DeepLearning

模型部署

1、硬件环境

1、服务器（网络）

2、PC

3、IOT（嵌入式设备，板卡等）

4、移动端（Android，IOS）

2、部署平台

Android

参考链接：https://pytorch.org/mobile/android/

IOS

参考链接：https://pytorch.org/mobile/ios/

3、模型调用方式

Python Flask 方式：

参考链接：https://pytorch.org/tutorials/intermediate/flask_rest_api_tutorial.html

C++方式：

参考链接：https://pytorch.org/tutorials/advanced/cpp_export.html

4、模型打包关键代码

```
# 打包模型
```

```
model=torchvision.models.resnet18()
model.eval() #  一定要有这句
example=torch.rand(1,3,224,224)       # N 可以变，CHW 固定
module=torch.jit.trace(model,example)
module.save(file_path)

#加载模型
module = torch.jit.load(module_path)
```

5、ONNX

ONNX 介绍

ONNX：是一种针对机器学习所设计的开放式的文件格式，用于存储训练好的模型。它使得不同的人工智能框架（如 Pytorch, MXNet）可以采用相同格式存储模型数据并交互。

ONNX RUNTIME：是一种用于将 ONNX 模型部署到生产环境的高性能推理引擎。它针对云和 Edge 进行了优化，适用于 Linux、Windows 和 Mac。 它使用 C++ 编写，还包含 C、Python、C#、Java 和 Javascript (Node.js) API，可在各种环境中使用。 ONNX RUNTIME 同时支持 DNN 和传统 ML 模型，并与不同硬件上的加速器（例如，NVidia GPU 上的 TensorRT、Intel 处理器上的 OpenVINO、Windows 上的 DirectML 等）集成。 通过使用 ONNX 运行时，可以从大量的生产级优化、测试和不断改进中受益。

ONNX 打包及使用实例

参考链接：

https://pytorch.org/tutorials/advanced/super_resolution_with_onnxruntime.html

视频链接：https://www.lieweiai.net/center/190/713

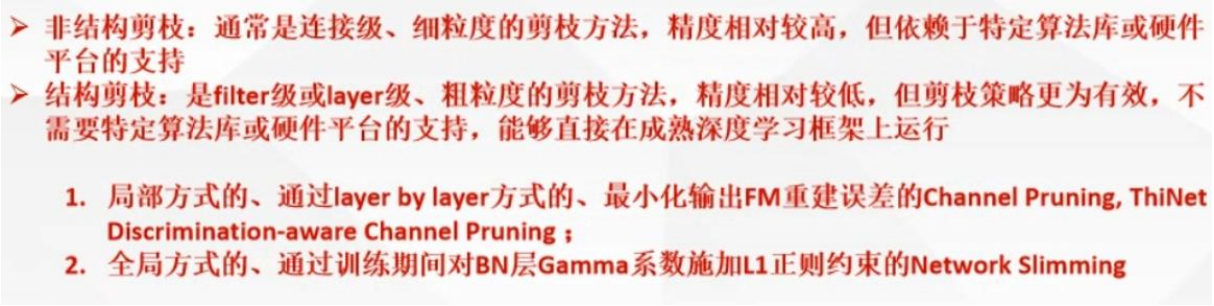
模型压缩

通常我们训练出的模型都比较大，将这些模型部署到例如手机、机器人等移动设备上时比较困难。模型压缩（model compression）可以将大模型压缩成小模

型，压缩后的小模型也能得到和大模型接近甚至更好的性能。

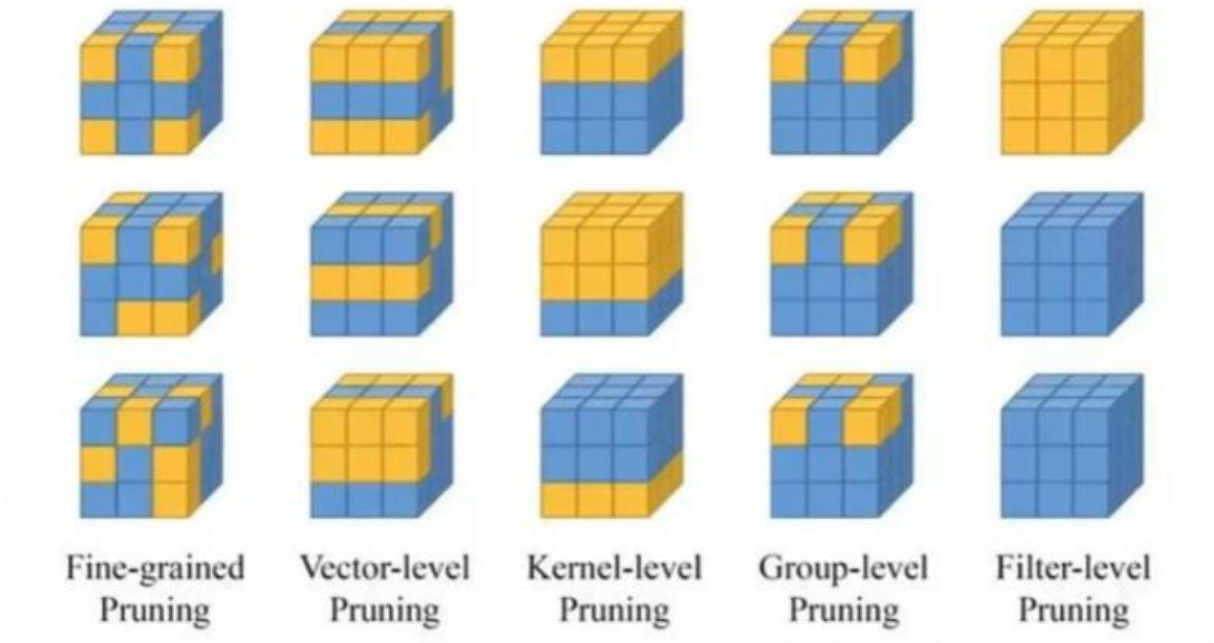
1、剪枝

剪枝方式：



注：结构化剪枝为剪掉某一层神经元，非结构化剪枝为随机剪掉某一些神经元

CNN 剪枝：



注：最右为结构化剪枝，其余为非结构化剪枝

剪枝操作实例：

参考链接：https://pytorch.org/tutorials/intermediate/pruning_tutorial.html

2、量化

学术界工作：

低精度（Low precision）可能是最通用的概念。常规精度一般使用 FP32（32 位浮点，单精度）存储模型权重；低精度则表示 FP16（半精度浮点），INT8（8 位的定点整数）等等数值格式。不过目前低精度往往指代 INT8。

混合精度（Mixed precision）在模型中使用 FP32 和 FP16 。 FP16 减少了一半的内存大小，但有些参数或操作符必须采用 FP32 格式才能保持准确度。

量化一般指 INT8 。不过，根据存储一个权重元素所需的位数，还可以包括：

二值神经网络：在运行时权重和激活只取两种值（例如 +1，-1）的神经网络，以及在训练时计算参数的梯度。

三元权重网络：权重约束为+1,0 和-1 的神经网络。

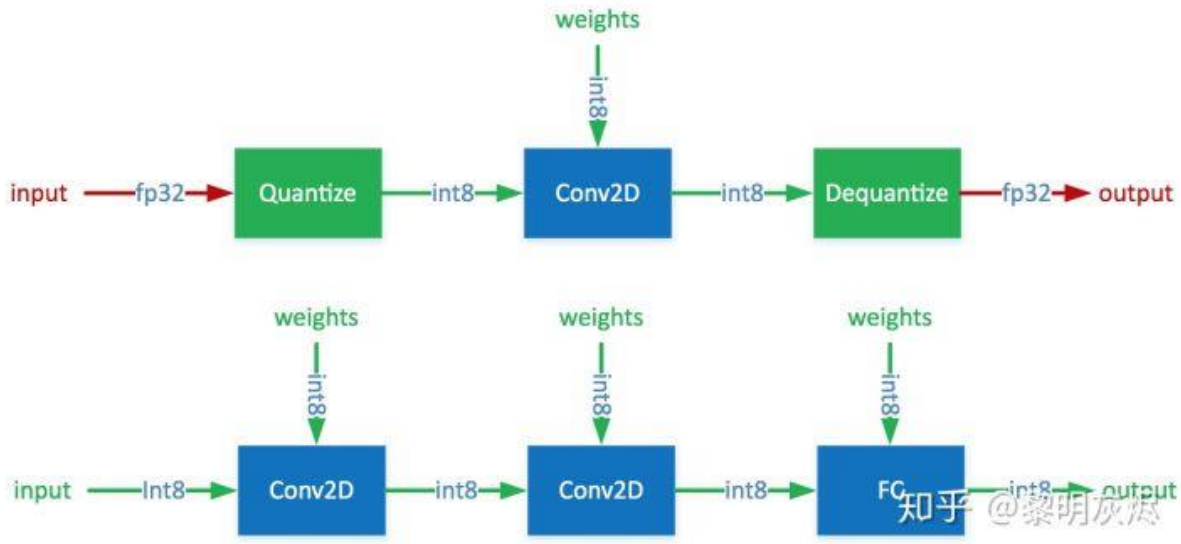
XNOR 网络：过滤器和卷积层的输入是二进制的。 XNOR 网络主要使用二进制运算来近似卷积。

工业界工作：

理论是一回事，实践是另一回事。如果一种技术方法难以推广到通用场景，则需要进行大量的额外支持。花哨的研究往往是过于棘手或前提假设过强，以至几乎无法引入工业界的软件栈。

工业界最终选择了 INT8 量化—— FP32 在推理（inference）期间被 INT8 取代，而训练（training）仍然是 FP32。TensorRT，TensorFlow，PyTorch，MxNet 和许多其他深度学习软件都已启用（或正在启用）量化。

通常，可以根据 FP32 和 INT8 的转换机制对解决方案进行分类。一些框架简单地引入了 Quantize 和 Dequantize 层，当从卷积或全链接层送入或取出时，它将 FP32 转换为 INT8 或相反。在这种情况下，如图四的上半部分所示，模型本身和输入/输出采用 FP32 格式。深度学习框架加载模型，重写网络以插入 Quantize 和 Dequantize 层，并将权重转换为 INT8 格式。



图四：混合 FP32/INT8 和纯 INT8 推理。红色为 FP32，绿色为 INT8 或量

量化 Tensor 代码：

```
# 量化
Q = torch.quantize_per_tensor(input,scale=0.025 , zero_point=0, dtype = torch.quint8)
# 反量化
Q = Q.dequantize()
```

代码说明：

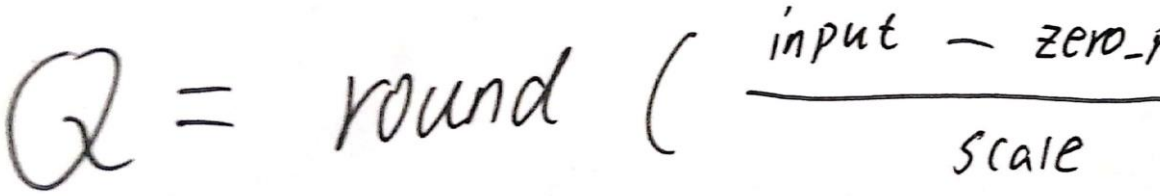
input 为准备量化的 32 位浮点数，Q 为量化后的 8 位定点数

dtype 为量化类型，quint8 代表 8 位无符号数，qint8 代表 8 位带符号数，最高位是符号位

假设量化为 qint8,设量化后的数 Q 为 0001_1101,最高位为 0（符号位），所以是正数；后 7 位转换为 10 进制是 29，所以 Q 代表的数为 ：zero_point + Q * scale = 0 + 29 * 0.025 = 0.725

所以最终使用 print 显示 Q 时，显示的不是 0001_1101 而是 0.725，但它在计算机中存储时，是 0001_1101

量化公式：

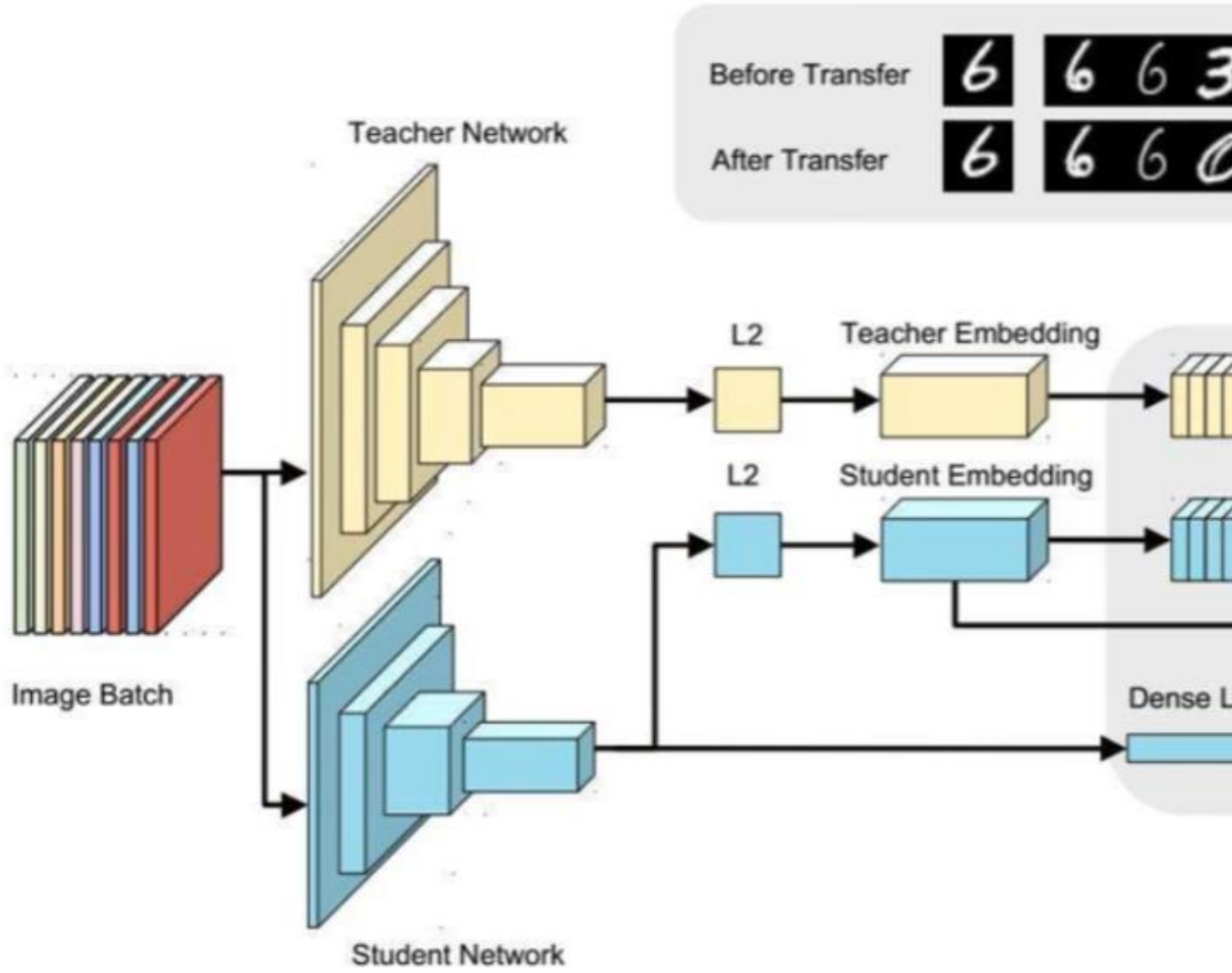


量化卷积网络模型：

参考链接：静态量化，动态量化

视频链接：https://www.lieweiai.net/center/190/713

3、蒸馏



提出背景

虽然在一般情况下，我们不会去区分训练和部署使用的模型，但是训练和部署之间存在着一定的不一致性:

在训练过程中，我们需要使用复杂的模型，大量的计算资源，以便从非常大、高度冗余的数据集中提取出信息。在实验中，效果最好的模型往往规模很大，甚至由多个模型集成得到。而大模型不方便部署到服务中去，常见的瓶颈如下:

1、推断速度慢

2、对部署资源要求高(内存，显存等)

在部署时，我们对延迟以及计算资源都有着严格的限制。

蒸馏关键点

在知识蒸馏时，由于我们已经有了一个泛化能力较强的 Net-T，我们在利用 Net-T 来蒸馏训练 Net-S 时，可以直接让 Net-S 去学习 Net-T 的泛化能力。

一个很直白且高效的迁移泛化能力的方法就是使用 softmax 层输出的类别的概率来作为“soft target”。

传统 training 过程(hard targets)：对 ground truth 求极大似然

KD 的 training 过程(soft targets)：用 large model 的 class probabilities（softmax 的输出值）作为 soft targets，当 softmax 输出值较小时，其所携带的信息量很小可忽略不计，此时便引入温度系数 T 来平衡，以下公式时加了温度 T 之后的 softmax 公式：

$$q_i = \frac{exp\left(z_i/T\right)}{\sum_j exp\left(z_j/T\right)}$$

即是求两个损失，一个 soft targets，一个 hard targets。

另：温度 T 的选取？

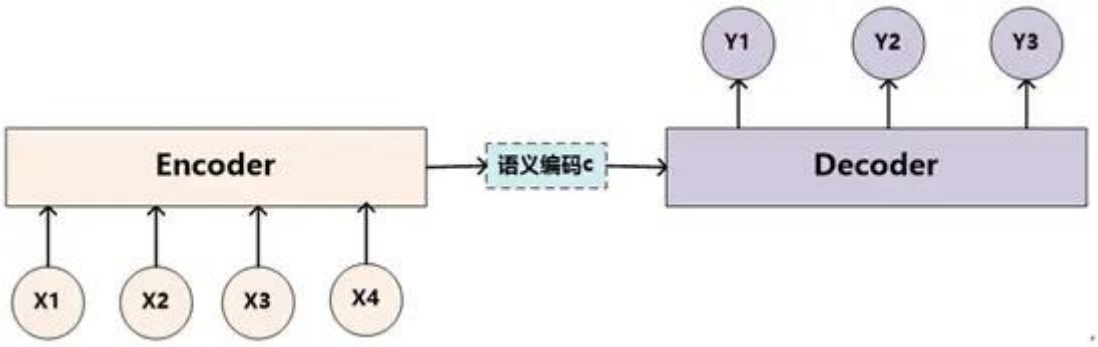
参考博客：https://nlplearning.blog.csdn.net/article/details/107744390

视频地址：https://www.lieweiai.net/center/190/713（第五课）

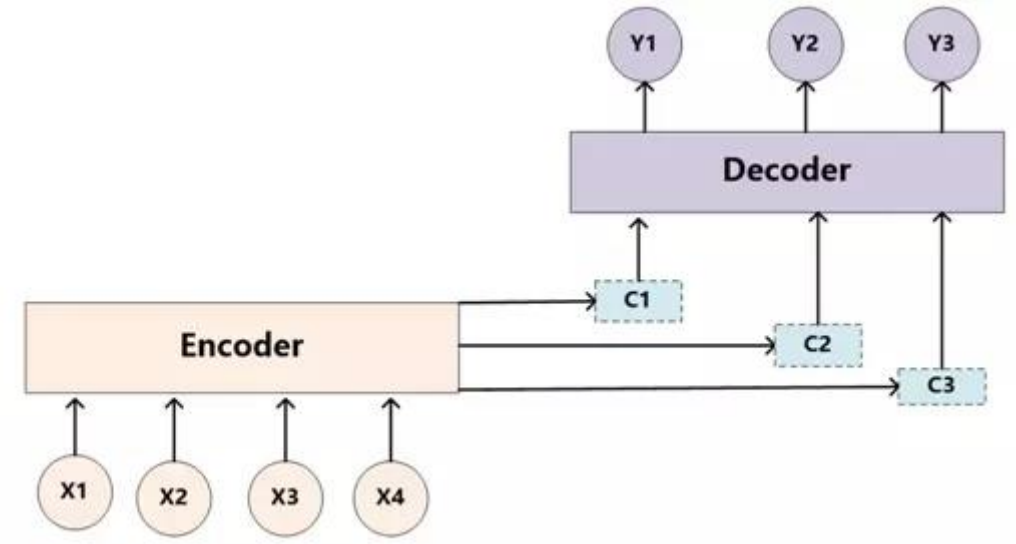
注意力机制

编解码结构的注意力机制

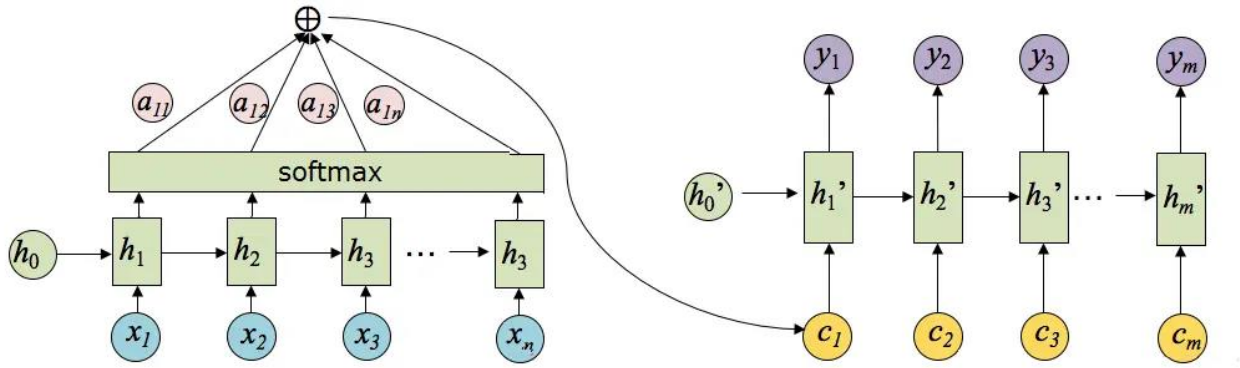
1、普通编解码模型



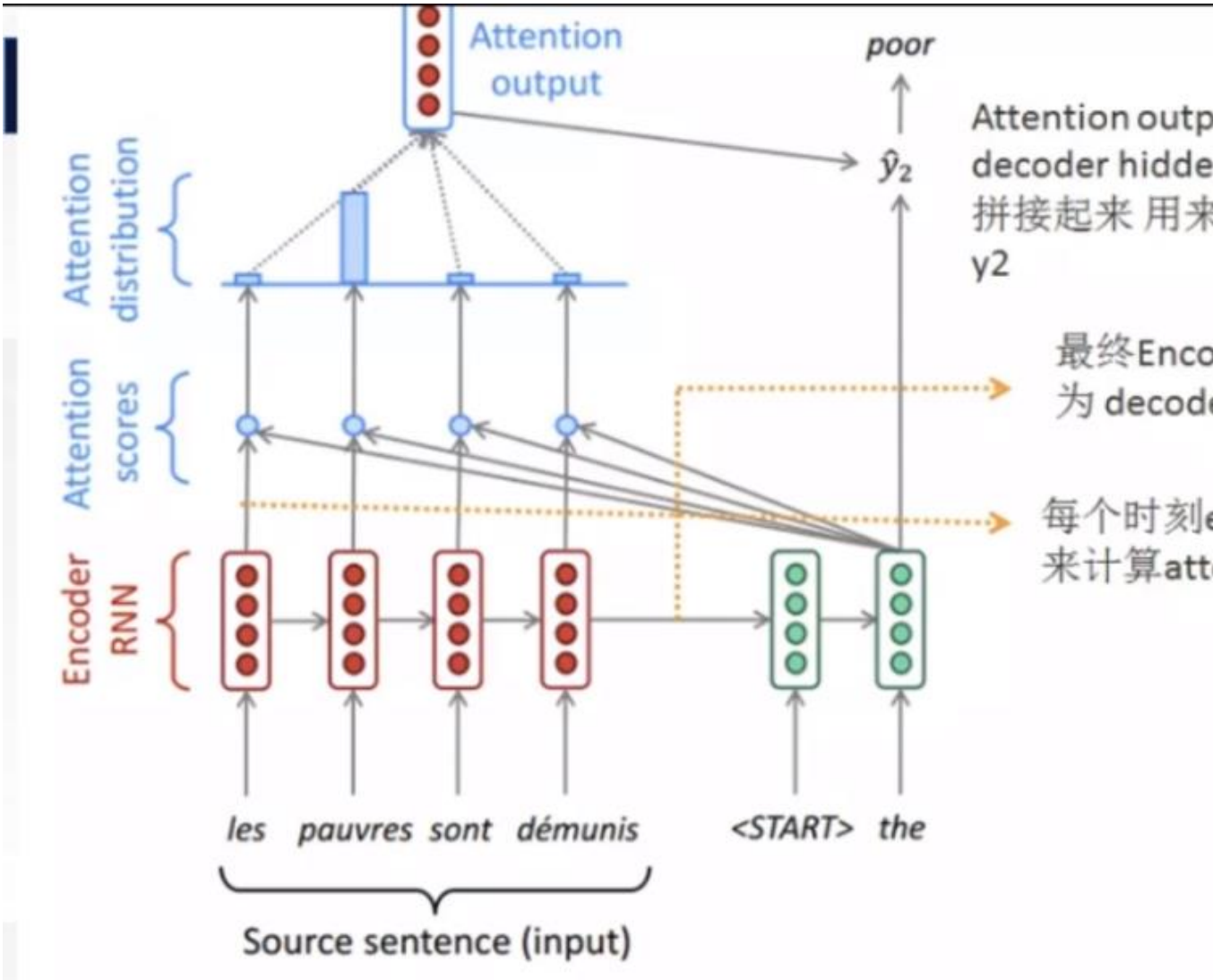
2、加注意力的编解码模型



3、最终模型

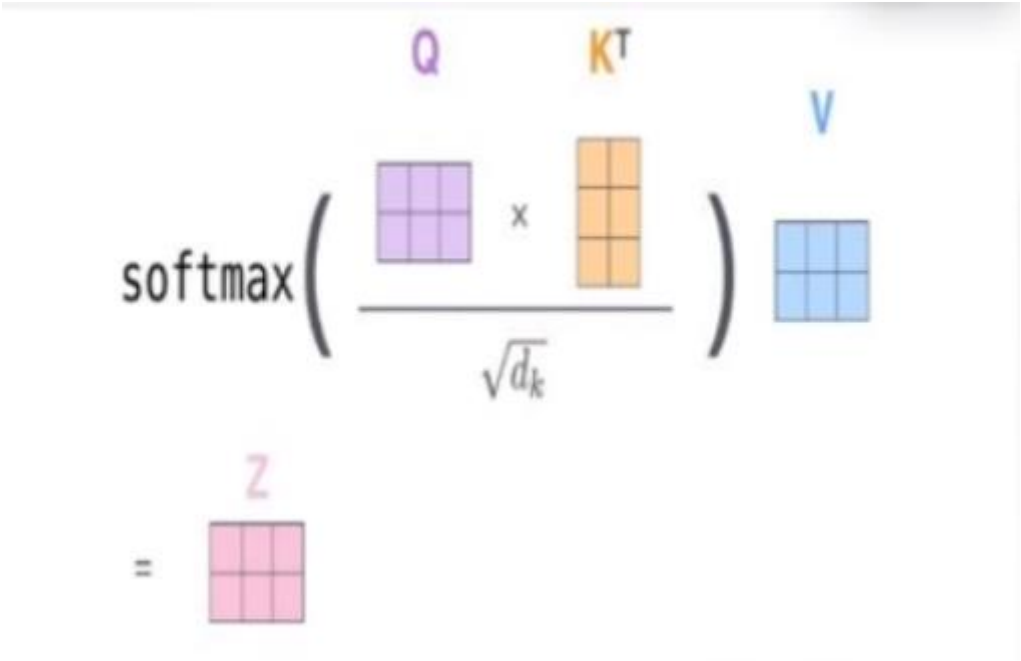


其中，hi 表示 Encoder 阶段的转换函数，ci 表示语义编码，h'i 表示 Decoder 阶段的转换函数。以上介绍的就是经典的 Soft-Attention 模型，而注意力模型按不同维度还有其它很多分类。



4、注意力公式

一般来说 Q=V=编码器的输入，K=解码器的输出，d 代表维度，除以根号下 d 简单理解为降维。



注意力模型的分类

按注意力的可微性，可分为：

Hard-Attention，就是 0/1 问题，某个区域要么被关注，要么不关注，这是一个不可微的注意力；

Soft-Attention，[0,1]间连续分布问题，用 0 到 1 的不同分值表示每个区域被关注的程度高低，这是一个可微的注意力。

按注意力的关注域，可分为：

空间域(spatial domain)

通道域(channel domain)

层域(layer domain)

混合域(mixed domain)

时间域(time domain)

单头和多头注意力机制

由上述例子，单头注意力即是 K 只有一个为单一向量，多头即是指 K 有多个，是一个矩阵。

自注意力机制

由上述例子，自注意力即是 Q=K=V，用以衡量自己的每个部分（特征）占整体多大的比重。

BERT 模型和 GPT 模型

transformer

为 U-net 添加注意力机制

优化器

参考链接：https://blog.csdn.net/fengchao03/article/details/78208414

视频链接：https://www.lieweiai.net/center/190/716 第三课时

TensorRT 推理加速

简介

TensorRT 是一个高性能的深度学习推理（Inference）优化器，可以为深度学习应用提供低延迟、高吞吐率的部署推理。TensorRT 可用于对超大规模数据中心、嵌入式平台或自动驾驶平台进行推理加速。TensorRT 现已能支持 TensorFlow、Caffe、Mxnet、Pytorch 等几乎所有的深度学习框架，将 TensorRT 和 NVIDIA 的 GPU 结合起来，能在几乎所有的框架中进行快速和高效的部署推理。

TensorRT 是一个 C++库，从 TensorRT 3 开始提供 C++ API 和 Python API，主要用来针对 NVIDIA GPU 进行 高性能推理（Inference）加速。现在最新版 TensorRT 是 4.0 版本。

目前 TensorRT4.0 几乎可以支持所有常用的深度学习框架，对于 caffe 和 TensorFlow 来说，tensorRT 可以直接解析他们的网络模型；对于 caffe2，pytorch，mxnet，chainer，CNTK 等框架则是首先要将模型转为 ONNX 的通用深度学习模型，

然后对 ONNX 模型做解析。而 tensorflow 和 MATLAB 已经将 TensorRT 集成到框架中去了。

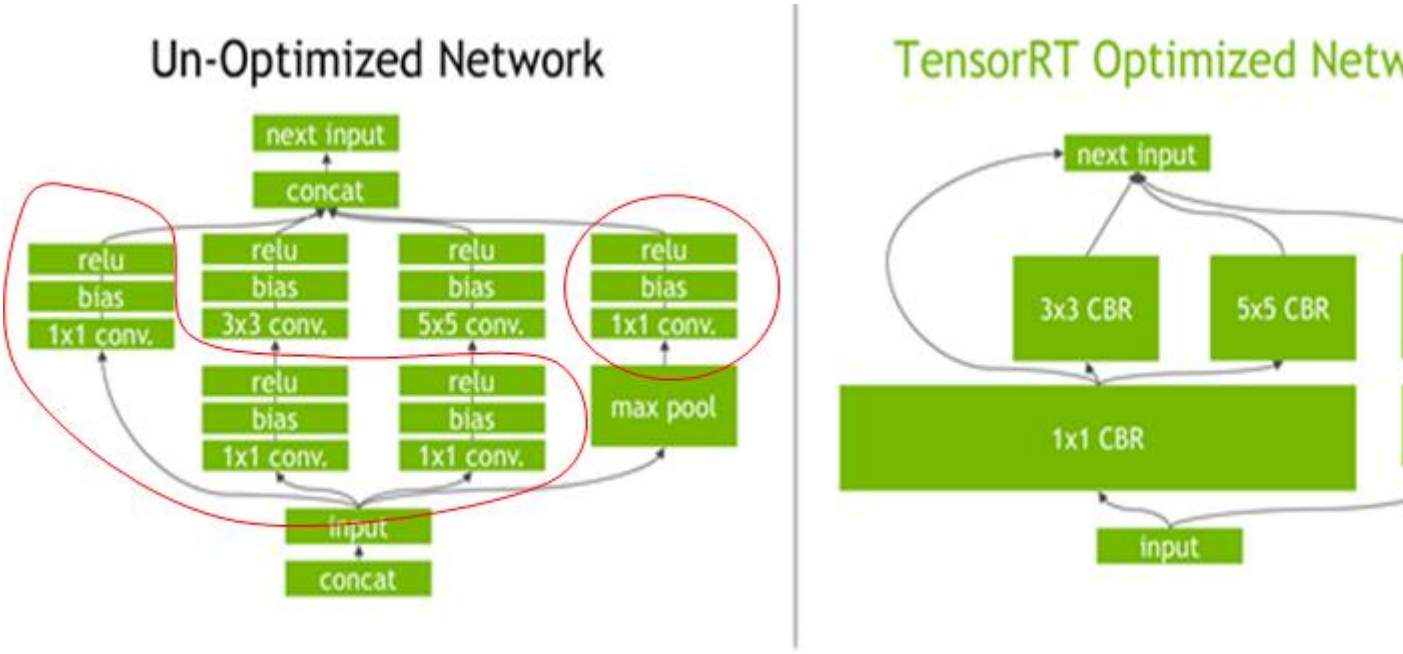
TentsorRT 优化方式

主要有以下两种：

1、层间融合或张量融合（Layer & Tensor Fusion）

如下图左侧是 GoogLeNetInception 模块的计算图。这个结构中有很多层，在部署模型推理时，这每一层的运算操作都是由 GPU 完成的，但实际上是 GPU 通过启动不同的 CUDA（Compute unified device architecture）核心来完成计算的，CUDA 核心计算张量的速度是很快的，但是往往大量的时间是浪费在 CUDA 核心的启动和对每一层输入/输出张量的读写操作上面，这造成了内存带宽的瓶颈和 GPU 资源的浪费。

TensorRT 通过对层间的横向或纵向合并（合并后的结构称为 CBR，意指 convolution, bias, and ReLU layers are fused to form a single layer），使得层的数量大大减少。横向合并可以把卷积、偏置和激活层合并成一个 CBR 结构，只占用一个 CUDA 核心。纵向合并可以把结构相同，但是权值不同的层合并成一个更宽的层，也只占用一个 CUDA 核心。合并之后的计算图（图 4 右侧）的层次更少了，占用的 CUDA 核心数也少了，因此整个模型结构会更小，更快，更高效。



2、数据精度校准（Weight &Activation Precision Calibration）

大部分深度学习框架在训练神经网络时网络中的张量（Tensor）都是 32 位浮点数的精度（Full 32-bit precision，FP32），一旦网络训练完成，在部署推理的过程中由于不需要反向传播，完全可以适当降低数据精度，比如降为 FP16 或 INT8 的精度。更低的数

据精度将会使得内存占用和延迟更低，模型体积更小。

如下表为不同精度的动态范围：

Precision	**Dynamic Range**
FP32	−3.4×1038+3.4×1038−3.4×1038 +3.4×1038
FP16	−65504+65504−65504 +65504
INT8	−128+127−128 +127

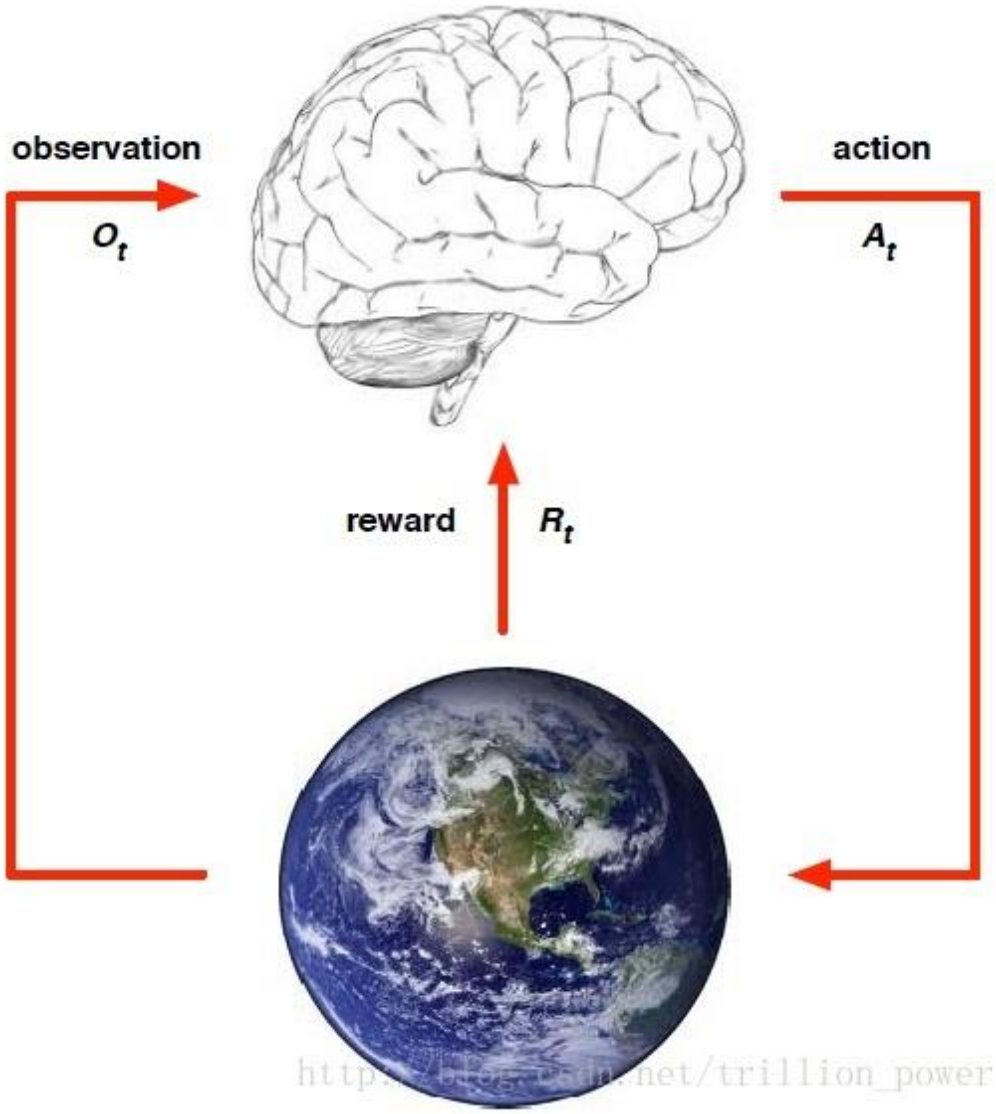
INT8 只有 256 个不同的数值，使用 INT8 来表示 FP32 精度的数值，肯定会丢失信息，造成性能下降。不过 TensorRT 会提供完全自动化的校准（Calibration ）过程，会以最好的匹配性能将 FP32 精度的数据降低为 INT8 精度，最小化性能损失。关于校准过程，后面会专门做一个探究。

参考链接：https://www.cnblogs.com/qccz123456/p/11767858.html

强化学习部分

1、 概述

模型：



强化学习中的一些概念：

Summary

Terminologies

- Agent 
- Environment
- State s.
- Action a.
- Reward r.
- Policy $\pi(a|s)$
- State transition $p(s'|s,a)$.

Return ar

- Return:

$$U_t = R_t + \gamma R_{t+}$$

- Action-value fu

$$Q_\pi(s_t, a_t) =$$

- Optimal action-

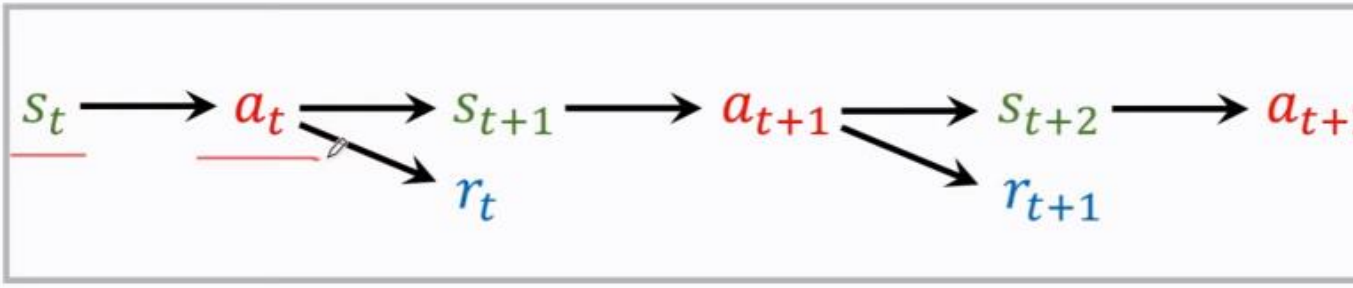
$$Q^\star(s_t, a_t) = \mathrm{m}$$

- State-value fun

$$V_\pi(s_t) = \mathbb{E}_A$$

强化学习的流程：

- Observe state s_t, make action a_t, environment gives s_t



强化学习的目的就是让 agent 根据当前的状态 s 来做出相应的动作 a，争取能够在未来获得尽量多的奖励，流程如上图。

强化学习的具体是要学什么呢？通常是要学策略函数 π（a|s）或者最优价值函数

Qmax（s|a），由此引出强化学习的三种流派，一种是基于策略（Policy-base learning），一种是基于价值（Value-base learning），另外是同时考虑两者（Actor-critic method）

2、 Q-Learning 算法

QLearning 是强化学习算法中 value-based 的算法，Q 即为 Q（s,a）就是在某一时刻的 s 状态下(s□S)，采取 动作 a (a□A)动作能够获得收益的期望，环境会根据 agent 的动作反馈相应的回报 reward r，所以算法的主要思想就是将 State 与 Action 构建成一张 Q-table 来存储 Q 值，然后根据 Q 值来选取能够获得最大的收益的动作。

贝尔曼方程：

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} \cdots +$$

Rt 当前时刻的价值 = 当前时刻的回报+后续每一时刻的回报乘以打折率的总和

Q 学习价值方程：

$$Q(s,a) = r + \gamma max_{a'} Q(s',a')$$

由上述方程可构建 Q 矩阵（Q 矩阵就是我们需要求的目标），上述方程由贝尔曼方程推出表示，当前时刻的价值 等于 当前时刻的回报（回报矩阵 R）和下一时刻的最大价值（价值矩阵 Q） 之和。

3、 DQN

Q-Learning 的缺陷：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 不现实。而深度神经网络可以自动提取复杂特征，因此，面对高维且连续的状态使用深度神经网络最合适不过了。

DRL 是将深度学习（DL）与强化学习（RL）结合，直接从高维原始数据学习控制策略。而 DQN 是 DRL 的其中一种算法，它要做的就是将卷积神经网络（CNN）和 Q-Learning 结合起来，CNN 的输入是原始图像数据（作为状态 State），输出则是每个动作 Action 对应的价值评估 Value Function（Q 值）。

DQN 中的一些 tips：

1）经验池的填充：深度学习中，采用小批量梯度下降来优化参数，DQN 中引入经验池的概念来实现批量，经验池未满时，在动作库中随机采样动作，经验池满时，引入探索值，随着网络的学习，探索动作应越来越少，即大部分为根据当前动作输入已训练的 Q 网络中，采样其中价值最大的动作放入经验池，小部分为随机采样的动作。

存入经验池的值有（state, reward, action, next_state, done）。

2）回报值：根据每一步动作网络输出的信息，确定一个回报值（可以为存活时间、游戏分数、通关等），回报值即是本轮游戏的分数。

3）损失函数的设计：从经验池中随机采样一条信息，将当前状态传入 Q 网络，再取出其中最大的价值 Qmax；而标签则通过贝尔曼方程算出，将下一时刻的状态传入 Q 网络，取出其中最大的价值 Qmax_，将此价值和当前时刻的回报相加即可得到 targetQ

```
exps = random.choices(self.exp_pool, k=100)
_state = torch.tensor([exp[0] for exp in exps])
_reward = torch.tensor([[exp[1]] for exp in exps])
_action = torch.tensor([[exp[2]] for exp in exps])
_next_state = torch.tensor([exp[3].tolist() for exp in exps])
_done = torch.tensor([[int(exp[4])] for exp in exps])

# 得到估计值
_Qs = self.q_net(_state.float())
_Q = torch.gather(_Qs, 1, _action)

# 目标值
_next_Qs = self.q_net(_next_state)
_max_Q = _next_Qs.max(dim=1, keepdim=True)[0]
_target_Q = _reward + (1 - _done) * 0.9 * _max_Q
```

loss = self.loss_fn(_Q, _target_Q.detach())

损失函数总结

1、 常见损失函数

普通回归：

1）L1 损失函数

L1 损失，又称为 MAE 损失，平均绝对误差，用 L1 范数来衡量其距离。L1 损失由于 L1 损失函数导数不连续，可能存在多个解，当数据集有一个微小的变化，解可能会有一个很大的跳动，L1 的解不稳定。

$$S=\sum_{i=1}^{n}\left|Y_i-f(x_i)\right|.$$

2）L2 损失函数

L2 损失，又称为 MSE 损失，均方差损失，L2 损失函数导数连续，损失下降会很平滑，稳点，但 L2 损失函数对异常点比较敏感，因为 L2 将误差平方化，使得异常点的误差过大，模型需要大幅度的调整，这样会牺牲很多正常的样本。

$$S=\sum_{i=1}^{n}\left(Y_i-f(x_i)\right)^2.$$

3）smooth L1 损失函数

smooth L1 损失函数结合了 L1、L2 的优点，可以让离群点更加的鲁棒，相比于 L1，其对离群点、异常值不敏感，可以控制梯度的量级使训练不容易跑飞。另外，smooth l1 在零点处变得可导，函数更加的平滑。总体而言 smooth-L1 可以让损失下降更快、更平稳。

$$L_{\text{loc}}(t^u, v) = \sum_{i \in \{x,y,w,h\}} \text{smooth}_{L_1}(t_i^u - v_i), \qquad (2)$$

in which

$$\text{smooth}_{L_1}(x) = \begin{cases} 0.5x^2 & \text{if } |x| < 1 \\ |x| - 0.5 & \text{otherwise,} \end{cases} \qquad (3)$$

边框回归：

1）IOU Loss

2）GIOU Loss

3）DIOU Loss

4）CIOU Loss

分类：

1）多分类交叉熵（Softmax）

交叉熵损失函数刻画的是两个概率分布之间的**距离**。如下式，交叉熵刻画的是通过概率分布 q 来表达概率分布 p 的困难程度，其中 p 为真实分布，q 为预测，交叉熵越小，两个概率分布越接近。

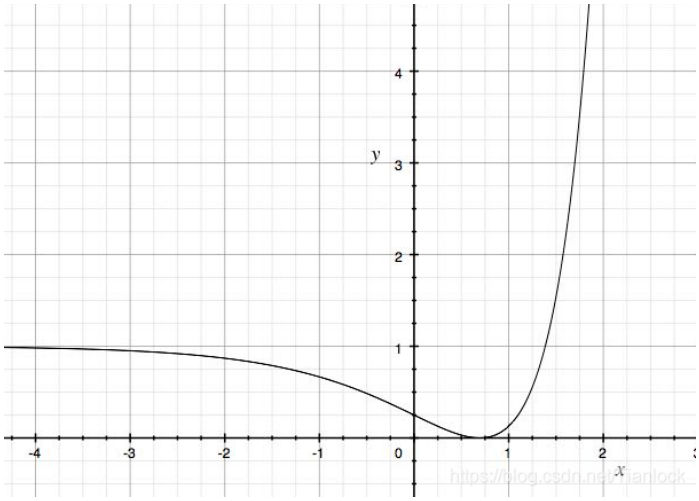
$$H(p,q) = -\sum p(x)\log(q(x))$$

其实交叉熵损失函数来源于信息论中 KL 散度这一概念。在深度学习中，二分类通 常通过 sigmoid 函数作为预测的输出，得到预测的分布，那对于多分类是怎样得到 预测的分布的呢。答案是 softmax。

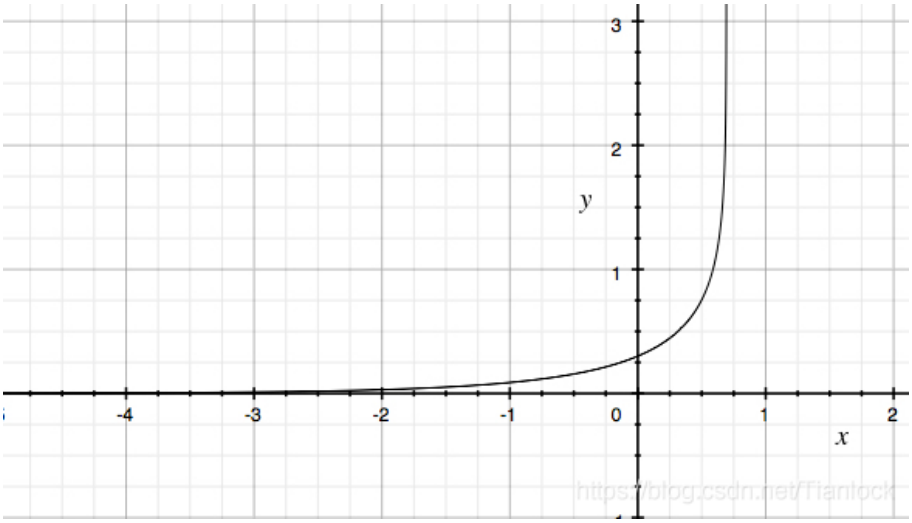
$$soft\max(y)_i = \frac{e^{y_i}}{\sum_{j=1}^{n} e^{y_j}}$$

神经网络经过 softmax 之后，把神经网络的输出变成了一个概率分布。从而可以通过交叉熵来计算预测的概率分布与真实分布之间的距离了。

Softmax 交叉熵与 MSE 的区别：



MSE 均方差损失函数



Softmax 损失函数

由上图可见，L2 损失函数是一个非凸函数，梯度下降法不一定能够保证达到全局最优解，Softmax 损失函数则是一个凸函数，曲线整体呈单调性，loss 越大，梯度越大。便于反向传播时的快速优化。所以通常使用交叉熵损失函数。

2）二分类交叉熵（Sigmoid）

其实 logistics loss 和交叉熵损失差不多，该函数处处光滑，交叉熵在二分类的时候就是 logistic loss。

$$H(p,q) = -\sum_{x} p(x)\log q(x) = -y\log\hat{y} - (1-y)\log(1-\hat{y})$$

衡量分布相似度：

1）KL 散度

KL 散度是用于衡量两个分布之间的**相似度**，与交叉熵略有不同，在图像生成

如 VAE 中常用作损失函数。

$$KLD(p||q) = \sum_{i=1}^{k} p(x^i) \log \frac{p(x^i)}{q(x^i)} = -\sum_{i=1}^{k} p(x^i) \log q(x^i) - H(p)$$

另补充：

交叉熵：

$$H(p,q) = -\sum_{i=1}^{k} p(x^i) \log q(x^i)$$

信息量：

$$I(X = X^i) = \log \frac{1}{p(X^i)} = -\log p(X^i)$$

熵（信息的期望）:

$$H\S = \sum_{i=1}^{K} P(x^i) \log \frac{1}{P(x^i)} = -\sum_{i=1}^{K} P(x^i) \log P(x^i)$$

交叉熵 = 相对熵 + 熵

$$H(p,q) = KLD(p||q) + H(p)$$

2、 focal_loss

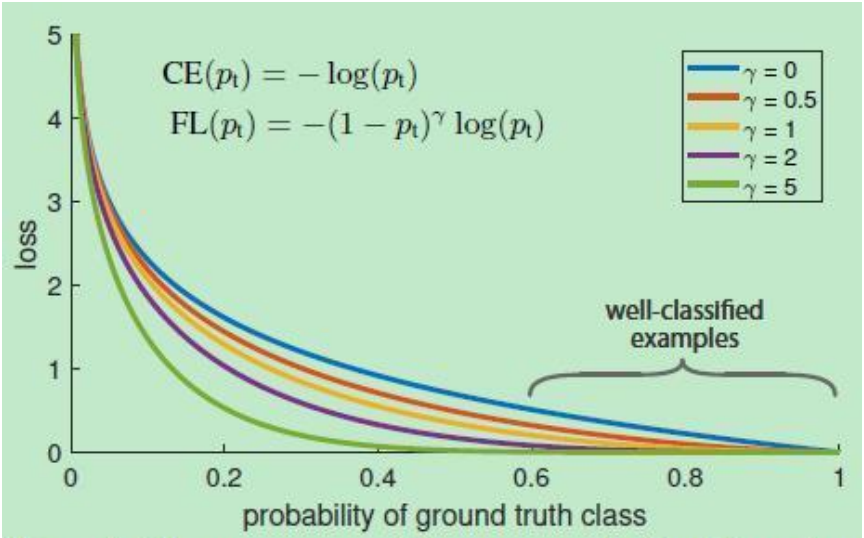


Figure 1. We propose a novel loss we term the *Focal Loss* that adds a factor $(1 - p_t)^\gamma$ to the standard cross entropy criterion. Setting $\gamma > 0$ reduces the relative loss for well-classified examples ($p_t > .5$), putting more focus on hard, misclassified examples. As our experiments will demonstrate, the proposed focal loss enables training highly accurate dense object detectors in the presence of vast numbers of easy background examples.

focal_loss 的提出：

在分类实验中，负样本数量太大，占总的 loss 的大部分，而且多是容易分类的，因此使得模型的优化方向并不是我们所希望的那样。针对类别不均衡问题，KAIMING 等大神作者提出一种新的损失函数：focal loss，这个损失函数是在标准交叉熵损失基础上修改得到的。这个函数可以通过减少易分类样本的权重，使得模型在训练时更专注于难分类的样本

以二分类为例：

$$\mathrm{FL}(p_t) = -\alpha_t(1-p_t)^\gamma \log(p_t). \quad (5)$$

由上述公式，当一个样本被分错的时候，pt 是很小的（pt 小于 0.5 才是错误分类），(1-pt) 偏大，整体的调制系数就趋于 1，也就是说相比原来的 loss 是没有什么大的改变的。当 pt 趋于 1 的时候（此时分类正确而且是易分类样本），调制系数趋于 0，也就是对于总的 loss 的贡献很小。另外当 γ=0 的时候，focal loss 就是传统的交

叉熵损失，当 γ 增加的时候，调制系数也会增加。

focal loss 其实就是用一个合适的函数去度量难分类和易分类样本对总的损失的贡献。作者在实验中采用的上述公式的 focal loss（添加了 at，这样既能调整正负样本的权重，又能控制难易分类样本的权重）

激活函数总结

目标检测评价指标总结

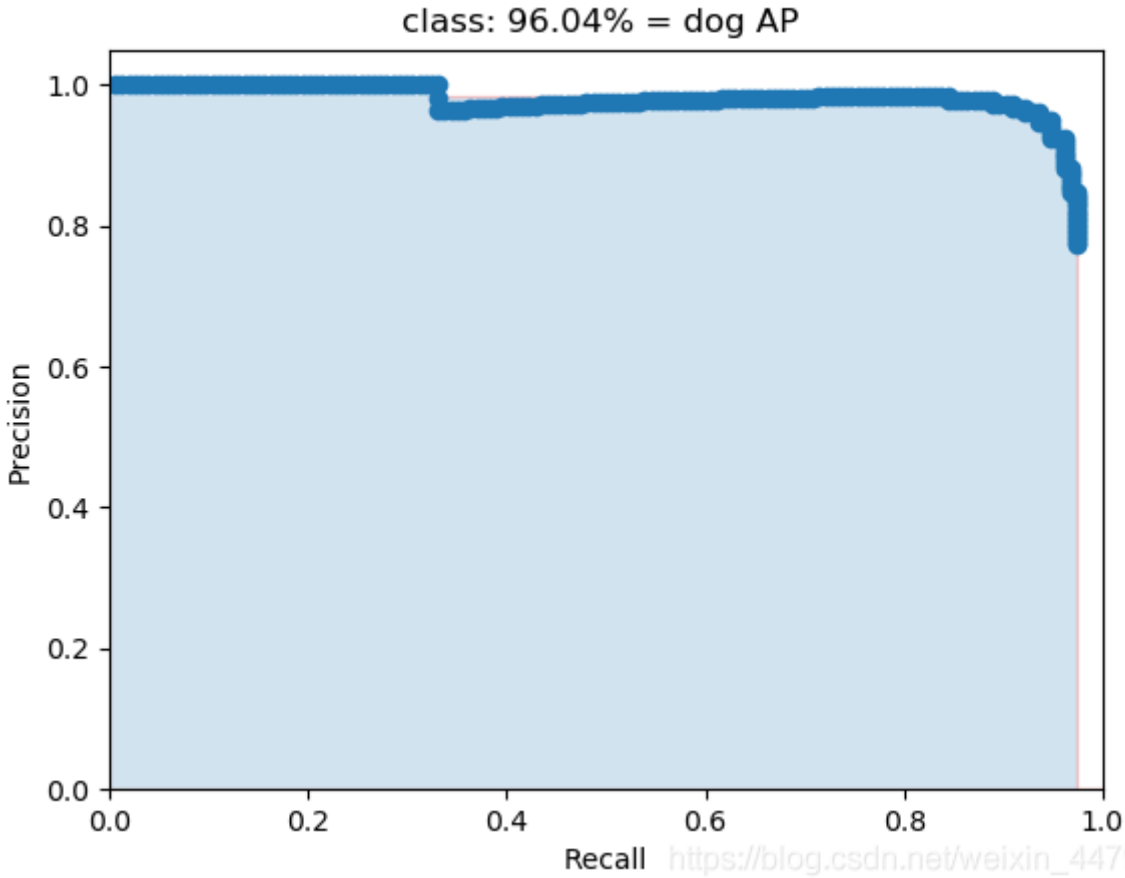
混淆矩阵和基本指标

PR 曲线

mAP 概念

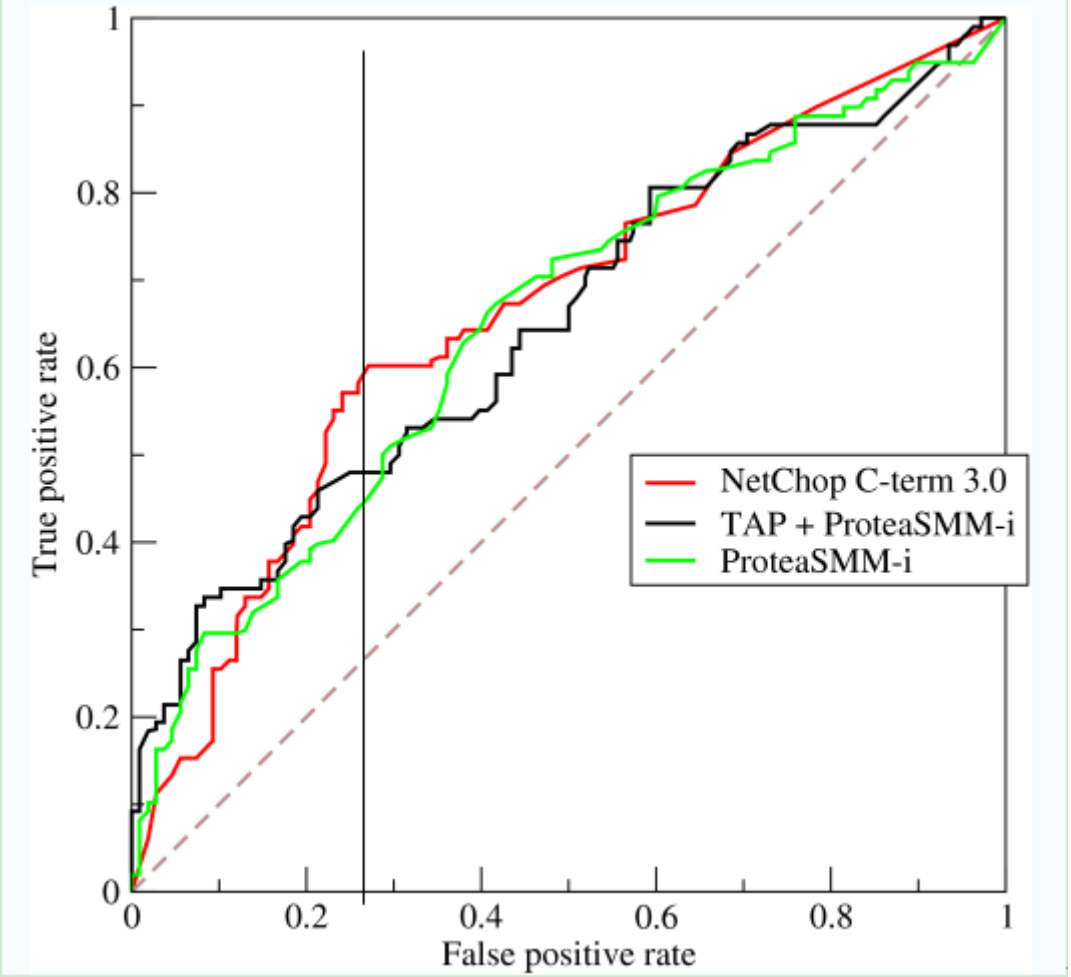
什么是 AP？

AP 事实上指的是，利用不同的 Precision 和 Recall 的点的组合，画出来的曲线下面的面积。如下面这幅图所示。



当我们取不同的置信度，可以获得不同的 Precision 和不同的 Recall，当我们取得置信度够密集的时候，就可以获得非常多的 Precision 和 Recall。此时 Precision 和 Recall 可以在图片上画出一条线，这条线下部分的面积就是某个类的 AP 值。mAP 就是所有的类的 AP 值求平均。

ROC 曲线和 AUC 值



数据集总结