

# Learning Report

Bingnan Guo

July 2021

## Contents

<b>1 Regression and Classification</b>	<b>2</b>
1.1 Regression . . . . .	4
1.2 Classifaction . . . . .	8
1.3 Optimization . . . . .	10
1.3.1 Batch . . . . .	10
1.3.2 Momentum . . . . .	12
1.3.3 RMSProp and Adam . . . . .	13
<b>2 Convolutional Neural Network</b>	<b>16</b>
<b>3 Self-attention</b>	<b>21</b>
<b>4 Transformer</b>	<b>24</b>
4.1 Encoder . . . . .	25
4.2 Decoder . . . . .	27
4.3 Encoder-Decoder . . . . .	29
<b>5 To Learn more</b>	<b>30</b>

# 1 Regression and Classification

首先学习的是回归 (regression) 与分类 (classification)，以及相关训练过程中的优化方法，图 1 是一个 machine learning 的基本框架，给定训练数据 (包含输入与标签)，需要拟合一个函数 (模型)，并通过相应的损失函数来约束监督拟合的函数。

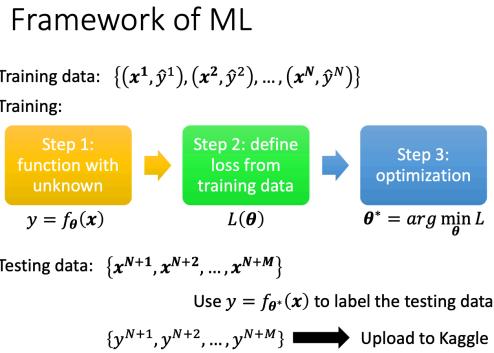
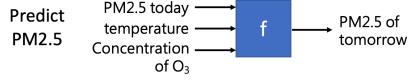


Figure 1: Framework of Machine Learning

对于回归与分类，主要是输出类型不同，回归任务输出一串连续的值，而分类任务中主要输出类别。在一开始的时候，我一直没搞清楚为什么逻辑回归的输出是不连续的 0 1，但却是属于回归，但其实该任务本质上属于二分类问题。并且在回归与分类问题中，损失函数也往往不同，在回归任务中，常用的有均方误差 (MSE)、平均绝对误差 (MAE)，在分类任务中，主要有二值交叉熵 (Binary Cross Entropy) 和交叉熵 (Cross Entropy) 等。

**Regression:** The function outputs a scalar.



**Classification:** Given options (**classes**), the function outputs the correct one.



Figure 2: Regression and Classification

特性	分类(监督学习)	回归
输出类型	离散数据	连续数据
目的	寻找决策边界	找到最优拟合
评价方法	精度 (accuracy)、混淆矩阵等	SSE (sum of square errors) 或拟合优度

Figure 3: Regression and Classification2

在定义了任务类型以及损失函数后，较为重要的就是如何去拟合参数来使得我们定义的损失函数最小，其中梯度下降 (Gradient Descent) 是极为重要的方法。并给出其公式。

$$\theta = \theta - \alpha \frac{\partial L}{\partial \theta} \quad (1)$$

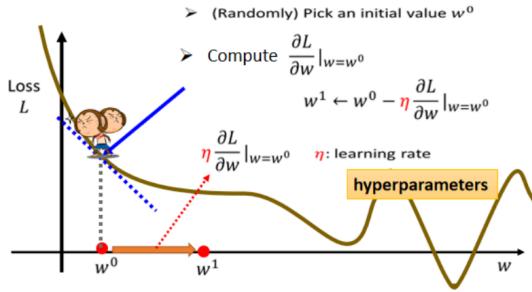


Figure 4: gradient descent

在梯度下降之后，一个较为重要的概念则是反向传播 (Back propagation)，

我们都知道在计算 loss 并使用梯度下降的过程中，首先初始化参数，并计算损失函数对各个参数的偏导来完成梯度下降，反向传播主要根据链式法则求导来加快偏导的计算。因此在梯度下降的过程中，首先使用前向传播，将得到的参数放入内存中，再使用反向传播，计算对各个参数的梯度，最后完成梯度下降，最后给出反向传播的链式法则公式及示意图。

$$\frac{\partial L}{\partial \theta} = \frac{\partial L}{\partial z} \times \frac{\partial z}{\partial \theta} \quad (2)$$

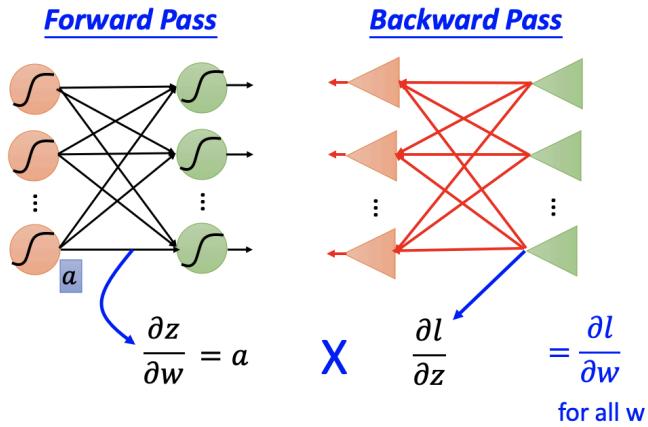
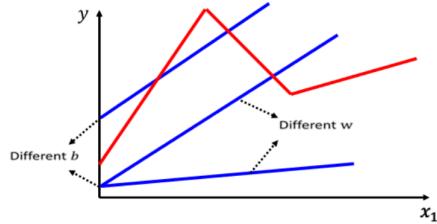


Figure 5: Back propagation

## 1.1 Regression

在回归任务中，可以通过线性回归来引出神经网络模型。对于简单的线性模型，我们可以拟合少量参数来完成线性回归，但实际上，线性模型相对来说会比较复杂，如图 6 所示。因此我们需要考虑如何拟合稍复杂的模型，以图 6 的红曲线为例，该曲线可以看成常数加上蓝色的函数曲线。

Linear models are too simple ... we need more sophisticated modes.



Linear models have severe limitation. ***Model Bias***

We need a more flexible model!

Figure 6: Different LinerModel

因此在图 7 中，将稍微复杂的红色曲线表示为多条蓝色曲线以及常数项的叠加，那么如果是更加复杂的线形曲线，我们也同样可以找到多种不同的蓝色曲线，然后叠加常数项得到。

## All Piecewise Linear Curves

= constant + sum of a set of

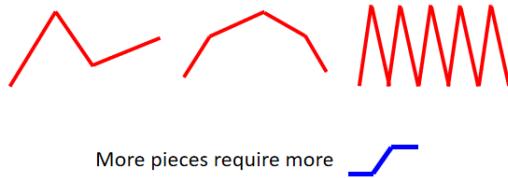


Figure 7: Different Linear Curves

我们都知道一条曲线可以由多个线性函数拟合而成，即将曲线分为一小段一小段，然后用线段拟合，因此就可以使用曲线来逼近任意函数，那么我们通常用 sigmoid 函数来表示最开始的蓝色曲线函数。sigmoid 函数的具体形式如公示 3 所示。

$$y = c \frac{1}{1 + e^{-(b + wx_1)}} \quad (3)$$

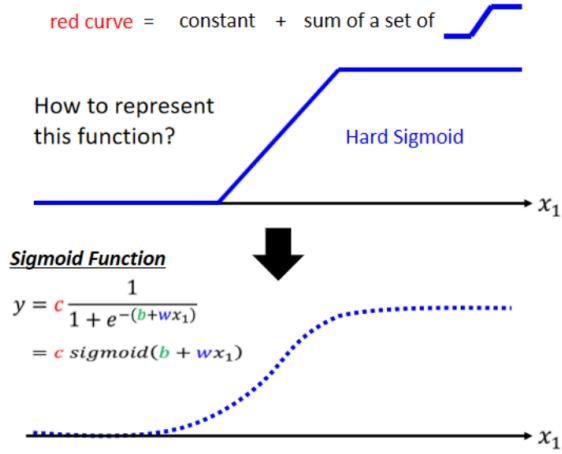


Figure 8: Sigmoid

通过调整 sigmoid 的不同参数，我们可以拟合不同的曲线，为了使得我们的函数能够表达更多不同的曲线，给出如下的公式：

$$y = b + \sum_i c_i \text{sigmoid}(b_i + w_i x_i)$$

$$y = b + \sum_i c_i \text{sigmoid} \left( b_i + \sum_j w_{ij} x_i \right)$$

在上述公式中，假设 i、j 为 1 2 3，那么将表达式列举出来，具体如图 9 所示，可以发现 sigmoid 中可以表示为  $r=b+wx$  的，那么就相当于经过线性变化再通过 sigmoid 函数。

$$y = b + \sum_i c_i \text{sigmoid} \left( b_i + \sum_j w_{ij} x_j \right) \quad i: 1,2,3 \quad j: 1,2,3$$

$$\begin{aligned} r_1 &= b_1 + w_{11}x_1 + w_{12}x_2 + w_{13}x_3 \\ r_2 &= b_2 + w_{21}x_1 + w_{22}x_2 + w_{23}x_3 \\ r_3 &= b_3 + w_{31}x_1 + w_{32}x_2 + w_{33}x_3 \end{aligned}$$

$$\begin{bmatrix} r_1 \\ r_2 \\ r_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} + \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$\mathbf{r} = \mathbf{b} + \mathbf{W} \mathbf{x}$$

Figure 9: Explanation of Sigmoid function

因此我们就可以得到一个简单的神经网络模型，输入向量经过线性变化，再经过 sigmoid 激活函数以及最后的线性变化得到输出，构成了一个极其简单的网络模型。

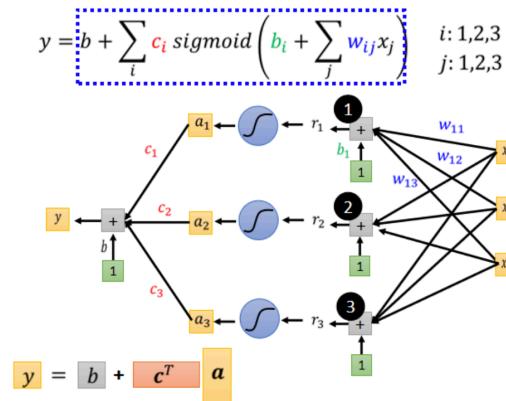


Figure 10: simple neural network

在网络模型中，激活函数通常使用 Relu，而在本节的 Regulation 中，主要叙述了从线性模型引出基本神经网络模型，这是该课程以独特的角度为我们介绍神经网络模型，因此我们能从不同的角度理解神经网络的构成。

## 1.2 Classification

简单来说，分类任务是输入一些向量（可以是表示图像、文字序列的向量），然后输出  $\hat{y}$ ，可以使用固定的数字来表示类别，但是在分类任务中，更多的使用 one-hot 向量来表示，具体如下所示，三个向量分别表示 class1、class2、class3。

$$\hat{y} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \text{ or } \hat{y} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \text{ or } \hat{y} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

但是在分类的过程中，往往会将神经网络输出的  $y$  再通过 Soft-max 的 function 得到  $y'$ ，然后才去计算 loss，至于加上 Soft-max 的原因，一个比较简单的解释是我们所表示类别的 One-hot 向量中表示的值只有 0 和 1，而神经网络输出的有任何值，所以通过 Soft-max 层将其标准化为 0, 1。

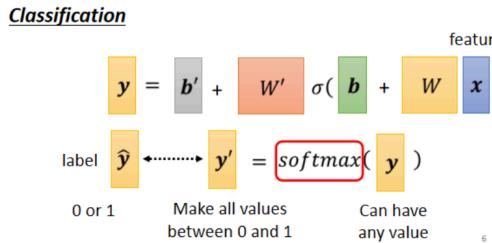


Figure 11: Classification

因此将会初步的介绍一下 Soft-max 的运行机制，具体如下图所示。并给出其具体的计算公式。从图中可知，输入任何值，Soft-max 函数都会根据权重输出  $0 \sim 1$  间的数字，因此在后续的数据处理中，我们可以设定阈值，将超出最大的值赋值为 1，其他赋值为 0，就可以得到预测结果，这点在 HW3 中的半监督学习法中有所体现。

$$y'_i = \frac{\exp(y_i)}{\sum_j \exp(y_j)} \quad (4)$$

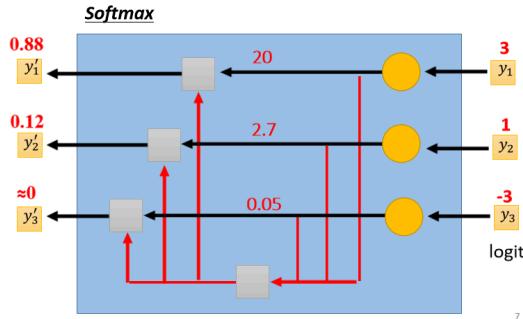


Figure 12: Soft-max

在分类任务中，还有一个比较重要的是分类的损失函数，在分类任务中较为常见的是使用 Cross-entropy 即使用交叉熵损失函数，并给出其公式以及调用方式。

$$e = - \sum_i \hat{y}_i \ln y'_i \quad (5)$$

```

1 # Soft-max 的函数定义
2 torch.nn.functional.softmax(input, dim=None, _stacklevel=3,
   dtype=None)
3 # Cross-entropy 的函数定义
4 torch.nn.CrossEntropyLoss(weight: Optional[torch.Tensor] = None
   , size_average=None, ignore_index: int = -100, reduce=None,
   reduction: str = 'mean')

```

需要注意的是，在 pytorch 中 Cross-entropy 和 Soft-max 是绑定在一起的，如果你使用了 Cross-entropy 来计算 loss，那么 Soft-max 将会自动加入你构建神经网络的最后一层。

## 1.3 Optimization

紧接着就是在训练过程中遇到问题如何去优化，包括如何根据训练过程中训练数据或者是验证数据的 loss 来判断你构建的神经网络的性能，是欠拟合还是过拟合？是否需要加深你的网络结构？或者是给更多的数据？又或者是采用正则化以及其他优化器，因此将在图 13 中给出一个判断网络性能的概览图。

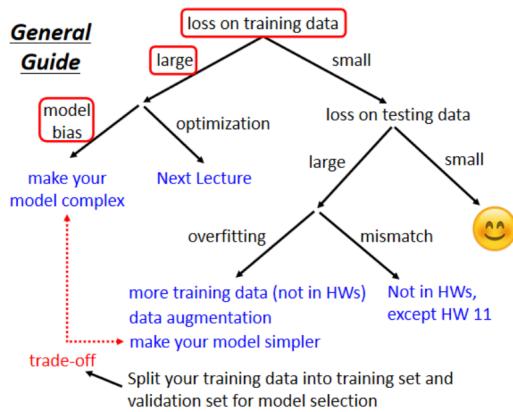


Figure 13: General Guide

如果在训练数据上发现 loss 很高，并没有训练好，那么你就需要考虑是什么原因导致，那么会有两个可能，其中一个则是 Model bias，即模型的结构设置过于简单。这个时候就需要重新设计模型使得模型具有更大的弹性。当然，在训练数据上 loss 高的另一个原因则是优化问题，其实梯度下降往往存在着很多问题，接着将会介绍训练策略及优化器。

### 1.3.1 Batch

在下图中，给出了 Batch 的介绍，在神经网络的训练中，会有一个参数叫 batch size，其代表将需要训练的数据分为多个大小为 batch size 的 Batch。所以训练的过程可以概括为，使用一个 Batch 的数据计算 loss，再计算 gradient，最后 update 参数，将所有的 Batch 训练过一遍则称为一轮 Epoch，其中值得注意的是，在每一轮 Epoch 开始前，会进行 Shuffle 使得每次 Epoch 开始前 Batch 中的数据不同。

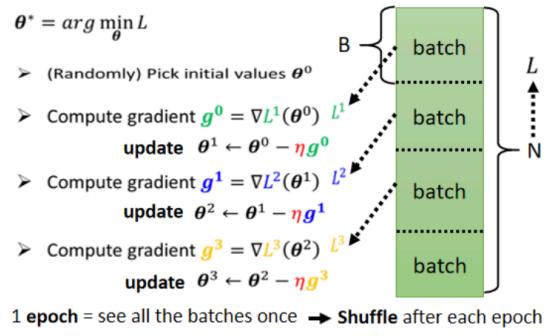


Figure 14: optimization with batch

那么不同大小的 batch size 对性能是否有什么影响，这里引用了在 MNIST 以及 CIFAR-10 中的两个实验结果。

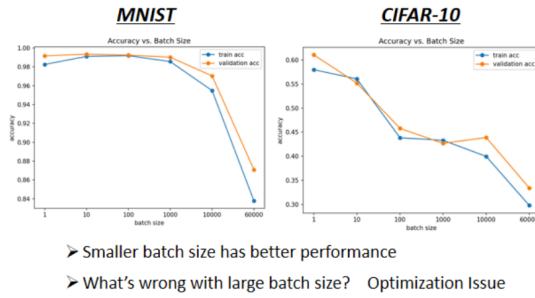


Figure 15: optimization with batch

可以发现极大的 batch size 的 (超过 1000) 结果并不好，相对来说偏小的 batch size(小于 1000) 有着就好的结果，但是具体的 batch size 作为一项超参需要在训练过程中根据不同的任务去调整。

### 1.3.2 Momentum

接着会介绍几个优化器，可以解决 Saddle Point(鞍点) 或者 Local Minima(局部最小值)，首先给出梯度下降 +Momentum 的公式，其中  $m$  为每次迭代需要移动的参数。通过  $\theta$  来计算梯度  $g$ 。

$$\begin{aligned} m^0 &= 0 \\ m^1 &= \lambda m^0 - \eta g^0 \\ \theta^1 &= \theta^0 + m^1 \\ m^2 &= \lambda m^1 - \eta g^1 \\ &\dots\dots \end{aligned}$$

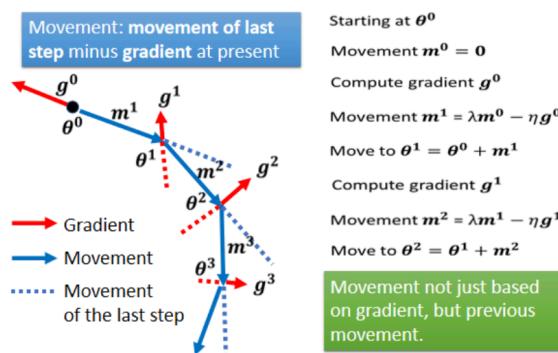


Figure 16: optimization with batch

以此类推，可以发现在梯度下降加上 Momentum 后，可以表述为梯度的负反方向加上前一次移动的方向。从其他角度看，当加上 Momentum 的时候，更新参数的方向不是只考虑现在的梯度，而是考虑过去所有梯度的总和。在 pytorch 中，Momentum 直接被集成到梯度下降函数中，可以直接设定其参数值。

```

1 # SGD-梯度下降中可以直接使用Momentum设置参数
2 torch.optim.SGD(params, lr=<required parameter>,
3                  momentum=0, dampening=0, weight_decay=0,
4                  nesterov=False)

```

### 1.3.3 RMSProp and Adam

当训练一个神经网络的过程中，训练后发现 loss 不再下降的时候，可能卡在 local minima，也可能卡在 saddle point，但有时候根本两个都不是，只是单纯的 loss 没有办法再下降。

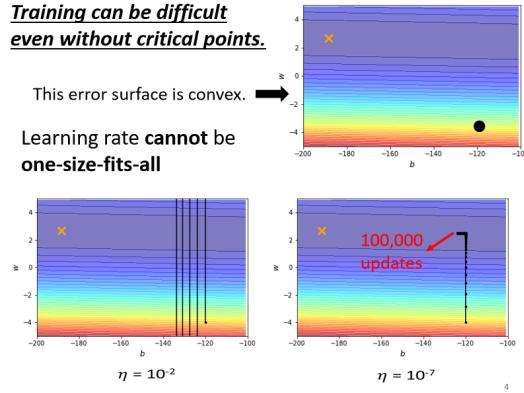


Figure 17: Why traning loss donot change

在上图中可以简单的可视化训练过程，如果以较大的学习率来训练，可以发现参数不断的在震荡，无法收敛，若以较小的学习率来训练，始终在一个点而无法继续前进，因此这就需要自适应学习率。给出公式。

$$\begin{aligned}\theta_i^{t+1} &\leftarrow \theta_i^t - \frac{\eta}{\sigma_i^t} g_i^t \\ g_i^t &= \frac{\partial L}{\partial \theta_i} |_{\theta = \theta^t}\end{aligned}\tag{6}$$

其中  $g_i^t$  代表在第  $t$  个 iteration 中的微分，因此就可以不同的参数、迭代次数中会存在不同的学习率，那么接下来则会考虑如何计算这个自适应的学习率。公式如下所示。

$$\sigma_i^t = \sqrt{\frac{1}{t+1} \sum_{i=0}^t (g_i^t)^2} \tag{7}$$

上述公式是被使用于一个叫做 Adagrad 的方法中，可以能够根据每一次参数梯度的不同来调整学习率的大小，但是在某些情况下，在同一个参数的同一个梯度也需要学习率能够进行动态调整，因此就提出了 RMS Prop，具体如下图所示，并给出公式。

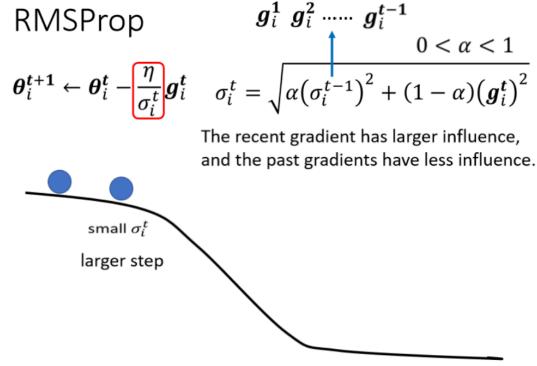


Figure 18: RMSProp

通过  $\alpha$  这一项可以决定  $g_i^t$  相较于之前的  $\sigma_i^{t-1}$  中的  $g_i^{t-1}$  而言, 它的重要性有多大, 如果使用 RMSProp, 就可以动态调整  $\sigma$  这一项。

$$\sigma_i^t = \sqrt{\alpha(\sigma_i^{t-1})_2 + (1-\alpha)(g_i^t)_2} \quad (8)$$

在优化策略中, 最常用的就是 Adam, Adam 中的策略即为 RMSProp+Momentum, 该优化器在 pytorch 中可以直接调用。

## Adam: RMSProp + Momentum

**Algorithm 1:** Adam, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation.  $g_t^2$  indicates the elementwise square  $g_t \odot g_t$ . Good default settings for the tested machine learning problems are  $\alpha = 0.001$ ,  $\beta_1 = 0.9$ ,  $\beta_2 = 0.999$  and  $\epsilon = 10^{-8}$ . All operations on vectors are element-wise. With  $\beta_1^t$  and  $\beta_2^t$  we denote  $\beta_1$  and  $\beta_2$  to the power  $t$ .

```

Require:  $\alpha$ : Stepsize
Require:  $\beta_1, \beta_2 \in [0, 1]$ : Exponential decay rates for the moment estimates
Require:  $f(\theta)$ : Stochastic objective function with parameters  $\theta$ 
Require:  $\theta_0$ : Initial parameter vector
 $m_0 \leftarrow 0$  (Initialize 1st moment vector)  $\rightarrow$  for momentum
 $v_0 \leftarrow 0$  (Initialize 2nd moment vector)  $\rightarrow$  for RMSprop
 $t \leftarrow 0$  (Initialize timestep)
while  $\theta_t$  not converged do
     $t \leftarrow t + 1$ 
     $g_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1})$  (Get gradients w.r.t. stochastic objective at timestep  $t$ )
     $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$  (Update biased first moment estimate)
     $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$  (Update biased second raw moment estimate)
     $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$  (Compute bias-corrected first moment estimate)
     $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$  (Compute bias-corrected second raw moment estimate)
     $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$  (Update parameters)
end while
return  $\theta_t$  (Resulting parameters)
  
```

Figure 19: the implementation Adam

```

1 # Adam优化器及其初始参数
2 torch.optim.Adam(params, lr=0.001, betas=(0.9, 0.999), eps=1e
                  -08, weight_decay=0)

```

在上述的公式中我们可以发现  $\eta$  是个固定值，那么考虑以下的情况，如果梯度一直很小，那么就累积了很小的  $\sigma$ ，当累计的  $\sigma$  足够小，则会忽然变的很大，因此需要一些策略来解决这些问题，即 Learning rate Decay 与 Warm up。

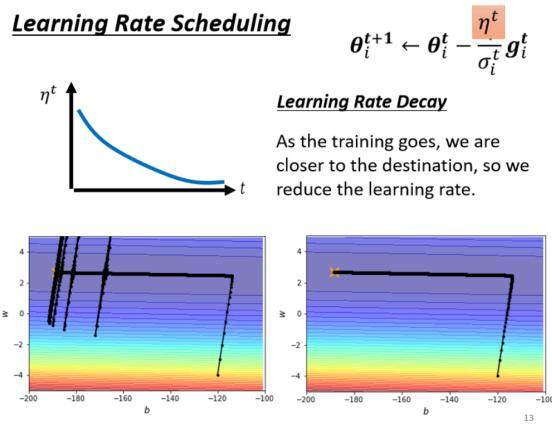


Figure 20: Weight Decay

Weight Decay 是持续让学习率逐步减少，逐渐慢慢变小。

Warm Up 这个方法是让学习率先变大后变小，具体变小的速度与点是作为超参需要调整，但是总体策略是，学习率要先变大后变小，其往往被 ResNet、BERT、Transformer 用到。

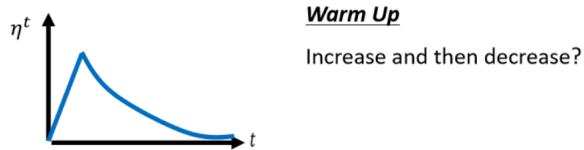


Figure 21: the implementation Adam

## 2 Convolutional Neural Network

在这一节中，将会记录一下课程中学习的卷积神经网络 (Convolutional Neural Network) 即 CNN，该网络架构主要应用于视觉研究领域。在图像分类的任务中，将图片作为输入并输出预测值，较为常见的处理方法是将所有的数据图片 Rescale 成一样的大小，再放入模型中进行训练预测。我们所期望的输出是一个 One-Hot 的 Vector，包括对类别的预测输出。

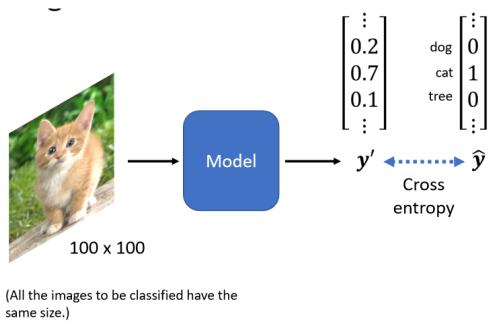


Figure 22: Image Classification

那么首先考虑输入，一张图片其实可以看成三维的 Tensor，即相当于 width\*height\*channel 的 Tensor，可以将其进行拼接，转化成一维的 Tensor，作为网络的输入。

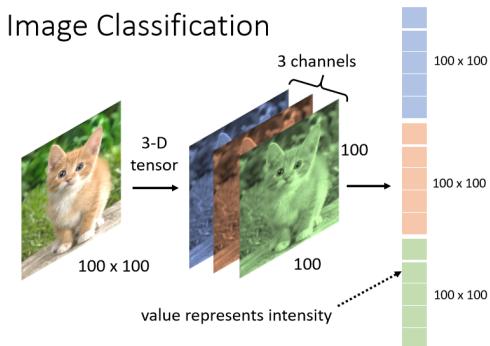


Figure 23: Image Classification2

因此可以发现，输入的向量数据非常的多，如果使用全连接层，随着模型层数的增加，可能会增加模型过拟合的风险，因此设计一个比较特别的模型来解决图像分类的问题是比较有意义的。

在图像分类的过程中，我们通常会根据一个动物的相应特征来判断，那么在模型中，是否可以设计令某些神经元关注于图像的某个位置，进行学习更新参数，因此就可以使用卷积。

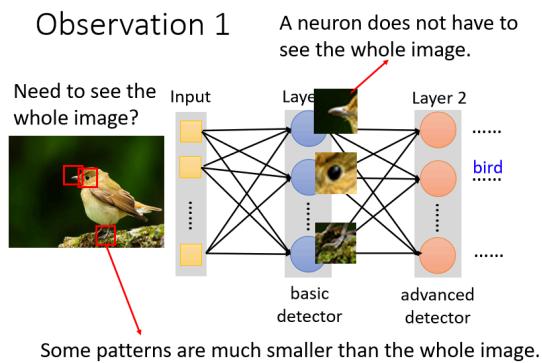


Figure 24: Pattern Design

如上图所示，对于某些神经元来说，只需要将图片的一部分作为输入，使它能够检测相应区域的特征有无出现就可以了，因此在下图中给出卷积的相应图示。

在图 25 中，设定的红框区域称为 Receptive Field(感受野)，每一个神经元只关注自己的区域，在下图中感受野的区域是  $3 \times 3 \times 3$ ，也就是 27 维向量，也就是有 27 个权重以及一个偏置项。通常所说的 kernel size 卷积核就是  $3 \times 3$ ，stride 即代表每次卷积核移动的步长，padding 代表超出范围时补值的范围。

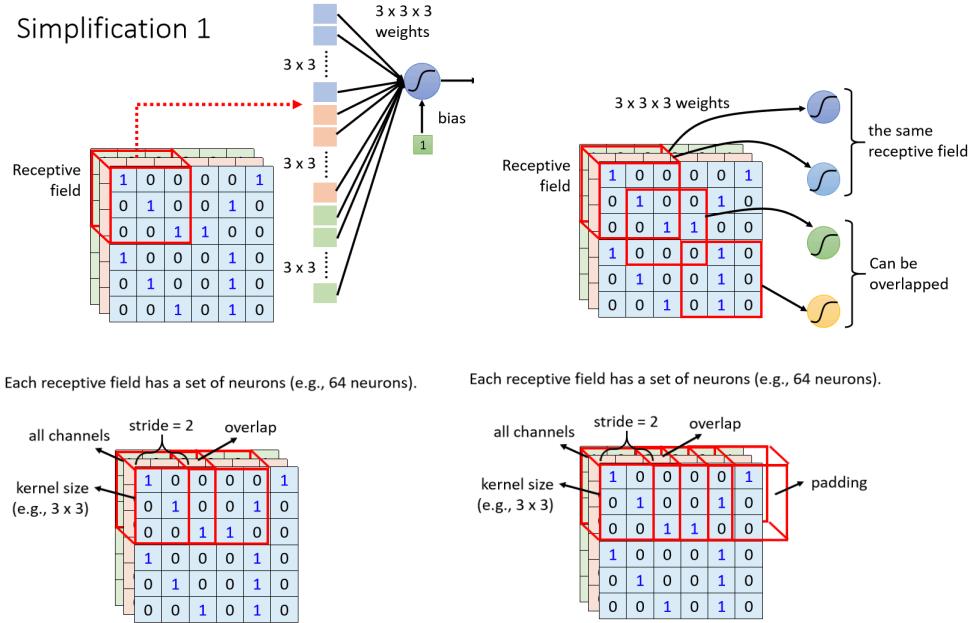


Figure 25: Neuron Design

我们可以根据这些参数来确定卷积的运行过程，与此同时给出 pytorch 中卷积层的代码调用。

```

1 # 卷积层
2 torch.nn.Conv2d(in_channels, out_channels, kernel_size,
3 stride=1, padding=0, dilation=1, groups=1, bias=True,
padding_mode='zeros')

```

用更加通俗易懂的方式解释卷积，将考虑一些 Filter，即使用 Filter 对图像进行卷积操作，一个卷积层中有一排的 Filter，当使用 Filter 扫过一张图片，这也就意味着参数共享，即针对每个感受野中的每个位置的相应参数相同。当使用这些 Filter 扫过图片后，就可以生成 FeatureMap，那么生成的 FeatureMap 可以经过下一层卷积层输出。

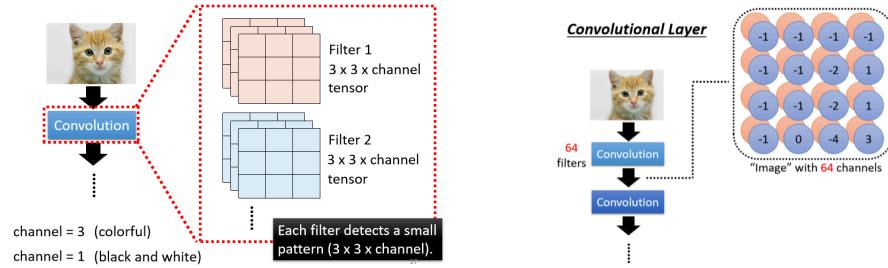


Figure 26: FeatureMap

在卷积层后，往往会有池化层。也就是 Pooling，Pooling 的作用主要是将原图像的尺寸缩小，更具体的说就是针对卷积提取到的每个 FeatureMap 的局部特征进行筛选或融合，选取适当的有代表性的点来表示一个区域，这样做的目的是进行泛化以及降维。

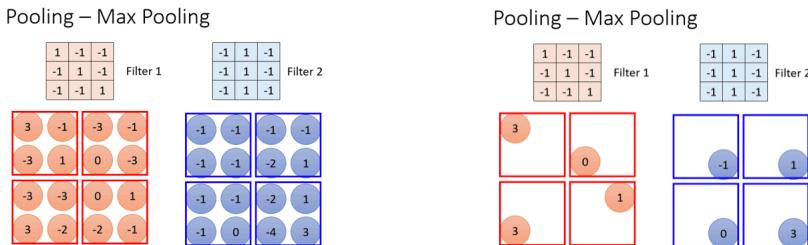


Figure 27: MaxPooling

```

1 # 池化层
2 torch.nn.MaxPool2d(kernel_size, stride=None, padding=0,
3 dilation=1, return_indices=False, ceil_mode=False)

```

通常来说，经过卷积层、池化层后的维度可以很轻松的确定，但是经过这两层之后的图像的尺寸会比较难确定，因此给出了输入、输出前后图像尺寸大小变换的公式。

$$\begin{aligned} Height_{out} &= (Height_{in} - Height_{kernel} + 2 * padding) / stride + 1 \\ Width_{out} &= (Width_{in} - Width_{kernel} + 2 * padding) / stride + 1 \end{aligned} \tag{9}$$

### 3 Self-attention

记录完了 CNN 网络结构后，将记录另一个较为常见的网络架构，即 Self-Attention，通常称为自注意力模型。考虑输入一串语句，那么如何将它们进行联系起来，首先介绍一下 Word Embedding，该策略将会给输入的每一个词汇一个向量，该向量会带有一定的语义资讯，而一个句子就是一排长度不一的向量。假设我们使用全连接层作为网络模型，那么我们输入一段文本文字，例如含有相同单词但是不同意思的文本，对于全连接层网络来说，相同单词它会输出相同的东西，而我们所期望的是网络能够根据上下文信息来学习输出，因此就需要 Self-Attention。

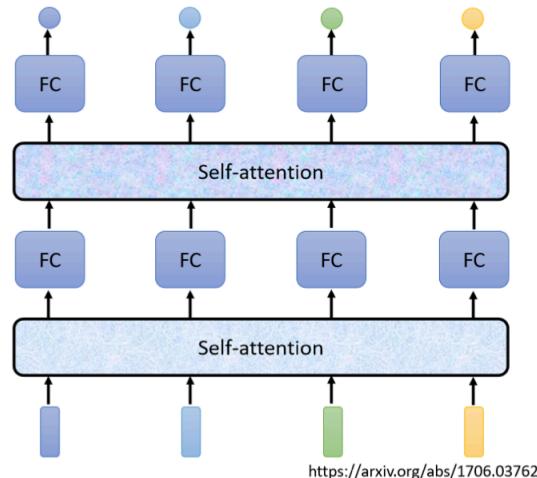


Figure 28: Self-Attention

上面给出了一些使用 Self-Attention 的网络模型架构，在 Self-Attention 中，输入和输出的数量相同，且其输出的向量都是考虑整个输入序列的咨询而产生的。并且可以把 Self-Attention 与全连接层交替使用，Self-Attention 处理整个输入序列的资讯，而全连接层专注于处理某一个位置的资讯处理。

接着将会记录一下 Self-Attention 的整个过程，假设 Self-Attention 的输入输出均为 4 个，如下图所示。

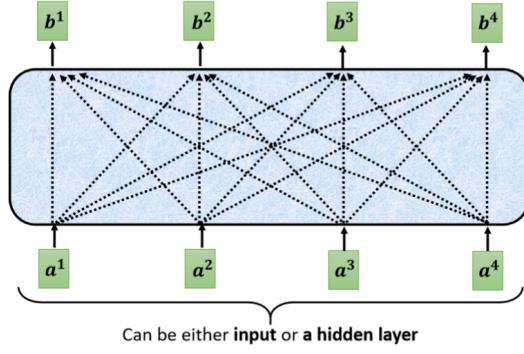


Figure 29: Explanation of Self-Attention

每一个  $b$  均考虑了所有  $a$  才产生出来，并且同时将会考虑输入序列中其他向量与  $a^1$  的关联度，那么如何计算它们的关联度  $\alpha$  呢，因此就需要计算 attention 的模组。

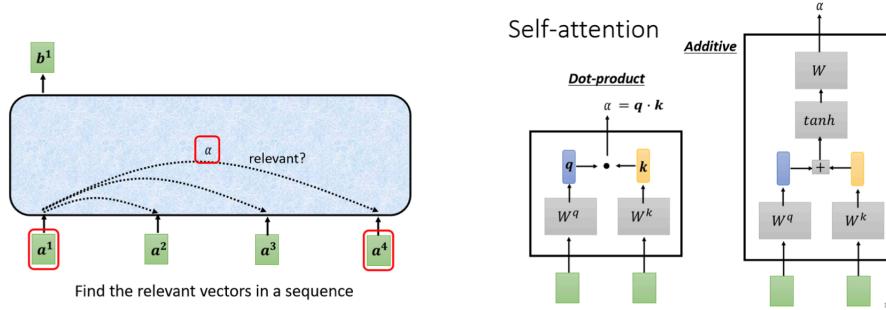
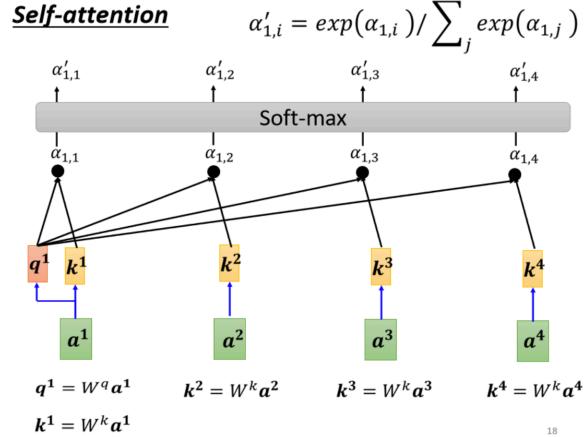


Figure 30: Computation of Self-Attention

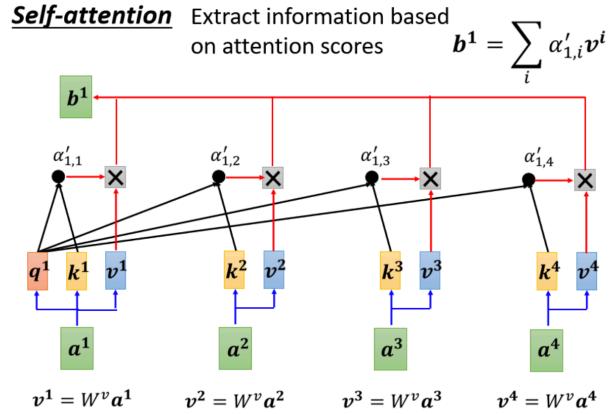
而计算 attention 最常用的方式就是 dot-product，其主要的计算方法就是  $q^1 = a^1 W^q$ ，其中通常称  $q$  为 Query，而对于  $a^2 a^3 a^4$  乘以  $W^k$  则会得到  $k$ ，通常称之为 Key，最后计算 attention 分数  $\alpha_{i,j} = q^i k^j$ ，值得注意的是，往往也会计算自关联，并且在计算出 attention 分数后（即计算出关联性后），往往还会添加一层 softmax。并将  $a^1$  到  $a^4$  的输入向量乘上  $W^v$  得到新的  $v^1 v^2 v^3 v^4$  向量，最后将得到的向量与 softmax 层的输出像乘，得到  $b^i$ ，也就是 Self-Attention 的

输出。

可以发现，在 Self-Attention 中，较为重要的是  $W^q$   $W^k$   $W^v$  三个参数，这些参数是需要通过训练来拟合的。



(a)



(b)

Figure 31: Computation of Self-Attention

接下来则会简要的记录一下 Multi-head Self-Attention，相比于普通的 Self-Attention，该网络模型的变化则是增加参数来计算更多的相关性。简单来

说就是增加一层  $W^q$   $W^k$   $W^v$  参数而已。

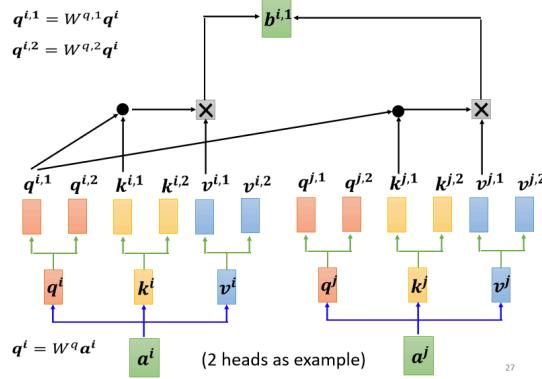


Figure 32: Pattern Design

## 4 Transformer

Transformer 是变形金刚的英文，而 Transformer 模型的特点也和其有些相似，因此将在这一章记录有关 Transformer 相关的知识要点。

Transformer 是 Sequence-to-Sequence 的模型，其通常写作 Seq2seq，序列到序列的模型顾名思义，输入是序列，输出也是序列。更加具体的解读则是输入一段序列，输出序列的长度及内容由模型预测而给出。



Figure 33: Transformer

一般在 Seq2seq 的模型中，一般会分为两块，一块是 Encoder，另一块则是 Decoder，Transformer 中有一个 Encoder 架构以及一个 Decoder 架构，接下来会在两节中记录这两种结构的处理细节。

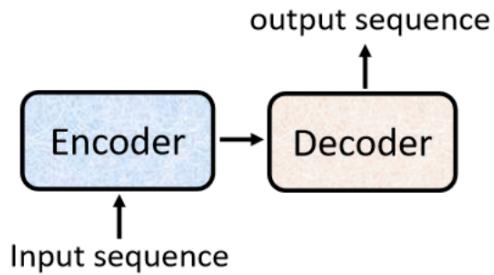


Figure 34: Encoder-Decoder

#### 4.1 Encoder

首先将会从总体来介绍 Encoder 的结构，具体如下图所示，Encoder 所需要做的事情就是输入一排向量并输出另一排向量，因此许多模型都可以做到，例如 RNN、CNN，在 Transformer 中的 Encoder 则是使用 Self-Attention。Encoder 的结构如下图所示。

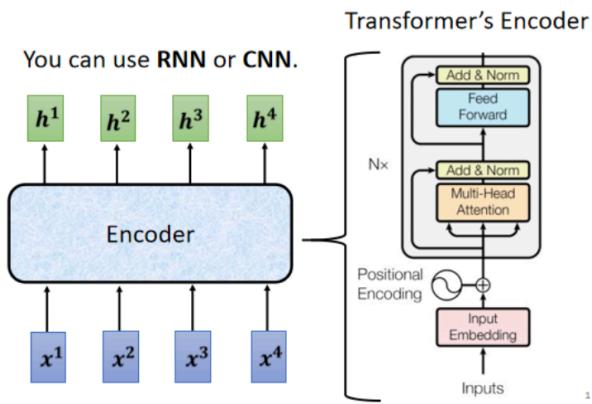


Figure 35: Encoder Architecture

在 Encoder 中，往往会分成很多个 block，每一个 block 由多个神经网络层组，举例来说买，一个 block 可以由一层 Self-Attention 以及一层全联接层组成。

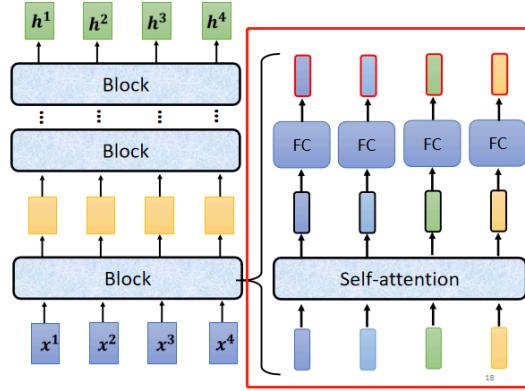


Figure 36: Encoder Block

而在 Transformer 中，block 往往会比较复杂，如下图所示。在经过 Self-Attention 后得到输出向量，在 Transformer 中，不只是输出 vector，还要将输入拉过来加给输出，得到新的 Output。其实这种网络架构称为 residual connection，得到输出后再对其进行标准化 (normlization)。

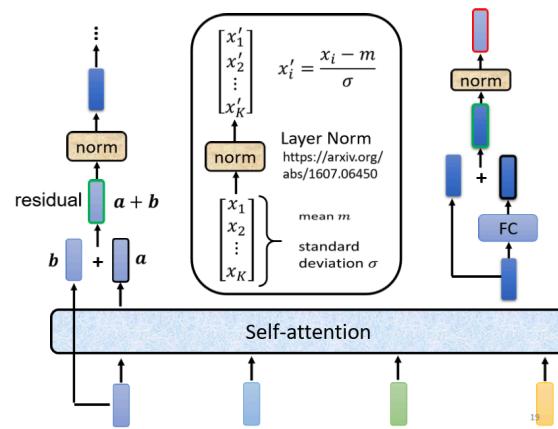


Figure 37: Encoder Self-Attention

这边给出 layer normalization 的公式。相同的，针对全连接层，也同样可以使用残差架构以及标准化来得到输出，最后再看图 35，可以发现 Transformer 的 encoder 所做的就是这些事情。

$$x'_i = \frac{x_i - m}{\sigma} \quad (10)$$

这边给出 layer normalization 的公式。相同的，针对全连接层，也同样可以使用残差架构以及标准化来得到输出，最后再看图 35，可以发现 Transformer 的 encoder 所做的就是这些事情。

## 4.2 Decoder

Decoder 主要分为两种，第一种是 Autoregressive(AT)，另一种是 Non-autoregressive(NAT)，比较常见及经常使用的是 AT。由上节可知，Encoder 的作用是输入一串向量，输出一串向量，而 Decoder 就是将输出向量作为输入得到最终的结果。

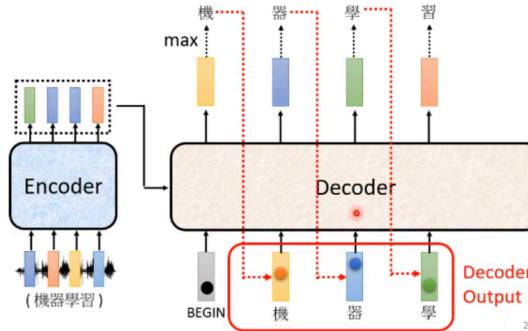


Figure 38: Decoder

从上图可知，Decoder 会接收 Encoder 的输入，也会把自己的输入当作接下來的输出，并且在下图也会给出 Transfomer 的 Decoder 结构，其实可以发现，相比于 Encoder，Decoder 比较不同的方面就是多了一个 Masked Multi-head Attention 以及最后输出的线形、softmax 层。

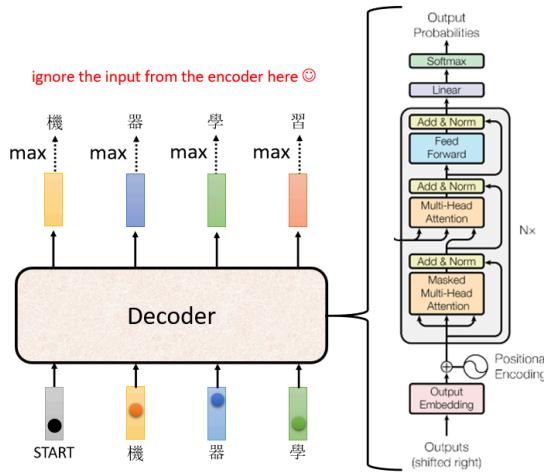


Figure 39: Decoder Architecture

而加入 Masked 与不加 Masked 的区别则是考虑到 Decoder 的运作方式，它是一个一个输出，所以先有  $a^1$  再有  $a^2$ ..... 因此在 Masked Multi-head Attention 中计算关联 Attention 分数的时候，只将当前的输入向量与前面的向量进行计算操作，而不与后面的向量计算分数。具体如下图所示。

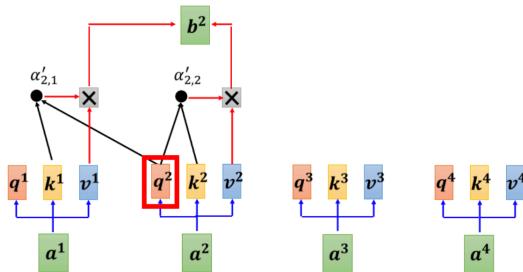


Figure 40: Masked Multi-head Attention

Decoder 中还有一个关键的问题则是它会自己推断输出的序列长度，因此它需要考虑什么时候结束输出，通常在输出预测中会有一个 end 符号，当模型输出 end 时则代表输出结束。

### 4.3 Encoder-Decoder

在分别记录了 Encoder 结构以及 Decoder 的结构之后，则需要考虑 Encoder 与 Decoder 的是如何连接的，这里给出 Transformer 的整个结构，其中连接的部分被框出，其被称为 Cross Attention，其中有两个输入来自 Encoder，Decoder 提供一个输入。

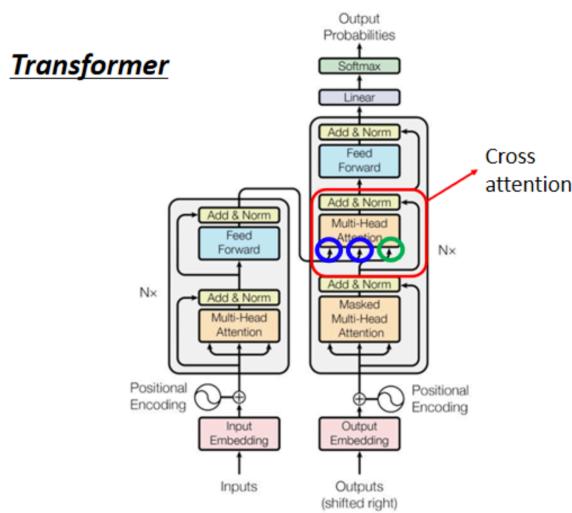


Figure 41: Transformer Architecture

Cross Attention 的具体运作形式如下图所示，简单来说是将 Encoder 的输出向量与 Decoder 的经过一部分变换的输出 (Masked Multi-Head Attention + Add + normalization)，对应于图 41， $q$  来自于 Decoder， $k, v$  来自 Encoder，交叉计算 attention。

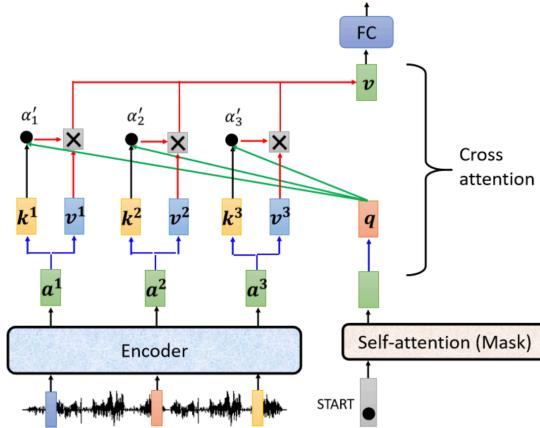


Figure 42: Cross Attention

Transformer 的整体构造还是有些复杂的，更多的训练细节需要在论文中阅读理解。

## 5 To Learn more

至此，已经了解了几个在视觉领域比较基础重要的网络结构，但是课程中涉及更多的是网络结构的设计理解，而更多的小细节需要从论文中去获取，与此同时也需要去复现一些经典论文在实践中提升自己对网络架构的理解。因此按照年份列举一些比较重要的网络结构，以及下图课程中所提到的。将会在下面列出一些网络结构的论文网址

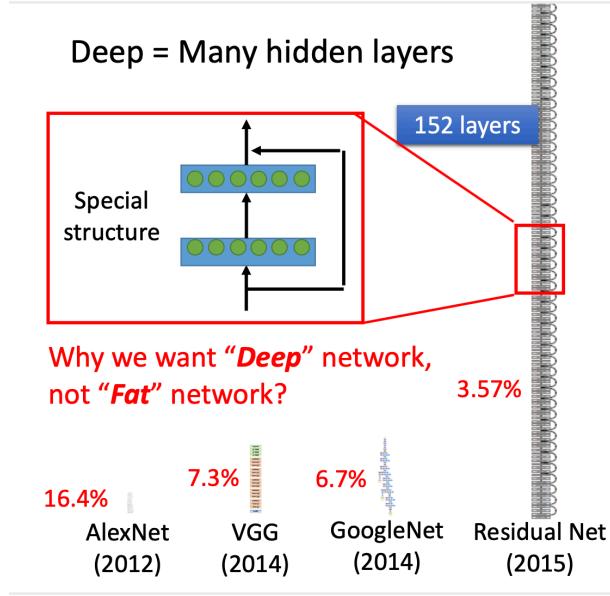


Figure 43: Tipycal Neural Network

AlexNet : <https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf>

VGG : <https://arxiv.org/abs/1409.1556>

GoogLeNet : <https://arxiv.org/abs/1409.4842>

ResNet : <https://arxiv.org/abs/1512.03385>

R-CNN : <https://arxiv.org/abs/1311.2524>

Faster R-CNN : <https://arxiv.org/abs/1506.01497>

Transformer : <https://arxiv.org/abs/1706.03762>