cs231n(2024’s)

DL 笔记_CS231n（他人的总结笔记）

what get from the cs231n assignments and its codes(2024’s):

assignment 1

CIFAR10数据集为10分类图集

Q1:knn.ipynb

np.linalg.norm(x, ord=None, axis=None, keepdims=False)

1. 矩阵化的范数计算的体验： 不用：

```
def compute_distances_no_loops(self, X):
    """
    Compute the distance between each test point in X and each training poi
nt
    in self.X_train using no explicit loops.

    Input / Output: Same as compute_distances_two_loops
    """
    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)***
**
    # Note: (a - b)^2 = -2ab + a^2 + b^2
    dists = np.sqrt(
      -2 * (X @ self.X_train.T) +
      np.power(X, 2).sum(axis=1, keepdims=True) +
      np.power(self.X_train, 2).sum(axis=1, keepdims=True).T
    )

    # dists = np.sqrt(np.sum(np.power(np.expand_dims(X, axis=1) - self.X_tr
ain, 2), axis=2))
```

```
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    return dists
```

2. **k 折交叉验证（k-fold cross-validation）**

“尽可能既用于训练又用于验证” 的策略：

1. 先把数据 **平均切成 k 份**（折）；
2. 每次 **用其中 1 份做验证集，其余 k-1 份做训练集**；
3. 这样 **轮流 k 次**，再把 k 次得到的评估结果取平均，作为模型最终性能估计。

Q2:svm.ipynb

1. 这里做中心化和bias trick

`# only has to worry about optimizing a single weight matrix W.`

`eg.X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))])`

2. SVM可形式化成一个求解凸二次规划(convex quadratic programming)问题，也等价于正则化的合页损失函数的最小化问题。

$$L_i = \sum_{j \neq y_i} \max(0, w_j^T x_i - w_{y_i}^T x_i + \Delta)$$

$$L = \underbrace{\frac{1}{N}\sum_i L_i}_{\text{data loss}} + \underbrace{\lambda R(W)}_{\text{regularization loss}}$$

$$L = \frac{1}{N}\sum_i \sum_{j \neq y_i} \left[\max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + \Delta)\right] + \lambda \sum_k \sum_l W_{k,l}^2$$

```
def svm_loss_naive(W, X, y, reg):
    """
    Structured SVM loss function, naive implementation (with loops).
    Inputs have dimension D, there are C classes, and we operate on minibat
ches
    of N examples.
    Inputs:
    - W: A numpy array of shape (D, C) containing weights.
    - X: A numpy array of shape (N, D) containing a minibatch of data.
    - y: A numpy array of shape (N,) containing training labels; y[i] = c mean
s
      that X[i] has label c, where 0 <= c < C.
    - reg: (float) regularization strength
    Returns a tuple of:
    - loss as single float
    - gradient with respect to weights W; an array of same shape as W
    """
    dW = np.zeros(W.shape)  # initialize the gradient as zero
    # compute the loss and the gradient
    num_classes = W.shape[1]
    num_train = X.shape[0]
    loss = 0.0
    for i in range(num_train):
        scores = X[i].dot(W)
        correct_class_score = scores[y[i]]
        for j in range(num_classes):
            if j == y[i]:
                continue
            margin = scores[j] - correct_class_score + 1  # note delta = 1
            if margin > 0:
                loss += margin
                dW[:, j] += X[i]    # update gradient for incorrect label
                dW[:, y[i]] -= X[i] # update gradient for correct label
    # Right now the loss is a sum over all training examples, but we want it
    # to be an average instead so we divide by num_train.
    loss /= num_train
    # Add regularization to the loss.
    loss += reg * np.sum(W * W)

    dW /= num_train   # scale gradient ovr the number of samples
    dW += 2 * reg * W # append partial derivative of regularization term
    return loss, dW
```

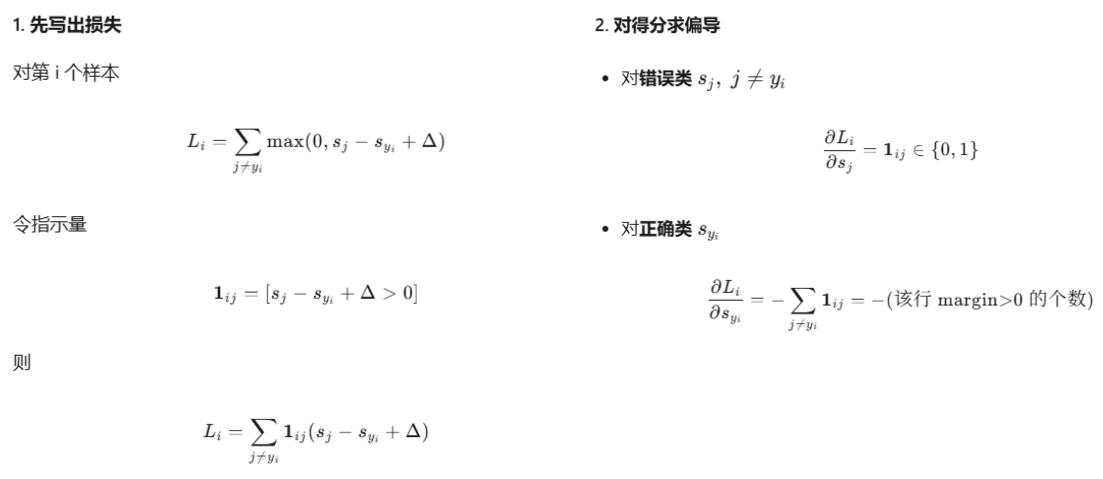
```
def svm_loss_vectorized(W, X, y, reg):
    """
        Structured SVM loss function, vectorized implementation.Inputs and o
utputs are the same as svm_loss_naive.
```

```
    """
    loss = 0.0
    dW = np.zeros(W.shape)  # initialize the gradient as zero

    N = len(y)     # number of samples
    Y_hat = X @ W  # raw scores matrix

    y_hat_true = Y_hat[range(N), y][:, np.newaxis]    # scores for true labels
    margins = np.maximum(0, Y_hat - y_hat_true + 1)   # margin for each sco
re
    loss = margins.sum() / N - 1 + reg * np.sum(W**2) # regularized loss for
batch

    dW = (margins > 0).astype(int)    # initial gradient with respect to Y_hat
    dW[range(N), y] -= dW.sum(axis=1) # update gradient to include correct
labels
    dW = X.T @ dW / N + 2 * reg * W   # gradient with respect to W

return loss, dW
```

Y_hat[range(N), y]这是高级索引（fancy indexing）的“同时索引两个维度”写法：
第一个维度给的是行索引 range(N)；第二个维度给的是列索引 y。
结果是一个一维数组(返回的是形状 (N,) 的视图，但是-= 实现原地操作)，长度 N，里面依次是Y_hat[0, y[0]], Y_hat[1, y[1]], … , Y_hat[N-1, y[N-1]]也就是每个样本在其真实类别上的得分。

[:, np.newaxis]（或 [:, None]）把刚才得到的一维数组** reshape 成列向量** (N, 1)。np.newaxis 是 None 的别名，用来增加一个长度为 1 的轴。

反向传播时，先对s求偏导，再链式对w偏导：


1. 先写出损失

对第 i 个样本

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta)$$

令指示量

$$\mathbf{1}_{ij} = [s_j - s_{y_i} + \Delta > 0]$$

则

$$L_i = \sum_{j \neq y_i} \mathbf{1}_{ij}(s_j - s_{y_i} + \Delta)$$

2. 对得分求偏导

- 对**错误类** $s_j,\ j \neq y_i$

$$\frac{\partial L_i}{\partial s_j} = \mathbf{1}_{ij} \in \{0, 1\}$$

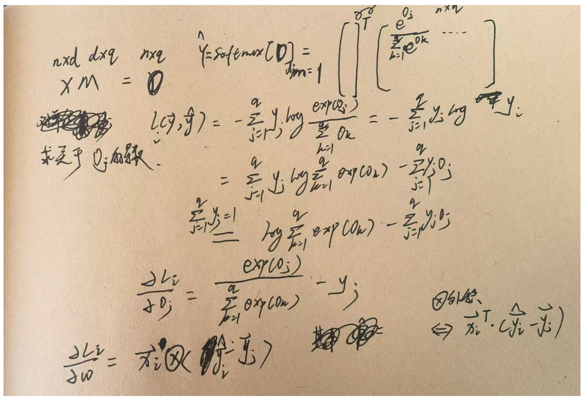
- 对**正确类** s_{y_i}

$$\frac{\partial L_i}{\partial s_{y_i}} = -\sum_{j \neq y_i} \mathbf{1}_{ij} = -(\text{该行 margin>0 的个数})$$

Q3softmax.ipynb

1. loss 与 梯度更新

```
def softmax_loss_naive(W, X, y, reg):
    """
    Softmax loss function, naive implementation (with loops)
    Inputs have dimension D, there are C classes, and we operate on minibat
ches
    of N examples.
    Inputs:
    - W: A numpy array of shape (D, C) containing weights.
    - X: A numpy array of shape (N, D) containing a minibatch of data.
    - y: A numpy array of shape (N,) containing training labels; y[i] = c mean
s
      that X[i] has label c, where 0 <= c < C.
    - reg: (float) regularization strength
    Returns a tuple of:
    - loss as single float
    - gradient with respect to weights W; an array of same shape as W
    """
    # Initialize the loss and gradient to zero.
    loss = 0.0
    dW = np.zeros_like(W)
    N = X.shape[0] # num samples
    for i in range(N):
        y_hat = X[i] @ W                    # raw scores vector
        y_exp = np.exp(y_hat - y_hat.max()) # numerically stable exponent vec
tor
        softmax = y_exp / y_exp.sum()       # pure softmax for each score
        loss -= np.log(softmax[y[i]])       # append cross-entropy
        softmax[y[i]] -= 1                  # update for gradient
        dW += np.outer(X[i], softmax)       # gradient

    loss = loss / N + reg * np.sum(W**2)    # average loss and regularize
    dW = dW / N + 2 * reg * W               # finish calculating gradient
    return loss, dW
```



$n\times d \quad d\times q \quad n\times q$

$XW = O$

$\hat{Y} = \mathrm{softmax}(O)_{dim=1} = \left[\ \right]^T \begin{bmatrix} \frac{e^{O_j}}{\sum_{k=1}^{q} e^{O_k}} & \cdots \end{bmatrix}_{n\times q}$

$L(y,\hat{y}) = -\sum_{j=1}^{q} y_j \log \frac{\exp(O_j)}{\sum_{k=1}^{q} O_k} = -\sum_{j=1}^{q} y_j \log \hat{y}_j$

求关于 O_j 的导数.

$= \sum_{j=1}^{q} y_j \log \sum_{k=1}^{q} \exp(O_k) - \sum_{j=1}^{q} y_j O_j$

$\overset{\sum_{j=1}^{q} y_j = 1}{=} \log \sum_{k=1}^{q} \exp(O_k) - \sum_{j=1}^{q} y_j O_j$

$\frac{\partial L_i}{\partial O_j} = \frac{\exp(O_j)}{\sum_{k=1}^{q} \exp(O_k)} - y_j$

$\frac{\partial L_i}{\partial W} = \vec{x}_i \otimes (\hat{y}_i - \vec{y}_i)$

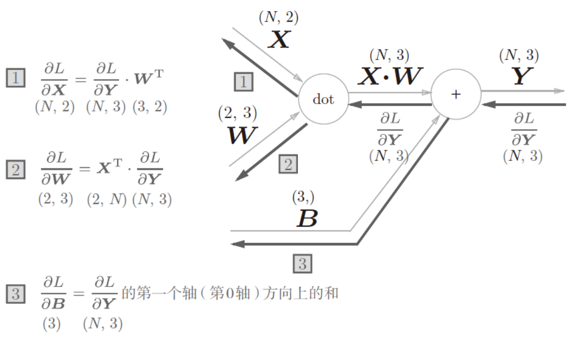
$\otimes$ 外积、

$\Leftrightarrow \vec{x}_i^{\,T} \cdot (\hat{\vec{y}}_i - \vec{y}_i)$

```
def softmax_loss_vectorized(W, X, y, reg):
    """
    Softmax loss function, vectorized version.

    Inputs and outputs are the same as softmax_loss_naive.
    """
    # Initialize the loss and gradient to zero.
    loss = 0.0
    dW = np.zeros_like(W)
    N = X.shape[0] # number of samples
    Y_hat = X @ W  # raw scores matrix
    P = np.exp(Y_hat - Y_hat.max())      # numerically stable exponents
    P /= P.sum(axis=1, keepdims=True)    # row-wise probabilities (softmax)

    loss = -np.log(P[range(N), y]).sum() # sum cross entropies as loss #P[ra
nge(N), y]:
    #一次性取出 N 个样本在各自正确类别上的概率值，返回的是一个长度 N 的一维数组，P不变
    loss = loss / N + reg * np.sum(W**2) # average loss and regularize

    P[range(N), y] -= 1                # update P for gradient
    dW = X.T @ P / N + 2 * reg * W      # calculate gradient
    return loss, dW
```

Q4 two_layer_net.ipynb

1. 实现各层的forward与backward
2. affine layer:



$\boxed{1}$ $\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} \cdot W^T$ (N, 2) (N, 3) (3, 2)
$\boxed{2}$ $\frac{\partial L}{\partial W} = X^T \cdot \frac{\partial L}{\partial Y}$ (2, 3) (2, N) (N, 3)
$\boxed{3}$ $\frac{\partial L}{\partial B} = \frac{\partial L}{\partial Y}$ 的第一个轴（第0轴）方向上的和 (3) (N, 3)
(N, 2) X
(2, 3) W
(3,) B
dot
(N, 3) X·W
+
(N, 3) Y
$\frac{\partial L}{\partial Y}$ (N, 3)
$\frac{\partial L}{\partial Y}$ (N, 3)

```
def affine_forward(x, w, b):
    """
    Computes the forward pass for an affine (fully-connected) layer.
    The input x has shape (N, d_1, ..., d_k) and contains a minibatch of N
    examples, where each example x[i] has shape (d_1, ..., d_k). We will
    reshape each input into a vector of dimension D = d_1 * ... * d_k, and
    then transform it to an output vector of dimension M.
    Inputs:
    - x: A numpy array containing input data, of shape (N, d_1, ..., d_k)
    - w: A numpy array of weights, of shape (D, M)
    - b: A numpy array of biases, of shape (M,)
    Returns a tuple of:
    - out: output, of shape (N, M)
    - cache: (x, w, b)
    """
    out = None
    #########################################################
######################
    x_reshaped = x.reshape(x.shape[0], -1)
    out = x_reshaped @ w + b                     #
    #########################################################
######################
    cache = (x, w, b)
    return out, cache

def affine_backward(dout, cache):
    """
    Computes the backward pass for an affine layer.

    Inputs:
    - dout: Upstream derivative, of shape (N, M)
    - cache: Tuple of:
      - x: Input data, of shape (N, d_1, ... d_k)
      - w: Weights, of shape (D, M)
      - b: Biases, of shape (M,)

    Returns a tuple of:
    - dx: Gradient with respect to x, of shape (N, d1, ..., d_k)
    - dw: Gradient with respect to w, of shape (D, M)
    - db: Gradient with respect to b, of shape (M,)
    """
    x, w, b = cache
    dx, dw, db = None, None, None
    #########################################################
######################
    x_reshaped = x.reshape(x.shape[0], -1)
    dx = (dout @ w.T).reshape(x.shape[0], *x.shape[1:])
    dw = x_reshaped.T @ dout
    db = dout.sum(axis=0)                        #
    #########################################################
######################
    return dx, dw, db
```

3. relu

```
#forward   for any shape
out = np.maximum(0, x)
#backward
dx = dout * (x > 0)
#可将层封装:
def affine_relu_forward(x, w, b):
    a, fc_cache = affine_forward(x, w, b)
    out, relu_cache = relu_forward(a)
    cache = (fc_cache, relu_cache)
    return out, cac
def affine_relu_backward(dout, cache):
    fc_cache, relu_cache = cache
    da = relu_backward(dout, relu_cache)
    dx, dw, db = affine_backward(da, fc_cache)
    return dx, dw, db
```

4. 两层网络:

cs231n/classifiers/fc_net.py 中 `TwoLayerNet` 类的实现，可视为model实例，model.loss 返回 （test:scores train:loss,grads）

cs231n/solver.py 中Solver类实现.与 `cs231n/optim.py` 中实现 `sgd`

eg.

```
    data = {
      'X_train': # training data
      'y_train': # training labels
      'X_val': # validation data
      'y_val': # validation labels
    }
    model = MyAwesomeModel(hidden_size=100, reg=10)
    solver = Solver(model, data,
                    update_rule='sgd',
                    optim_config={
                      'learning_rate': 1e-3,
                    },
                    lr_decay=0.95,
                    num_epochs=10, batch_size=100,
                    print_every=100)
    solver.train()
```

Q5features.ipynb:

这里特此提取;

- **HOG** – Histogram of Oriented Gradients
- **HSV** – Hue, Saturation, Value

该实现，特征提取最高增10个点

assignment 2

Q1FullyConnectedNets.ipynb

1. `cs231n/classifiers/fc_net.py` 中的 `FullyConnectedNet` 类
   这里实现了可选任意隐藏层的Net.
2. optimizer

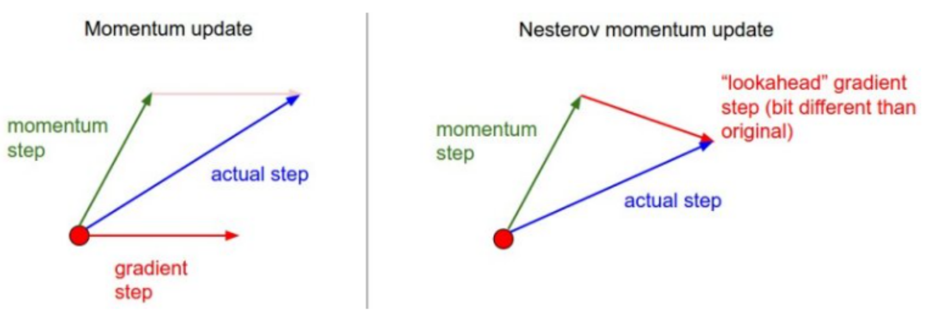
SGD:

```
w -= config["learning_rate"] * dw
```

sgd_momentum:

```
v = config['momentum'] * v - config['learning_rate'] * dw # update velocity
next_w = w + v            # update position
```

Nesterov Momentum:



```
x_ahead = x + mu * v
# evaluate dx_ahead (the gradient at x_ahead instead of at x)
v = mu * v - learning_rate * dx_ahead
x += v
```

然而，在实际应用中，人们倾向于使更新表达式尽可能接近原始 SGD 或先前的动量更新。这可以通过对上述更新进行变量变换 `x_ahead = x + mu * v` 来实现，然后以 `x_ahead` 代替 `x` 来表示更新。也就是说，我们实际存储的参数向量始终是前一个版本。用 `x_ahead` 表示的方程（但将其改回 `x`）则变为：

```
v_prev = v # back this up
v = mu * v - learning_rate * dx # velocity update stays the same
x += -mu * v_prev + (1 + mu) * v # position update changes form
```

Adagrad:

```
# Assume the gradient dx and parameter vector x
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

RMSprop :

Here, `decay_rate` is a hyperparameter and typical values are [0.9, 0.99, 0.999].

```
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

```
keys = ['learning_rate','decay_rate','epsilon','cache'] # keys in this order
lr, dr, eps, cache = (config.get(key) for key in keys)   # vals in this order

config['cache'] = dr * cache + (1 - dr) * dw**2          # update cache
next_w = w - lr * dw / (np.sqrt(config['cache']) + eps) # update w
```

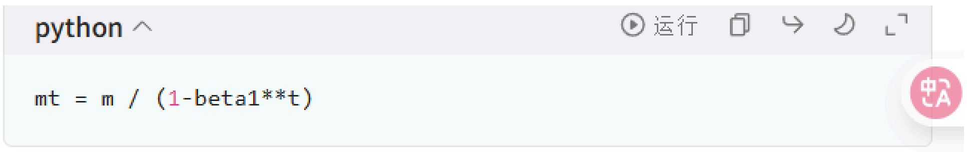
对比维度	Adagrad	RMSprop
平方梯度处理方式	累积求和（$G_t = G_{t-1} + (dw_t)^2$）	指数移动平均（ $E[(dw)^2]_t = \gamma E[(dw)^2]_{t-1} + (1-\gamma)(dw_t)^2$）
学习率变化趋势	持续衰减，后期趋近于零	稳定在近期梯度对应的水平，不轻易停滞
训练后期表现	参数更新停滞，无法收敛	保持有效更新，持续优化
适用场景	数据简单、训练轮次少的浅层网络	数据复杂、训练轮次多的深层网络

Adam(结合RMSprop+Momentum):

```
m = beta1*m + (1-beta1)*dx
v = beta2*v + (1-beta2)*(dx**2)
x += - learning_rate * m / (np.sqrt(v) + eps)
```

```
# t is your iteration counter going from 1 to infinity
m = beta1*m + (1-beta1)*dx
mt = m / (1-beta1**t)
v = beta2*v + (1-beta2)*(dx**2)
vt = v / (1-beta2**t)
x += - learning_rate * mt / (np.sqrt(vt) + eps)
```

实际用的时候，会有个：**偏差修正**



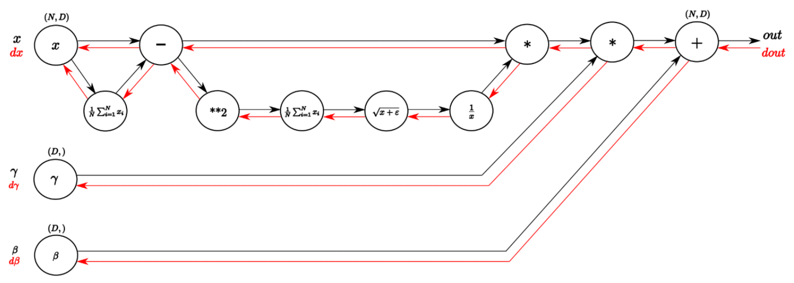
```python
mt = m / (1-beta1**t)
```

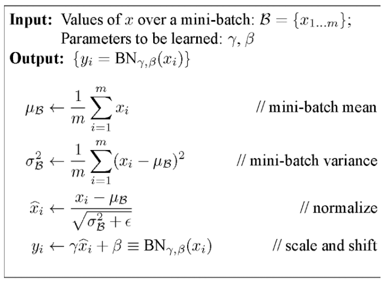
- **作用**：对一阶矩进行**偏差修正。**
- **原理**：由于初始时 `m=0`，前几步的 `m` 会偏小（偏向 0）。修正公式为 $\hat{m}_t = \frac{m_t}{1-\beta_1^t}$，随着 `t` 增大，`1 - beta1**t` 趋近于 1，修正作用逐渐消失。
- **必要性**：保证训练初期（`t` 较小时）的更新量不会因初始值为 0 而过小。

```
keys = ['learning_rate','beta1','beta2','epsilon','m','v','t'] # keys in this order
lr, beta1, beta2, eps, m, v, t = (config.get(k) for k in keys) # vals in this order

config['t'] = t = t + 1                                    # iteration counter
config['m'] = m = beta1 * m + (1 - beta1) * dw             # gradient smoothing (Momentum)
mt = m / (1 - beta1**t)                                    # bias correction
config['v'] = v = beta2 * v + (1 - beta2) * (dw**2)  # gradient smoothing (RMSprop)
vt = v / (1 - beta2**t)                                    # bias correction
next_w = w - lr * mt / (np.sqrt(vt) + eps)                 # weight update
```

Q2BatchNormalization.ipynb

1502.03167(论文)





Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$ // mini-batch mean

$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2$ // mini-batch variance

$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$ // normalize

$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i)$ // scale and shift

https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html（网页）

(99+ 封私信 / 80 条消息) 笔记: Batch Normalization及其反向传播 - 知乎（更多微分运算）

目的解决内部**协变量偏移**。

关于偏移：

分布偏移就是训练数据和测试数据的分布不一致，类似于训练的猫和狗，测试的牛和马

协变量偏移⁺：指的是训练数据和测试数据的输入特征分布不一致，类似于训练的猫和狗，测试的汤姆杰瑞等动画人物

标签偏移⁺：指的是训练数据和测试数据的输出分布不一致，类似于对于缅因猫，训练集标注的是缅因猫，但测试集标注的是狮子

关于协变量偏移：

对于理论情况，我们知道联合分布情况下，经验risk:

$$\mathbb{E}_{p(\mathbf{x},y)}[l(f(\mathbf{x}),y)] = \int\int l(f(\mathbf{x}),y)p(\mathbf{x},y)\,d\mathbf{x}\,dy$$

但其实，往往不知道，知道就直接可以求边缘分布，在进而求条件概率分布了（分类要的分布模型）

协变量偏移，是指训练数据和测试数据之间，**特征变量（协变量）**x的边际分布发生了变化，但**条件分布** P(y|x)保持不变的情况。换句话说，训练数据和测试数据中的特征分布 P(x)可能不同，但在给定特征x的情况下，标签y的分布不变：
$P_{\text{train}}(x) \neq P_{\text{test}}(x)$; $P_{\text{train}}(y|x) = P_{\text{test}}(y|x)$

以上即，分类规则不变，但是输入特征变了。

解决：改写如下，p真实分布，q所用分布

$$\int\int l(f(\mathbf{x}),y)p(\mathbf{x},y)\,d\mathbf{x}\,dy = \int\int l(f(\mathbf{x}),y)p(y|\mathbf{x})p(\mathbf{x})\,d\mathbf{x}\,dy$$

$$\int\int l(f(\mathbf{x}),y)p(y|\mathbf{x})p(\mathbf{x})\,d\mathbf{x}\,dy = \int\int l(f(\mathbf{x}),y)q(y|\mathbf{x})q(\mathbf{x})\frac{p(\mathbf{x})}{q(\mathbf{x})}\,d\mathbf{x}\,dy$$

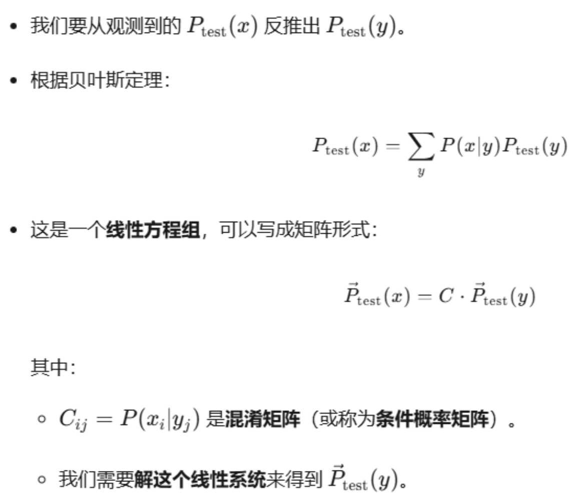
再改写如上，可以看做用q分布求E_risk时加上p/q的权重。

训练集和测试集（所用的）的**输入分布**不同，但是两者我们都可以拿到手。直接可以用一个**二分类器**（如逻辑回归）来估计这个比值即可（p/q）。
关于标签偏移：

同样改写：

$$\int\int l(f(\mathbf{x}),y)p(x|\mathbf{y})p(\mathbf{y})\,d\mathbf{x}\,dy = \int\int l(f(\mathbf{x}),y)q(x|\mathbf{y})q(\mathbf{y})\frac{p(\mathbf{y})}{q(\mathbf{y})}\,d\mathbf{x}\,dy$$

这里与上面不同点是：不能直接观测p（y）,只能观测(x)，所以：


- 我们要从观测到的 $P_{\text{test}}(x)$ 反推出 $P_{\text{test}}(y)$。
- 根据贝叶斯定理：

$$P_{\text{test}}(x) = \sum_y P(x|y)P_{\text{test}}(y)$$

- 这是一个**线性方程组**，可以写成矩阵形式：

$$\vec{P}_{\text{test}}(x) = C \cdot \vec{P}_{\text{test}}(y)$$

  其中：
  - $C_{ij} = P(x_i|y_j)$ 是**混淆矩阵**（或称为**条件概率矩阵**）。
  - 我们需要**解这个线性系统**来得到 $\vec{P}_{\text{test}}(y)$。

而q即训练集可得的分布，好估计。

code

```
def batchnorm_forward(x, gamma, beta, bn_param):
    """Forward pass for batch normalization.
    Input:
    - x: Data of shape (N, D)
    - gamma: Scale parameter of shape (D,)
    - beta: Shift paremeter of shape (D,)
    - bn_param: Dictionary with the following keys:
      - mode: 'train' or 'test'; required
      - eps: Constant for numeric stability
      - momentum: Constant for running mean / variance.
      - running_mean: Array of shape (D,) giving running mean of features
      - running_var Array of shape (D,) giving running variance of features
    Returns a tuple of:
    - out: of shape (N, D)
    - cache: A tuple of values needed in the backward pass
    """
    mode = bn_param["mode"]
    eps = bn_param.get("eps", 1e-5)
    momentum = bn_param.get("momentum", 0.9)

    N, D = x.shape
    running_mean = bn_param.get("running_mean", np.zeros(D, dtype=x.dty
pe))
    running_var = bn_param.get("running_var", np.zeros(D, dtype=x.dtype))

    out, cache = None, None
    if mode == "train":
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*
***
        mu = x.mean(axis=0)        # batch mean for each feature对列的mean，
即对每个特征
        var = x.var(axis=0)        # batch variance for each feature
        std = np.sqrt(var + eps)   # batch standard deviation for each feature
        x_hat = (x - mu) / std     # standartized x
        out = gamma * x_hat + beta # scaled and shifted x_hat

        shape = bn_param.get('shape', (N, D))             # reshape used in back
prop
        axis = bn_param.get('axis', 0)                    # axis to sum used in backpr
op
        cache = x, mu, var, std, gamma, x_hat, shape, axis # save for backpro
p

        if axis == 0:                # if batchnorm
            running_mean = momentum * running_mean + (1 - momentum) * mu
# update overall mean
```

```
        running_var = momentum * running_var + (1 - momentum) * var  # u
pdate overall variance

    elif mode == "test":

        x_hat = (x - running_mean) / np.sqrt(running_var + eps)
        out = gamma * x_hat + beta

                                                    ##############################
    else:
        raise ValueError('Invalid forward batchnorm mode "%s"' % mode)

    # Store the updated running means back into bn_param
    bn_param["running_mean"] = running_mean
    bn_param["running_var"] = running_var

    return out, cache
```

```
def batchnorm_backward(dout, cache):###正常backward
    """Backward pass for batch normalization.
    Inputs:
    - dout: Upstream derivatives, of shape (N, D)
    - cache: Variable of intermediates from batchnorm_forward.
    Returns a tuple of:
    - dx: Gradient with respect to inputs x, of shape (N, D)
    - dgamma: Gradient with respect to scale parameter gamma, of shape
(D,)
    - dbeta: Gradient with respect to shift parameter beta, of shape (D,)
    """
    dx, dgamma, dbeta = None, None, None
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)***
**
    # https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-
through-the-batch-normalization-layer.html

    x, mu, var, std, gamma, x_hat, shape, axis = cache         # expand cache

    dbeta = dout.reshape(shape, order='F').sum(axis)          # derivative w.r.
```

```
t. beta
    dgamma = (dout * x_hat).reshape(shape, order='F').sum(axis) # derivati
ve w.r.t. gamma

    dx_hat = dout * gamma                                  # derivative w.t.r. x_hat
    dstd = -np.sum(dx_hat * (x-mu), axis=0) / (std**2)          # derivative w.t.
r. std
    dvar = 0.5 * dstd / std                                # derivative w.t.r. var
    dx1 = dx_hat / std + 2 * (x-mu) * dvar / len(dout)          # partial derivative
w.t.r. dx
    dmu = -np.sum(dx1, axis=0)                             # derivative w.t.r. mu
    dx2 = dmu / len(dout)                                  # partial derivative w.t.r. dx
    dx = dx1 + dx2                                         # full derivative w.t.r. x
    ############################################################
######################

    return dx, dgamma, dbeta
```

```
def batchnorm_backward_alt(dout, cache):###更多数学计算，详细推导见笔
记或知乎：
#https://zhuanlan.zhihu.com/p/45614576
    dx, dgamma, dbeta = None, None, None
    ############################################################
######################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)***
**

    _, _, _, std, gamma, x_hat, shape, axis = cache # expand cache
    S = lambda x: x.sum(axis=0)                     # helper function

    dbeta = dout.reshape(shape, order='F').sum(axis)            # derivative w.r.
t. beta
    dgamma = (dout * x_hat).reshape(shape, order='F').sum(axis) # derivati
ve w.r.t. gamma

    dx = dout * gamma / (len(dout) * std)          # temporarily initialize scale v
alue
    dx = len(dout)*dx  - S(dx*x_hat)*x_hat - S(dx) # derivative w.r.t. unnorma
```

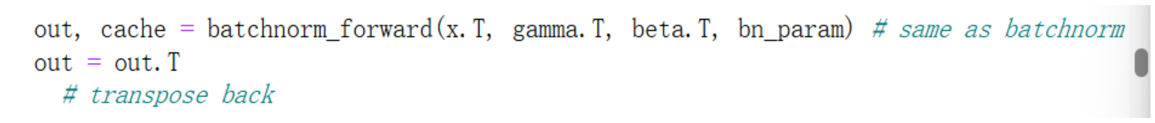
```
lized x
    ###########################################################
#######################

    return dx, dgamma, dbeta
```

1. **提升了最终性能**：通过创造更易于优化的损失地形。归一化，确保了输入分布的稳定性（缓解了**内部协变量偏移**）。这使得损失函数的拓扑结构**更加平滑、更易于优化**。梯度下降的路径因此更加直接地指向损失最小值，而不易在崎岖的损失地形中卡住或震荡。没有Batch Norm时，网络前层的微小变化会被后层放大，导致损失函数地形复杂，对初始化尺度极其敏感。
2. **增强了鲁棒性**：使模型性能不再依赖于寻找一个精确的初始化。**权重的尺度主要影响的是该层输出的缩放，而Batch Norm的"缩放和平移"参数（γ和β）会学习并补偿这种缩放效应**。因此，网络的实际输出对初始权重的绝对值不那么敏感。
3. **保证了训练稳定性**：有效缓解了梯度消失和爆炸问题。**有效地将激活值（和反向传播的梯度）重新拉回到一个合理的、稳定的动态范围**。

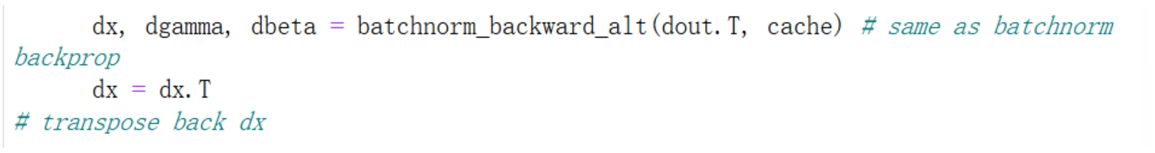
(上面为与权重初始化的关系)

Layer Normalization：

1607.06450（论文）实现了batch normalization　.T就可layer normal。同时不像batch n对批量大小有依赖性。但是当特征维度非常小时，特征向量中只有几个值，无法很好地近似这些特征的均值和方差，也无法将它们置于相似的尺度上。这会导致性能出现噪声。



```
out, cache = batchnorm_forward(x.T, gamma.T, beta.T, bn_param) # same as batchnorm
out = out.T
  # transpose back
```



```
      dx, dgamma, dbeta = batchnorm_backward_alt(dout.T, cache) # same as batchnorm
backprop
      dx = dx.T
# transpose back dx
```

Q3：Dropout.ipynb

1207.0580(论文)

在训练过程中，Dropout 可以被理解为在全神经网络中采样一个神经网络，并且仅根据输入数据更新采样网络的参数。（然而，可能的采样网络呈指数级增长，因为它们共

享参数。）在测试过程中不应用 dropout，其解释为评估所有子网络（呈指数规模）的集成平均预测。

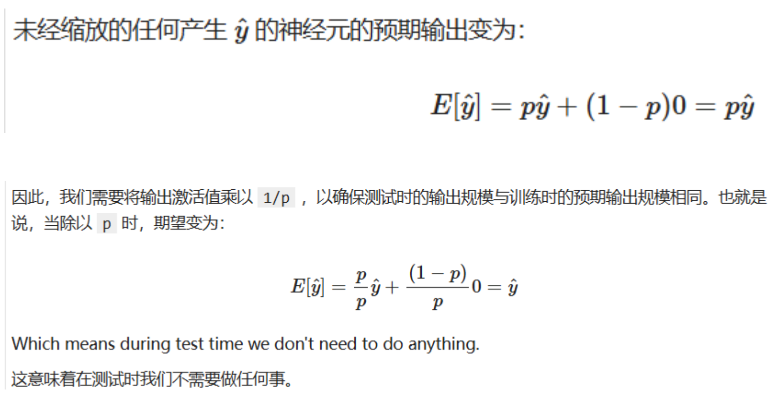
Dropout Training as Adaptive Regularization: “we show that the dropout regularizer is first-order equivalent to an L2 regularizer applied after scaling the features by an estimate of the inverse diagonal Fisher information matrix”. （dropout 正则化器与特征通过逆对角 Fisher 信息矩阵估计缩放后应用 L2 正则化器是一阶等价的）

1. code

```
def dropout_forward(x, dropout_param):
    """Forward pass for inverted dropout.
    Note that this is different from the vanilla version of dropout.
    Here, p is the probability of keeping a neuron output, as opposed to
    the probability of dropping a neuron output.
    See http://cs231n.github.io/neural-networks-2/#reg for more details.
    Inputs:
    - x: Input data, of any shape
    - dropout_param: A dictionary with the following keys:
      - p: Dropout parameter. We keep each neuron output with probability p.
      - mode: 'test' or 'train'. If the mode is train, then perform dropout;
        if the mode is test, then just return the input.
      - seed: Seed for the random number generator. Passing seed makes thi
s
        function deterministic, which is needed for gradient checking but not
        in real networks.
    Outputs:
    - out: Array of the same shape as x.
    - cache: tuple (dropout_param, mask). In training mode, mask is the drop
out
      mask that was used to multiply the input; in test mode, mask is None.
    """
    p, mode = dropout_param["p"], dropout_param["mode"]
    if "seed" in dropout_param:
        np.random.seed(dropout_param["seed"])
    mask = None
    out = None
    if mode == "train":
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*
****
        mask = (np.random.rand(*x.shape) < p) / p
        out = x * mask
        ###########################################################
####################
    elif mode == "test":
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*
****
        out = x
        ###########################################################
####################

    cache = (dropout_param, mask)
    out = out.astype(x.dtype, copy=False)

    return out, cache
```

```
def dropout_backward(dout, cache):
    """Backward pass for inverted dropout.
    Inputs:
    - dout: Upstream derivatives, of any shape
    - cache: (dropout_param, mask) from dropout_forward.
    """
    dropout_param, mask = cache
    mode = dropout_param["mode"]
    dx = None
    if mode == "train":
        ###########################################################
####################
        dx = dout * mask
        ###########################################################
####################
    elif mode == "test":
        dx = dout
    return dx
```

2. 为什么要除以p


未经缩放的任何产生 $\hat{y}$ 的神经元的预期输出变为：

$$E[\hat{y}] = p\hat{y} + (1-p)0 = p\hat{y}$$

因此，我们需要将输出激活值乘以 `1/p` ，以确保测试时的输出规模与训练时的预期输出规模相同。也就是说，当除以 `p` 时，期望变为：

$$E[\hat{y}] = \frac{p}{p}\hat{y} + \frac{(1-p)}{p}0 = \hat{y}$$

Which means during test time we don't need to do anything.

这意味着在测试时我们不需要做任何事。

Q4ConvolutionalNetworks.ipynb

关于im2col方法技巧

：(99+ 封私信 / 80 条消息) im2col方法实现卷积算法 - 知乎

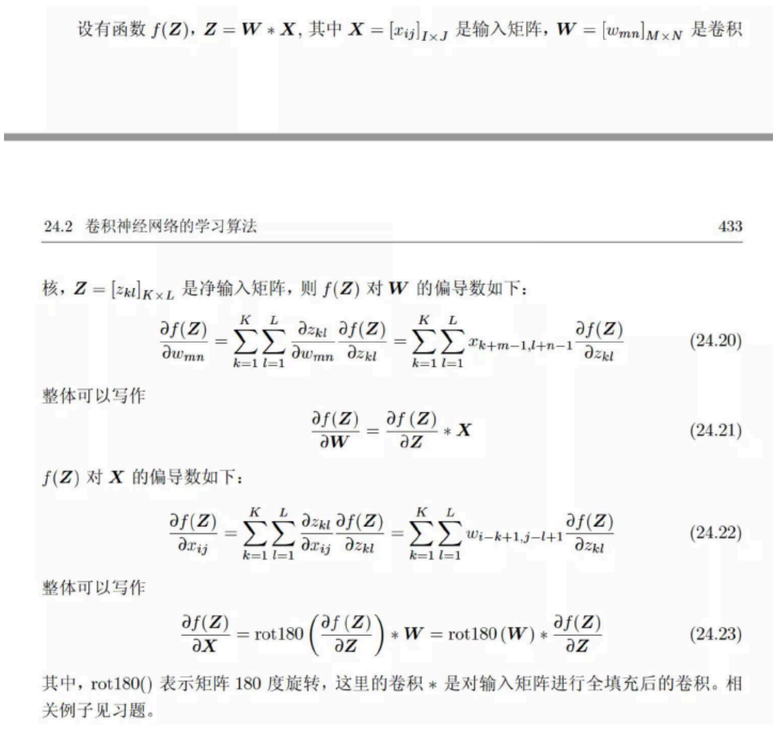
"im2col"技巧：

将输入图像的每个局部感受野展平为一列

将滤波器权重展平为矩阵的行

通过一次矩阵乘法完成所有位置的卷积运算

1. CNN的学习算法（数学推导）


设有函数 $f(\boldsymbol{Z})$，$\boldsymbol{Z} = \boldsymbol{W} * \boldsymbol{X}$，其中 $\boldsymbol{X} = [x_{ij}]_{I\times J}$ 是输入矩阵，$\boldsymbol{W} = [w_{mn}]_{M\times N}$ 是卷积

24.2 卷积神经网络的学习算法 433

核，$\boldsymbol{Z} = [z_{kl}]_{K\times L}$ 是净输入矩阵，则 $f(\boldsymbol{Z})$ 对 $\boldsymbol{W}$ 的偏导数如下：

$$\frac{\partial f(\boldsymbol{Z})}{\partial w_{mn}} = \sum_{k=1}^{K}\sum_{l=1}^{L}\frac{\partial z_{kl}}{\partial w_{mn}}\frac{\partial f(\boldsymbol{Z})}{\partial z_{kl}} = \sum_{k=1}^{K}\sum_{l=1}^{L} x_{k+m-1,l+n-1}\frac{\partial f(\boldsymbol{Z})}{\partial z_{kl}} \tag{24.20}$$

整体可以写作

$$\frac{\partial f(\boldsymbol{Z})}{\partial \boldsymbol{W}} = \frac{\partial f(\boldsymbol{Z})}{\partial \boldsymbol{Z}} * \boldsymbol{X} \tag{24.21}$$

$f(\boldsymbol{Z})$ 对 $\boldsymbol{X}$ 的偏导数如下：

$$\frac{\partial f(\boldsymbol{Z})}{\partial x_{ij}} = \sum_{k=1}^{K}\sum_{l=1}^{L}\frac{\partial z_{kl}}{\partial x_{ij}}\frac{\partial f(\boldsymbol{Z})}{\partial z_{kl}} = \sum_{k=1}^{K}\sum_{l=1}^{L} w_{i-k+1,j-l+1}\frac{\partial f(\boldsymbol{Z})}{\partial z_{kl}} \tag{24.22}$$

整体可以写作

$$\frac{\partial f(\boldsymbol{Z})}{\partial \boldsymbol{X}} = \mathrm{rot180}\left(\frac{\partial f(\boldsymbol{Z})}{\partial \boldsymbol{Z}}\right) * \boldsymbol{W} = \mathrm{rot180}(\boldsymbol{W}) * \frac{\partial f(\boldsymbol{Z})}{\partial \boldsymbol{Z}} \tag{24.23}$$

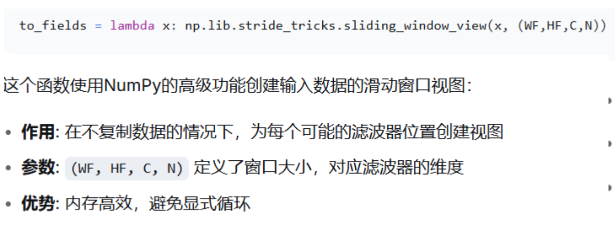
其中，rot180() 表示矩阵 180 度旋转，这里的卷积 $*$ 是对输入矩阵进行全填充后的卷积。相关例子见习题。

对于pooling:(max pooling就记录最大值位置，向该位置back;mean pooling 就除以核大小，平均“抹”back)

需要保证传递的loss（或者梯度）总和不变

2. code

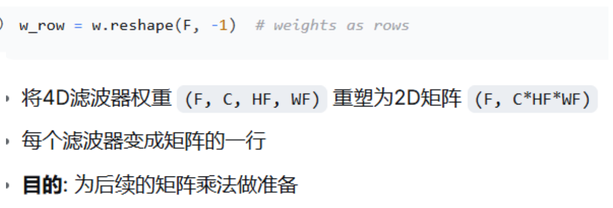
```
def conv_forward_naive(x, w, b, conv_param):
    """A naive implementation of the forward pass for a convolutional layer.
    The input consists of N data points, each with C channels, height H and
    width W. We convolve each input with F different filters, where each filter
    spans all C channels and has height HH and width WW.
    Input:
    - x: Input data of shape (N, C, H, W)
    - w: Filter weights of shape (F, C, HH, WW)
    - b: Biases, of shape (F,)
    - conv_param: A dictionary with the following keys:
      - 'stride': The number of pixels between adjacent receptive fields in the
        horizontal and vertical directions.
      - 'pad': The number of pixels that will be used to zero-pad the input.
    During padding, 'pad' zeros should be placed symmetrically (i.e equally
on both sides)
    along the height and width axes of the input. Be careful not to modfiy the
original
    input x directly.
    Returns a tuple of:
    - out: Output data, of shape (N, F, H', W') where H' and W' are given by
      H' = 1 + (H + 2 * pad - HH) / stride
      W' = 1 + (W + 2 * pad - WW) / stride
    - cache: (x, w, b, conv_param)
    """
    out = None
    ###########################################################################

    P1 = P2 = P3 = P4 = conv_param['pad'] # padding: up = right = down = l
eft
    S1 = S2 = conv_param['stride']       # stride:  up = down
    N, C, HI, WI = x.shape               # input dims
    F, _, HF, WF = w.shape               # filter dims
    HO = 1 + (HI + P1 + P3 - HF) // S1   # output height
    WO = 1 + (WI + P2 + P4 - WF) // S2    # output width

    # Helper function (warning: numpy version 1.20 or above is required for
usage)
    to_fields = lambda x: np.lib.stride_tricks.sliding_window_view(x, (WF,HF,
C,N))

    w_row = w.reshape(F, -1)                    # weights as rows
    x_pad = np.pad(x, ((0,0), (0,0), (P1, P3), (P2, P4)), 'constant')   # padded
inputs
    x_col = to_fields(x_pad.T).T[...,::S1,::S2].reshape(N, C*HF*WF, -1) # input
s as cols

    out = (w_row @ x_col).reshape(N, F, HO, WO) + np.expand_dims(b, axis=
(2,1))

    x = x_pad # we will use padded version as well during backpropagation

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        cache = (x, w, b, conv_param)
    return out, cache
```



```
to_fields = lambda x: np.lib.stride_tricks.sliding_window_view(x, (WF,HF,C,N))
```

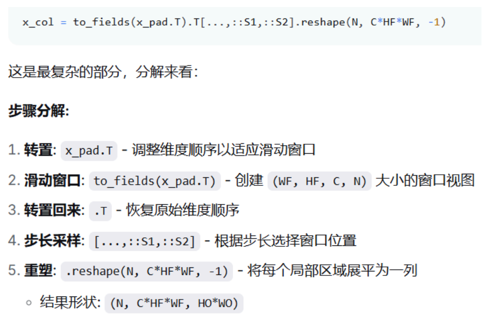
这个函数使用NumPy的高级功能创建输入数据的滑动窗口视图：

- **作用**: 在不复制数据的情况下，为每个可能的滤波器位置创建视图
- **参数**: `(WF, HF, C, N)` 定义了窗口大小，对应滤波器的维度
- **优势**: 内存高效，避免显式循环



```
w_row = w.reshape(F, -1)  # weights as rows
```

- 将4D滤波器权重 `(F, C, HF, WF)` 重塑为2D矩阵 `(F, C*HF*WF)`
- 每个滤波器变成矩阵的一行
- **目的**: 为后续的矩阵乘法做准备

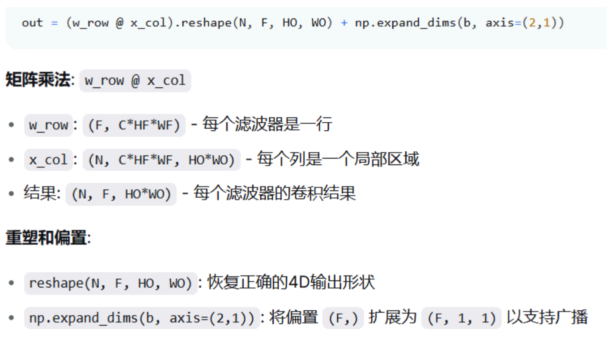


```
x_col = to_fields(x_pad.T).T[...,::S1,::S2].reshape(N, C*HF*WF, -1)
```

这是最复杂的部分，分解来看：

步骤分解:

1. **转置**: `x_pad.T` - 调整维度顺序以适应滑动窗口
2. **滑动窗口**: `to_fields(x_pad.T)` - 创建 `(WF, HF, C, N)` 大小的窗口视图
3. **转置回来**: `.T` - 恢复原始维度顺序
4. **步长采样**: `[...,::S1,::S2]` - 根据步长选择窗口位置
5. **重塑**: `.reshape(N, C*HF*WF, -1)` - 将每个局部区域展平为一列
    - 结果形状: `(N, C*HF*WF, HO*WO)`



```
out = (w_row @ x_col).reshape(N, F, HO, WO) + np.expand_dims(b, axis=(2,1))
```

矩阵乘法: `w_row @ x_col`

- `w_row`: `(F, C*HF*WF)` - 每个滤波器是一行
- `x_col`: `(N, C*HF*WF, HO*WO)` - 每个列是一个局部区域
- 结果: `(N, F, HO*WO)` - 每个滤波器的卷积结果

重塑和偏置:

- `reshape(N, F, HO, WO)`: 恢复正确的4D输出形状
- `np.expand_dims(b, axis=(2,1))`: 将偏置 `(F,)` 扩展为 `(F, 1, 1)` 以支持广播

```
def conv_backward_naive(dout, cache):
    """A naive implementation of the backward pass for a convolutional layer.
    Inputs:
    - dout: Upstream derivatives.
    - cache: A tuple of (x, w, b, conv_param) as in conv_forward_naiv
    Returns a tuple of:
    - dx: Gradient with respect to x
    - dw: Gradient with respect to w
    - db: Gradient with respect to b
    """
    dx, dw, db = None, None, None
    ###########################################################################
    to_fields = np.lib.stride_tricks.sliding_window_view

    x_pad, w, b, conv_param = cache       # extract parameters from cache
    S1 = S2 = conv_param['stride']        # stride:  up = down
    P1 = P2 = P3 = P4 = conv_param['pad'] # padding: up = right = down = left
    F, C, HF, WF = w.shape                # filter dims
    N, _, HO, WO = dout.shape             # output dims

    dout = np.insert(dout, [*range(1, HO)] * (S1-1), 0, axis=2)         # "missing" rows
    dout = np.insert(dout, [*range(1, WO)] * (S2-1), 0, axis=3)         # "missing" columns
    dout_pad = np.pad(dout, ((0,), (0,), (HF-1,), (WF-1,)), 'constant') # for full convolution

    x_fields = to_fields(x_pad, (N, C, dout.shape[2], dout.shape[3]))   # input local regions w.r.t. dout
    dout_fields = to_fields(dout_pad, (N, F, HF, WF))                  # dout local regions w.r.t. filter
    w_rot = np.rot90(w, 2, axes=(2, 3))                                # rotated kernel (for convolution)

    db = np.einsum('ijkl->j', dout)                                    # sum over
    dw = np.einsum('ijkl,mnopiqkl->jqop', dout, x_fields)                    # cor
relate
    dx = np.einsum('ijkl,mnopqikl->qjop', w_rot, dout_fields)[..., P1:-P3, P2:-P
4] # convolve                   #
    ###########################################################
######################
    return dx, dw, db
```

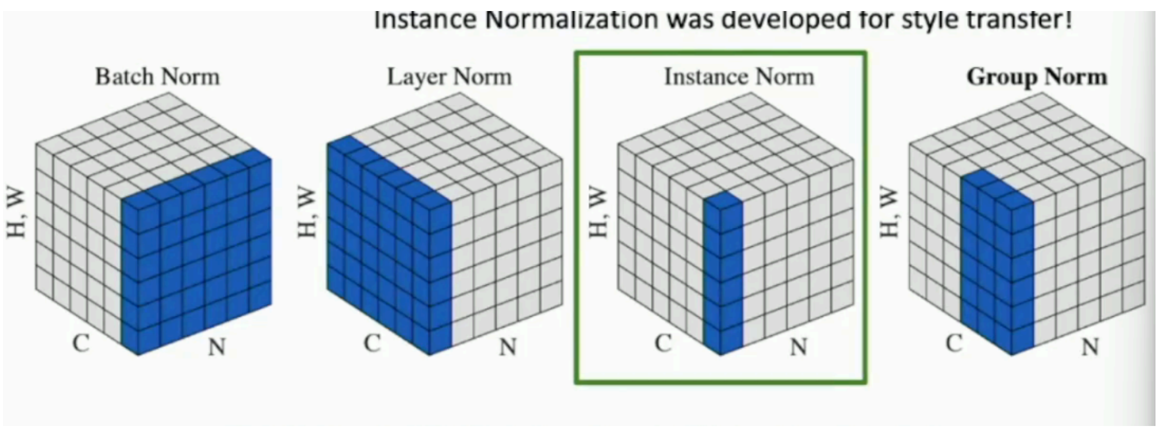
3. max_pool

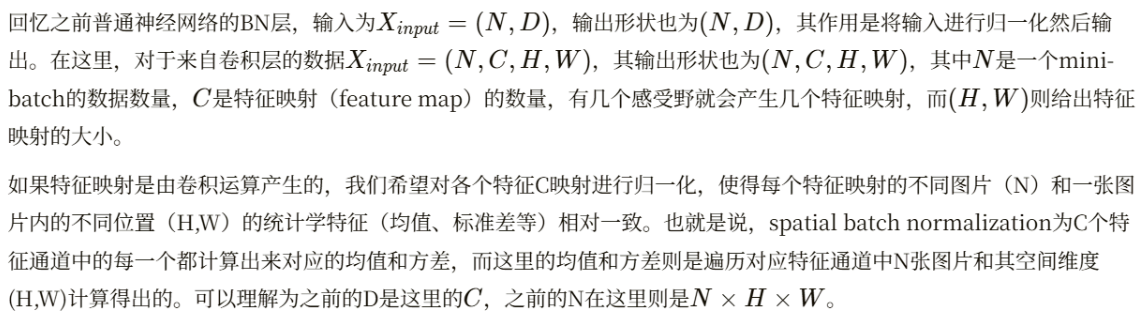
```
def max_pool_forward_naive(x, pool_param):
    """A naive implementation of the forward pass for a max-pooling layer.
    Inputs:
    - x: Input data, of shape (N, C, H, W)
    - pool_param: dictionary with the following keys:
      - 'pool_height': The height of each pooling region
      - 'pool_width': The width of each pooling region
      - 'stride': The distance between adjacent pooling regions
    No padding is necessary here, eg you can assume:
      - (H - pool_height) % stride == 0
      - (W - pool_width) % stride == 0
    Returns a tuple of:
    - out: Output data, of shape (N, C, H', W') where H' and W' are given by
      H' = 1 + (H - pool_height) / stride
      W' = 1 + (W - pool_width) / stride
    - cache: (x, pool_param)
    """
    out = None
    ###########################################################
######################
    S1 = S2 = pool_param['stride'] # stride: up = down
    HP = pool_param['pool_height'] # pool height
    WP = pool_param['pool_width']  # pool width
    N, C, HI, WI = x.shape         # input dims
    HO = 1 + (HI - HP) // S1       # output height
    WO = 1 + (WI - WP) // S2       # output width

    # Helper function (warning: numpy version 1.20 or above is required for
usage)
    to_fields = lambda x: np.lib.stride_tricks.sliding_window_view(x, (WP,HP,
C,N))

    x_fields = to_fields(x.T).T[...,::S1,::S2].reshape(N, C, HP*WP, -1) # input lo
cal regions
    out = x_fields.max(axis=2).reshape(N, C, HO, WO)                 # pooled o
utput                    #
    ##############################################################
########################
    cache = (x, pool_param)
    return out, cache
```

```
def max_pool_backward_naive(dout, cache):
    """A naive implementation of the backward pass for a max-pooling laye
    Inputs:
    - dout: Upstream derivatives
    - cache: A tuple of (x, pool_param) as in the forward pass.
    Returns:
    - dx: Gradient with respect to x
    """
    dx = None
    ##############################################################
########################
    x, pool_param = cache     # expand cache
    N, C, HO, WO = dout.shape # get shape values
    dx = np.zeros_like(x)     # init derivative

    S1 = S2 = pool_param['stride'] # stride: up = down
    HP = pool_param['pool_height'] # pool height
    WP = pool_param['pool_width']  # pool width

    for i in range(HO):
        for j in range(WO):
            [ns, cs], h, w = np.indices((N, C)), i*S1, j*S2    # compact indexing
            f = x[:, :, h:(h+HP), w:(w+WP)].reshape(N, C, -1)  # input local fields
            k, l = np.unravel_index(np.argmax(f, 2), (HP, WP)) # offsets for max
vals
```

```
          dx[ns, cs, h+k, w+l] += dout[ns, cs, i, j]        # select areas to updat
e                    #
   ############################################################
#######################
   return dx
```

4. **(Spatial)Batch Normalization(BatchNorm2D)**




回忆之前普通神经网络的BN层，输入为$X_{input} = (N, D)$，输出形状也为(N, D)，其作用是将输入进行归一化然后输出。在这里，对于来自卷积层的数据$X_{input} = (N, C, H, W)$，其输出形状也为(N, C, H, W)，其中N是一个mini-batch的数据数量，C是特征映射（feature map）的数量，有几个感受野就会产生几个特征映射，而(H, W)则给出特征映射的大小。

如果特征映射是由卷积运算产生的，我们希望对各个特征C映射进行归一化，使得每个特征映射的不同图片（N）和一张图片内的不同位置（H,W）的统计学特征（均值、标准差等）相对一致。也就是说，spatial batch normalization为C个特征通道中的每一个都计算出来对应的均值和方差，而这里的均值和方差则是遍历对应特征通道中N张图片和其空间维度(H,W)计算得出的。可以理解为之前的D是这里的C，之前的N在这里则是$N \times H \times W$。

1803.08494（Group Normalization）:

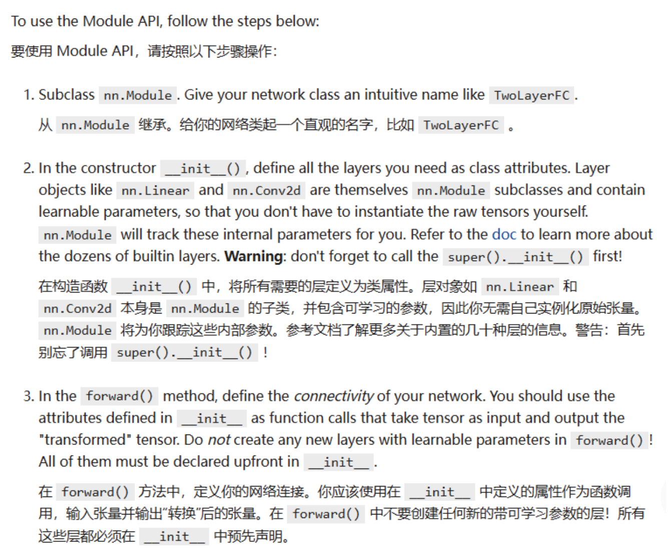
他们用来说明这一点的一个例子是，传统计算机视觉中许多高性能的手工特征都有明确分组在一起的项。例如直方图梯度方向直方图{用于人体检测的定向梯度直方图 |IEEE 会议出版物 |IEEE Xplore}——在计算每个空间局部块的直方图后，每个块直方图在连接在一起形成最终特征向量之前会被归一化。

(相关代码对FNN的batch norm复用，改"保留通道维数即可")

Q5 PyTorch.ipynb

1. Justin Johnson 's tutorial for PyTorch

barebone即用torch.tensor（参数要手动requires_grad = True） 与 import torch.nn.functional as F `nn.Module` API 供你定义任意网络架构，同时自动跟踪所有可学习参数。



To use the Module API, follow the steps below:

要使用 Module API，请按照以下步骤操作:

1. Subclass `nn.Module`. Give your network class an intuitive name like `TwoLayerFC`.

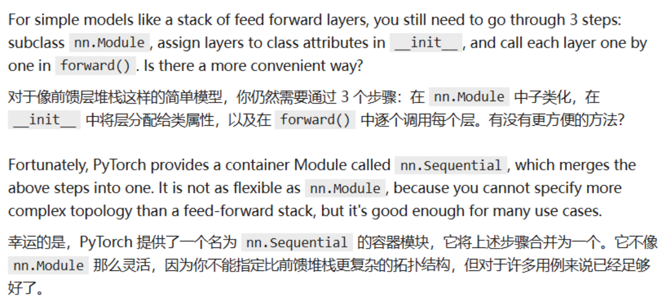
   从 `nn.Module` 继承。给你的网络类起一个直观的名字，比如 `TwoLayerFC` 。

2. In the constructor `__init__()`, define all the layers you need as class attributes. Layer objects like `nn.Linear` and `nn.Conv2d` are themselves `nn.Module` subclasses and contain learnable parameters, so that you don't have to instantiate the raw tensors yourself. `nn.Module` will track these internal parameters for you. Refer to the doc to learn more about the dozens of builtin layers. **Warning**: don't forget to call the `super().__init__()` first!

   在构造函数 `__init__()` 中，将所有需要的层定义为类属性。层对象如 `nn.Linear` 和 `nn.Conv2d` 本身是 `nn.Module` 的子类，并包含可学习的参数，因此你无需自己实例化原始张量。`nn.Module` 将为你跟踪这些内部参数。参考文档了解更多关于内置的几十种层的信息。警告：首先别忘了调用 `super().__init__()` ！

3. In the `forward()` method, define the *connectivity* of your network. You should use the attributes defined in `__init__` as function calls that take tensor as input and output the "transformed" tensor. Do *not* create any new layers with learnable parameters in `forward()`! All of them must be declared upfront in `__init__`.

   在 `forward()` 方法中，定义你的网络连接。你应该使用在 `__init__` 中定义的属性作为函数调用，输入张量并输出"转换"后的张量。在 `forward()` 中不要创建任何新的带可学习参数的层！所有这些层都必须在 `__init__` 中预先声明。



For simple models like a stack of feed forward layers, you still need to go through 3 steps: subclass `nn.Module`, assign layers to class attributes in `__init__`, and call each layer one by one in `forward()`. Is there a more convenient way?

对于像前馈层堆栈这样的简单模型，你仍然需要通过 3 个步骤：在 `nn.Module` 中子类化，在 `__init__` 中将层分配给类属性，以及在 `forward()` 中逐个调用每个层。有没有更方便的方法?

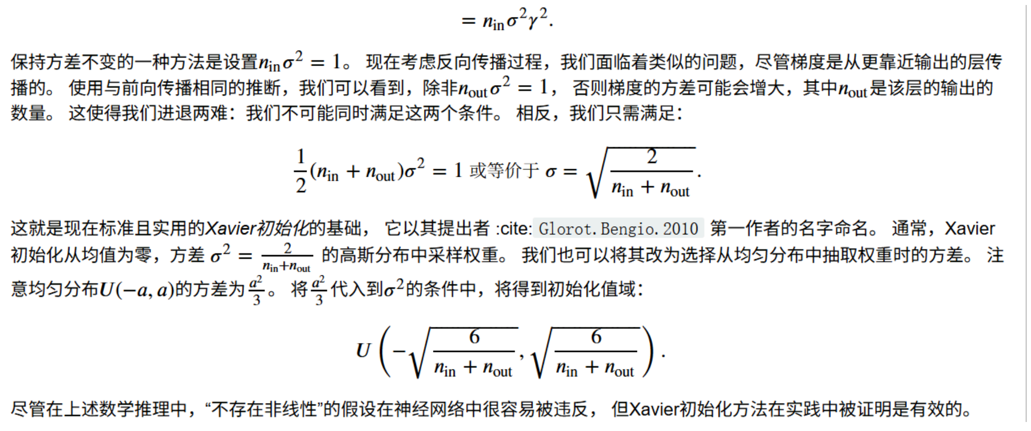
Fortunately, PyTorch provides a container Module called `nn.Sequential`, which merges the above steps into one. It is not as flexible as `nn.Module`, because you cannot specify more complex topology than a feed-forward stack, but it's good enough for many use cases.

幸运的是，PyTorch 提供了一个名为 `nn.Sequential` 的容器模块，它将上述步骤合并为一个。它不像 `nn.Module` 那么灵活，因为你不能指定比前馈堆栈更复杂的拓扑结构，但对于许多用例来说已经足够好了。

API	Flexibility 灵活性	Convenience 便利性
Barebone 基础	High 高	Low 低
`nn.Module`	High 高	Medium
`nn.Sequential`	Low 低	High 高

2. 初始化

Kaiming 正态初始化方法: 1502.01852(论文)



$$= n_{\text{in}}\sigma^2\gamma^2.$$

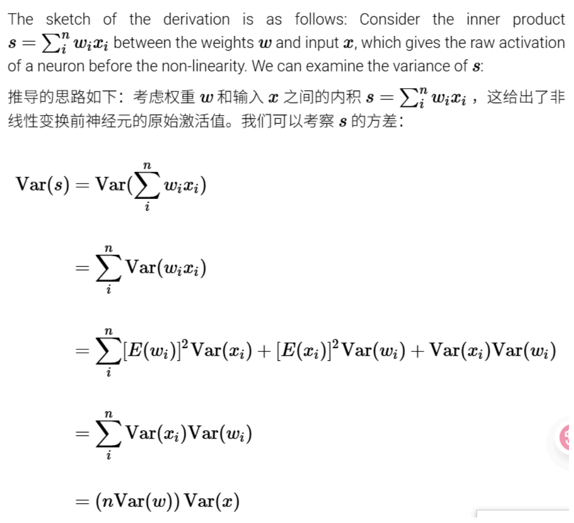
保持方差不变的一种方法是设置$n_{\text{in}}\sigma^2 = 1$。 现在考虑反向传播过程，我们面临着类似的问题，尽管梯度是从更靠近输出的层传播的。 使用与前向传播相同的推断，我们可以看到，除非$n_{\text{out}}\sigma^2 = 1$， 否则梯度的方差可能会增大，其中$n_{\text{out}}$是该层的输出的数量。 这使得我们进退两难：我们不可能同时满足这两个条件。 相反，我们只需满足：

$$\frac{1}{2}(n_{\text{in}} + n_{\text{out}})\sigma^2 = 1 \text{ 或等价于 } \sigma = \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}.$$

这就是现在标准且实用的*Xavier初始化*的基础， 它以其提出者 :cite:`Glorot.Bengio.2010` 第一作者的名字命名。 通常，Xavier初始化从均值为零，方差 $\sigma^2 = \frac{2}{n_{\text{in}}+n_{\text{out}}}$ 的高斯分布中采样权重。 我们也可以将其改为选择从均匀分布中抽取权重时的方差。 注意均匀分布$U(-a, a)$的方差为$\frac{a^2}{3}$。 将$\frac{a^2}{3}$代入到σ^2的条件中，将得到初始化值域：

$$U\left(-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right).$$

尽管在上述数学推理中，“不存在非线性”的假设在神经网络中很容易被违反， 但Xavier初始化方法在实践中被证明是有效的。



The sketch of the derivation is as follows: Consider the inner product $s = \sum_i^n w_i x_i$ between the weights w and input x, which gives the raw activation of a neuron before the non-linearity. We can examine the variance of s:

推导的思路如下：考虑权重 w 和输入 x 之间的内积 $s = \sum_i^n w_i x_i$，这给出了非线性变换前神经元的原始激活值。我们可以考察 s 的方差：

$$
\begin{aligned}
\mathrm{Var}(s) &= \mathrm{Var}(\sum_i^n w_i x_i) \\
&= \sum_i^n \mathrm{Var}(w_i x_i) \\
&= \sum_i^n [E(w_i)]^2 \mathrm{Var}(x_i) + [E(x_i)]^2 \mathrm{Var}(w_i) + \mathrm{Var}(x_i)\mathrm{Var}(w_i) \\
&= \sum_i^n \mathrm{Var}(x_i)\mathrm{Var}(w_i) \\
&= (n\mathrm{Var}(w))\,\mathrm{Var}(x)
\end{aligned}
$$

DenseNets 1608.06993(assignment2 PyTorch.ipynb有实现一简单nets，正确get 79.73%)

3. more

- Alternative optimizers: you can try Adam, Adagrad, RMSprop, etc.

  其他优化器：你可以尝试 Adam、Adagrad、RMSprop 等。

- Alternative activation functions such as leaky ReLU, parametric ReLU, ELU, or MaxOut.

  替代激活函数，如 leaky ReLU、parametric ReLU、ELU 或 MaxOut。

- Model ensembles

  模型集成

- Data augmentation

  数据增强

- New Architectures

  新架构

  - ResNets where the input from the previous layer is added to the output.

    ResNets 中，前一层的输入会加到输出上。

  - DenseNets where inputs into previous layers are concatenated together.

    DenseNets 中，输入到前一层的会连接在一起。

  - This blog has an in-depth overview这篇博客有深入的概述

4. about code

assignment 3

Q1 RNN_Captioning.ipynb

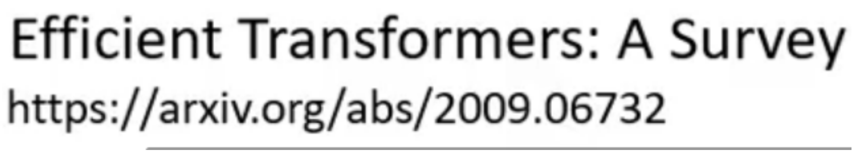
1. 文件 `cs231n/rnn_layers.py` 包含了循环神经网络所需的各类层实现，而文件 `cs231n/classifiers/rnn.py` 则使用这些层来实现图像描述模型。
2. about embedding：

词嵌入的"学习"本质上是：

通过大量文本数据中单词的共现模式和预测任务，让每个词的向量逐渐调整到能够最佳配合其各种上下文用法的位置。

3. 数学推导见李航（P450→P452）
4. code这里不贴了。`cs231n/rnn_layers.py`

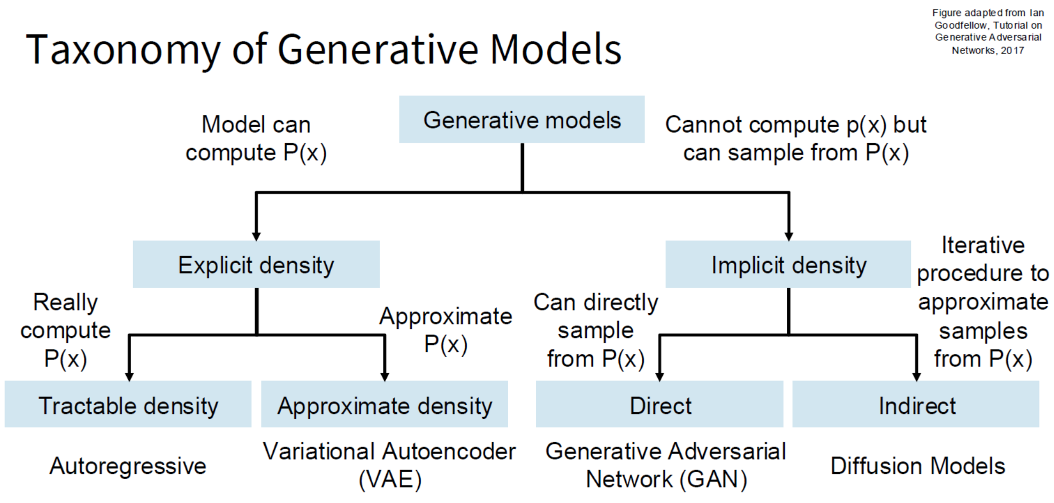
Q2Transformer_Captioning.ipynb



Efficient Transformers: A Survey
https://arxiv.org/abs/2009.06732

Q3Generative_Adversarial_Networks.ipynb

else.


Taxonomy of Generative Models
Figure adapted from Ian Goodfellow, Tutorial on Generative Adversarial Networks, 2017
Generative models
Model can compute P(x)
Cannot compute p(x) but can sample from P(x)
Explicit density
Implicit density
Really compute P(x)
Approximate P(x)
Can directly sample from P(x)
Iterative procedure to approximate samples from P(x)
Tractable density
Approximate density
Direct
Indirect
Autoregressive
Variational Autoencoder (VAE)
Generative Adversarial Network (GAN)
Diffusion Models

Variational Autoencoders (VAEs):

Variational Autoencoders (VAEs).pdf

https://arxiv.org/pdf/1406.2661（GAN）

https://arxiv.org/pdf/1611.04076（Least Squares Generative Adversarial Networks）

https://arxiv.org/pdf/1511.06434(DCGAN)

https://arxiv.org/pdf/1606.03657(InfoGAN)

https://arxiv.org/pdf/2006.11239(Denoising Diffusion Probabilistic Models)

https://arxiv.org/pdf/2103.00020(Learning transferable visual models from natural language supervision)(CLIP)

https://arxiv.org/abs/2207.12598

Q4Self_Supervised_Learning.ipynb

自监督学习的方法主要可以分为 3 类：1. 基于上下文（Context based） 2. 基于时序（Temporal Based）3. 基于对比（Contrastive Based）

自监督学习 | (1) Self-supervised Learning入门_carl 等人[]通过名为拼图的方式来构造辅助任务,通过预测图像各个图像块的相对位置-CSDN博客

Q5LSTM_Captioning.ipynb

第25章 循环神经网络 （这里有LSTM的backward数学推导）