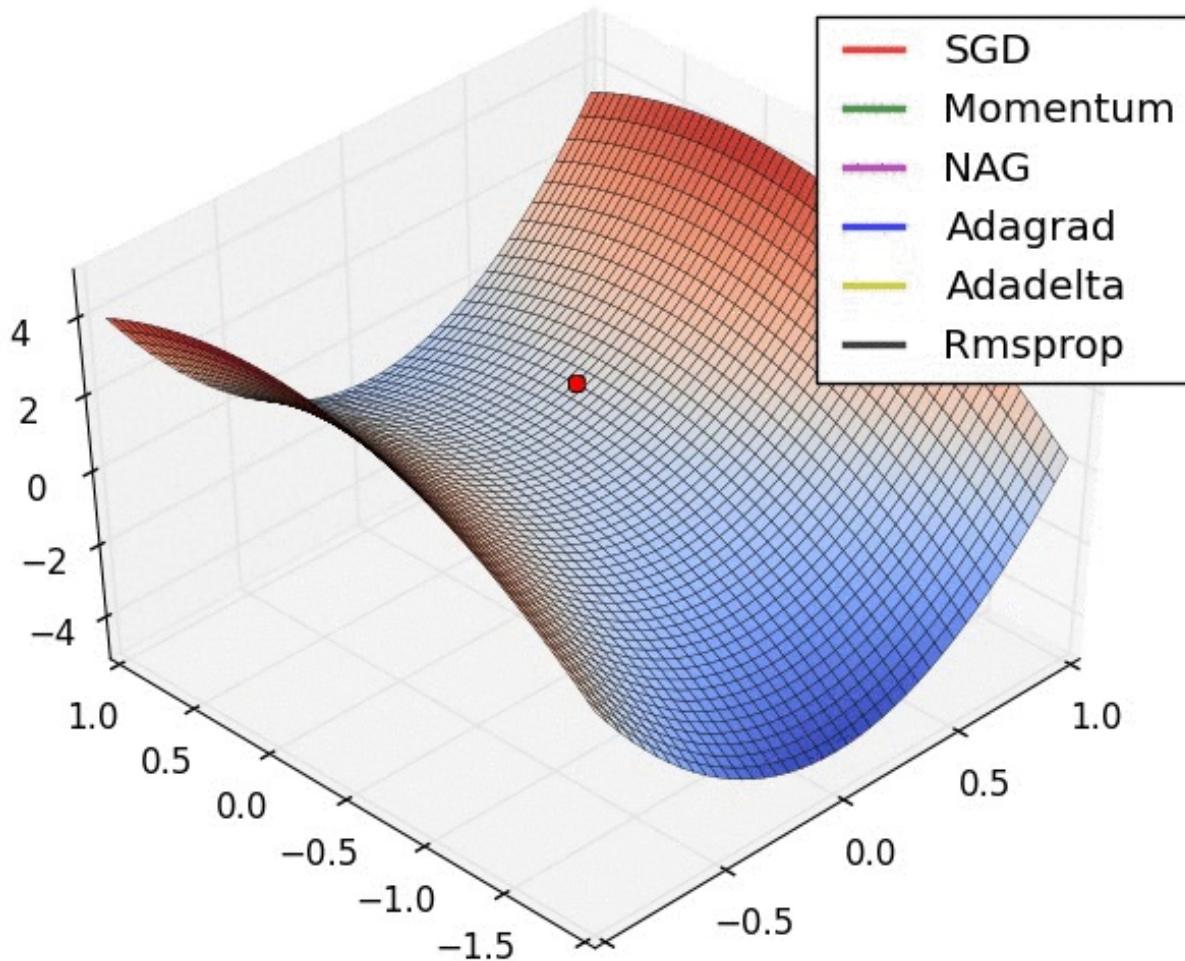
三个自适应学习优化器**Adagrad**、**RMSProp**与**AdaDelta**的下降速度明显比**SGD**要快，其中，**Adagrad**和**RMSProp**齐头并进，要比**AdaDelta**要快。

两个动量优化器**Momentum**和**NAG**由于刚开始走了岔路，初期下降的慢；随着慢慢调整，下降速度越来越快，其中**NAG**到后期甚至超过了领先的**Adagrad**和**RMSProp**。

下降轨迹：

SGD和三个自适应优化器轨迹大致相同。两个动量优化器初期走了“岔路”，后期也调整了过来。

示例二

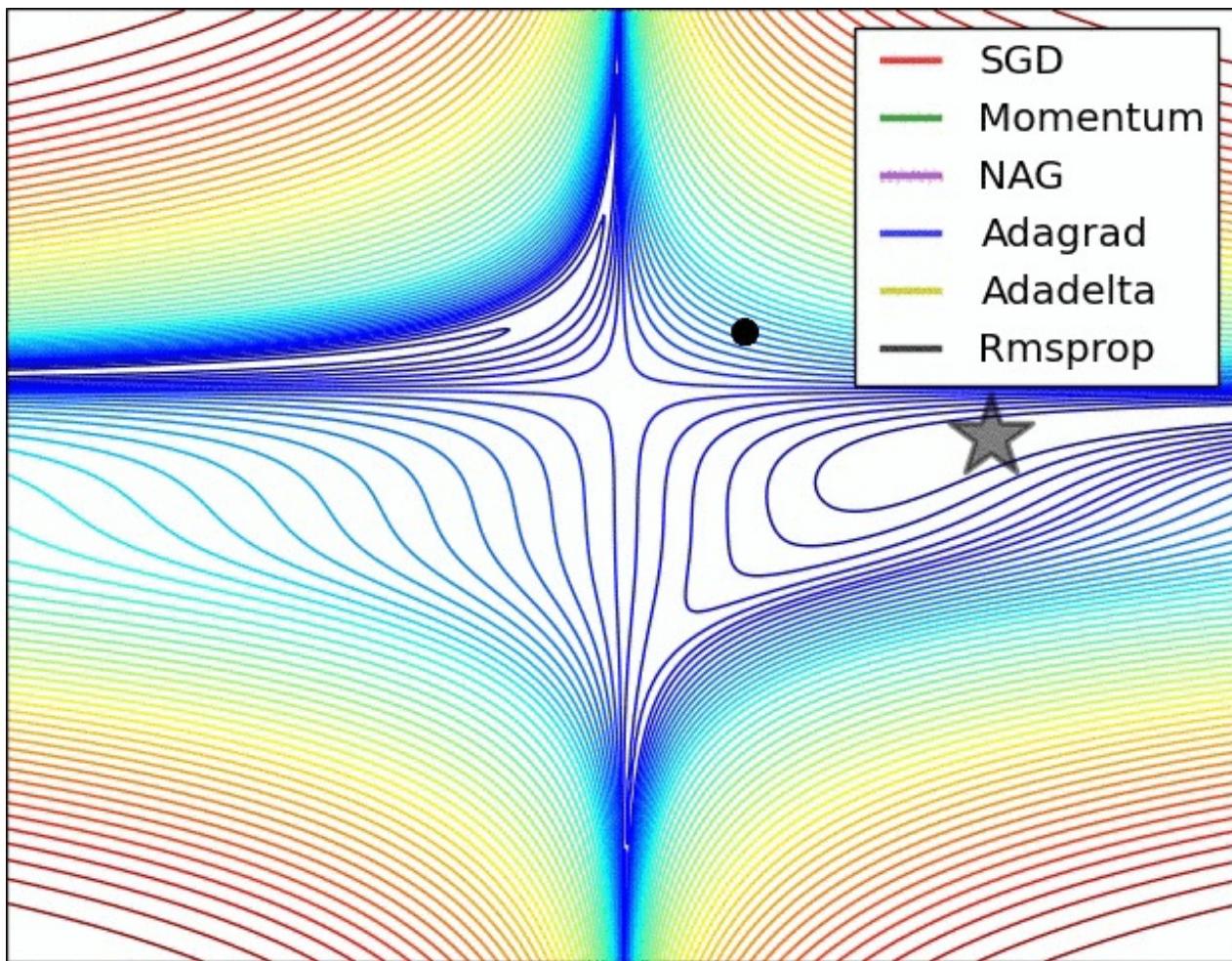


上图在一个存在鞍点的曲面，比较六种优化器的性能表现，从图中大致可以看出：三个自适应学习率优化器没有进入鞍点，其中，**AdaDelta**下降速度最快，**Adagrad**和**RMSprop**则齐头并进。

两个动量优化器**Momentum**和**NAG**以及**SGD**都顺势进入了鞍点。但两个动量优化器在鞍点抖动了一会，就逃离了鞍点并迅速地下降，后来居上超过了**Adagrad**和**RMSProp**。

很遗憾，**SGD**进入了鞍点，却始终停留在了鞍点，没有再继续下降。

示例三



在运行速度方面：

两个动量优化器**Momentum**和**NAG**的速度最快，其次是三个自适应学习率优化器**AdaGrad**、**AdaDelta**以及**RMSProp**，最慢的则是**SGD**。

在收敛轨迹方面：

两个动量优化器虽然运行速度很快，但是初中期走了很长的“岔路”。三个自适应优化器中，**Adagrad**初期走了岔路，但后来迅速地调整了过来，但相比其他两个走的路最长；**AdaDelta**和**RMSprop**的运行轨迹差不多，但在快接近目标的时候，**RMSProp**会发生很明显的抖动。

SGD相比于其他优化器，走的路径是最短的，路子也比较正。

6.5 优化器最终选择

三种类型的优化算法，通过以上分析**自适应优化算法**的效果最好。所以这里考虑训练集的标准差为 $\text{IMG_STD} = (0.28836174 \ 0.32873901 \ 0.33058995)$ 。差距小于 $\frac{1}{3}$ 。说明这里样本差距

不是非常大。那么**Adagrad**就不是一个很好的选择，其余的**RMSProp**、**AdaDelta**。在比较中**AdaDelta**效果要稍微好一些。而**Adam**再上述中并没有讨论，所以这里采用**AdaDelta**与**Adam**两种优化器来调试模型。

7 损失函数选取

损失函数最终选取

这里我采用 $\log loss$ 来对项目做最终的评估。 $\log loss = -\frac{1}{N} \sum_{i=1}^N \sum_{j=1}^M y_{ij} \log(p_{ij})$ 。选用此的原因是kaggle选用了此函数作为评估函数。

8 第一次实验

第一次实验已经有了利用特征图去做训练的想法。思考期间其实还有用 $heatmap$ 来做训练样本。但是没有付诸实施。其思路就是从去掉噪声的角度去考虑。

8.1 实施步骤

1. 使用 $vgg16$ 做迁移学习。（锁定了特征提取部分）
2. 使用 $xception$ 做迁移学习。（锁定了特征提取部分）
3. 使用 $vgg16$ 做特征提取并反卷得到特征图。
4. 使用 $xception$ 通过特征图判别类型。

8.2 实施过程

因为在最初就想用掩码图的方式对图片进行处理，但是得到掩码本身又是一个巨大的问题，所以就想到利用反卷积后的操作生成类似**掩码图**类似的图形，这样天然去除了一些图像不必要的因素。这样的图相当于做了掩码操作。之后再通过**XCEPTION**进行迁移学习进行分类。

这里可以说明的是，我使用了冻结全部特征层仅训练线性分类层的方式进行迁移学习。

期间的一个插曲是我使用`keras_applications`内的预置模型做迁移学习，而使用了`tensorflow`作为后端，安装`tensorflow`时会默认安

装`keras_applications`与`keras_processing`两个包，我使用`keras`作为前端框架进行编写代码。这里由于`keras_applications`与`keras`版本的冲突，使用前者导出的模型，后者不能和好的加载。尝试几次不同版本匹配后，果断的放弃了使用`keras+ tensorflow`的组合。转而使用`pytorch`重写编写代码。可以说`pytorch`比起`tensorflow`更加清爽与方便使用。

下边列出了 **VGG16** 的网络结构与原始论文。

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224×224 RGB image)					
conv3-64	conv3-64 LRN	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 conv1-256	conv3-256 conv3-256 conv3-256	conv3-256 conv3-256 conv3-256
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

在**VGG16**上，在线性分类之前，我对特征做了逐层的反卷积操作。

```
1.          (deconv1): ConvTranspose2d(512, 512, kernel_size=(3, 3), stride=(2
, 2), padding=(1, 1), output_padding=(1, 1))
2.          (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
3.          (deconv2): ConvTranspose2d(512, 256, kernel_size=(3, 3), stride=(2
, 2), padding=(1, 1), output_padding=(1, 1))
4.          (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
5.          (deconv3): ConvTranspose2d(256, 128, kernel_size=(3, 3), stride=(2
, 2), padding=(1, 1), output_padding=(1, 1))
6.          (bn3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
7.          (deconv4): ConvTranspose2d(128, 64, kernel_size=(3, 3), stride=(2,
2), padding=(1, 1), output_padding=(1, 1))
8.          (bn4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
9.          (deconv5): ConvTranspose2d(64, 32, kernel_size=(3, 3), stride=(2,
2), padding=(1, 1), output_padding=(1, 1))
10.         (bn5): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

在这之后因为要得到一个**3**通道的原始输入尺寸(**320 × 480**)(这里的尺寸是经过我剪切的)，我又用一个尺寸为**1**的核做了三个卷积操作。得到符合分类层模型需要的图像。

```
1.      (masker): Sequential(
2.          (0): Conv2d(32, 16, kernel_size=(1, 1), stride=(1, 1))
3.          (1): Conv2d(16, 8, kernel_size=(1, 1), stride=(1, 1))
4.          (2): Conv2d(8, 3, kernel_size=(1, 1), stride=(1, 1))
5.      )
```

目前到这里为止，后跟一个做了**FineTune**的**XCEPTION**分类网络即可。但是由于这个网络过大，在我的显卡上并不能顺利的跑起来。需要大概**40GB** + 显存空间。幸运的是我有**4**块显卡，所以我将特征网络也就是使用**VGG16**做反卷的网络在**GPU:1**上运行，而分类网络在**GPU:2**上运行。中间当有数据过渡时通过**CPU**做中转。

但是由于空间原因，这两个网络并不能串行到一起进行训练。这样我必须有两套优化器分别对两个不同的网络。但是对于特征网络来说并没有训练目标，所以这里我又添加了特征网络的分类层。

```
1.     (excessive): Conv2d(3, 1, kernel_size=(1, 1), stride=(1, 1))
2.     (classifier): Sequential(
3.         (0): Dropout(p=0.5)
4.         (1): Linear(in_features=153600, out_features=2048, bias=True)
5.         (2): ReLU(inplace)
6.         (3): Dropout(p=0.5)
7.         (4): Linear(in_features=2048, out_features=10, bias=True)
8.     )
```

先将3通道变成单通道的图像，然后展开做全链接的分类运算。

而对于分类层，我只是调节一下全链接的情况。做了如下添加：

```
1.     (last_linear): Sequential(
2.         (0): Dropout(p=0.5)
3.         (1): Linear(in_features=2048, out_features=1000, bias=True)
4.         (2): ReLU(inplace)
5.         (3): Dropout(p=0.5)
6.         (4): Linear(in_features=1000, out_features=10, bias=True)
7.     )
```

完整所有工作以后，我使用了 `dropout=0.5` , `learn_rate=0.001` , `epochs=20` 这样的参数进行了训练。得到了如下结果(由于屏幕有限，这里仅列出前4次迭代)：

```
Trainning: 1
Train Epoch: 1    Masker Loss: 0.068282    Classifier Loss: 0.695285

Valid set: Average loss: 0.8115, Accuracy: 1109/1560 (71%)

Trainning: 2
Train Epoch: 2    Masker Loss: 0.004174    Classifier Loss: 0.267701

Valid set: Average loss: 0.8432, Accuracy: 1081/1560 (69%)

Trainning: 3
Train Epoch: 3    Masker Loss: 0.000421    Classifier Loss: 0.618786

Valid set: Average loss: 0.7693, Accuracy: 1212/1560 (78%)

Trainning: 4
```

上图中**Masker Loss**是特征网络在**训练集**的损失值，**Classifier Loss**是分类网络在**训练集**的损失值。可以看出的是，在第一次迭代完毕后，特征网络基本就收敛了。**0.068282**相当于**98%**以上的精确度。

从这里可以看出使用特征图再做分类，会加速网络的收敛与过拟合。因用转置得到的图像是特征图，用特征图在与原输出叠加。相当于把训练集的重要特征放大了。所以再调整权的时候，对这些放大的特征给予更多的调节。而激励函数选取relu 把那些小的输出为0。泛化能力降低。但对训练过的图片记忆增强。

这里也可以看到在**验证集**上的效果还算一般。在**20**次迭代之后，最高达到**86**。

```
Valid set: Average loss: 0.7823, Accuracy: 1300/1560 (83%)  
Trainning: 17  
Train Epoch: 17 Masker Loss: 0.000000 Classifier Loss: 0.003399  
  
Valid set: Average loss: 0.8998, Accuracy: 1258/1560 (81%)  
Trainning: 18  
Train Epoch: 18 Masker Loss: 0.000000 Classifier Loss: 0.099811  
  
Valid set: Average loss: 0.8566, Accuracy: 1258/1560 (81%)  
Trainning: 19  
Train Epoch: 19 Masker Loss: 0.000000 Classifier Loss: 0.000031  
  
Valid set: Average loss: 0.6661, Accuracy: 1348/1560 (86%)  
Trainning: 20  
Train Epoch: 20 Masker Loss: 0.000000 Classifier Loss: 0.001570  
  
Valid set: Average loss: 0.9861, Accuracy: 1205/1560 (77%)  
□
```

可以看到**Masker Loss**为**0**，这说明验证的过拟合了。因为特征提取的过拟合特性，使得分类层也有过拟合的倾向。在第**19**轮，达到**0.000031**。这相当于接近**0**了。在**训练集**上这个网络应该可以**100%**的识别目标。但是在**验证集**上的表现一般了。

由于问题是过拟合产生的，所以首先要加大**dropout**的处理，这里选择了**0.75**，其次为了让网络得到更充分的训练。我训练了**50**次。而学习率也降低了一些**0.00075**。进行了第二次训练。但是结果仍然和上次结果相差无几。

这个原因是出在特征网络的过拟合造成的，而特征网络使用了反卷积做特征提取在进行分类。所以问题可能是出在反卷积叠加了每层的输出，所以我去掉每层的卷积输出。只保留最后一层的输出。这样形成的特征可能更具有防滑效果。

```
1.     def forward(self, x):  
2.         output = self.model(x)  
3.         x5 = output['x5']  
4.         x4 = output['x4']
```

```

5.         x3 = output['x3']
6.         x2 = output['x2']
7.         x1 = output['x1']
8.
9.         mask = self.bn1(self.relu(self.deconv1(x5)))
10.        # mask = mask + x4
11.        mask = self.bn2(self.relu(self.deconv2(mask)))
12.        # mask = mask + x3
13.        mask = self.bn3(self.relu(self.deconv3(mask)))
14.        # mask = mask + x2
15.        mask = self.bn4(self.relu(self.deconv4(mask)))
16.        # mask = mask + x1
17.        mask = self.bn5(self.relu(self.deconv5(mask)))
18.        self.mask = self.masker(mask)

```

但是特征网络仍然会过快的拟合，这样分类网络也过快拟合了。随后我又改进了模型，使用了在获取到mask层后与原始的输入进行并集的操作，然后再输入到xception网络中。

```

1.         mask = self.masker(mask)
2.
3.         #select = mask.max()*0.5 if mask.max()*0.7 > mask.mean() else m
ask.mean()
4.         select = mask.mean()
5.         mask[mask<select] = 0
6.         mask[mask>=select] = 1
7.
8.         image = x * mask
9.         #show = transforms.functional.to_pil_image(image.cpu()
[0].detach())
10.        #show.show()
11.
12.        pred = self.xception(image)

```

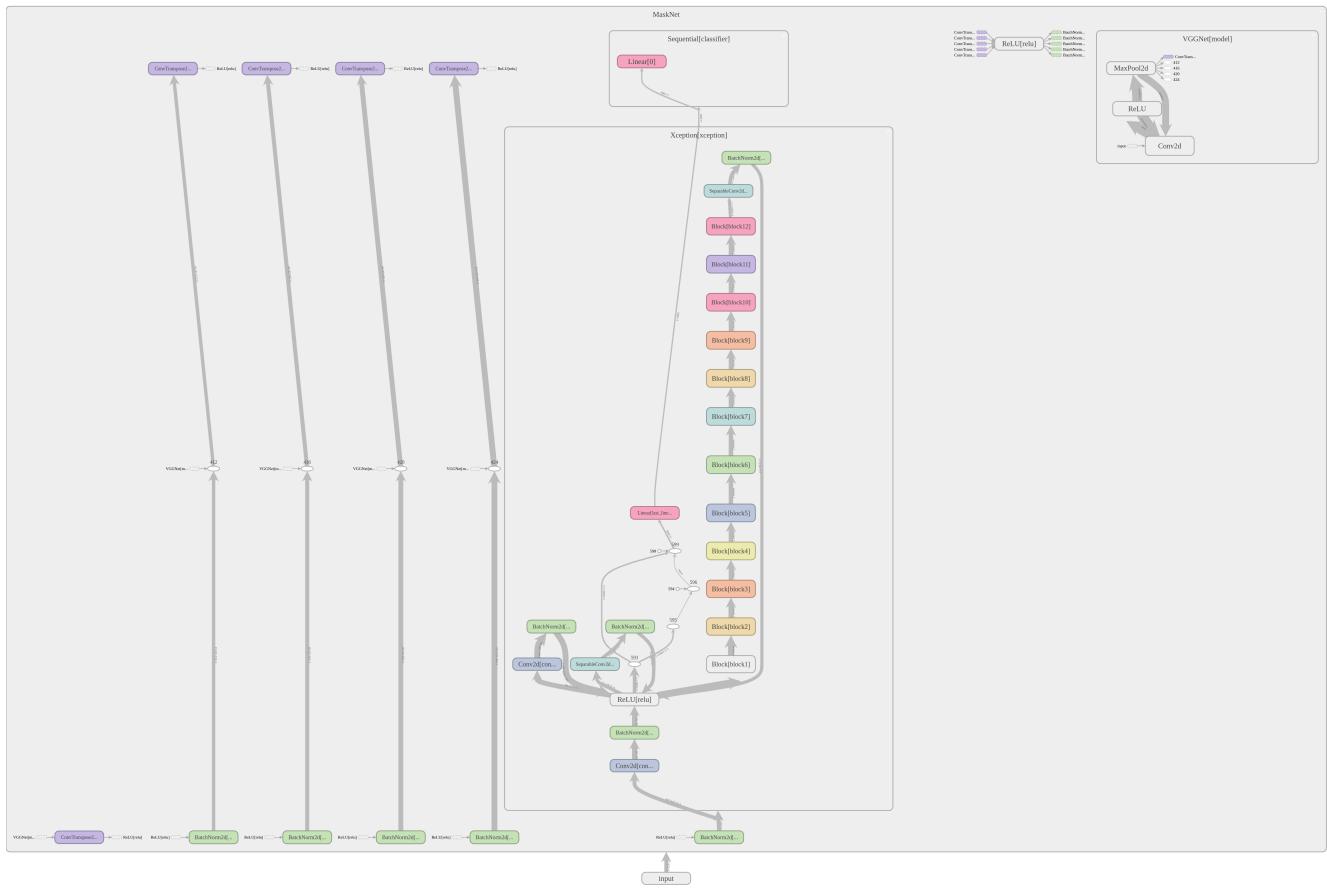
结果却有下降的趋势。所以直接使用上述的模型进行测试。

8.3 最终结果

最终的实验结果并不是非常理想，只得到**loss**在1.0左右的分值。这里并没有到达预期。

8.4 xvgg的网络结构

下图展示了第一次实验所采用的网络结构。



9 第二次实验

通过第一次实验，有了通过掩码图来屏蔽噪声来提高精确度的想法。所以就进行了以下尝试。

9.1 实施步骤

1. 尝试使用`vgg16`直接生成掩码图。（这里的`vgg16`锁定了特征提取d）
2. 对`train`、`valid`、`test`的数据集转换为掩码图。
3. 使用`xception`用掩码图进行训练。

9.2 实施过程

当进行了第一次实验测试后，有了提取`Mask`图的想法。然后进行了以下的测试，自动提

取mask图的流程如下：

1. 通过VGG16模型进行特征提取生成**卷积特征图**。
2. 通过对**卷积特征图**进行反卷积的操作生成原图大小的**特征图**。
3. 通过对**特征图**的**做众数池化**进一步去噪声生成**原始掩码图**。
4. 通过对**原始掩码图**取一个阀值(这里取均值)生成**掩码图**。
5. 将**原图与掩码图**取并集生成最终的图片。

通过反卷积+卷积层输出得到的特征图像



通过特征图像提取的掩码图



但从感观上看效果还不错，保留了我想让网络学习的特征。

9.2.1 众数池化

为了让得到的特征图可以进一步去噪，让相临的数值保持一致性，做了利用过滤核取众数的操作。

代码如下：

```
1.     def ModePool2d(self, input_tensor, kernel_size=3):
2.         row_size = input_tensor.size(0)
3.         col_size = input_tensor.size(1)
4.
5.         row_count = row_size // kernel_size
6.         col_count = col_size // kernel_size
7.
8.         for i in range(0, row_count):
9.             for j in range(0, col_count):
10.                 m = input_tensor[i*kernel_size: i*kernel_size + kernel_size,
11.                                 j*kernel_size:
12.                                 j*kernel_size+kernel_size]
13.                 feature_value = m.mode()[0].mode()[0]
14.                 input_tensor[i*kernel_size: i*kernel_size + kernel_size,
15.                             j*kernel_size: j*kernel_size+kernel_size] =
feature_value
16.             return input_tensor
```

我将训练集合以及测试与验证集都输出到一个目录，是对原始数据预处理。

使用预处理后的文件再次进行训练，这次采用的模型是**xception**做迁移学习只是修改了分类层：

```
1.     finetune = nn.Sequential(
2.         nn.Linear(in_dim, 4096),
3.         nn.ReLU(True),
4.         nn.Dropout(self.dropout),
5.         nn.Linear(4096, 4096),
6.         nn.ReLU(True),
7.         nn.Dropout(self.dropout),
8.         nn.Linear(4096, self.num_classes)
9.     )
```

得到如下结果：

```
Trainning: 5
Train Epoch: 5    Loss: 2.087347
Valid set: Average of loss: 2.1793, Accuracy: 259/1560 (17%)
Trainning: 6
Train Epoch: 6    Loss: 2.260529
Valid set: Average of loss: 2.1866, Accuracy: 262/1560 (17%)
Trainning: 7
Train Epoch: 7    Loss: 2.133332
Valid set: Average of loss: 2.1703, Accuracy: 269/1560 (17%)
Trainning: 8
Train Epoch: 8    Loss: 2.257448
Valid set: Average of loss: 2.2072, Accuracy: 223/1560 (14%)
Trainning: 9
Train Epoch: 9    Loss: 2.029122
Valid set: Average of loss: 2.2507, Accuracy: 226/1560 (14%)
Trainning: 10
Train Epoch: 10   Loss: 2.146322
Valid set: Average of loss: 2.1843, Accuracy: 273/1560 (18%)
```

结果非常糟糕，不过这只是我的一次尝试。在后续的工作中我会再调整策略，以及改进训练方式以及取掩码的方式再进行尝试。

10 第三次实验

这次实验，我准备首先按照正常套路来调教对比迁移学习的模型，然后再使用**bagging**的训练方式来进行调教。每次做训练，我都想一些很奇怪套路来，结果并不是非常有效果。这次按照最正统的方式，先将一个模型调节到一个比较不错的精度再实验一些其他想法。

10.1 实施步骤

1. 使用*xception*做迁移学习。（使用*imagenet*作为初始值，完全重新训练网络）
2. 使用*pnasnet5large*做迁移学习。（使用*imagenet*作为初始值，完全重新训练网络）
3. 通过**bagging**的方式测试*xception*的学习的结果。
4. 通过**bagging**的方式测试*pnasnet5large*的学习的结果。
5. 对比两个网络产生的结果。

10.2 实施过程

使用*xception*、*resnet152*、*pnasnet5large*，采用学习率在0.001，迭代次数在100的超参对比如下：

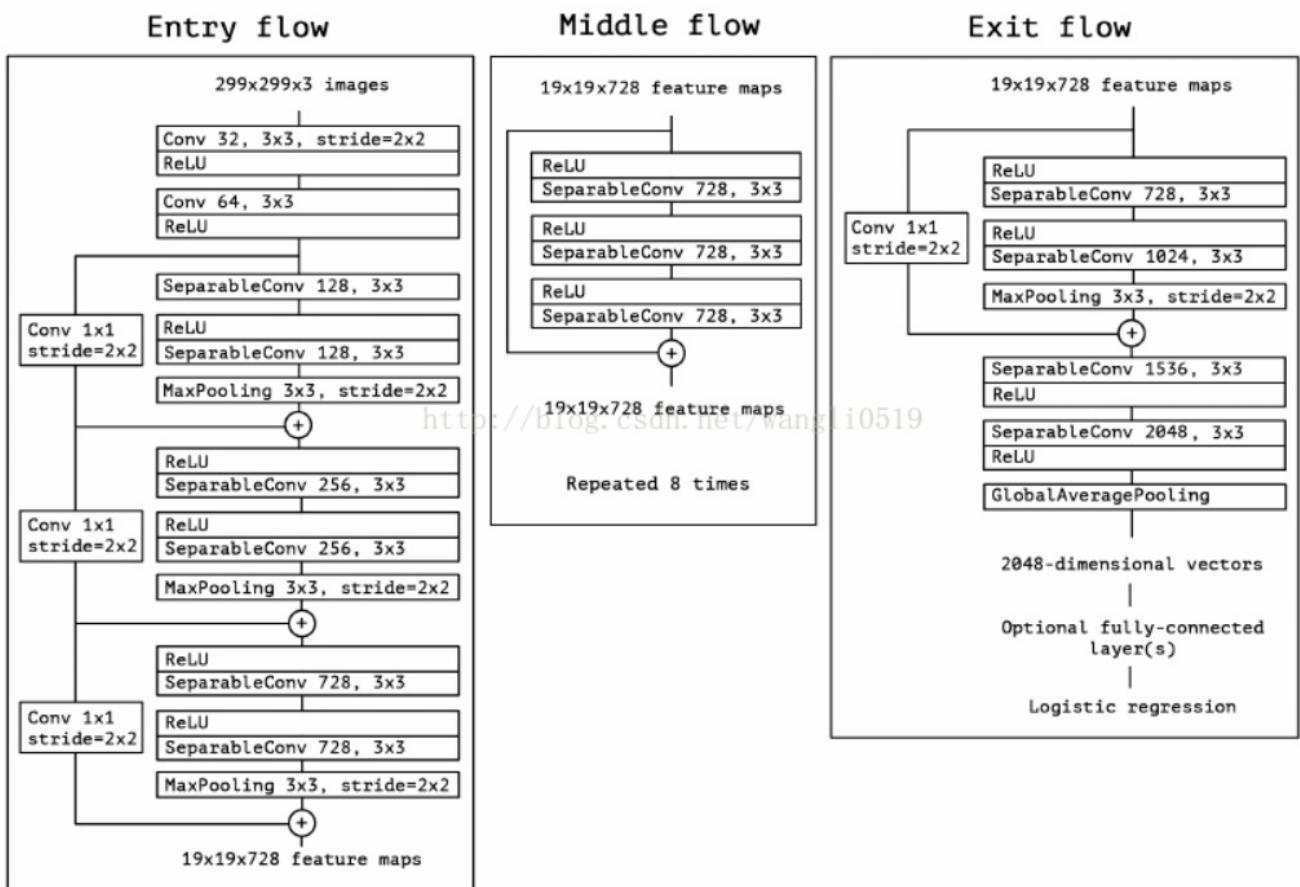
名称	验证集精度	训练集loss	时间
<i>xception</i>	83%	0.0	5小时
<i>resnet152</i>	80%	0.0	8小时
<i>pnasnet5large</i>	87%	0.0	> 10小时

这里表现最好的是*pnasnet5large*网络，但是这个网络的训练时间太长了。这里选取两种一个是*xception*，一个是*pnasnet5large*来接下来通过*bagging*的训练方式调教网络。这里我以司机进行分类，总共分为4个司机。

```
1. # 四个模型每个模型使用了6个训练集
2. model_1_subjects = ['p041', 'p066', 'p016', 'p024', 'p056', 'p061']
3. model_2_subjects = ['p075', 'p015', 'p042', 'p022', 'p049', 'p039']
4. model_3_subjects = ['p035', 'p026', 'p072', 'p012', 'p047', 'p045']
5. model_4_subjects = ['p081', 'p021', 'p050', 'p002', 'p064', 'p051']
6.
7. # 验证集使用了两个集合
8. valid_subjects = ['p014', 'p052']
```

10.2.1 使用*xception*做迁移学习

下图展示了*xception*网络的基本结构以及原始论文出处。



Xception: Deep Learning with Depthwise Separable Convolutions

首先使用了学习率**0.001**, 丢包率**0.5**, 迭代次数**100**, 批次**32**的超参下训练4个模型。

在第三个训练集上在训练到**20**轮左右一直是**11%**，感觉模型收敛到一个局部解里出不来，所以，这里中断尝试其他的学习率。选择了学习率**0.005**, 丢包率**0.75**的组合。

次数	模型序号	学习率	丢包率	迭代次数	批次	验证集精确度
1	1	0.001	0.5	100	32	76.98%
1	2	0.001	0.5	100	32	56.61%
1	3	0.005	0.75	100	32	75.93%
1	4	0.001	0.5	100	32	68.38%
2	1	0.0005	0.75	100	32	78.96%
2	2	0.001	0.75	100	32	52.29%

次数	模型序号	学习率	丢包率	迭代次数	批次	验证集精确度
2	3	0.0005	0.75	100	32	75.62%
2	4	0.0005	0.75	100	32	62.07%
3	1	0.00001	0.8	100	32	80.75%
3	2	0.005	0.6	100	32	70.98%
3	3	0.0001	0.6	100	32	76.92%
3	4	0.0001	0.6	100	32	62.44%
4	1	0.001	0.85	100	32	最好精度，没超过上一次的 80.75%
4	2	0.001	0.5	100	32	70.98%
4	3	0.00001	0.85	100	32	77.29%
4	4	0.0001	0.85	100	32	63.18%

10.2.2 使用 $pnasnnet5large$ 做迁移学习

四个模型都使用了学习率0.0001，丢包率0.5，批次8的超参数进行训练。

次数	模型序号	学习率	丢包率	迭代次数	批次	验证集精确度
1	1	0.0001	0.5	100	8	89.98%
1	2	0.0001	0.5	100	8	90.10%
1	3	0.0001	0.5	100	8	89.60%
1	4	0.0001	0.5	100	8	89.79%

$pnasnnet5large$ 训练非常慢，也非常消耗显存。所以这里只做了一次调教，每批次的数量也削减到8个样本。根据结果而言，单个模型的精度还是可以再提高的。

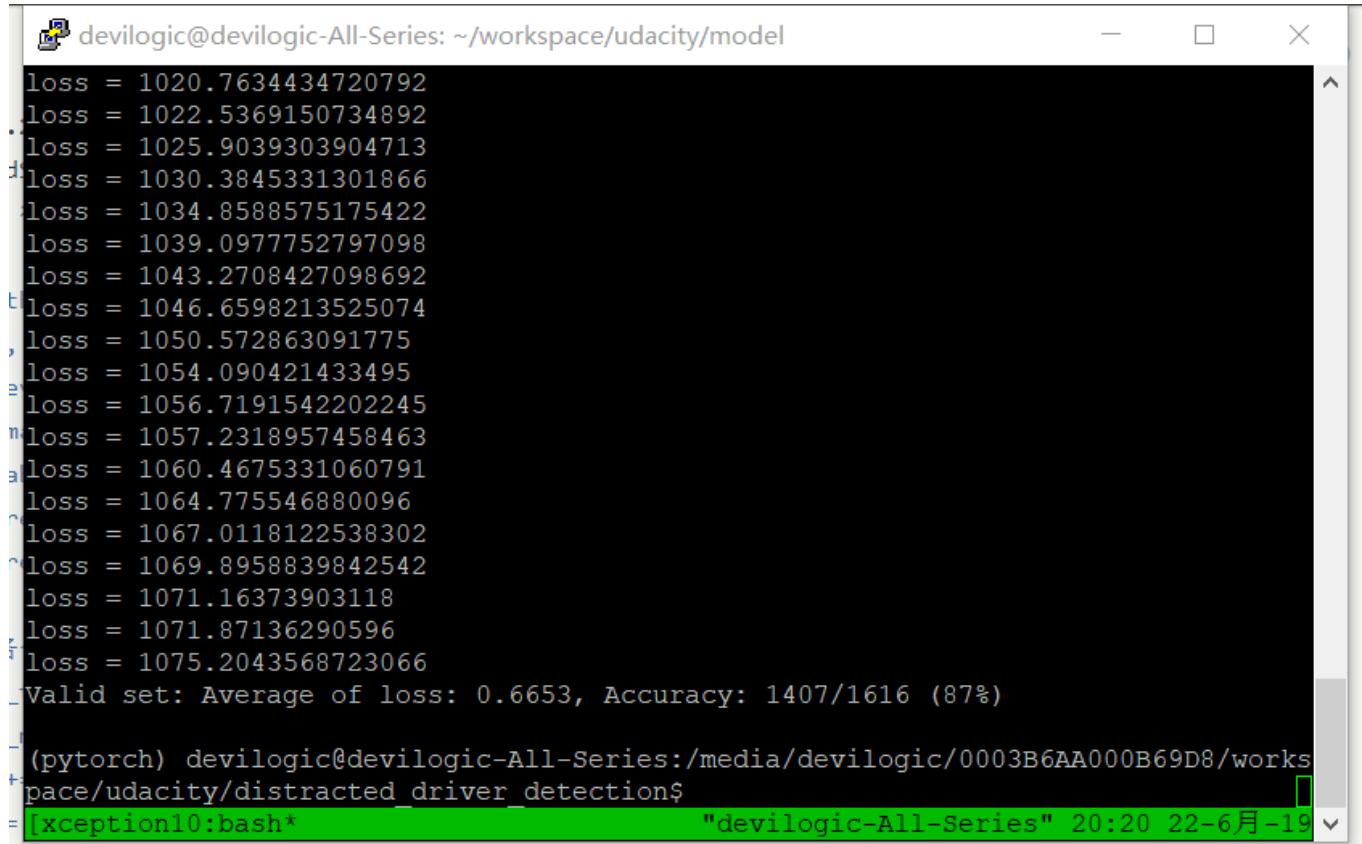
10.2.3 通过bagging的方式测试

在每个模型的源文件中，我提供了 `bagging_test` 函数对模型进行测试。通过四个模型对目标独立进行识别，得到结果后，通过取均值来获得最后预测结果。最后通过选取最大值来确定识别目标。

```
1.     for i, model in enumerate(models):
2.         device = devices[i]
3.         image = image.to(device)
4.         label = label.to(device)
5.         pred = model(image)
6.         preds.append(pred.cuda().data.cpu().numpy())
7.
8.     # 取各个模型预测的均值
9.     preds_tensor = torch.tensor(preds, dtype=torch.float64)
10.    preds_mean = preds_tensor.mean(dim=0)
11.    loss += criterion(preds_mean, label_clone).item()
12.    pred = preds_mean.max(1, keepdim=True)[1]
13.    correct += pred.eq(
14.        label_clone.view_as(pred)).sum().item()
```

以下是两个模型在验证集上的预测结果：

*xception*的预测结果



A terminal window showing the output of an *xception* model's predictions. The window title is "devilogic@devilogic-All-Series: ~/workspace/udacity/model". The terminal displays a series of loss values followed by a summary line: "Valid set: Average of loss: 0.6653, Accuracy: 1407/1616 (87%)". The prompt at the bottom right indicates the session is still active.

```
devilogic@devilogic-All-Series: ~/workspace/udacity/model
loss = 1020.7634434720792
loss = 1022.5369150734892
loss = 1025.9039303904713
loss = 1030.3845331301866
loss = 1034.8588575175422
loss = 1039.0977752797098
loss = 1043.2708427098692
loss = 1046.6598213525074
loss = 1050.572863091775
loss = 1054.090421433495
loss = 1056.7191542202245
loss = 1057.2318957458463
loss = 1060.4675331060791
loss = 1064.775546880096
loss = 1067.0118122538302
loss = 1069.8958839842542
loss = 1071.16373903118
loss = 1071.87136290596
loss = 1075.2043568723066
Valid set: Average of loss: 0.6653, Accuracy: 1407/1616 (87%)
(pytorch) devilogic@devilogic-All-Series:/media/devilogic/0003B6AA000B69D8/workspace/udacity/distracted_driver_detection$ [xception10: bash*] "devilogic-All-Series" 20:20 22-6月-19
```

预测精度: 89% , 平均 loss 0.6653。

*pnasnet5large*的预测结果

```
devilogic@devilogic-All-Series: ~/workspace/udacity/model
pace/udacity/distracted_driver_detection$ python ddd_pnasnet5large.py bagging_te
st --valid_data_dir=../data/bagging/valid --cpp1=../model/pnasnet1_Sat_Jun_22_02
15_32_2019.json --cpp2=../model/pnasnet2_Sat_Jun_22_02_03_29_2019.json --cpp3=.
./model/pnasnet3_Sat_Jun_22_02_41_09_2019.json --cpp4=../model/pnasnet4_Sat_Jun_
22_01_31_05_2019.json
[INFO]read checkpoint: ../model/pnasnet1_Sat_Jun_22_02_15_32_2019.json
    read model: /media/devilogic/0003B6AA000B69D8/workspace/udacity/model/pnas
net1_Sat_Jun_22_02_15_32_2019.pth
, [INFO]read checkpoint: ../model/pnasnet2_Sat_Jun_22_02_03_29_2019.json
    read model: /media/devilogic/0003B6AA000B69D8/workspace/udacity/model/pnas
net2_Sat_Jun_22_02_03_29_2019.pth
n[INFO]read checkpoint: ../model/pnasnet3_Sat_Jun_22_02_41_09_2019.json
a    read model: /media/devilogic/0003B6AA000B69D8/workspace/udacity/model/pnas
net3_Sat_Jun_22_02_41_09_2019.pth
[INFO]read checkpoint: ../model/pnasnet4_Sat_Jun_22_01_31_05_2019.json
    read model: /media/devilogic/0003B6AA000B69D8/workspace/udacity/model/pnas
net4_Sat_Jun_22_01_31_05_2019.pth
[INFO]models generated
[INFO]generate datasets ok
Valid set: Average of loss: 0.1528, Accuracy: 1563/1616 (97%)
(pytorch) devilogic@devilogic-All-Series:/media/devilogic/0003B6AA000B69D8/works
pace/udacity/distracted_driver_detection$ 
= [pnasnet1]0:bash*                                     "devilogic-All-Series" 20:20 22-6月-19
```

预测精度: 97% , 平均 loss 0.1528

10.2.4 对比两个网络产生的结果

实际的对比是*pnasnet5large*要比*xception*的预测结果高出10%个点。这里选取*pnasnet5large*的模型作为最终模型。

10.3 Kaggle测试结果

上传kaggle后，最终得分:0.91667。

11 第四次实验

由于上一次bagging没有充分应用到数据集，这次实验准备采用*k-fold*的方式进

行 $bagging$ ，将训练集分成 k 份， $k - 1$ 份作为训练集合，其余1份作为验证集，这样总共做4份模型独立进行训练。每个模型的验证集都不同。然后最终在测试时使用几个模型的均值进行判别。

之前的 $xception$ 模型的线性分类层分化过多，过拟合太严重。所以这里修改为一层分类层，并且增加一个 $dropout$ 层来进一步调参防止过拟合的出现。

***xception*分类层的修改**

这里由于我的分类层太长可能会产生过拟合的现象。

修改前

```
1. finetune = nn.Sequential(
2.     nn.Linear(in_dim, 4096),
3.     nn.ReLU(True),
4.     nn.Dropout(self.dropout),
5.     nn.Linear(4096, 4096),
6.     nn.ReLU(True),
7.     nn.Dropout(self.dropout),
8.     nn.Linear(4096, self.num_classes)
9. )
```

修改后

```
1. in_dim = self.model.last_linear.in_features
2. last_linear = nn.Sequential(
3.     nn.Dropout(self.dropout),
4.     nn.Linear(in_dim, self.num_classes)
5. )
```

11.1 对过拟合与泛化的数学原理的一点思考

我在处理以上模型时，就是忘记对训练集过拟合的处理，至此我对防止过拟合的手段进一步思考。

因为我一直将神经网络看作是**线性代数** + **数值分析** + **概率统计**的一个综合应用。所以在思考 $dropout$ 问题是我是从**线性代数**的角度来理解 $dropout$ 的。首先将神经网络本身就是**n做亚定方程组**(自由变量的个数大于方程组的个数)一般而言存在**无数**个解。这里方程的自由变量就是样本的维度，而方程组的个数就是每层输出神经元的个数。就一般而言一张图片的像素个数

而言都会大于当前层的输出个数。 $W\mathbf{x} + \mathbf{b} = \mathbf{y}$ ，一般而言 W 是一个 $m > n$ 的一个矩阵，*dropout*就是每次训练时，减少 m 的个数。但是形成了不同的 W_1, W_2, W_i 等等，但是他们的行个数都要大于列个数 n ，这样选取后使得每个线性组合的变量在不同的向量空间得到的训练。可以选择到一个对假设空间更好划分的线性组合。而我有兴趣的就是以数学的语言去解释这个过程。

以下是我读的一些*dropout*的论文。

关于*dropout*的论文：

[《Improving neural networks by preventing co-adaptation of feature detectors》](#)

[《Dropout:A Simple Way to Prevent Neural Networks from Overfitting》](#)

[《Improving Neural Networks with Dropout》](#)

[《Dropout as data augmentation》](#)

[《Dropout as a Bayesian approximation: representing model uncertainty in deep learning》](#)

但是感觉最好的是这篇[《Dropout as a Bayesian approximation: representing model uncertainty in deep learning》](#) 对我启发很大。

11.2 数据集划分

通过*k-fold*的对单个模型的数据集进行划分。在源文件`ddd_units/kfold_split.py`中有记录了划分规则。最终会形成一份配置文件，在训练时程序读取这份文件来确定当前模型的验证集合，并且每个验证集合都不相交。

以下是*kfold*配置文件的一个例子：

```
1.  {
2.      "number_models": 4,
3.      "number_valids": 2,
4.      "valid_drivers":
5.      {
6.          "model_1": ["p056", "p045"],
7.          "model_2": ["p051", "p026"],
8.          "model_3": ["p061", "p022"],
9.          "model_4": ["p039", "p075"]
10.     }
11. }
```

11.3 实验步骤

1. 使用 *xception* 做迁移学习。（使用 *imagenet* 作为初始值，完全重新训练网络，分类层进行了缩短）
2. 使用 *pnasnet5large* 做迁移学习。（使用 *imagenet* 作为初始值，完全重新训练网络，使用与 **实验三** 不同的超参数调整模型）
3. 通过 **bagging** 的方式测试 *xception* 的学习的结果。
4. 通过 **bagging** 的方式测试 *pnasnet5large* 的学习的结果。
5. 对比两个网络产生的结果。

11.4 实验过程

这里先采用 *xception* 进行迁移学习，随后采用 *pnasnet5* 进行迁移学习，在单体网络都可以达到一个比较不错的程度时，4个模型的验证集精度超过 85%。随后进行 **bagging** 测试。其后进行两个模型验证集精度对比，以及 **kaggle** 分数的对比。

11.4.1 第一轮 *xception* 进行迁移训练

这里先使用 *xception* 进行训练，日志记录如下：

次数	模型序号	学习率	丢包率	迭代次数	批次	最小验证集 loss	最优验证集精确度	最优精度迭代数
1	1	0.0001	0.75	100	32	0.2566	94%	57
1	2	0.0001	0.75	100	32	0.1608	97%	20
1	3	0.0001	0.75	100	32	0.2612	94%	25
1	4	0.0001	0.75	100	32	0.1884	95%	36

在选出验证集的结果

在 `['p064', 'p026']`，两个数据集上验证 0.0 的 loss，100% 的正确率，这也不稀奇，因为四个模型一起做训练，这两个集合已经包含在里面了。

kaggle 结果

私有分数为0.53680，公开分数0.70055。

11.4.2 第二轮xception进行迁移训练

由于上一次模型已经收敛到一个比较高的精度，平均在95%的水平，所以这次调节，将学习率降低一半，并且增加一些*dropout*。

次数	模型序号	学习率	丢包率	迭代次数	批次	最小验证集loss	最优验证集精确度	最优精度迭代数
2	1	0.00005	0.8	100	32	0.2037	94%	2
2	2	0.00005	0.8	100	32	0.1608	97%	0
2	3	0.00005	0.8	100	32	0.3293	95%	32
2	4	0.00005	0.8	100	32	0.2620	95%	92

从以上训练结果看，第二轮的训练没有第一轮好，第一个模型在略微提高了一点点精度的情况下也从局部最优解中跳了出来。第二个模型基本没有变动。第三个模型与第四个模型都没有什么大幅度的提高，但是这几个模型的*loss*却都增加了。所以决定舍弃这次训练。

11.4.3 第三轮xception进行迁移训练

决定还是在使用第一次的模型进行训练，由于感觉第一次模型训练的很好，应该还没到局部最优解，所以使用第一次的*dropout*的数值，但是使用更下的学习率使得它更加陷入到局部解中。

次数	模型序号	学习率	丢包率	迭代次数	批次	最小验证集loss	最优验证集精确度	最优精度迭代数
3	1	0.00001	0.75	100	32	0.3593	94%	13
3	2	0.00001	0.75	100	32	0.2397	97%	28
3	3	0.00001	0.75	100	32	0.3059	95%	3
3	4	0.00001	0.75	100	32	0.2197	96%	28

从以上看来这次训练也并不是非常好，在第一次训练后，应该已经达到一个局部最小点，这次是已第一次的模型为基础，然后调低了学习率来进行的。从训练结果可以看出精度并无大幅度提高，并且 $loss$ 却增大了。这并不是一个很好的结果。过拟合的现象也严重，这个不仅仅是调教 $dropout$ 可以解决的问题了。

11.4.4 第四轮*xception*进行迁移训练

从第三次实验可以看出模型3, 4的 $loss$ 比起第一次要小，所以第四次训练，1, 2两个模型将使用第一次实验的模型作为基础，而3, 4模型将采用第三轮的模型最为基础。因为在之前的训练中，都没有进行图像增强的方式，所以这次训练带入图像增强的方式，对原始图像随机进行反转，剪切，左右颠倒等操作。

对样本数据的增强

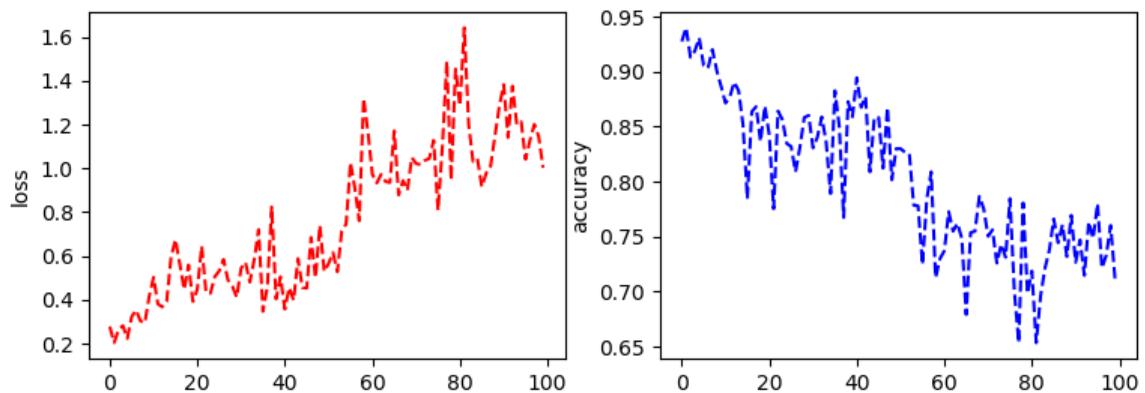
```
1.      train_transforms = transforms.Compose([
2.          transforms.CenterCrop(CROP_SIZE),
3.          #transforms.RandomAffine(degrees=10, translate=(0.1, 0.3)),
4.          # transforms.RandomHorizontalFlip(),
5.          # transforms.RandomResizedCrop(IMG_SIZE),
6.          transforms.Resize(IMG_SIZE),
7.          transforms.ToTensor(),
8.          transforms.Normalize(IMG_MEAN, IMG_STD)
9.      ])
```

次数	模型序号	学习率	丢包率	迭代次数	批次	最小验证集 $loss$	最优验证集精确度	最优精度迭代数
4	1	0.0001	0.75	100	32	0.2037	94%	2
4	2	0.0001	0.75	100	32	0.2397	97%	0
4	3	0.0001	0.75	100	32	0.3059	95%	0
4	4	0.0001	0.75	100	32	0.2197	96%	0

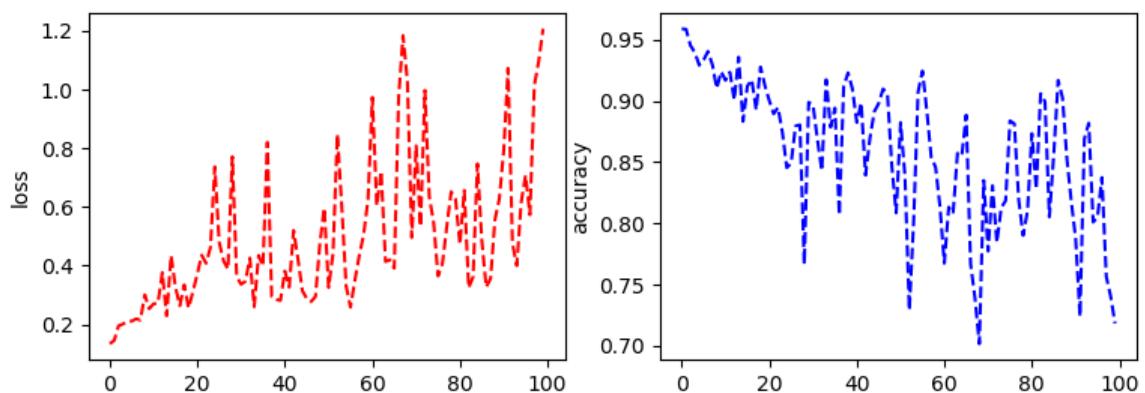
从以上数据来看，增强数据集后，模型并不是非常适应。基本最好的结果还是上一个基础模型。

下面展出了四个模型的 $loss$ 与精度图

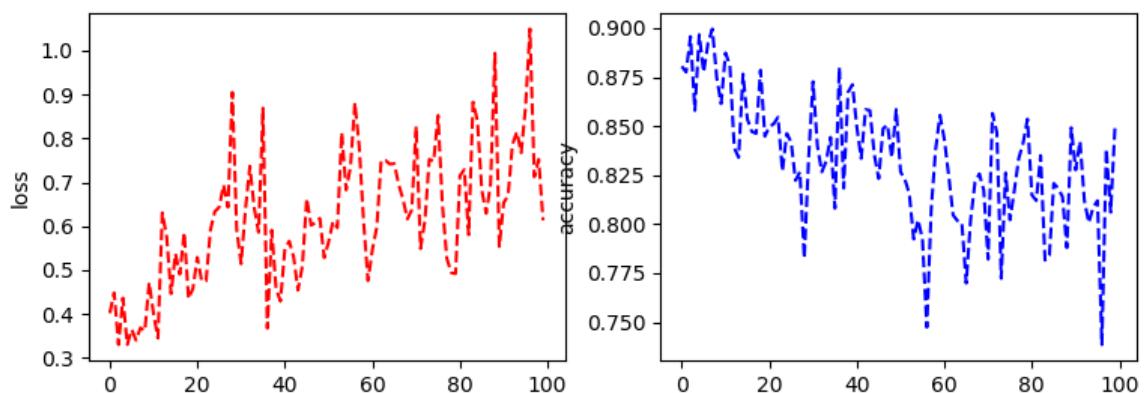
模型1



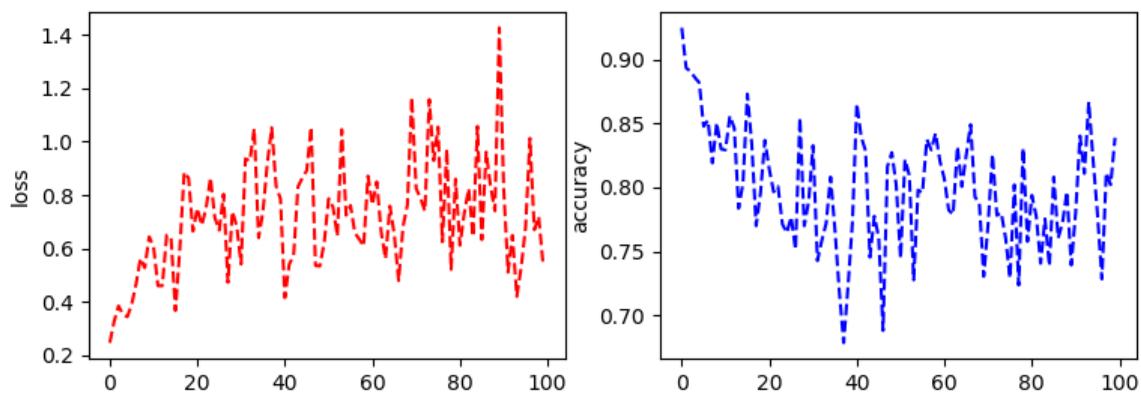
模型2



模型3



模型4



11.4.5 第五轮 *xception* 进行迁移训练

从第四次实验可以看到，图像增强后，并没有大幅度的提高，从日志文件中可以看到，训练集的 *loss* 都很高，所以这里再次进行一调节。将学习率降低，并且 *dropout* 率降低，进行尝试。

次数	模型序号	学习率	丢包率	迭代次数	批次	最小验证集 <i>loss</i>	最优验证集精确度	最优精度迭代数
5	1	0.00005	0.5	100	32	0.2037	94%	-
5	2	0.00005	0.5	100	32	0.2397	97%	-
5	3	0.00005	0.5	100	32	0.3059	95%	-
5	4	0.00005	0.5	100	32	0.2197	96%	-

这里没有任何一个模型超过上一次训练的验证精度。下一次实验准备换取一个新的模型进行训练。再此之前将这个结果提交到 *kaggle* 进行评分。

kaggle 评分结果

Private Score: 0.48936, Public Score: 0.59652

11.4.6 Progressive Neural Architecture Search 论文学习

由于本项目接下来要使用 *pnasnet5* 这种大型网络模型，所以对此原始论文进行了研究。神经网络结构搜索是谷歌的 *AutoML* 的一个具体分支。约翰斯霍普金斯大学刘晨曦博士和

Alan Yellie教授，以及Google AI的李飞飞、李佳等多名研究者提出渐进式神经网络结构搜索技术，论文被ECCV 2018接收作为Oral。

论文提出了一种比之前的方法更高效的用于学习**CNN**结构的方法：没有使用强化学习（**RL**）或遗传算法（**GA**），而是使用了基于序列模型的优化（**SMBO**）策略，我们在其中按复杂度逐渐增大的顺序搜索架构，同时学习一个用于引导该搜索的代理函数（*surrogate function*），类似于A*搜索。此方法在**CIFAR-10**数据集上找到了一个与 Zoph *et al.* (2017) 的强化学习方法有同等分类准确度（3.41% 错误率）的**CNN**结构，但速度却快2倍（在所评估的模型的数量方面）。我们的方法的表现也优于 Liu *et al.* (2017) 的遗传算法方法，它得到的模型的表现更差（3.63% 错误率），而且耗时还长5倍。最后我们还表明我们在**CIFAR**上学习到的模型也能在**ImageNet**分类任务上取得良好的效果。尤其是我们得到的表现足以与当前最佳水平媲美：*top-1*准确度达 82.9%，*top-5*准确度达 96.1%。

该算法的研究基础是Zoph *et al.* (2017) 所提出的**结构化搜索空间**，其中搜索算法的任务是搜索优良的**卷积单元**，而不是搜索完整的**CNN**。一个单元包含**B**个**模块**，其中一个模块是一个应用于两个输入（**张量**）的组合算子（比如加法），这两个输入在被组合之前都可以进行变换（**比如使用卷积**）。然后将这种单元结构堆叠一定的次数，具体数量取决于训练集的大小以及我们想要的最终模型的运行时间。尽管结构化空间能显著简化搜索过程，但可能的单元结构的数量仍然非常巨大。因此，在更高效地搜索这一空间的研究上还存在很多机会。

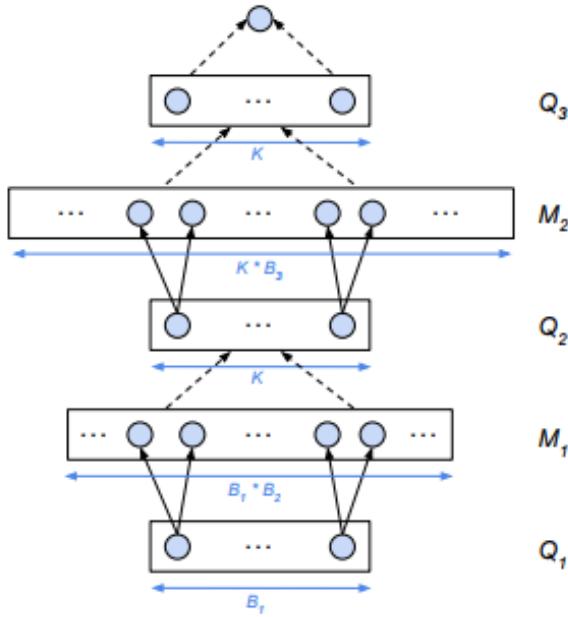
此方法类似于A*算法，从简单到复杂搜索模型空间，并在前进过程中剪枝处理掉没有前途的模型。这些模型（单元）按照它们所包含的模块的数量进行排序。从带有一个模块的单元开始。评估这些单元（通过训练它们并在一个验证集上计算它们的损失，然后使用观察得到的奖励来训练一个基于**RNN**的启发式函数（也被称为代理函数），其可以预测任何模型的奖励。通过**RNN**模型可以使用这个学习到的启发式函数来决定应该评估哪些带有2个模块的单元。在对它们进行了评估之后，再对这个启发式函数进行更新。重复这一过程，直到找到带有所想要的模块数量的优良单元。

通过渐进式（从简单到复杂）方法有一些优点。

1. 简单模型的训练速度更快，所以可以快速得到一些用于训练该代理函数的初始结果。
2. 仅要求该代理函数预测模型的质量，而这些模型与它之前已经见过的模型仅有少许不同（参考信赖域方法（*trust-region method*））。
3. 将搜索空间分解成了更小的搜索空间的积。

总结来说，该论文提出了一种用于**CNN**结构学习的方法，该方法的效率是之前最好方法的大约2倍，同时也实现了同等质量的结果。

渐进式神经架构搜索 (PNAS)



在上图所展示的是当最大模块数量 $B = 3$ 时的 PNAS 搜索过程图示。其中 Q_b 表示有 b 个模块的候选单元的集合。从所有单元有 1 个模块开始，即 $Q_1 = B_1$ ；训练和评估了所有这些单元，并更新了该预测器。在第 2 轮迭代时，我们扩展 Q_1 中的每个单元使所有单元都有 2 个模块，即 $M_2 = B_{1:2}$ ；预测它们的分数，选择其中最好的 K 个得到 Q_2 ，然后训练和评估它们并更新预测器。在第 3 轮迭代中，扩展 Q_2 中的每个单元，得到带有 3 个模块的单元的一个子集，即 $M_3 \subseteq B_{1:3}$ 预测它们的分数，选择其中最好的 K 个得到 Q_3 ，然后训练和评估它们并返回优胜者。蓝色横线表示每个集合的大小，其中 $B_b = |B_b|$ ，是在层级 b 处可能模块的数量， K 是束的大小。

算法流程

Input :

F 在第一层过滤器的数量

N 单元展开的数量

B 每个单元块的最大数量

K 每次迭代模型评估的最大数量

$trainSet$ 训练集

$valSet$ 验证集

begin

$Q_1 = B_1$ 设置模型的一个块

$pred = init - predictor()$ 代理预测器(RNN)

```

for  $b = 1 : B - 1$  do
    // 训练并且评估当前在  $Q$  中的模型
     $D_b = []$  // 保存评估模型与分数列表
    for  $m \in Q_b$  do
         $model = train(unroll(m, F, N), trainSet)$ 
         $score = eval(models[m], valSet)$ 
         $D_b.push(model, score)$ 
    end for

    // 基于新的数据更新预测器
     $pred = pred.update(D_b)$ 

    // 扩展并且预测搜索评估
     $M_{b+1} = []$  // 保存候选者模型
     $S_{b+1} = []$  // 保存预测分数
    for  $m \in Q_b$  do
         $C = expand-by-one-block(m)$  // 扩展
        for  $c \in C$  do
             $M_{b+1}.push(c)$ 
             $S_{b+1}.push(pred.predict(c))$ 
        end for
    end for

    // 挑选出最优秀的候选者模型
     $Q_{b+1} = top-K(M_{b+1}, S_{b+1}, K)$ 
end for

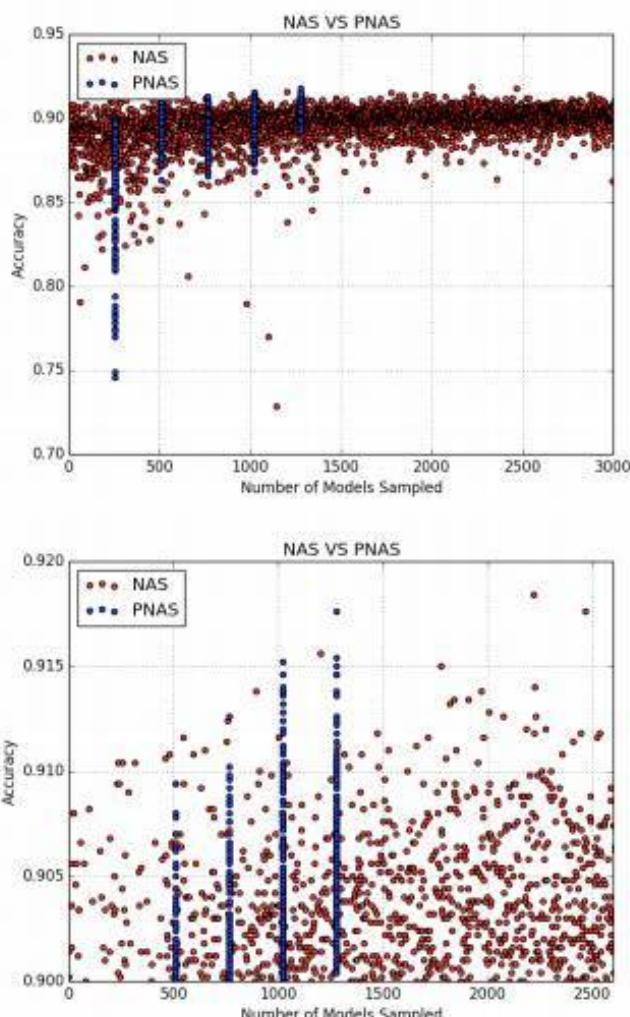
 $Return top-K(M_b, S_b, 1)$ 
end

```

使用编码器**RNN**进行准确度预测

论文众选择使用一个循环神经网络（**RNN**）作为预测器，因为**RNN**很准确而且可以轻松应对大小会改变的输入；此外监督式学习方法比强化学习方法要远远更具样本效率。

PNAS的效率



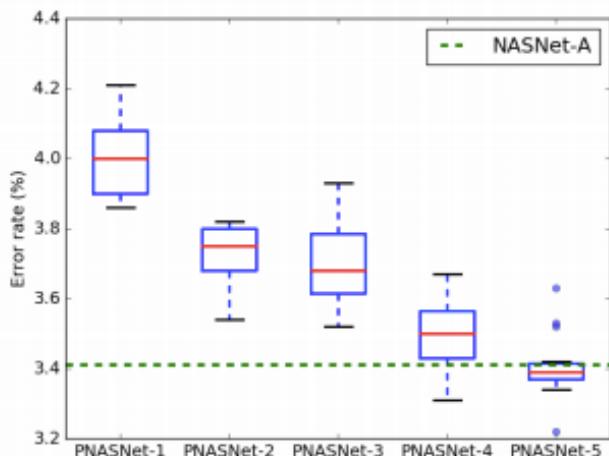
上图是**NAS**和**PNAS**算法的相对效率比较。蓝色是**PNAS**访问的前1280个模型在**CIFAR-10**验证集上的验证准确度，红色是**NAS-RL**访问的前3000个模型在**CIFAR-10**验证集上的验证准确度。下图是上图的局部放大。**PNAS**的结果呈现出 $B = 5$ 个集群，反映了该算法的 B 个阶段。**PNAS**经过1280个模型之后得到了0.917的top-1准确度、0.915的平均top-5准确度和0.912的平均top-25准确度。**NAS-RL**在经过2280个模型之后得到了0.917的top-1准确度、0.915的平均top-5准确度和0.913的平均top-25准确度。注意这些架构搜索算法的平均top准确度是平滑变化的（有一些随机波动）

在 **CIFAR-10** 图像分类任务上的结果

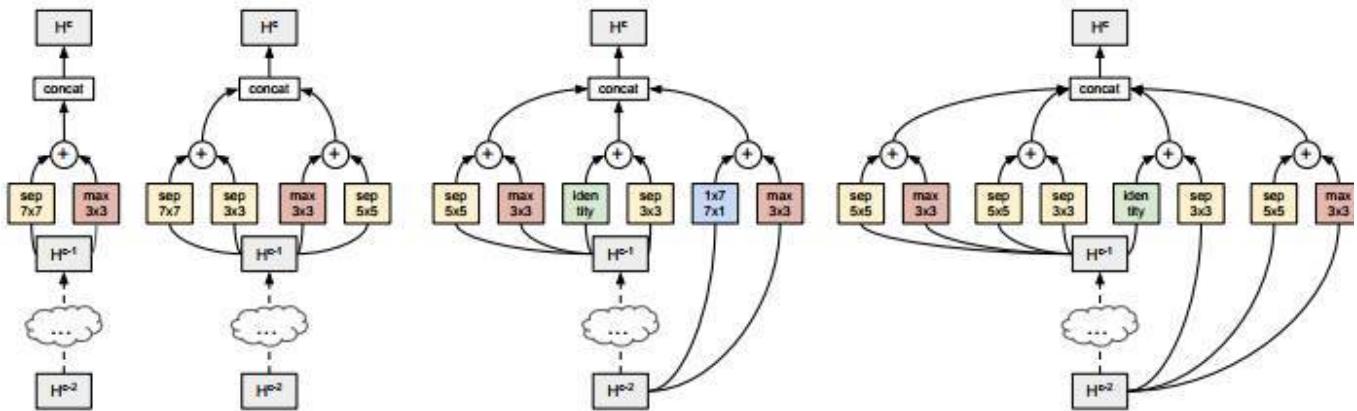
Method	Algo.	Error	Params	Models
NASNet-A ($N = 6, F = 32$) (Zoph et al., 2017)	RL	3.41	3.3M	20,000
NASNet-B ($N = 4$) (Zoph et al., 2017)	RL	3.73	2.6M	20,000
NASNet-C ($N = 4$) (Zoph et al., 2017)	RL	3.59	3.1M	20,000
Hierarchical ($N = 2, F = 128$) (Liu et al., 2017)	EA	3.63 ± 0.10	61.3M	7000
Hierarchical ($N = 2, F = 64$) (Liu et al., 2017)	EA	3.75 ± 0.12	15.7M	7000
Hierarchical ($N = 2, F = 64$) (Liu et al., 2017)	Random	3.91 ± 0.15	14.1M	7000
Hierarchical ($N = 2, F = 64$) (Liu et al., 2017)	Random	4.04 ± 0.2	14.1M	200
PNASNet-5 ($N = 3, F = 48$)	SMBO	3.41 ± 0.09	3.2M	1280
PNASNet-4 ($N = 4, F = 44$)	SMBO	3.50 ± 0.10	3.0M	1024
PNASNet-3 ($N = 6, F = 32$)	SMBO	3.70 ± 0.12	1.8M	768
PNASNet-2 ($N = 6, F = 32$)	SMBO	3.73 ± 0.09	1.7M	512
PNASNet-1 ($N = 6, F = 44$)	SMBO	4.01 ± 0.11	1.6M	256

一些最近的结构学习方法在**CIFAR-10** 上的结果；最下面的方法来自该论文。**Algo** 一列是这些方法所用的对应**搜索算法的类型**（**RL**=*强化学习，****EA**=进化算法。**SMBO**=基于序列模型的优化）。**Error** 一列是对应最佳模型的 *top-1* 分类错误率。**Params** 一列是对应的参数数量。**Models** 一列是为了寻找最佳模型而训练和评估的模型的数量。其中错误率以 $\mu \pm \sigma$ 的形式给出，其中 μ 是在15次拟合和评估每个模型的随机试验上的平均验证表现， σ 是标准差。

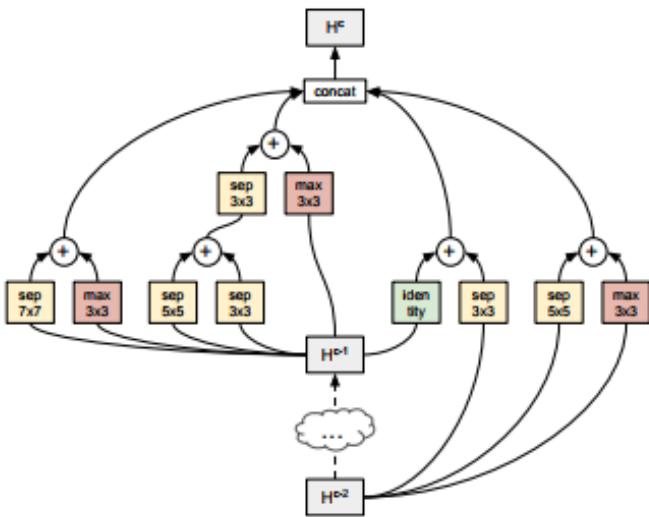
更小模型的表现



在 $b \in \{1, 2, 3, 4, 5\}$ 时，PNAS 找到的在**CIFAR-10** 上的最佳模型的测试表现的箱线图。论文训练和测试了每个模型15次，每次600迭代；所以质量的范围（纵轴）是由参数初始化中的随机性、**SGD** 优化过程等造成的。其中的横线是**NAS-RL** 单元方法所找到的最佳模型的表现，这种方法的一个单元也是5个模块。



上图为PNASNet- $\{1, 2, 3, 4\}$ 中所用的单元的结构



上图为PNASNet-5中所用的单元的结构

剪枝过程的有效性

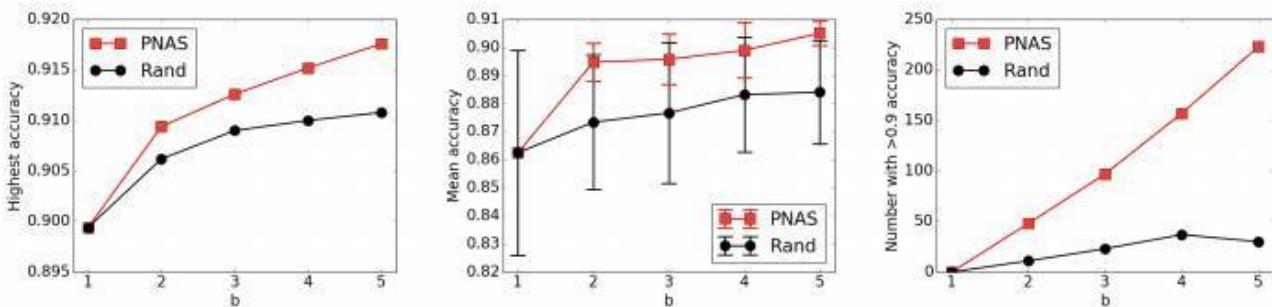


Figure 7. Measuring the effectiveness of the pruning process. We plot three metrics: (a) highest accuracy, (b) mean accuracy (with error bars), and (c) the number of models with > 0.9 accuracy within the $K = 256$ models at each level b . “PNAS” is our algorithm. “Rand” means randomly selecting K models from $\mathcal{B}_{1:b}$.

测量剪枝过程的有效性。上图描绘了三个指标：

- (a) 最高准确度。
- (b) 平均准确度 (带有误差范围)
- (c) 在每个层级 b 的 $K = 256$ 个模型内准确度大于 0.9 的模型的数量。PNAS 是文中的算法。Rand 是从 $B(1 : b)$ 中随机选择的 K 个模型。

在 ImageNet 图像分类上的结果

Model	Params	Mult-Adds	Top 1 (%)	Top 5 (%)
MobileNet-224 (Howard et al., 2017)	4.2M	569M	70.6	89.5
ShuffleNet (2x) (Zhang et al., 2017)	5M	524M	70.9	89.8
NASNet-A ($N = 4, F = 44$) (Zoph et al., 2017)	5.3M	564M	74.0	91.6
PNASNet-5 ($N = 3, F = 54$)	5.1M	588M	74.2	91.9

上表中展示了在 Mobile 与 Large 分别设置下的 ImageNet 分类结果。

可参见原始的论文：[Progressive Neural Architecture Search](#)

以下是论文中给出的源代码实现：

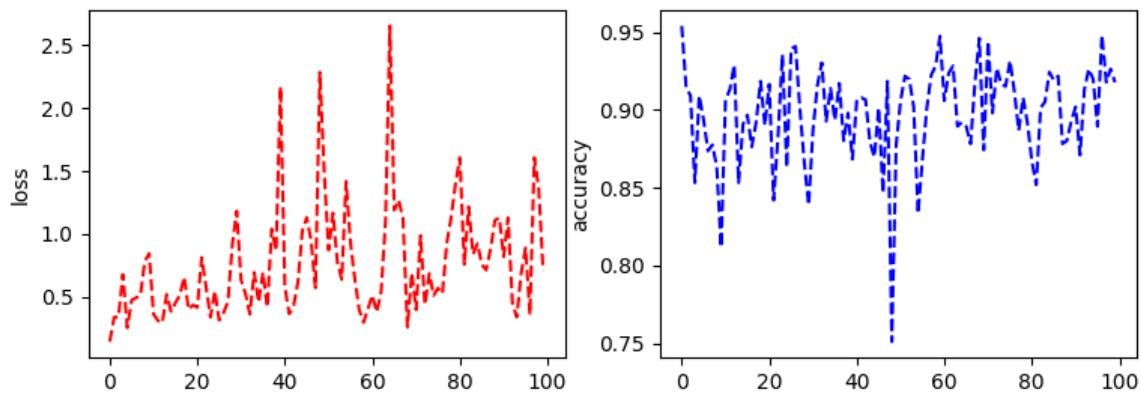
地址	说明
http://github.com/tensorflow/models/	<i>tensorflow</i> 的 <i>models</i> 模块中的实现
https://github.com/chenxi116/PNASNet.pytorch	作者的 <i>pytorch</i> 实现
https://github.com/chenxi116/PNASNet.TF	作者的 <i>tensorflow</i> 的实现

11.4.7 第一轮 *pnasnet5* 进行迁移训练

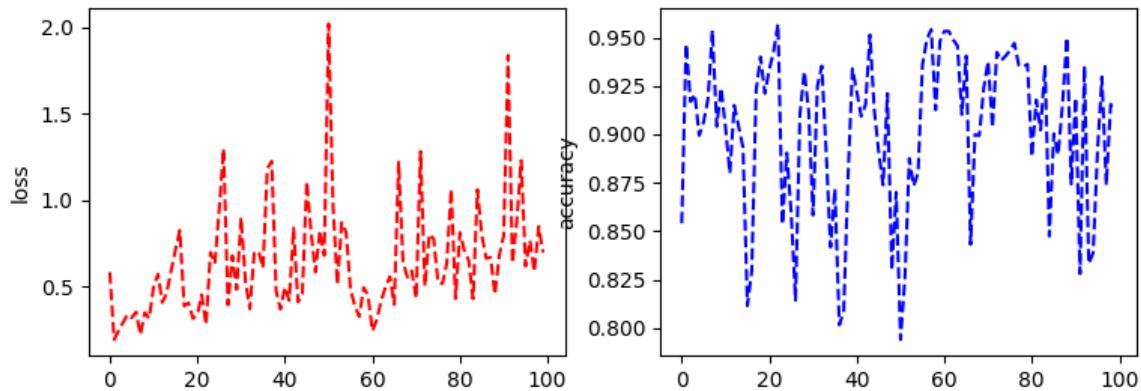
次数	模型序号	学习率	丢包率	迭代次数	批次	最小验证集 loss	最优验证集精确度	最优精度迭代数
1	1	0.0001	0.75	100	8	0.1457	95.39%	1
1	2	0.0001	0.75	100	8	0.2779	95.7%	23
1	3	0.0001	0.75	100	8	0.2522	94.76%	8
1	4	0.0001	0.75	100	8	0.3458	95.42%	86

从以上数据来看，增强数据集后，模型并不是非常适应。基本最好的结果还是上一个基础模型。

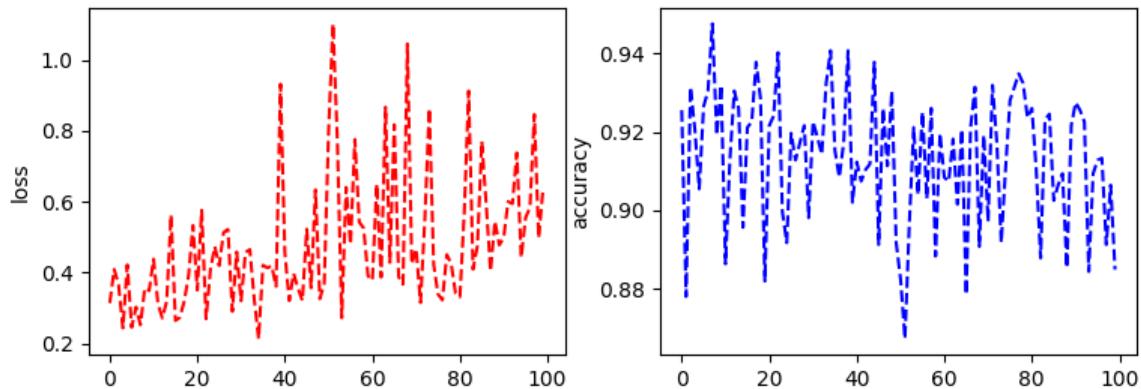
模型1



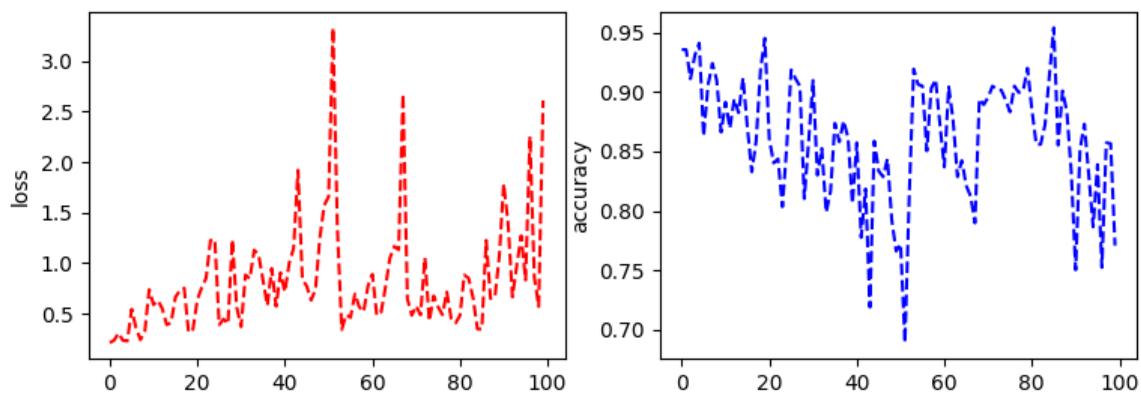
模型2



模型3



模型4



再下一论的训练中调低学习率，尝试在已经收敛的解中提高精度。

11.4.8 第二轮 *pnasnet5* 进行迁移训练

次数	模型序号	学习率	丢包率	迭代次数	批次	最小验证集 loss	最优验证集精确度	最优精度迭代数
2	1	0.00001	0.75	100	8	0.1457	95.39%	-
2	2	0.00001	0.75	100	8	0.2422	96.5%	1
2	3	0.00001	0.75	100	8	0.2455	95.24%	11
2	4	0.00001	0.75	100	8	0.3458	95.42%	-

从结果来看基本上是收敛状态了。准备提交到 *kaggle* 评分。

kaggle 的预测结果

Private Score: 0.39471, Public Score: 0.51678

kaggle 的评分是使用 *loss* 进行的，所以我准备再次尝试使用最小 *loss* 的模型保存模型，而不是最高精度。

11.4.9 第三轮 *pnasnet5* 进行迁移训练

这次训练不依赖第二轮的模型，单独进行训练。以一个比较通常的学习率与丢弃率开始。

次数	模型序号	学习率	丢包率	迭代次数	批次	最小验证集loss	最优验证集精确度	最优精度迭代数
3	1	0.0001	0.75	20	8	0.1457	95.39%	-
3	2	0.0001	0.75	20	8	0.2422	96.5%	1
3	3	0.0001	0.75	20	8	0.2455	95.24%	11
3	4	0.0001	0.75	20	8	0.3458	95.42%	-

kaggle的预测结果

Private Score: 0.39471, Public Score: 0.51678, 这个分数比上次要好些。

11.4.10 第四轮pnasnet5进行迁移训练

这次训练依赖第三轮的模型，但是进行增强图像样本的。在第三轮训练中，使用了较高的dropout，因为样本是随机增强的，这样样本本身就存在变动，所以这次训练我增高了dropout率到0.5。

次数	模型序号	学习率	丢包率	迭代次数	批次	最小验证集loss	最优验证集精确度	最优精度迭代数
4	1	0.00001	0.5	20	8	0.1177	96.97%	5
4	2	0.00001	0.5	20	8	0.1639	95.98%	9
4	3	0.00001	0.5	20	8	0.1865	93.44%	15
4	4	0.00001	0.5	20	8	0.2248	93.86%	3

kaggle的预测结果

Private Score: 0.31896, Public Score: 0.35224, 还差目标50名的样子。通过这个结果，我想可以加大样本的增加的效果。

11.4.11 第五轮pnasnet5进行迁移训练

依托第四轮训练，进一步对图片进行增强尝试训练。这里我发现一个问题是，我在使用增强时，使用了CenterCrop函数，并且使用了(320, 480)进行切割，但是我发现，在几个样本上

查看切割结果，发现有些切多了。所以这次训练我使用了上次的训练参数，但是在增强时舍弃了这个切割函数，直接使用原始图片进行训练。几次尝试后，并没有超过上次的最好成绩。这里就不列出多余的实验结果了。

我想在对图片进行一些预处理操作减少一些不必要的噪声信息。基于上次使用的是自己编写的掩码图，这次使用**梯度图**来进行对图片进行预处理。调用了`opencv`中对梯度的实现。在文件`ddd_units/image_preprocessing.py`源文件中进行了编写。

```
1. def calc_image_gradient_inside(image, X_weight=0.5, Y_weight=0.5):
2.     image = np.array(image)
3.
4.     x = cv2.Sobel(image, cv2.CV_16S, 1, 0)
5.     y = cv2.Sobel(image, cv2.CV_16S, 0, 1)
6.
7.     absX = cv2.convertScaleAbs(x) # 转回uint8
8.     absY = cv2.convertScaleAbs(y)
9.
10.    dst = cv2.addWeighted(absX, X_weight, absY, Y_weight, 0)
11.    return dst
```

上述函数使用了`sobel`因子来进行梯度图的计算。重新对模型进行训练，学习率与`dropout`都使用了最通常的设定**0.0001**与**0.5**的配置。计算过后这次的评分也不是非常理想，最差的模型在**88%**，最好的模型也只在**93%**，随后又进行了多次调教。但是都不是非常理想。

11.4.12 第六轮`pnasnet5`进行迁移训练

在`pytorch`中提供了对样本增强的另外一个工具就是**`FiveCrop`**，这个函数可以对图片进行**5**种角度的切割，这里我将原本的训练样本进行扩充。在源文件"`ddd_units/image_preprocessing.py`"中的`generate_five_crop`函数中实现。这样我的训练样本就比原来扩充了**5**倍。

```
1. def generate_five_crop(image_path):
2.     image_path = os.path.abspath(image_path)
3.     dirname = os.path.dirname(image_path)
4.     filename = os.path.basename(image_path)
5.     filename = filename[0:filename.rfind('.')]
6.
7.     five_crop = transforms.FiveCrop(CROP_SIZE)
```

```
8.     image = Image.open(image_path)
9.     images = five_crop(image)
10.    for i, img in enumerate(images):
11.        img_path = os.path.join(dirname, filename)
12.        img_path += '_' + str(i+1) + '.jpg'
13.        img.save(img_path)
```

随后开启了随机增强的选项进行训练，并设定学习率**0.0001**，丢弃率**0.5**，可能是因为样本做了随机增强的原因。这轮训练效果非常不好，平均的精度只有**90%**。重新按照这个精度进行训练，但是去掉了随机增强的训练选项。效果仍然不是非常好。

11.4.13 结果融合

决定使用更多的模型进行*bagging*操作。因为目前只有**4**块显卡，所以这里采用了分两次进行*kfold_bagging*的方式，在源文件"*ddd_units/merge_result.py*"中实现了两个*kaggle result*文件的合并。通过均值将两个文件结果合并到一起，所以这次用**8**个模型来做*bagging*。两个模型都通过第六轮的方式进行训练。拿**第四轮与第三轮**的结果做融合。得到了**kaggle**分数为：*Private Score: 0.27005, Public Score: 0.28016*。

我又使用了*xception*进行训练，这次得到了单体模型在**98%**的模型，可能之前图像剪切多了。造成了精度下降。再次利用融合的方式，反复进行了几次提交。以下是近期的*kaggle*分数。

result_5.zip 21 hours ago by gA4ss add submission details	0.26581	0.27922	<input type="checkbox"/>
result_4.zip 21 hours ago by gA4ss add submission details	0.27385	0.28117	<input type="checkbox"/>
result_3.zip 21 hours ago by gA4ss add submission details	0.26660	0.29348	<input type="checkbox"/>
result_2.zip 21 hours ago by gA4ss add submission details	0.26266	0.28252	<input type="checkbox"/>
new_result.zip 21 hours ago by gA4ss add submission details	0.27335	0.29998	<input type="checkbox"/>

12 总结

我对神经网络的数学原理非常感兴趣，很想在这方面进行深入的研究。但是这次实战项目确实给我了很大的教训。深度学习在训练模型方面还是要靠不断的实验才可以取得更高的精度。毕竟不是考单纯的理论解决的问题。

13 后续工作

之后我想尝试一种新的模型，这种模型的思路借鉴自**bagging**训练方式，先用一个大的卷积网提取通用特征，随后每一个类型后跟一个小的卷积网络来提取特定图像的特征。最终每个小卷积网络链接自己的全连接层，这个全连接层最终只输出一个概率值，表示所属分类的概率。最终概率值最大优胜。

14 未解决的疑问

由于这个项目，让我产生以下几个待解决的问题：

1. 反卷积层的权重是否不需要训练，可以直接通过对应卷积层的权重的逆替代

卷积运算相当于一个非线性方程组，那么通过对调整好的权重进行运算 $\mathbf{W}\mathbf{x} = \mathbf{y}$ ，由于自由变量的个数(特征)比方程个数(样本)多。所以有多个解，这也是泛化的原理基础。而反卷积相当于知道 \mathbf{y} ，这时是否可以通过求 \mathbf{W} 的逆或者伪逆来 \mathbf{W}^{-1} 来直接进行输出。

直接使用梯度反向调节当网络收敛后应该会逼近卷基层的权重的逆。

2. 利用高斯分布对相同类型样本做图像生成

由于利用特征图做训练产生了过快的拟合，所以就想如果模型看多了图片。那么精确度会提升，那么我利用现有训练集的每个像素点，做高斯分布的统计。那么通过这个模型随机输出图片是否能达到增强的效果。这里的一个优化我想利用池化思想，并不用每个像素取分布。而是利用一个过滤器来取。

3. 使用特征图直接训练加快收敛速度以及过拟合原因的数学原理

通过反卷积后的特征图基本会在第一轮后就过拟合。这个从表面来看因用转置得到的图像是特征图，用特征图在与原输出叠加。相当于把训练集的重要特征放大了。所以再调整权的时候，对这些放大的特征给予更多的调节。而激励函数选取 $ReLU$ 把那些小的输出为 0。泛化能力降低。但对训练过的图片记忆增强。但这仅仅是我猜测，需要找到背后真正得原因。

4. 迁移学习成立的数学原理

因为用别人训练好的模型输出的 *mask* 图也能反映所要关注的图像区域。但是在之前别人训练中却使用了不同的样本，这些样本的图像轮廓与当前的图像样本明显不同。但是为什么还是有一定的有效性。这里神经网络能学习到的并不是目标样本的轮廓，而是对颜色数值的尤其是聚类连续数值的学习，例如将 **245, 240, 230, ...** 差不多大小的值统一归为一类。所以才可以在迁移学习中起到作用。在迁移学习中提到一个原理就是，如果当前的训练样本与之前的训练集相差不大，则直接调整分类层。这里我想可以加个定语是如果 **当前训练样本的颜色分布与之前的训练集相差不大，则直接调整分类层。**

这也可从神经网络学习的目标的梯度这一观点进行解释。

我还是相信利用掩码可以剔除图像的噪声使得图像识别的精度可以得到提高。之后会在这个点上继续进行研究。

5. *dropout* 的数学原理

感觉可以从 **亚定方程组** 的解这个角度去思考 *dropout* 为何可以选取到更合适的线性组合。

参考资料

1. Karen Simonyan* & Andrew Zisserman+V.:VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION
2. Kaiming He Xiangyu Zhang Shaoqing Ren Jian Sun.: DeepResidualLearningforImageRecognition
3. Wei Liu1, Dragomir Anguelov2, Dumitru Erhan3, Christian Szegedy3, Scott Reed4, Cheng-Yang Fu1, Alexander C. Berg1.: SSD:SingleShotMultiBoxDetector
4. Uijlings, J.R., van de Sande, K.E., Gevers, T., Smeulders, A.W.: Selective search for object recognition. IJCV (2013)
5. Ren, S., He, K., Girshick, R., Sun, J.: Faster R-CNN: Towards real-time object detection with region proposal networks. In: NIPS. (2015)
6. He,K.,Zhang,X.,Ren,S.,Sun,J.: Deepresiduallearningforimagerecognition. In:CVPR. (2016)
7. Redmon, J., Divvala, S., Girshick, R., Farhadi, A.: You only look once: Unified, real-time object detection. In: CVPR. (2016)
8. Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Alan Yuille, Jonathan Huang, Kevin Murphy.: Progressive Neural Architecture Search (2018)
9. https://blog.csdn.net/weixin_40170902/article/details/80092628
10. <https://blog.csdn.net/zhangjup3/article/details/80467350>