**问题描述：**使用conda自己搭建一个transformer架构和一个CNN，并上传至GitHub上。

为什么要使用conda？

创建 Conda 环境的主要作用是**为项目提供一个隔离的开发环境**，确保每个项目使用自己的依赖和包版本，避免冲突和干扰。具体来说：

**隔离环境，避免冲突：**在开发不同的项目时，可能需要使用不同版本的 Python 或第三方库。比如，一个项目可能需要 numpy 1.18，而另一个项目需要 numpy 1.20。如果你不使用虚拟环境，将所有项目的依赖直接安装在全局环境中，就会导致版本冲突。

通过创建 Conda 环境，你可以在不同的项目中使用各自需要的包和版本，不会互相干扰。

**方便管理和调试项目依赖：**每个 Conda 环境都有自己的包依赖和配置，所有安装的库和工具包只影响当前环境。你可以在不同的环境中安装不同版本的库，而不会影响到其他项目。

**便于项目分享与部署：**当你与他人合作或将项目部署到服务器时，只需提供环境配置（如 environment.yml 文件），对方可以一键安装你项目所需的依赖包，确保项目在其他机器上能正常运行，而无需逐个手动安装依赖。

**兼容性和稳定性：**通过 Conda 环境，你可以锁定项目中依赖包的具体版本，避免因为更新库版本而导致项目代码出错。比如，一个旧项目用 Python 3.6 开发，而另一个新项目需要 Python 3.8，那么你可以为每个项目创建不同的环境来满足各自的需求。

**举例：**

假设你有两个项目：

* **项目 A** 需要 Python 3.8 和 TensorFlow 1.x。
* **项目 B** 需要 Python 3.9 和 TensorFlow 2.x。

如果不使用 Conda 环境，安装新的 TensorFlow 版本会覆盖旧的版本，导致另一个项目可能无法运行。通过 Conda 环境，你可以分别为项目 A 和项目 B 创建各自的隔离环境，分别安装所需的依赖，不会发生冲突。

环境配置文件是什么？怎么生成和使用？

环境配置文件是用来描述和记录某个 Conda 环境中所有依赖包及其版本的文件，通常以 `.yml`（YAML 文件格式）结尾。这种文件允许你轻松地分享环境或在其他计算机上重建相同的环境。

当你开发某个项目时，为了确保你的代码在其他机器上能正常运行（无论是团队协作、服务器部署，还是在未来的开发中），你需要记录项目依赖的具体包和版本。通过环境配置文件，别人可以快速创建与你的开发环境相同的 Conda 环境，而不必手动安装和配置各个依赖。

Conda 环境配置文件的功能：

1. \*\*记录依赖包\*\*：记录项目所需的 Python 版本和第三方库的具体版本。

2. \*\*环境重现\*\*：你可以通过环境配置文件，在任何机器上精确地重建相同的开发环境。

3. \*\*简化协作\*\*：你只需要分享环境文件，对方就能快速创建相同的开发环境，减少版本冲突的可能性。

如何生成和使用环境配置文件？

1. 导出环境配置文件

当你在某个 Conda 环境中已经安装了所有需要的依赖后，可以将其导出为 `.yml` 文件。步骤如下：

- 首先，激活你要导出的 Conda 环境：

```bash

conda activate myenv

```

- 使用 `conda env export` 命令导出环境配置文件：

```bash

conda env export > environment.yml

```

这会生成一个名为 `environment.yml` 的文件，里面包含当前环境的所有包、依赖和版本信息。

2. 使用环境配置文件创建环境

如果你想在另一台机器上重现该环境，或者与他人分享环境，只需使用 `.yml` 文件即可快速创建相同的 Conda 环境。步骤如下：

- 下载或获取 `environment.yml` 文件，然后在终端中执行以下命令创建环境：

```bash

conda env create -f environment.yml

```

- Conda 会根据配置文件中记录的包和版本自动安装依赖，并创建与原始环境相同的环境。

3. 示例 `environment.yml` 文件

这是一个典型的 `environment.yml` 文件的内容示例：

```yaml

name: myenv # 环境的名称

channels:

- defaults

dependencies:

- python=3.8

- numpy=1.19

- pandas=1.2

- tensorflow=2.3

- pip:

- scikit-learn==0.24

- matplotlib==3.3

```

文件包含了：

- 环境名称 (`myenv`)。

- 使用的包来源渠道 (`channels`)。

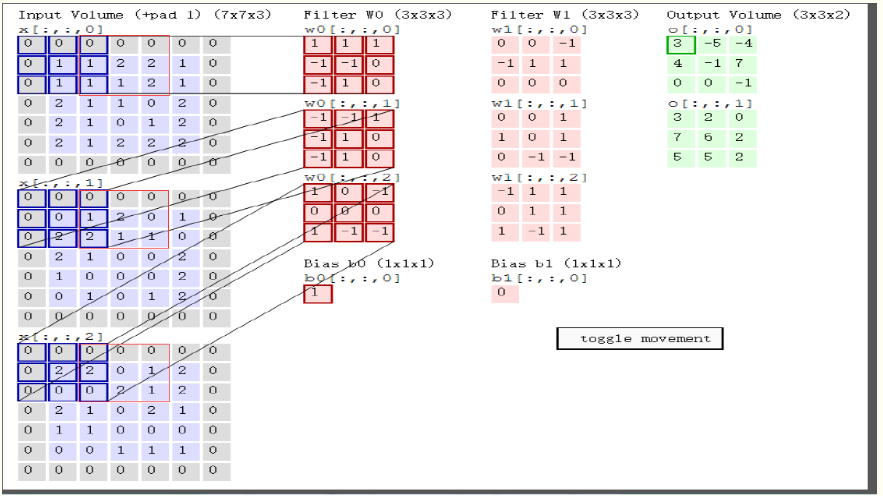
- 所需的包及其版本 (`dependencies`)。

- 如果你使用了 `pip` 安装的包，也可以在 `pip` 部分记录它们。

# CNN

卷积神经网络结构：输入层、卷积层、激活函数、池化层、全连接层、损失函数。实质是**特征提取**和**决策推断**。

卷积：输入是7\*7\*3因为pad=1，补零的作用是能够提取图片边界特征。对于彩色图片，一般都是RGB三种颜色，号称3通道，7\*7指图片的高和宽。Stride是步长，即每次窗口向右或向下滑动两个单位。有几个卷积核就有几个输出结果。



Relu激活函数：f(x)=max(0,x).在卷积层和全连接层之后都有激活函数Relu

为什么要用激活函数？它的作用是什么？

对于第1个问题：由 y = w \* x + b 可知，如果不用激活函数，每个网络层的输出都是一种线性输出，而我们所处的现实场景，其实更多的是各种非线性的分布。这也说明了激活函数的作用是将线性分布转化为非线性分布，能更逼近我们的真实场景。

在CNN中为什么要用Relu，相比于sigmoid，tanh，它的优势在什么地方？

Sigmoid和tanh在x->+∞时的输出变成了恒定值，在求梯度时他们的偏导为0，存在梯度消失问题，会导致w和b无法更新。在x>0时Relu的导数=1，在反向传播计算dw，db时能够简化运算。使用sigmoid还会存在梯度爆炸问题，在进行前向传播和反向传播迭代次数非常多的情况下，sigmoid因为是指数函数，其结果中某些值会在迭代中累积，并成指数级增长，最终会出现NaN而导致溢出

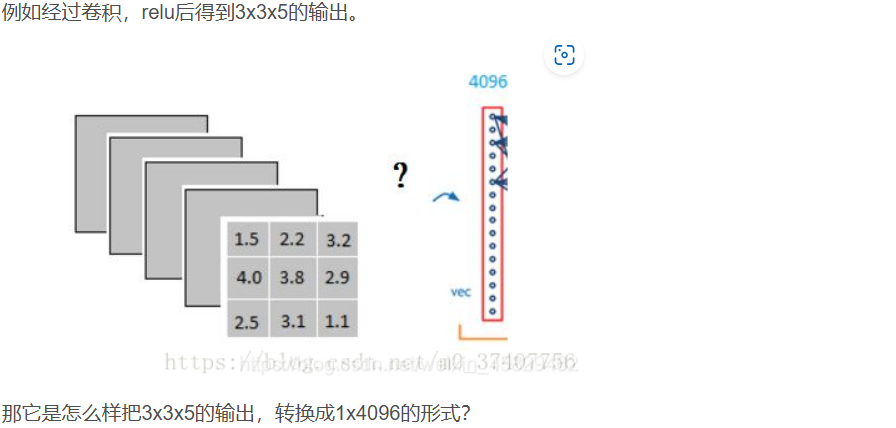
池化(pooling)：池化层一般在卷积层+ Relu之后，它的作用是：

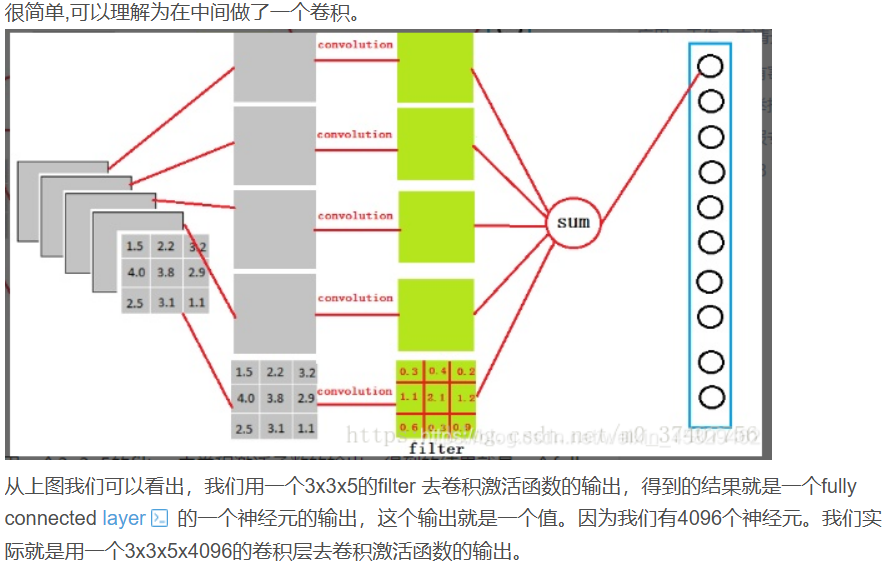
1、减小输入矩阵的大小（只是宽和高，而不是深度），提取主要特征。（不可否认的是，在池化后，特征会有一定的损失，所以，有些经典模型就去掉了池化这一层）。

它的目的是显而易见的，就是在后续操作时能降低运算。

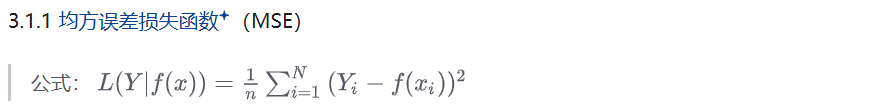
2、一般采用mean\_pooling（均值池化）和max\_pooling（最大值池化），对于输入矩阵有translation（平移），rotation（旋转），能够保证特征的不变性。

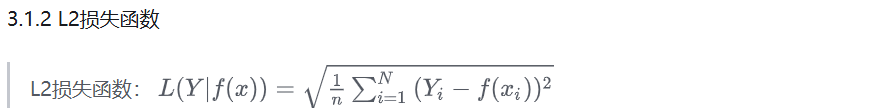
全连接：在整个卷积神经网络中起到“分类器”的作用。如果说卷积层、池化层和激活函数等操作是将原始数据映射到隐层特征空间的话，全连接层则起到将学到的“分布式特征表示”映射到样本标记空间的作用。在实际使用中，全连接层可由卷积操作实现：对前层是全连接的全连接层可以转化为卷积核为1x1的卷积；而前层是卷积层的全连接层可以转化为卷积核为hw的全局卷积，hw分别为前层卷积结果的高和宽。

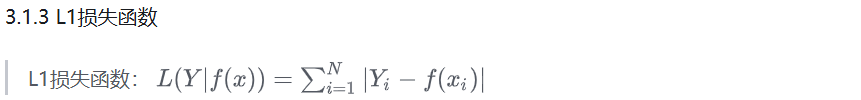


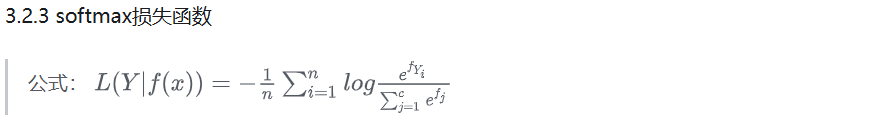
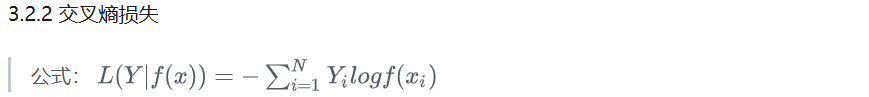


损失函数：计算损失loss，从而求出梯度grad。常用损失函数有:MSE均方误差，SVM（支持向量机）合页损失函数，Cross Entropy交叉熵损失函数。







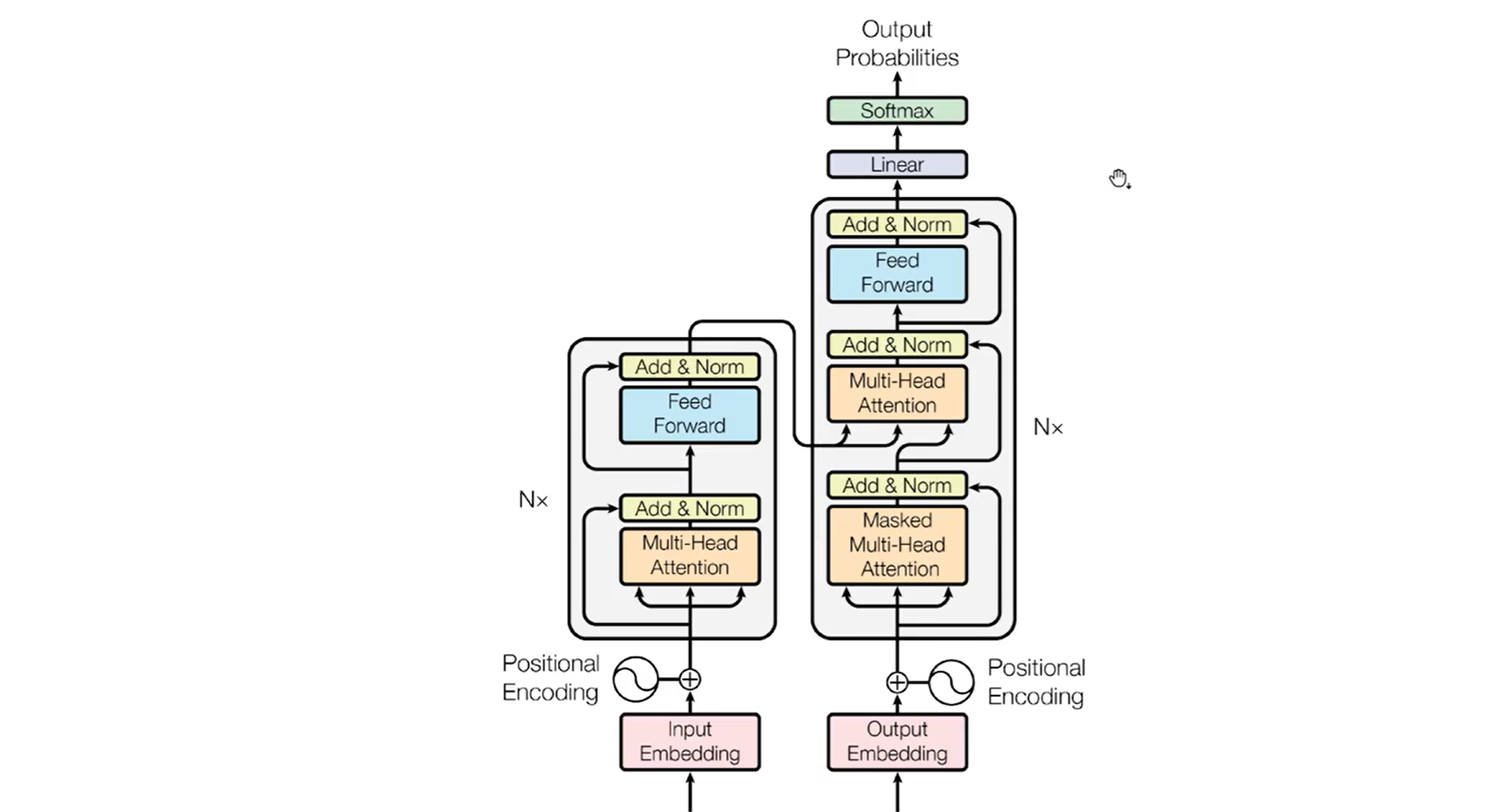


前向传播：包含卷积，Relu激活函数，池化（pool），全连接(fc)，可以说，在损失函数之前操作都属于前向传播。主要是权重参数w , b 初始化，迭代，以及更新w, b,生成分类器模型。

反向传播：更新模型的权重，以最小化损失函数。步骤：前向传播、计算损失、计算梯度、更新权重。

# Transformer

编码器(提取特征信息)和解码器(根据特征信息“破译”目标内容)通常配套使用，但是编码器和解码器彼此之间在模型结构和用途上都相互独立，因此我们也可以只用编码器或者只用解码器。



Encoder：多头自注意力机制的输入是input的三个copy：values、keys、queries

Decoder：多头自注意力机制的输入是来自encoder的values和keys，以及来自之前decoder的queries

Positional Encoding：使transformer注意到输入位置的变化