《人工智能与Python程序设计》——PyTorch神经网络

人工智能与Python程序设计 教研组

PyTorch神经网络

提纲

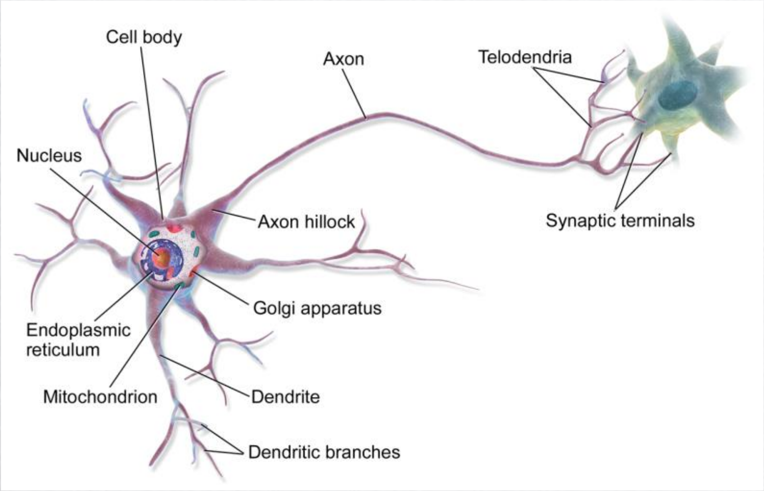
- 神经网络概述
- 使用PyTorch搭建神经网络
- 使用PyTorch训练神经网络
- 卷积神经网络与循环神经网络

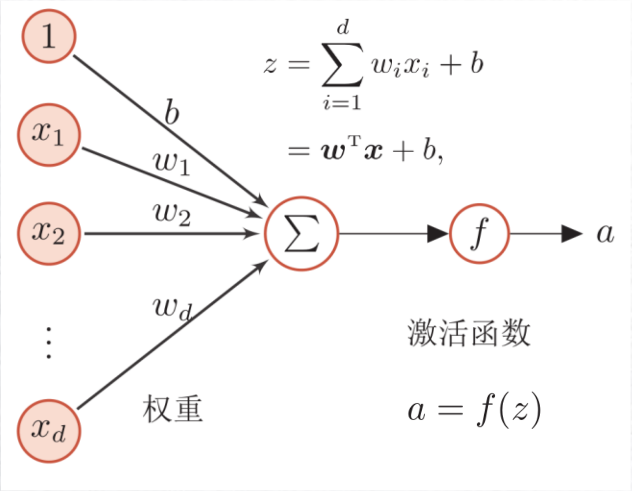
神经网络概述

- 神经网络
  - 神经网络可以指向两种
    - **生物**神经网络
    - **人工**神经网络
  - 生物神经网络
    - 一般指生物的大脑神经元、细胞、触点等组成的网络
    - 用于产生生物的意识，帮助生物进行思考和行动
  - 人工神经网络也简称为神经网络（Neural Network）
    - 模仿动物神经网络的行为特征，进行分布式、深层信息处理的算法模型

神经网络概述

- 生物神经元 *v.s.* 人工神经元





单个神经细胞有两种状态：
抑制或者兴奋

线性模型配上激活函数

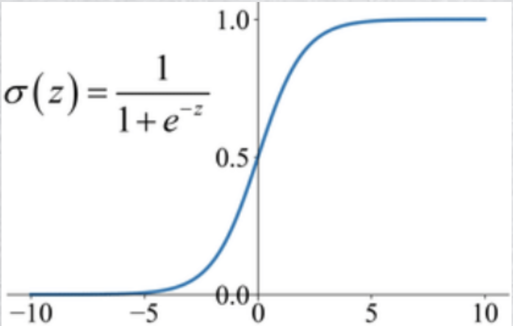
神经网络概述

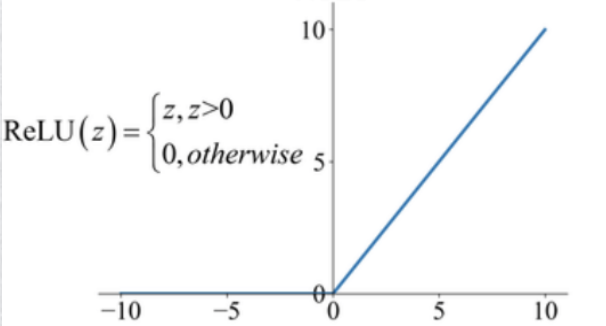
- 激活函数

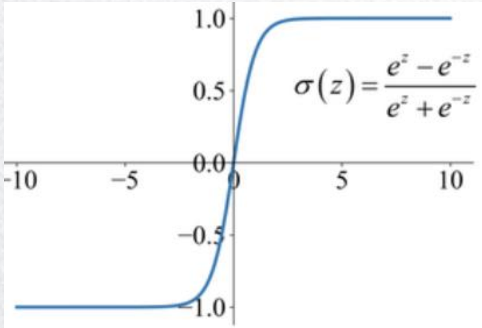
$$\sigma(x) = \frac{1}{1+\exp(-x)}$$

$$\begin{aligned}\mathrm{ReLU}(x) &= \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases} \\ &= \max(0, x).\end{aligned}$$

$$\tanh(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$$

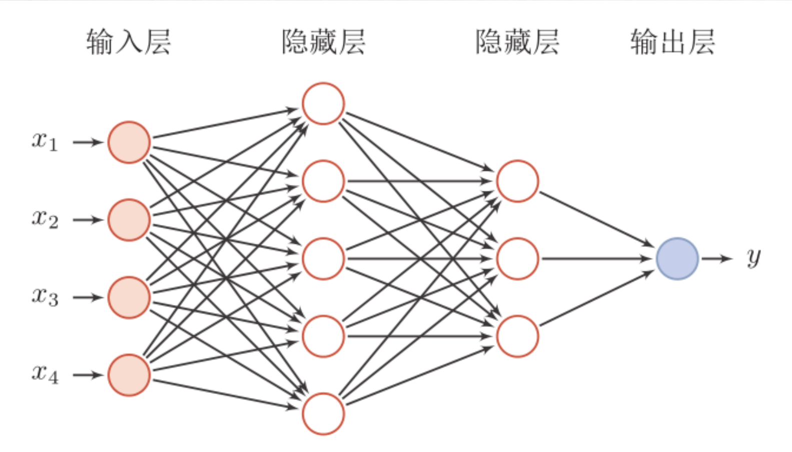






神经网络概述

- 最常见的网络结构
  - 前馈神经网络
    - 也称为多层感知机（MLP）或者全连接网络



$$\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)}\boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)},$$

$$\boldsymbol{a}^{(l)} = f_l(\boldsymbol{z}^{(l)}).$$

记号	含义
L	神经网络的层数
M_l	第 l 层神经元的个数
$f_l(\cdot)$	第 l 层神经元的激活函数
$\boldsymbol{W}^{(l)} \in \mathbb{R}^{M_l \times M_{l-1}}$	第 $l-1$ 层到第 l 层的权重矩阵
$\boldsymbol{b}^{(l)} \in \mathbb{R}^{M_l}$	第 $l-1$ 层到第 l 层的偏置
$\boldsymbol{z}^{(l)} \in \mathbb{R}^{M_l}$	第 l 层神经元的净输入（净活性值）
$\boldsymbol{a}^{(l)} \in \mathbb{R}^{M_l}$	第 l 层神经元的输出（活性值）

神经网络概述

- 神经网络本质上是一种复合函数

$$\hat{y} = f^L(\dots(f^3(f^2(f^1(x)))))$$

- 两个重要过程
  - 前向计算 (Forward computation)

$$\boldsymbol{x} \rightarrow \boldsymbol{a}^{(0)} \rightarrow \boldsymbol{z}^{(1)} \rightarrow \boldsymbol{a}^{(1)} \rightarrow \boldsymbol{z}^{(2)} \rightarrow \cdots \rightarrow \boldsymbol{a}^{(L-1)} \rightarrow \boldsymbol{z}^{(L)} = \hat{y}$$

  - 后向求导 (Back-propagation) [感兴趣的同学阅读注释链接]

$$\boxed{\delta^l} = ((w^{l+1})^T \boxed{\delta^{l+1}}) \odot \sigma'(z^l)$$

从后回传误差

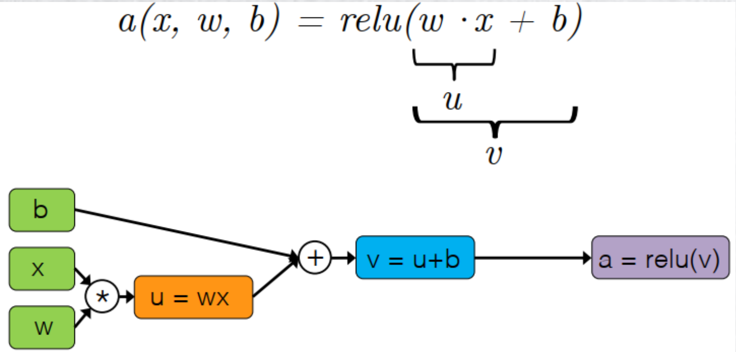
$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \text{ and } \frac{\partial C}{\partial b^l_j} = \delta^l_j.$$

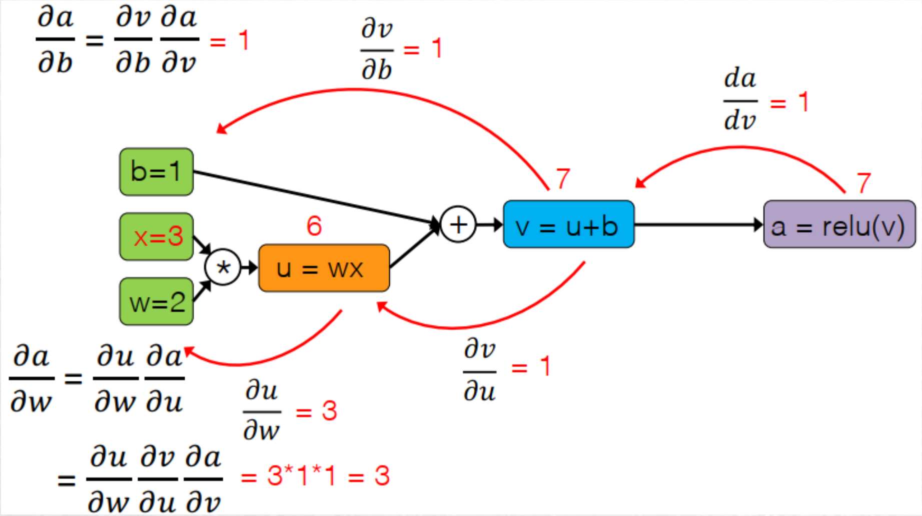
逐层计算导数

http://neuralnetworksanddeeplearning.com/chap2.html

神经网络概述

- 自动求导的本质
  - 构建计算图，通过可达的路径累积计算导数





PyTorch是如何完成自动求导的?

- 数学原理:
  - 以向量为输入，向量为输出的函数: $\boldsymbol{y} = f(\boldsymbol{x}), \boldsymbol{y} \in R^m, \boldsymbol{x} \in R^n$
  - 雅克比矩阵（Jacobian Matrix）：

$$\boldsymbol{J}_f = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \frac{\partial y_i}{\partial x_j} & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}$$

  - $l = g(\boldsymbol{y}), l \in R$

$$\boldsymbol{v} = \frac{\partial l}{\partial \boldsymbol{y}} = \begin{pmatrix} \frac{\partial l}{\partial y_1} & \cdots & \frac{\partial l}{\partial y_m} \end{pmatrix}$$

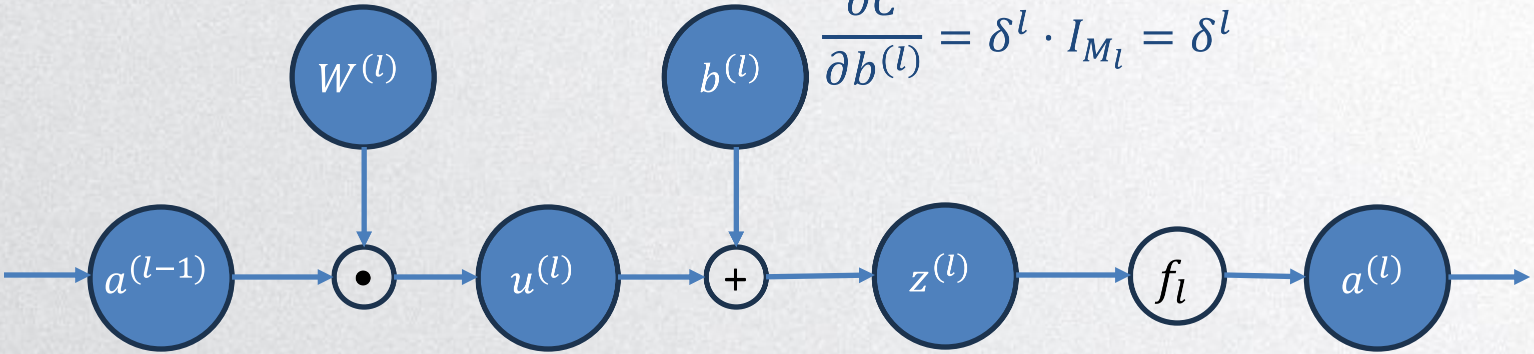
$$\frac{\partial l}{\partial \boldsymbol{x}} = \boldsymbol{v} \cdot \boldsymbol{J}_f$$

神经网络概述

- 反向传播求梯度 (Back-propagation)：

$$\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)}\boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)},$$
$$\boldsymbol{a}^{(l)} = f_l(\boldsymbol{z}^{(l)}).$$

$$\frac{\partial C}{\partial b^{(l)}} = \delta^l \cdot I_{M_l} = \delta^l$$



$$\frac{\partial C}{\partial u^{(l)}} = \delta^l \cdot I_{M_l} = \delta^l$$

$$\delta^l := \frac{\partial C}{\partial z^{(l)}}$$

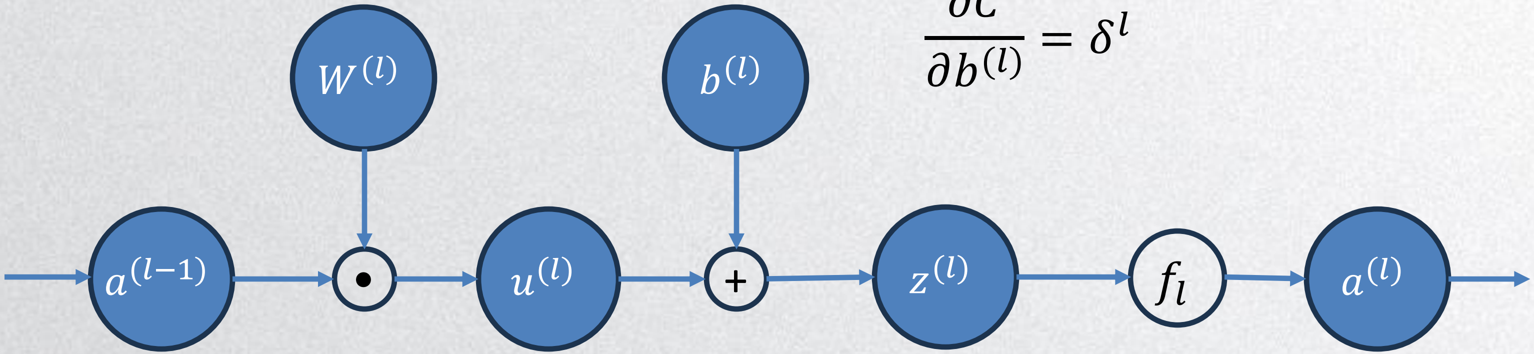
神经网络概述

- 反向传播求梯度(Back-propagation)：

$$\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)}\boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)},$$
$$\boldsymbol{a}^{(l)} = f_l(\boldsymbol{z}^{(l)}).$$

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \delta_j^l \cdot a_k^{(l-1)}$$

$$\frac{\partial C}{\partial b^{(l)}} = \delta^l$$

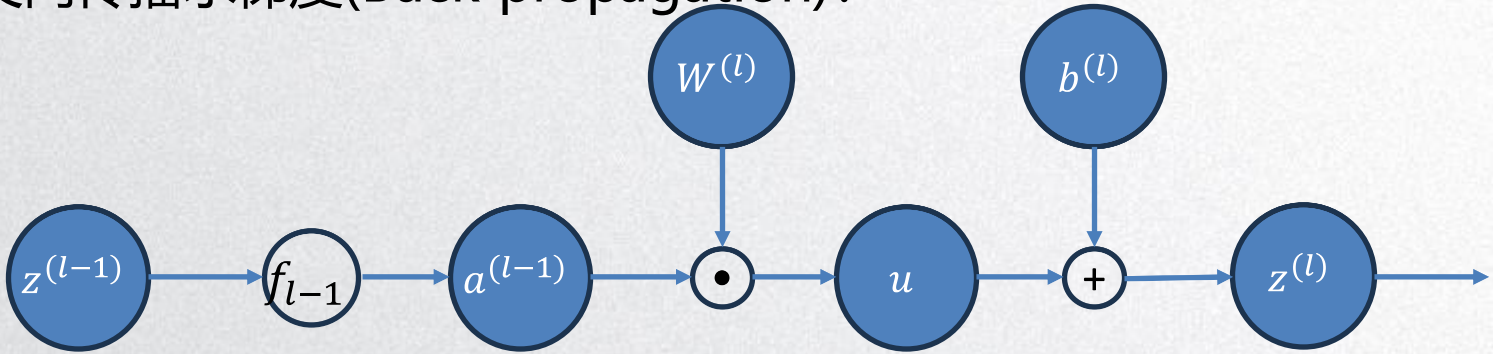


$$\frac{\partial C}{\partial a^{(l-1)}} = \delta_j^l \cdot {W^{(l)}}^T \qquad \frac{\partial C}{\partial u^{(l)}} = \delta^l \qquad \delta^l := \frac{\partial C}{\partial z^{(l)}}$$

神经网络概述

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)},$$
$$a^{(l)} = f_l(z^{(l)}).$$

- 反向传播求梯度(Back-propagation)：



$$\delta^{l}:=\frac{\partial C}{\partial z^{(l)}}$$

$$\delta^{l-1}=\frac{\partial C}{\partial a^{(l-1)}}\cdot J_{f_{l-1}} \qquad \frac{\partial C}{\partial a^{(l-1)}}=\delta_j^l\cdot {W^{(l)}}^T$$

$$=\left(\delta_j^l\cdot {W^{(l)}}^T\right)*f'_{l-1}(z^{(l-1)})$$

$$J_{f_{l-1}}=\begin{pmatrix} f'_{l-1}(z_1^{(l-1)}) & \cdots & \\ \vdots & \ddots & \vdots \\ & \cdots & f'_{l-1}(z_{M_{l-1}}^{(l-1)}) \end{pmatrix}$$

神经网络概述

- 优化的一般流程
  - 规定函数形式，明确如下
    - 参数（需要学习，一般用w表示）
    - 输入（始终给定，一般用x表示）
    - 输出（训练时给定，一般用y表示）
    - 预测（由模型的出来，一般用$\hat{y}$表示）
  - 构建关于输出y与预测$\hat{y}$之间的损失函数 $L(y, \hat{y})$
  - 以参数为待求目标，进行反向求导 $\frac{\partial L}{\partial w}$
  - 使用梯度下降（或者其变种）进行参数的更新:

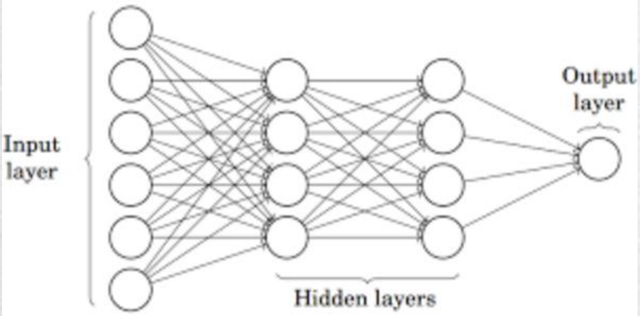
$$w^{new} = w^{old} - lr * \frac{\partial L}{\partial w}$$

  - 迭代多轮，直至收敛

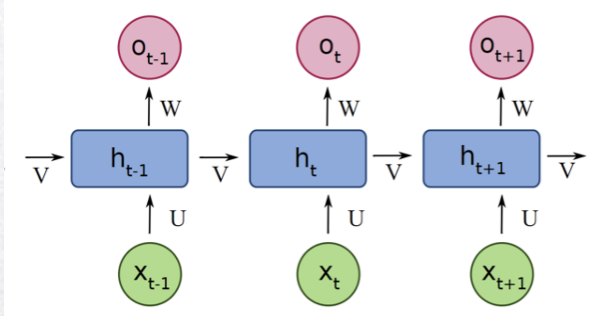
神经网络概述

- 常见的神经网络

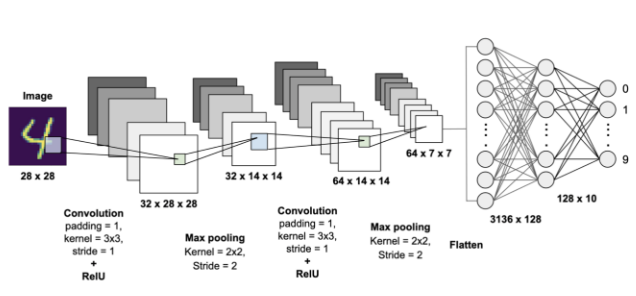
多层感知机



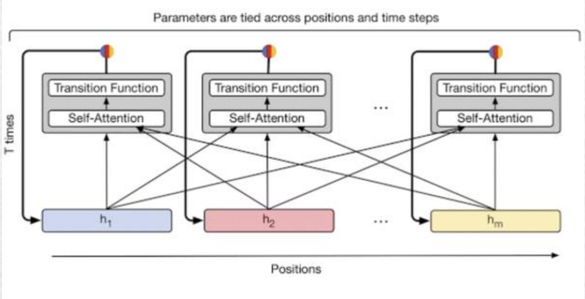
循环神经网络



卷积神经网络



自注意力机制网络



神经网络概述

- 之前的函数
  - 多元线性回归
    - 只有一个线性层
    - $\hat{y}=\boldsymbol{x}\cdot\boldsymbol{w}+b$
  - 多元逻辑回归
    - 只有一个线性层+sigmoid激活函数
    - $\hat{y}=\sigma(\boldsymbol{x}\cdot\boldsymbol{w}+b)$

PyTorch神经网络

提纲

- 神经网络概述
- **使用PyTorch搭建神经网络**
- 使用PyTorch训练神经网络
- 卷积神经网络与循环神经网络

回顾：使用PyTorch实现线性回归模型

- 优化的一般流程
  - **规定函数形式，明确如下**
    - **参数（需要学习，一般用w表示）**
    - **输入（始终给定，一般用x表示）**
    - **输出（训练时给定，一般用y表示）**
    - **预测（由模型的出来，一般用$\hat{y}$表示）**
  - 构建关于输出y与预测$\hat{y}$之间的损失函数 $L(y, \hat{y})$
  - 以参数为待求目标，进行反向求导 $\frac{\partial L}{\partial w}$
  - 使用梯度下降（或者其变种）进行参数的更新:

$$w^{new} = w^{old} - lr * \frac{\partial L}{\partial w}$$

  - 迭代多轮，直至收敛

回顾：使用PyTorch实现线性回归模型

- 继承nn.Module，实现线性回归模型

```
class LinearRegression(nn.Module):
    def __init__(self, in_dim): #构造函数，需要调用nn.Mudule的构造函数
        super().__init__()          #等价于nn.Module.__init__()
        self.w=nn.Parameter(torch.randn(in_dim+1, 1))

    def forward(self, x):
        x = torch.cat([x, torch.ones((x.shape[0],1))], dim = 1)
        x = x.matmul(self.w)
        return x
```

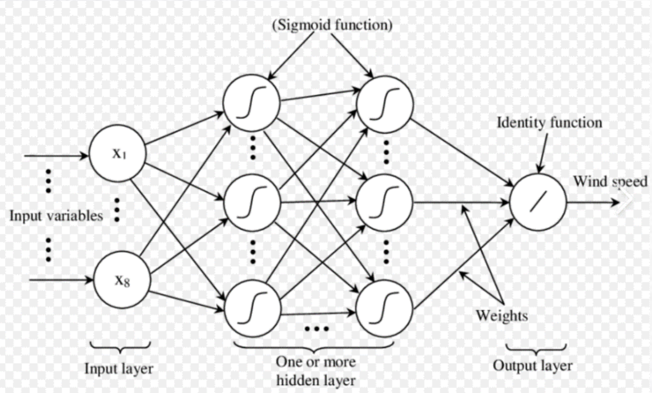
- 自己实现了一个线性层

回顾：使用PyTorch实现线性回归模型

- 继承nn.Module，实现线性回归模型：
  - 步骤1：实现__init__方法，初始化线性回归参数，保存到self.w属性中
    - 模型参数需要是nn.Parameter类型的，这样该属性才能被self.parameters()迭代器遍历，进而被optimizer优化和更新
    - 不要忘了调用super().__init__()
  - 步骤2：实现forward方法，计算$\hat{y} = \boldsymbol{x}\boldsymbol{w}$

目标：实现一个基于MLP的回归模型

- 继承nn.Module，实现MLP（Multi-Layer Perceptron）回归模型：
  - 步骤1：实现__init__方法
  - 步骤2：实现forward方法
- 使用torch.nn中自带的神经网络模块实现该模型
  - nn包中自带很多常用的神经网络模块
    - 线性层
    - 激活函数
  - 这些模块均继承nn.Module
  - 我们常把这些模块称为神经网络中的”层”



PyTorch多个线性层

- Pytorch堆积两个线性层（激活函数为sigmoid）

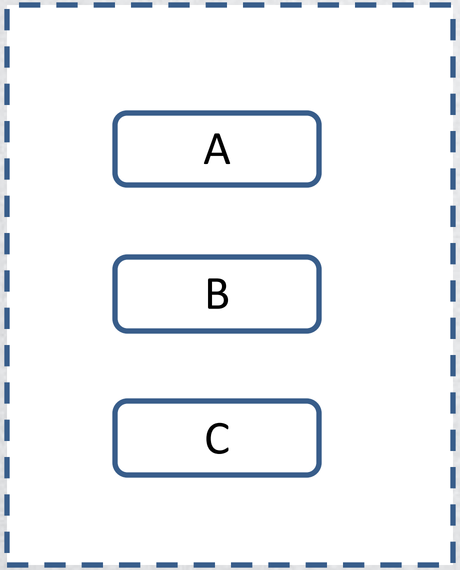
$$\boldsymbol{z} = f(\boldsymbol{x} \cdot \boldsymbol{W}_1 + \boldsymbol{b}_1)$$

$$\hat{y} = f(\boldsymbol{z} \cdot \boldsymbol{W}_2 + \boldsymbol{b}_2)$$

  – 如何搭建这样一个网络

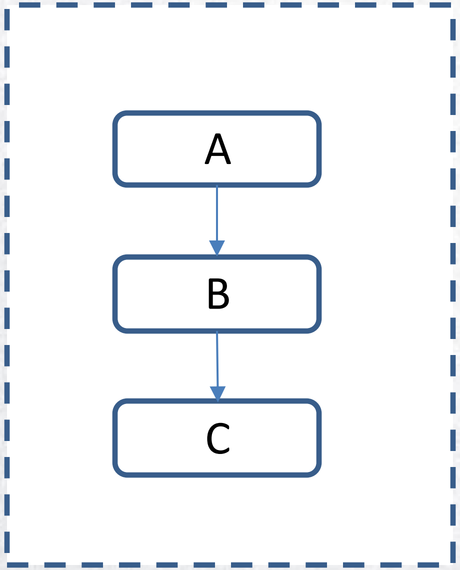
PyTorch多个线性层

- 构造函数与forward方法的“分工”：



__init__()

“挂载” 成员变量



forward()

连同数据流，形成前向计算

选零件：PyTorch中提供的“层” （Layer）

- 选择层函数
  - https://pytorch.org/docs/stable/nn.html

    - 线性层：
      - **nn.Linear**
    - 循环层
      - nn.RNN/nn.LSTM/nn.GRU/nn.RNNCell/nn.LSTMCell/nn.GRUCell
    - 卷积层
      - nn.Conv1d/ nn.Conv2d/ nn.Conv3d
    - 池化层
      - nn.MaxPool1d/ nn.MaxPool2d/nn.MaxPool3d

PyTorch线性层

- torch.nn.Linear(in_features, out_features, bias=True)
  - **in_features**: 输入特征的维数，即输入的张量shape：[data_size, in_dim]中的in_dim
  - **out_features**: 输出数据的维数，即输出的二维张量的形状为[data_size，out_dim]，也代表了该全连接层的神经元个数
    - data_size: 输入特征的总数（样本数）
  - **Bias**: 线性变换$\boldsymbol{x} \cdot \boldsymbol{W} + b$是否加b

PyTorch线性层

- 线性层（全连接层）
- $\hat{\boldsymbol{y}} = \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{b}$

$$
\begin{bmatrix}
x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(d)} \\
x_2^{(1)} & x_2^{(2)} & \cdots & x_2^{(d)} \\
\vdots & \vdots & & \vdots \\
x_N^{(1)} & x_N^{(2)} & \cdots & x_N^{(d)}
\end{bmatrix}
\begin{bmatrix}
w_1^{(1)} & w_1^{(2)} & \cdots & w_1^{(d\prime)} \\
w_2^{(1)} & w_2^{(2)} & \cdots & w_2^{(d\prime)} \\
\vdots & \vdots & & \vdots \\
w_d^{(1)} & w_d^{(2)} & \cdots & w_d^{(d\prime)}
\end{bmatrix}
+
\begin{bmatrix}
b_1 & b_2 & \vdots & b_{d\prime}
\end{bmatrix}
=
\begin{bmatrix}
y_1^{(1)} & y_1^{(2)} & \cdots & y_1^{(d\prime)} \\
y_2^{(1)} & y_2^{(2)} & \cdots & y_2^{(d\prime)} \\
\vdots & \vdots & & \vdots \\
y_N^{(1)} & y_N^{(2)} & \cdots & y_N^{(d\prime)}
\end{bmatrix}
$$

Pytorch线性回归

- 多目标回归：回归值是多维向量

```python
class Linear_Model():
    def __init__(self, in_dim, out_dim):
        """
        创建模型和优化器，初始化线性模型和优化器超参数
        """
        self.learning_rate = 0.01
        self.epoches = 10000
        #self.model = LinearRegression(in_dim) #torch.nn.Linear(in_dim,1)
        self.model = torch.nn.Linear(in_dim,out_dim)
        self.optimizer = torch.optim.SGD(self.model.parameters(), lr=self.learning_rate)
        self.loss_function = torch.nn.MSELoss()
```

选零件： PyTorch中提供的激活函数

- 选择激活函数
  - https://pytorch.org/docs/stable/nn.html
    - **nn.Sigmoid**
    - nn.Tanh
    - nn.ReLU
  - …

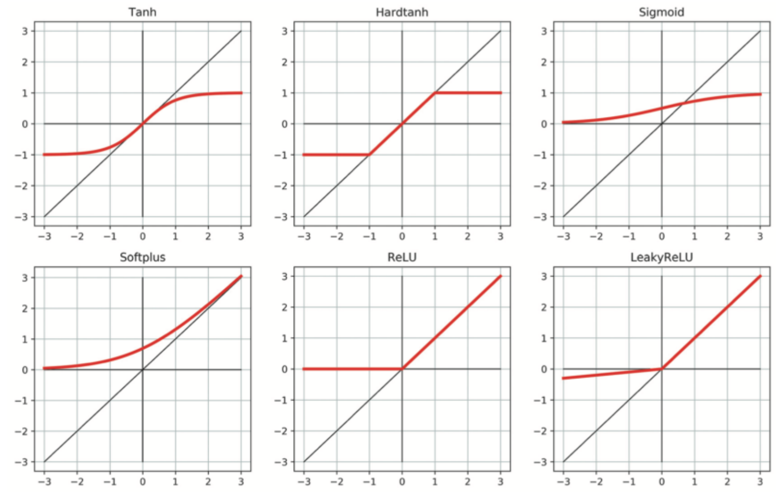


Figure 5.5 A collection of common and not-so-common activation functions

搭积木：PyTorch实现多个线性层

- PyTorch堆积两个线性层（激活函数为sigmoid）

$$z = f(x \cdot W_1 + b_1)$$

$$\hat{y} = f(z \cdot W_2 + b_2)$$

```python
class MLPRegression(nn.Module):
    def __init__(self, in_dim, hidden_dim):
        super().__init__()
        self.first_layer = torch.nn.Linear(in_dim, hidden_dim, bias=True) #set the first layer
        self.second_layer = torch.nn.Linear(hidden_dim, 1, bias=True) #set the second layer
        self.sigmoid = torch.nn.Sigmoid()

    def forward(self, x):
        hidden_layer = self.sigmoid( self.first_layer(x) )
        output = self.sigmoid( self.second_layer(hidden_layer) )
        return output
```

准备“积木”

搭积木：PyTorch实现多个线性层

- PyTorch堆积两个线性层（激活函数为sigmoid）

$$z = f(x \cdot W_1 + b_1)$$

$$\hat{y} = f(z \cdot W_2 + b_2)$$

```
class MLPRegression(nn.Module):
    def __init__(self, in_dim, hidden_dim):
        super().__init__()
        self.first_layer = torch.nn.Linear(in_dim, hidden_dim, bias=True) #set the first layer
        self.second_layer = torch.nn.Linear(hidden_dim, 1, bias=True) #set the second layer
        self.sigmoid = torch.nn.Sigmoid()

    def forward(self, x):
        hidden_layer = self.sigmoid( self.first_layer(x) )
        output = self.sigmoid( self.second_layer(hidden_layer) )
        return output
```

搭“积木”

搭积木：PyTorch实现多个线性层

- 如何确定参数在哪里？是什么
  - 通过 parameters() 或者 named_parameters()

```python
def testMLPRegression(in_dim, hidden_dim, data_size):
    mlpr = MLPRegression(in_dim, hidden_dim)
    input = torch.randn(data_size, in_dim)
    output = mlpr(input)
    for name, parameters in mlpr.named_parameters():#parameters()
        print('[', name, ']',  parameters)

testMLPRegression(4, 2, 1)
```

```
[ first_layer.weight ] Parameter containing:
tensor([[ 0.4034,  0.1987, -0.3451,  0.3046],
        [-0.0597,  0.3907,  0.4175,  0.0442]], requires_grad=True)
[ first_layer.bias ] Parameter containing:
tensor([ 0.0213, -0.4166], requires_grad=True)
[ second_layer.weight ] Parameter containing:
tensor([[0.4981, 0.0724]], requires_grad=True)
[ second_layer.bias ] Parameter containing:
tensor([0.1699], requires_grad=True)
```

- 一个nn.Module的子类（MLPRegression）的parameters()方法能够用来遍历它的所有**实例属性**中:
  - nn.Parameter类对象
  - nn.Module子类对象的parameters()方法能够遍历的对象
  - 递归思想！

PyTorch神经网络

提纲

- 神经网络概述
- 使用PyTorch搭建神经网络
- **使用PyTorch训练神经网络**
- 卷积神经网络与循环神经网络

如何训练神经网络

- 模型训练和优化的一般流程
  - 规定函数形式，明确如下
    - 参数（需要学习，一般用w表示）
    - 输入（始终给定，一般用x表示）
    - 输出（训练时给定，一般用y表示）
    - 预测（由模型的出来，一般用$\hat{y}$表示）
  - 构建关于输出y与预测$\hat{y}$之间的**损失函数** $L(y, \hat{y})$
  - 以参数为待求目标，进行反向求导 $\frac{\partial L}{\partial w}$
  - 使用**梯度下降（或者其变种）**进行参数的更新:

$$w^{new} = w^{old} - lr * \frac{\partial L}{\partial w}$$

  - 迭代多轮，直至收敛

如何训练神经网络

- 模型训练和优化的一般流程
  - 规定函数形式，明确如下
    - 参数（需要学习，一般用w表示）
    - 输入（始终给定，一般用x表示）
    - 输出（训练时给定，一般用y表示）
    - 预测（由模型的出来，一般用$\hat{y}$表示）
  - 构建关于输出y与预测$\hat{y}$之间的**损失函数** $L(y, \hat{y})$
  - 以参数为待求目标，进行反向求导 $\frac{\partial L}{\partial w}$
  - 使用**梯度下降（或者其变种）**进行参数的更新:

$$w^{new} = w^{old} - lr * \frac{\partial L}{\partial w}$$

  - 迭代多轮，直至收敛

如何使用PyTorch训练神经网络

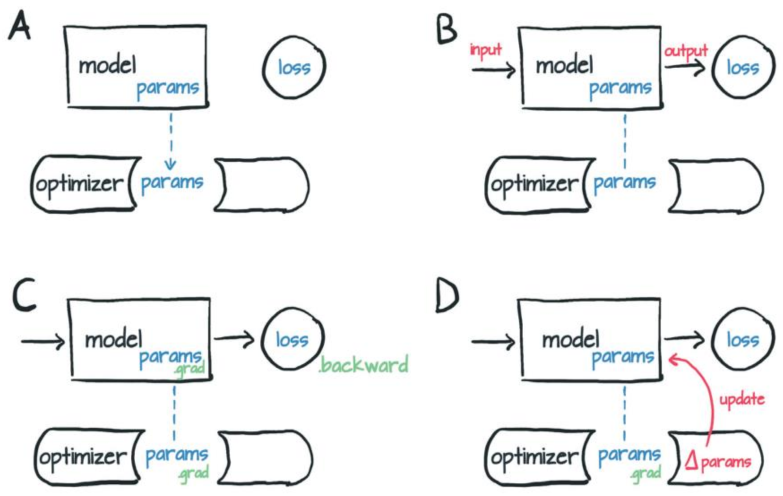


Figure 4.10 Conceptual representation of how an optimizer holds a reference to parameters (A), and after a loss is computed from inputs (B), a call to .backward leads to .grad being populated on parameter (C). At that point, the optimizer can access .grad and compute the parameter updates (D).

```python
def train(self, x, y):
    """
    训练模型并保存参数
    输入:
        model_save_path: saved name of model
        x: 训练数据
        y: 回归真值
    返回:
        losses: 所有迭代中损失函数值
    """
    losses = []
    for epoch in range(self.epoches):
        prediction = self.model(x)
        loss = self.loss_function(prediction, y)

        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        losses.append(loss.item())

        if epoch % 500 == 0:
            print("epoch: {}, loss is: {}".format(epoch, loss.item()))
```

- 我们需要根据不同的任务选择合适的**损失函数**和**优化器**

PyTorch中的损失函数

- torch.nn中同样提供了很多常用的损失函数：
  - https://pytorch.org/docs/stable/nn.html#loss-functions

Loss Functions

nn.L1Loss	
nn.MSELoss	
nn.CrossEntropyLoss	
nn.CTCLoss	
nn.NLLLoss	
nn.PoissonNLLLoss	
nn.GaussianNLLLoss	
nn.KLDivLoss	
nn.BCELoss	
nn.BCEWithLogitsLoss	
nn.MarginRankingLoss	
nn.HingeEmbeddingLoss	
nn.MultiLabelMarginLoss	
nn.SmoothL1Loss	
nn.SoftMarginLoss	Creates a criterion that optimizes a two-class classification logistic loss between input tensor x and target tensor y (containing 1 or -1).
nn.MultiLabelSoftMarginLoss	Creates a criterion that optimizes a multi-label one-versus-all loss based on max-entropy, between input x and target y of size (N, C).
nn.CosineEmbeddingLoss	Creates a criterion that measures the loss given input tensors x_1, x_2 and a *Tensor* label y with values 1 or -1.
nn.MultiMarginLoss	Creates a criterion that optimizes a multi-class classification hinge loss (margin-based loss) between input x (a 2D mini-batch *Tensor*) and output y (which is a 1D tensor of target class indices, $0 \leq y \leq \text{x.size}(1) - 1$):
nn.TripletMarginLoss	Creates a criterion that measures the triplet loss given an input tensors $x1$, $x2$, $x3$ and a margin with a value greater than 0.

PyTorch中的损失函数

- 损失函数$L(y,\hat{y})$：
  - 衡量模型预测$\hat{y}$和实际真实输出y之间的误差
  - 损失函数越小，模型预测越准确
  - 通过最小化训练集上的损失函数，来训练模型
- 如何选择损失函数?
  - 通常我们依据要完成的任务，选择合适的损失函数

PyTorch中的损失函数

- 常用损失函数：
  - **回归**模型：
    - 均方误差函数（Mean Squared Error， MSE）：$\frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$
      - [nn.MSELoss]()
    - 平均绝对值误差（Mean Absolute Error, MAE）：$\frac{1}{N}\sum_{i=1}^{N}|\hat{y}_i - y_i|$
      - [nn.L1Loss]()

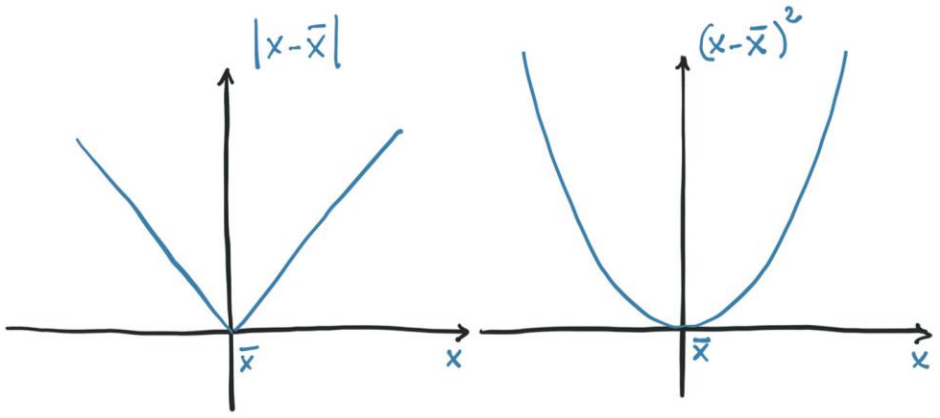


Figure 4.3 Absolute difference versus difference squared

PyTorch中的损失函数

- 常用损失函数：
  - **分类**模型：
    - 虽然可以强行通过回归方式解决，但是效果较差
    - 目前主流方法都是基于**概率相关方法**进行建模
      - 使得样本的分类概率达到最大

下雨	不下雨	下雨	不下雨	下雨	不下雨
0.1	0.9	0.3	0.7	0.4	0.6

使用概率的方法更符合真实情况

概率基础

- （离散）**概率分布**
  - 样本空间：一个事件所有发生的可能情况
    - 通常可以看做一个集合 $S = \{s_1, s_2, \dots, s_n\}$

  - 样本点概率： $P(\mathrm{s})$，满足 $P(\mathrm{s}) \geq 0$ 且 $P(\mathrm{s}) \leq 1$

  - 定义在一个样本空间S的概率分布，必须同时满足
    - **非负性**： $P(\mathrm{s}) \geq 0, \ \forall s \in S$
    - **归一性**： $\sum_{s \in S} P(\mathrm{s}) = 1$

概率基础

- 二分类概率分布
  - 样本空间：一个事件所有发生的可能情况
    - $S=\{发生，不发生\}$
    - $S=\{第0类，第1类\}$
  - 样本点概率： $P(\mathrm{s})$，满足 $P(\mathrm{s})\geq 0且P(\mathrm{s})\leq 1$
  - 定义在一个样本空间S的概率分布，满足
    - $P(\mathrm{s})\geq 0，\ \forall\, s\in S$
    - $P(第0类)+P(第1类)=1$
    - 可以用2个概率值（一个概率分布）表示对于一个事件发生的概率的估计
      - (0.1, 0.9)
      - (1, 0) 必定为第0类的概率
      - (0, 1) 必定为第1类的概率

PyTorch中的损失函数

- 问题:
  - 如何计算两个概率分布的“**相似程度**”？
  - 对于二分类:
    - y_i 是真实标签，取0或者1
      - 形成一个真实概率分布：$(1 - y_i, y_i)$
    - $\hat{y}_i$ 是预测值，是[0,1]之间的小数
      - 形成一个预测概率分布：$(1 - \hat{y}_i, \hat{y}_i)$

PyTorch中的损失函数

- 交叉熵:
  - $H(p,q) = -\sum_y p(y) \cdot \log q(y)$
  - 衡量真实分布$p(y)$和预测分布$q(y)$之间的差异
    - 如果$p(y) = q(y)$，那么交叉熵$H(p,q)$最小，且刚好等于$p(y)$的熵

1、枚举两个概率分布的值，对应点相乘
2、看起来很复杂，但是对于分类任务来说，只会有一项“**被激活**”（非0）

PyTorch中的损失函数

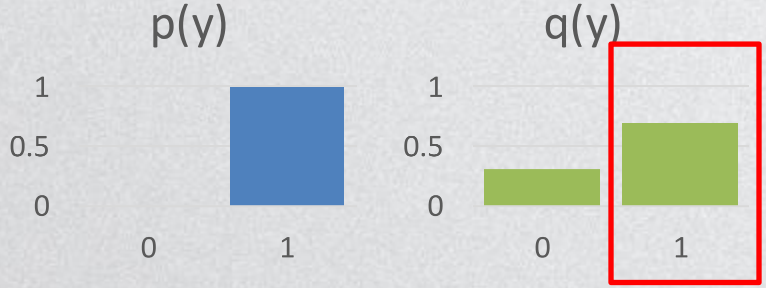
- 常用损失函数：
  - 二分类模型
    - 逻辑回归损失函数： $-\frac{1}{N}\sum_{i=1}^{N}[y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i)]$
      - nn.BCELoss
      - 二元交叉熵（Binary Cross Entropy ）

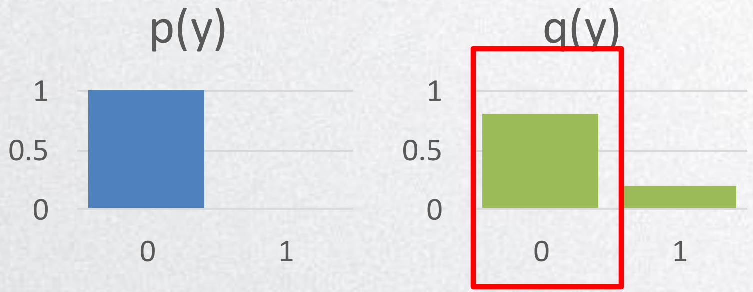
      - y_i 是真实标签，取0或者1
        - » 形成一个真实概率分布：$(1-y_i, y_i)$
      - $\hat{y}_i$ 是预测值，是[0,1]之间的小数
        - » 形成一个预测概率分布： $(1-\hat{y}_i, \hat{y}_i)$

PyTorch中的损失函数

- 二分类： $y \in \{0, 1\}$
  - $H(p,q) = -\sum_y p(y) \cdot \log q(y)$
    $= - \quad (y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i))$

    逻辑回归： $\hat{y}_i = \frac{1}{1+e^{-(xw+b)}}$
  - 在所有数据上取平均： $-\frac{1}{N}\sum_{i=1}^{N}[y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i)]$





举例：使用BCELoss计算二分类损失

- 10个数据
- 2个类别C={0,1}

```python
def testBCELoss(data_size):
    #target y
    y = torch.empty(data_size).random_(2) #ground-truth label
    print("[target y]", y)
    #creating y_pred
    m = nn.Sigmoid() #you can replace it with your trained function
    loss = nn.BCELoss()
    input = torch.randn(data_size, requires_grad=True)
    print("[input]", input)
    y_pred = m(input)
    print("[predicted y]", y_pred)
    #computing the loss
    output = loss(y_pred, y)
    print(output.item())
```

```
[target y] tensor([0., 1., 1., 0., 0., 0., 0., 1., 0., 0.])
[input] tensor([-1.5010, -0.5681,  1.9500,  1.4116,  0.4478,  0.1876,  0.8693, -0.5146,
        -0.3979,  0.1848], requires_grad=True)
[predicted y] tensor([0.1823, 0.3617, 0.8754, 0.8040, 0.6101, 0.5468, 0.7046, 0.3741, 0.4018,
        0.5461], grad_fn=<SigmoidBackward>)
0.822050929069519
```

如何训练神经网络

- 模型训练和优化的一般流程
  - 规定函数形式，明确如下
    - 参数（需要学习，一般用w表示）
    - 输入（始终给定，一般用x表示）
    - 输出（训练时给定，一般用y表示）
    - 预测（由模型的出来，一般用$\hat{y}$表示）
  - 构建关于输出y与预测$\hat{y}$之间的**损失函数** $L(y,\hat{y})$
  - 以参数为待求目标，进行反向求导 $\frac{\partial L}{\partial w}$ ← loss.backward()
  - 使用**梯度下降（或者其变种）**进行参数的更新:

$$w^{new} = w^{old} - lr * \frac{\partial L}{\partial w}$$

  - 迭代多轮，直至收敛

如何训练神经网络

- 模型训练和优化的一般流程
  - 规定函数形式，明确如下
    - 参数（需要学习，一般用w表示）
    - 输入（始终给定，一般用x表示）
    - 输出（训练时给定，一般用y表示）
    - 预测（由模型的出来，一般用$\hat{y}$表示）
  - 构建关于输出y与预测$\hat{y}$之间的**损失函数** $L(y,\hat{y})$
  - 以参数为待求目标，进行反向求导 $\frac{dL}{dw}$
  - 使用**梯度下降（或者其变种）**进行参数的更新:

$$w^{new} = w^{old} - lr * \frac{dL}{dw}$$

  - 迭代多轮，直至收敛

PyTorch中的优化器

- torch.optim中提供了多种优化器：
  - 在执行optimizer.step()时，优化器会根据**损失函数输出**反向传播计算的梯度，更新**模型**的参数

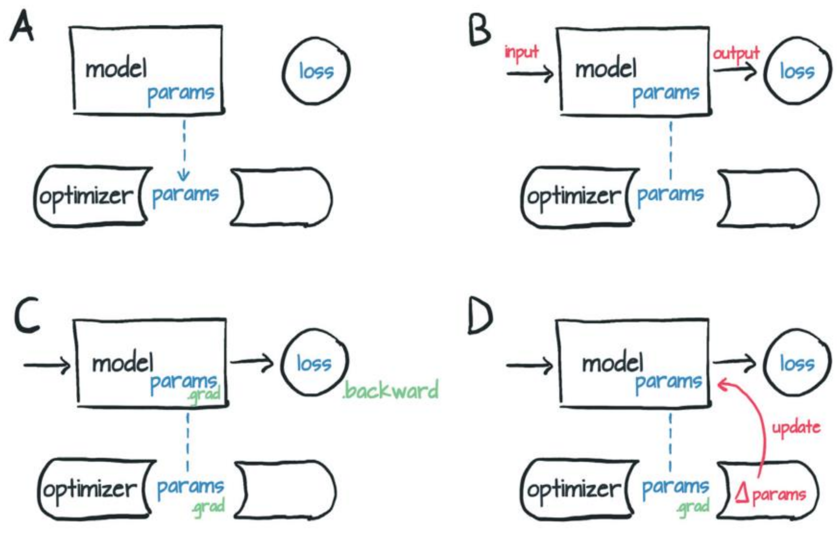


Figure 4.10 Conceptual representation of how an optimizer holds a reference to parameters (A), and after a loss is computed from inputs (B), a call to .backward leads to .grad being populated on parameter (C). At that point, the optimizer can access .grad and compute the parameter updates (D).

PyTorch中的优化器

- torch.optim中提供了多种优化器：
  - 不同的优化器在执行.step()时会采用不同的方式更新模型参数

```python
import torch.optim as optim
dir(optim)
```

```
['ASGD',
 'Adadelta',
 'Adagrad',
 'Adam',
 'AdamW',
 'Adamax',
 'LBFGS',
 'Optimizer',
 'RMSprop',
 'Rprop',
 'SGD',
 'SparseAdam',
```

PyTorch中的优化器

- 常用的优化器：
  - optim.SGD
    - 随机梯度下降
    - 每次更新的学习率是固定的

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^{t} - lr\left(\left.\frac{\partial L(\boldsymbol{w})}{\partial \boldsymbol{w}}\right|_{\boldsymbol{w}^{t}}\right)^{T}$$

  - optim.Adam
    - g_t为反向传播计算得到的梯度
    - 会在学习过程中自动的改变学习率
    - 参数不同维度间的大小差异对训练影响较小

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

使用SGD和Adam进行优化

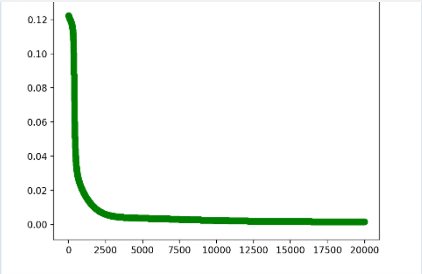
```python
        # 模型超参数
        self.learning_rate = lr
        self.epoches = epoches
        self.hidden_dim = hidden_dim

        # 模型
        self.model = MLPRegression(input_dim, self.hidden_dim)
        # 优化器
        self.optimizer = torch.optim.SGD(self.model.parameters(), lr=self.learning_rate)
        # 损失函数
        self.loss_function = torch.nn.MSELoss()

    def train(self, x, y):
        """..."""
        losses = []
        for epoch in range(self.epoches):
            pred = self.model(x)
            loss = self.loss_function(pred, y)

            self.optimizer.zero_grad()
            loss.backward()
            self.optimizer.step()

            losses.append(loss.item())
```

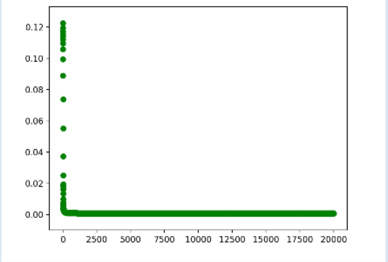



```python
        # 模型超参数
        self.learning_rate = lr
        self.epoches = epoches
        self.hidden_dim = hidden_dim

        # 模型
        self.model = MLPRegression(input_dim, self.hidden_dim)
        # 优化器
        self.optimizer = torch.optim.Adam(self.model.parameters(), lr=self.learning_rate)
        # 损失函数
        self.loss_function = torch.nn.MSELoss()

    def train(self, x, y):
        """..."""
        losses = []
        for epoch in range(self.epoches):
            pred = self.model(x)
            loss = self.loss_function(pred, y)

            self.optimizer.zero_grad()
            loss.backward()
            self.optimizer.step()
```



如何使用PyTorch训练神经网络

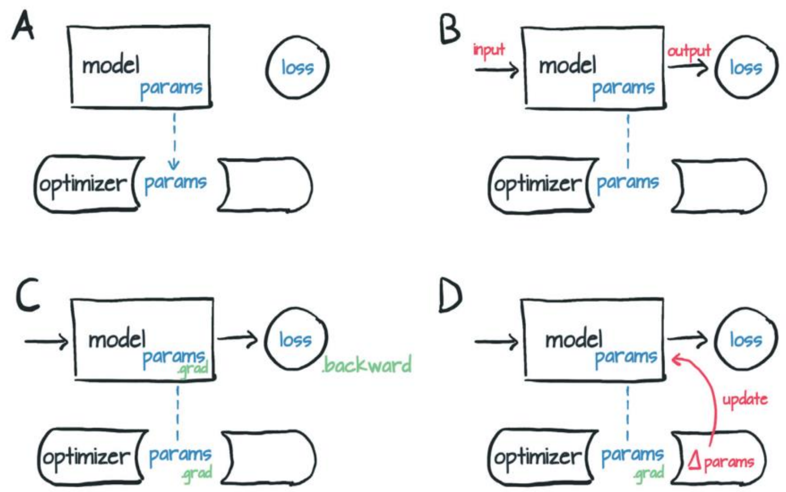


Figure 4.10 Conceptual representation of how an optimizer holds a reference to parameters (A), and after a loss is computed from inputs (B), a call to .backward leads to .grad being populated on parameter (C). At that point, the optimizer can access .grad and compute the parameter updates (D).

```python
def train(self, x, y):
    """
    训练模型并保存参数
    输入:
        model_save_path: saved name of model
        x: 训练数据
        y: 回归真值
    返回:
        losses: 所有迭代中损失函数值
    """
    losses = []
    for epoch in range(self.epoches):
        prediction = self.model(x)
        loss = self.loss_function(prediction, y)

        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        losses.append(loss.item())

        if epoch % 500 == 0:
            print("epoch: {}, loss is: {}".format(epoch, loss.item()))
```

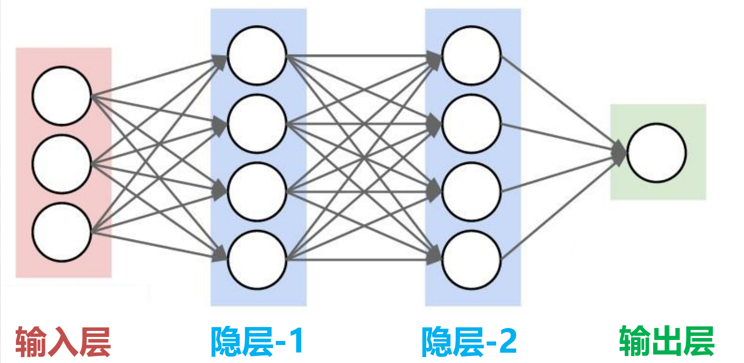
PyTorch神经网络

提纲

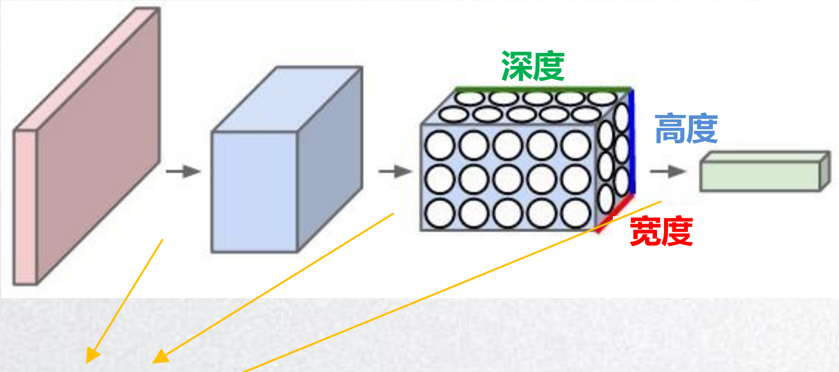
- 神经网络概述
- 使用PyTorch搭建神经网络
- 使用PyTorch训练神经网络
- **卷积神经网络**与循环神经网络

卷积神经网络

输入图像：深度为3（RGB通道），
高度和宽度为图像形状大小



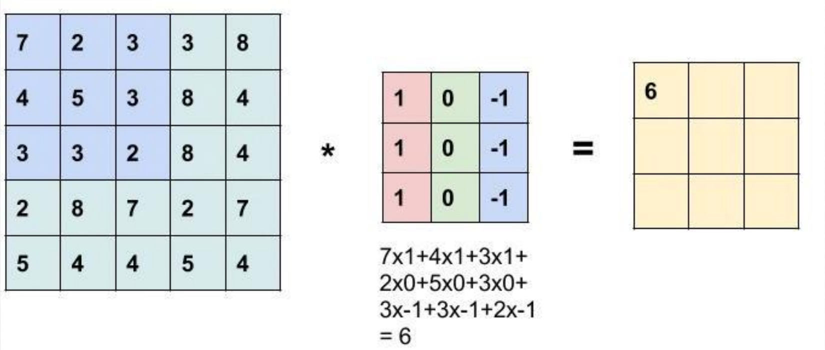
全连接网络



卷积操作

卷积网络

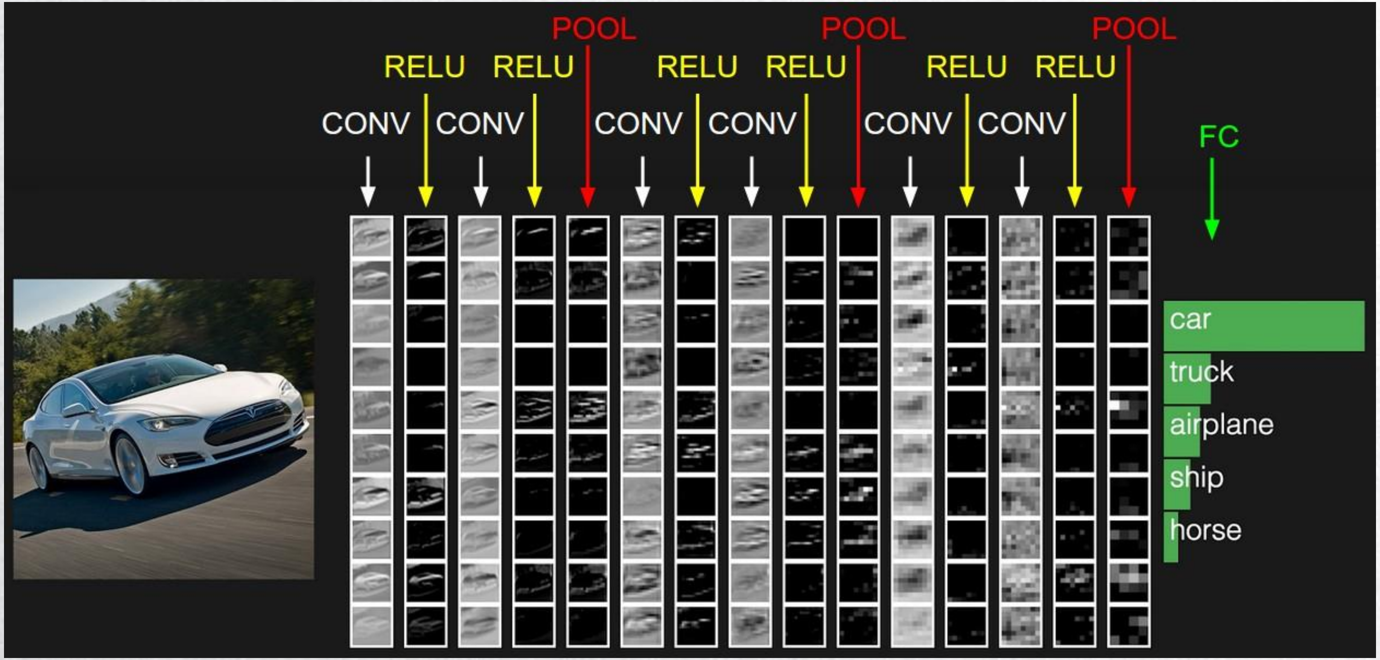
卷积神经网络



$$
\begin{aligned}
g(x, y) = {} & w(-1, -1)f(x - 1, y - 1) + w(-1, 0)f(x - 1, y) + \ldots \\
& + w(0, 0)f(x, y) + \ldots + w(1, 1)f(x + 1, y + 1)
\end{aligned}
$$

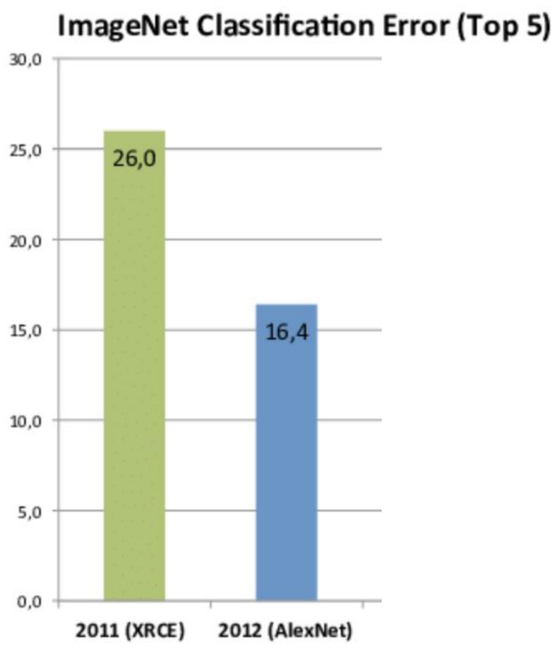
卷积（本质是相关）运算

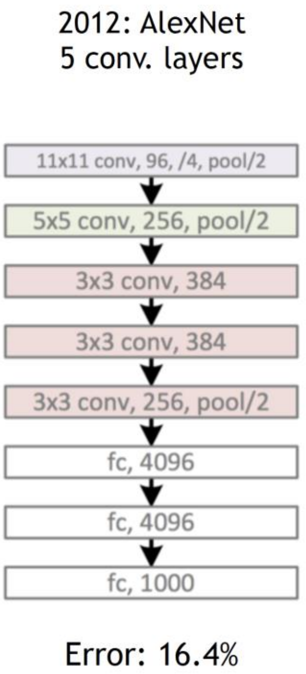
卷积神经网络



POOL
POOL
POOL
RELU
RELU
RELU
RELU
RELU
RELU
CONV
CONV
CONV
CONV
CONV
CONV
FC
car
truck
airplane
ship
horse

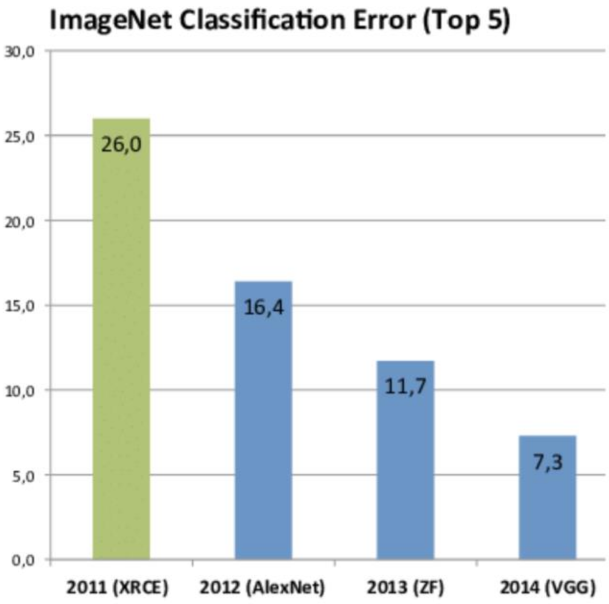
卷积神经网络



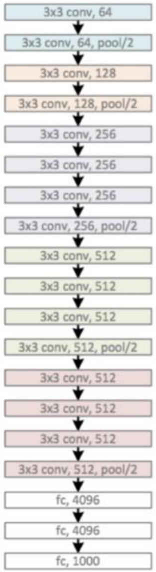


[Krizhevsky et al: ImageNet Classification with Deep Convolutional Neural Networks, NIPS 2012]

卷积神经网络



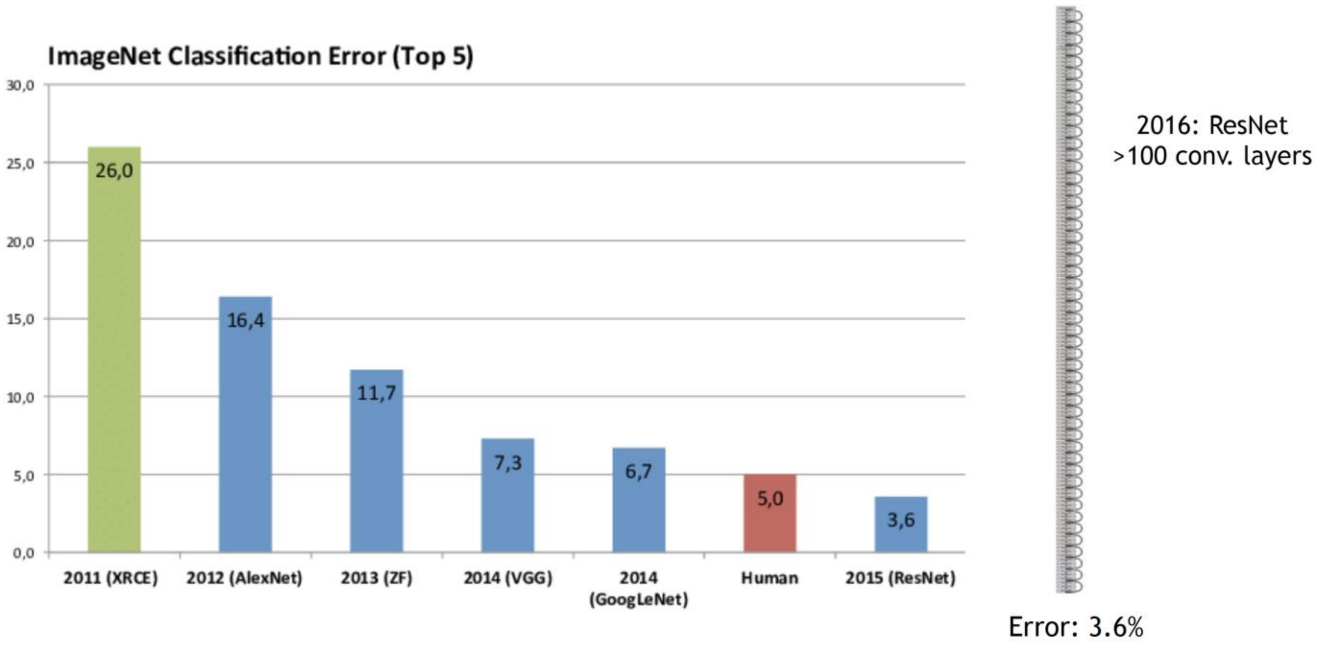
2014: VGG
16 conv. layers



Error: 7.3%

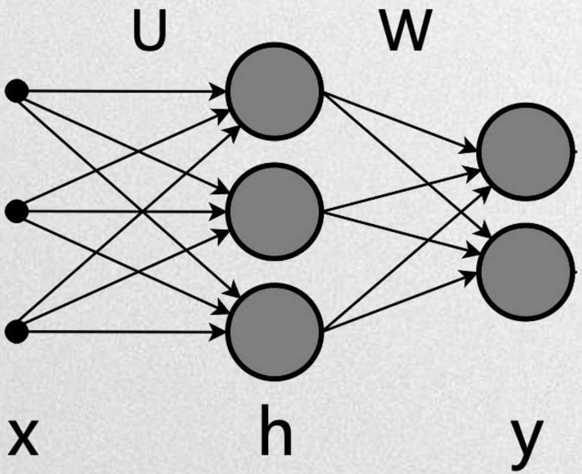
[Simonyan & Zisserman: Very Deep Convolutional Networks

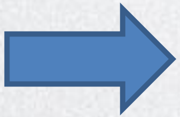
卷积神经网络

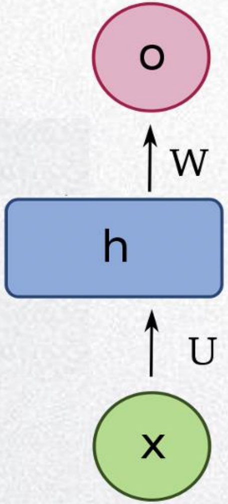


[He et al: Deep Residual Learning for Image Recognition, CVPR 2016]

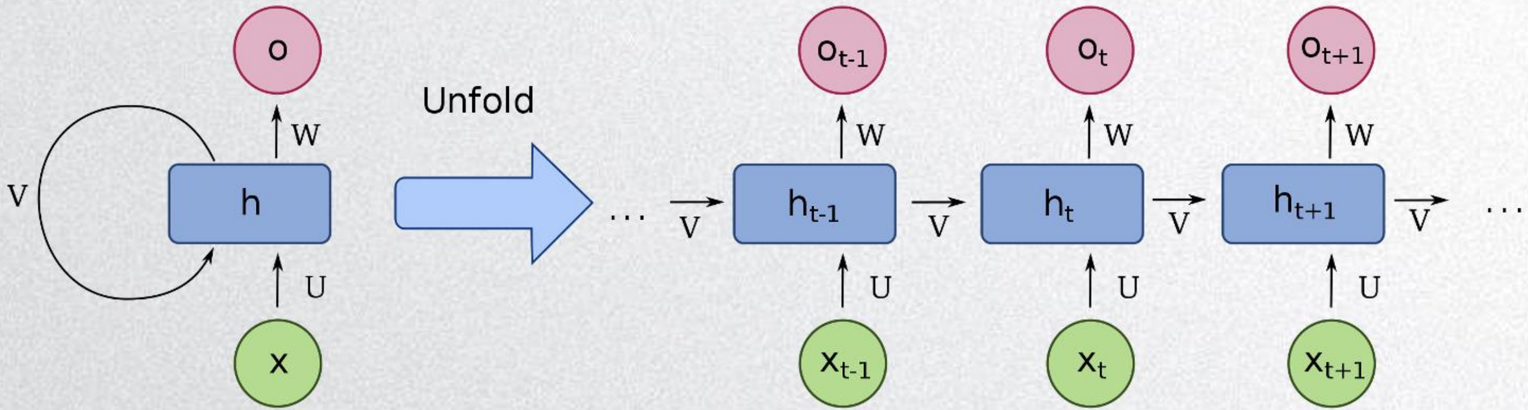
从MLP到循环神经网络





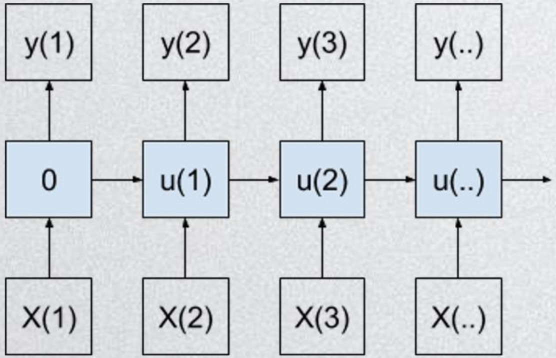


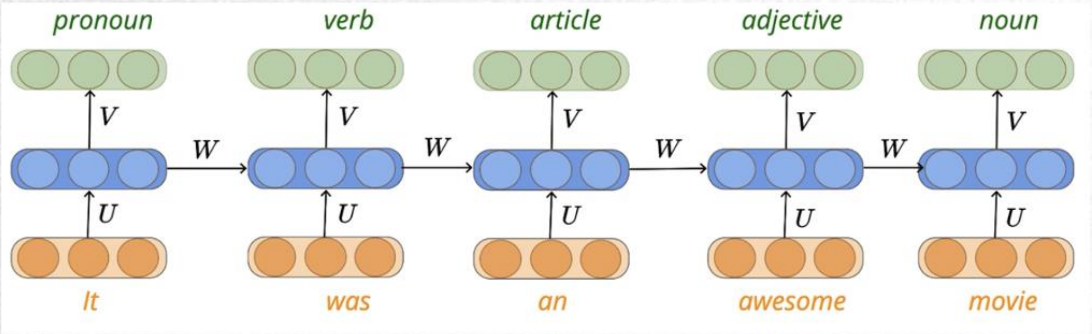
循环神经网络



循环神经网络

- One-to-one

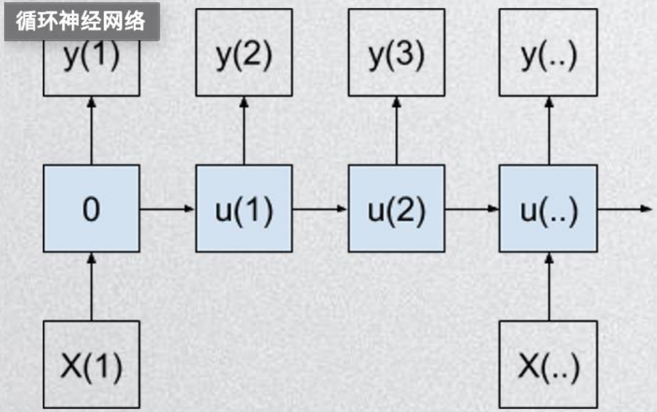


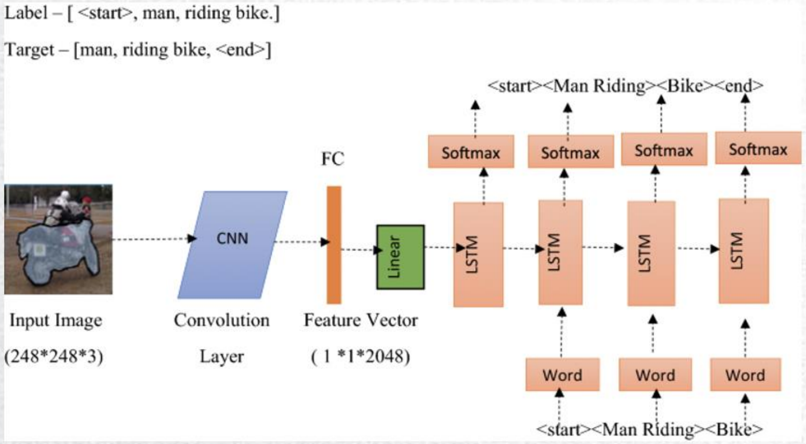


词性标注示例

循环神经网络

- One-to-many

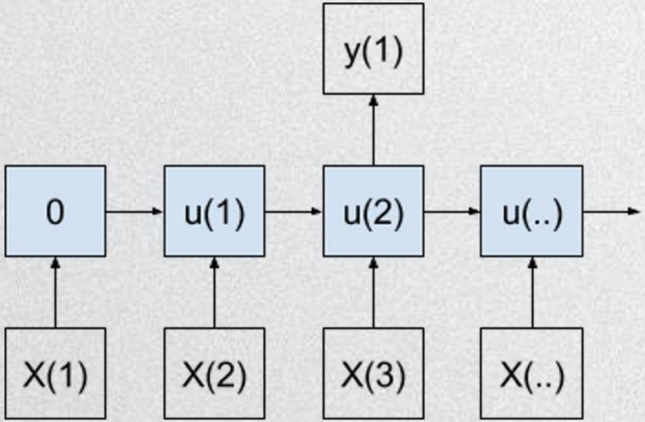


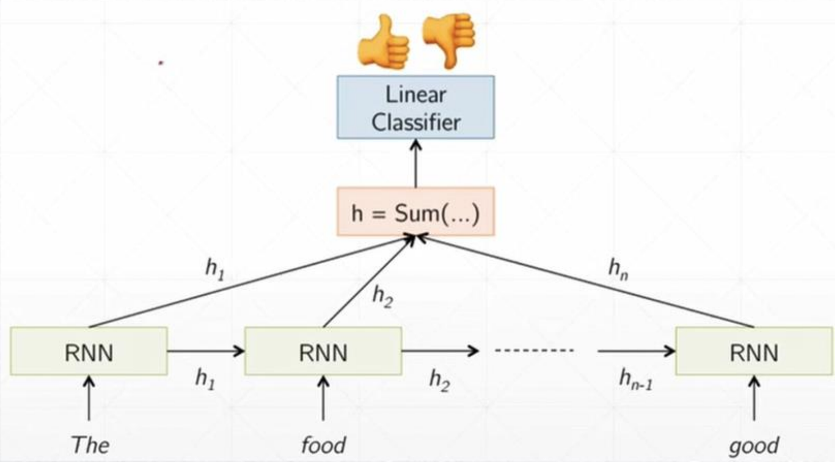


图像内容描述

循环神经网络

- Many-to-one

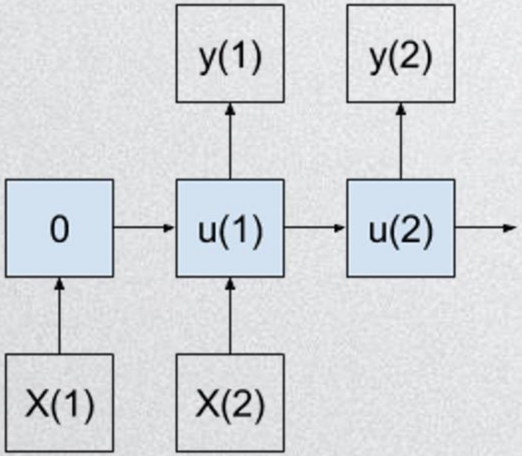


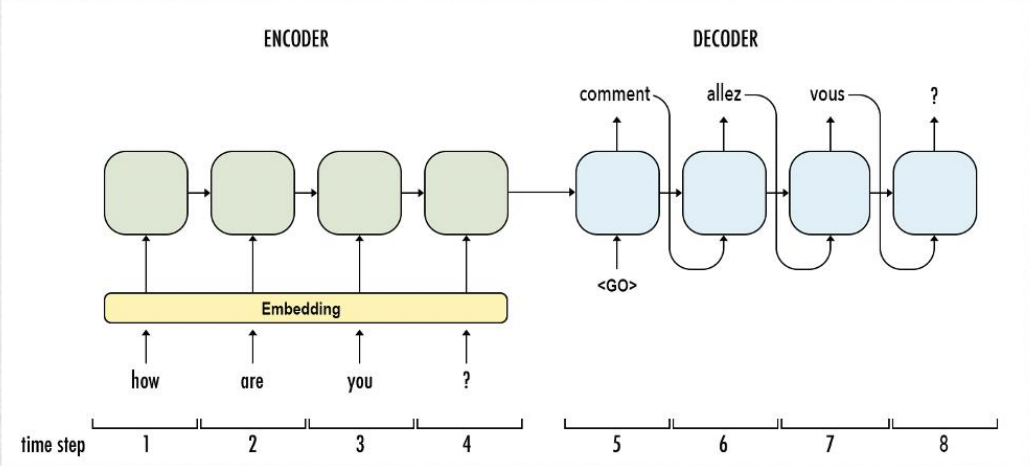


情感分类

循环神经网络

- Many-to-many





机器翻译

谢谢！