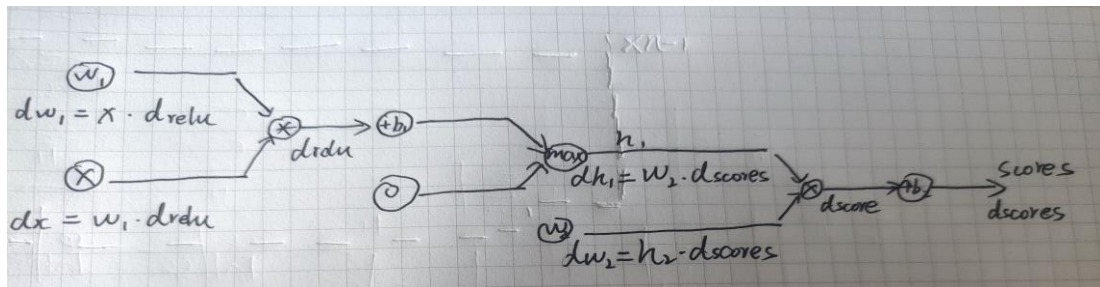


CS231n Assignment #2

● Fully-connected Neural Network:

Q1: 实现 n 层全连神经网络（前向传播和反向传播）。

A1: 本小节可视为 assignment1 two-layer network 的拓展，由 2 层网络向 n 层网络推进，其模型结构如下图所示：



可认为虚线框内前 n-1 层均做如下变换，其中激活函数 f 为 relu:

$$H = f(W_1 X + b_1)$$

最后一层（第 n 层）则为预测分数层 S，其中打分函数为 Softmax:

$$S = f(W_2 H + b_2)$$

因此，在正向传播的过程中，我们可以分两步进行计算，即先循环计算 1 到 n-1 的正向传播，再计算第 n 层的正向传播，相关代码如下（红色箭头标出）：

```
for i in range(self.num_layers - 1):
    if self.normalization == 'batchnorm':
        hidden, cache[i+1] = affine_bn_relu_forward(hidden,
            self.params['W' + str(i+1)],
            self.params['b' + str(i+1)],
            self.params['gamma' + str(i+1)],
            self.params['beta' + str(i+1)],
            self.bn_params[i])
    elif self.normalization == 'layernorm':
        hidden, cache[i + 1] = affine_ln_relu_forward(hidden,
            self.params['W' + str(i + 1)],
            self.params['b' + str(i + 1)],
            self.params['gamma' + str(i + 1)],
            self.params['beta' + str(i + 1)],
            self.bn_params[i])
    else:
        hidden, cache[i+1] = affine_relu_forward(hidden, self.params['W' + str(i+1)],
            self.params['b' + str(i+1)])
    if self.use_dropout:
        hidden, cache_dropout[i+1] = dropout_forward(hidden, self.dropout_param)
# 最后一层不用激活层
scores, cache[self.num_layers] = affine_forward(hidden, self.params['W' + str(self.num_layers)],
    self.params['b' + str(self.num_layers)])
```

反向传播阶段于此同理，先计算第 n 层的梯度，前 n-1 层的反向传播过程是相似的故可通过循环求解，相应代码如下（红色箭头标出，使用 L2 正则项）：

```

loss, dS = softmax_loss(scores, y)
# 最后一层没有relu激活层
dhidden, grads['W' + str(self.num_layers)], grads['b' + str(self.num_layers)] = \
    affine_backward(dS, cache[self.num_layers])
loss += 0.5 * self.reg * np.sum(self.params['W' + str(self.num_layers)] * self.params['W' + str(self.num_layers)])
grads['W' + str(self.num_layers)] += self.reg * self.params['W' + str(self.num_layers)]

for i in range(self.num_layers - 1, 0, -1):
    loss += 0.5 * self.reg * np.sum(self.params["W" + str(i)] * self.params["W" + str(i)])
    # 倒着求梯度
    if self.use_dropout:
        dhidden = dropout_backward(dhidden, cache_dropout[i])
    if self.normalization == 'batchnorm':
        dhidden, dw, db, dgamma, dbeta = affine_bn_relu_backward(dhidden, cache[i])
        grads['gamma' + str(i)] = dgamma
        grads['beta' + str(i)] = dbeta
    elif self.normalization == 'layernorm':
        dhidden, dw, db, dgamma, dbeta = affine_ln_relu_backward(dhidden, cache[i])
        grads['gamma' + str(i)] = dgamma
        grads['beta' + str(i)] = dbeta
    else:
        dhidden, dw, db = affine_relu_backward(dhidden, cache[i])
        grads['W' + str(i)] = dw + self.reg * self.params['W' + str(i)]
        grads['b' + str(i)] = db

```

Q2: 实现多种优化方法（SGD+Momentum, RMSprop, Adam）。

A2:

1. 冲量法（SGD+Momentum）:

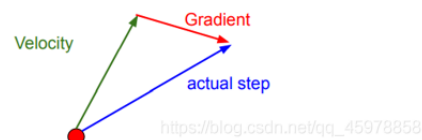
使用冲量法，意在使算法经过鞍点或是停滞区时，不至于停下来做过于缓慢的迭代，且经过不是很“深”的极值时，可借助冲量项带来的惯性冲出极值所在的坑。因此相较于普通的随机梯度下降，该方法能使神经网络更快速收敛。

公式如下：

Nesterov Momentum

$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$

$$x_{t+1} = x_t + v_{t+1}$$



相关代码实现如下：

```

# 梯度影响速度，而不直接影响位置
v = config['momentum'] * v - config['learning_rate'] * dw
next_w = w + v
config['velocity'] = v

```

此外，因为算法的变化，停止算法的标准也不再是梯度小于一个阈值，还可以是冲量小于某个值，梯度小于某个值，或是用户给定某个次数就停止。

2. AdaGrad（自适应学习率，补充）:

基本思想是对每个变量使用不同的学习率，在优化过程中，对已经下降很多的变量，减缓学习率，对还没怎么变化的变量，保持一个较大学习率：

$$\mathbf{g}_t = \partial_{\mathbf{w}} l(y_t, f(\mathbf{x}_t, \mathbf{w})),$$

$$\mathbf{s}_t = \mathbf{s}_{t-1} + \mathbf{g}_t^2,$$

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t} + \epsilon} \cdot \mathbf{g}_t.$$

本质上是每个变量都随着学习的进行，根据历史学习率累计总量来决定当前学习率减小的幅度。

3. RMSprop:

adagrad 中学习率按 $O(t^{-1/2})$ 时间预期降低。虽然这通常适用于凸问题，但对于非凸问题，例如深度学习中遇到的问题，可能并不理想。且其将梯度 \mathbf{g}_t 的平方积累成状态矢量 $\mathbf{s}_t = \mathbf{s}_t - 1 + \mathbf{g}_t^2$ ，缺乏规范化， \mathbf{s}_t 呈线性增长。从而导致训练速度极速减小。

RMSProp 维持一个第二动量，即梯度的平方和，使得梯度大得方向更新稍微缓慢一点，梯度小的方向更新稍微快一点。同时第二动量也会随着时间消减（可能会造成训练一直变慢）。其公式如下：

$$\begin{aligned}\mathbf{s}_t &\leftarrow \gamma \mathbf{s}_{t-1} + (1 - \gamma) \mathbf{g}_t^2, \\ \mathbf{x}_t &\leftarrow \mathbf{x}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t} + \epsilon} \odot \mathbf{g}_t.\end{aligned}$$

相关代码如下：

```
config['cache'] = config['decay_rate'] * config['cache'] + (1 - config['decay_rate']) * dw * dw
next_w = w - config['learning_rate'] * dw / (np.sqrt(config['cache'] + config['epsilon']))
```

4. Adam:

Adam 集成了上述优化算法的优点，同时维持一个第一动量（类似于之前 sgd 里的动量）和一个第二动量（即梯度的平方和）。同时还需要设置一个偏置修正项，防止刚开始第一动量大，第二动量小的时候学习率太大。公式如下：

$$\begin{aligned}\mathbf{v}_t &\leftarrow \beta_1 \mathbf{v}_{t-1} + (1 - \beta_1) \mathbf{g}_t, \\ \mathbf{s}_t &\leftarrow \beta_2 \mathbf{s}_{t-1} + (1 - \beta_2) \mathbf{g}_t^2. \\ \hat{\mathbf{v}}_t &= \frac{\mathbf{v}_t}{1 - \beta_1^t} \text{ and } \hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1 - \beta_2^t}. \\ \mathbf{g}'_t &= \frac{\eta \hat{\mathbf{v}}_t}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon}. \\ \mathbf{x}_t &\leftarrow \mathbf{x}_{t-1} - \mathbf{g}'_t.\end{aligned}$$

相应代码如下：

```
config['m'] = config['beta1'] * config['m'] + (1 - config['beta1']) * dw
# 第二动量
config['v'] = config['beta2'] * config['v'] + (1 - config['beta2']) * dw * dw
# 偏置修正
config['t'] += 1
m_unbias = config['m'] / (1 - config['beta1'] ** config['t'])
v_unbias = config['v'] / (1 - config['beta2'] ** config['t'])
# 更新参数
next_w = w - m_unbias * config['learning_rate'] / (np.sqrt(v_unbias) + config['epsilon'])
```

总结：

- 1、MLP 通过激活函数增加非线性，能够拟合更多非线性问题。
- 2、在深度学习优化过程中，Adam 往往能取得相对较好的优化时间、效果。

- 3、Relu 函数在输入值均为负数时可能产生梯度消失，可通过使用 leaky relu 解决。
- 4、在深度神经网络中，权重初始化显得十分重要，初始化过小可能导致梯度消失问题，而初始化太大，又可能导致梯度爆炸问题。

● Batch Normalization:

当输入数据是**不相关、零均值**以及**单元方差**的时候，我们的机器学习方法往往表现得很好。但是，当我们训练深度神经网络的时候，即便我们预处理数据使得输入数据服从这样的分布，不断的网络层的处理也会使得原始分布发生改变。更严重得使，随着权重得不断更新，每一层得输入特征的分布也会不断地发生偏移。

所以，科学家们假设，输入特征分布的偏移会使得深度神经网络的训练变得困难，从而提出插入一个批量归一化层来处理这个问题。

在训练阶段，采用一个小批量的数据来估计每一个特征维度的均值和方差，并用它来处理输入的小批量数据，使得它们零均值和去相关化。同时，维护一个训练集上得平均均值和方差，用来在测试集上处理数据。

但是，这样得 BN 层或许会因为改变的输入特的分布而影响网络的表达能力，即对于某些网络层，非零均值和单元方差的数据分布可能会更好。所以，对于每一个 BN 层，我们会学习一个**偏移因子**和**尺度变化因子**来适当的恢复每一个特征维度的分布，使得其不是严格服从我们得标准分布，这样增加网络的丰富性。

如果我们尝试使用大小为 1 的小批量应用批量归一化，我们将无法学到任何东西。这是因为在减去均值之后，每个隐藏单元将为 0。所以，只有使用足够大的小批量，批量归一化这种方法才是有效且稳定的。请注意，在应用批量归一化时，批量大小的选择可能比没有批量归一化时更重要。

前向传播如下图所示：

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots x_m\}$;

Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

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

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

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

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

相应代码如下：

```

batch_mean = np.mean(x, axis = 0)
batch_var = np.var(x, axis = 0)
# 存储训练时候的均值和方差
running_mean = momentum * running_mean + (1 - momentum) * batch_mean
running_var = momentum * running_var + (1 - momentum) * batch_var
x_std = (x - batch_mean) / (np.sqrt(batch_var) + eps)
out = gamma * x_std + beta
cache = [gamma, x_std, beta, x, batch_mean, batch_var, eps]

```

由于批量归一化层可视为独立层级结构，故其反向传播可通过对正向传播直接求导实现，推导过程如下：

$$\frac{\partial L}{\partial \hat{x}_i} = \frac{\partial L}{\partial y_i} \cdot r$$

$$\frac{\partial L}{\partial \sigma_\beta^2} = \sum_{i=1}^m \frac{\partial L}{\partial \hat{x}_i} (x_i - \mu_\beta) \cdot \left(-\frac{1}{2}\right) (\sigma_\beta^2 + \epsilon)^{-\frac{3}{2}}$$

$$\frac{\partial L}{\partial \mu_\beta} = \left(\sum_{i=1}^m \frac{\partial L}{\partial \hat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_\beta^2 + \epsilon}} \right) + \frac{\partial L}{\partial \sigma_\beta^2} \left(\frac{\sum_{i=1}^m -2(x_i - \mu_\beta)}{m} \right)$$

$$\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial \hat{x}_i} \cdot \frac{1}{\sqrt{\sigma_\beta^2 + \epsilon}} + \frac{\partial L}{\partial \sigma_\beta^2} \cdot \frac{2(x_i - \mu_\beta)}{m} + \frac{\partial L}{\partial \mu_\beta} \cdot \frac{1}{m}$$

$$\frac{\partial L}{\partial r} = \sum_{i=1}^m \frac{\partial L}{\partial y_i} \cdot \hat{x}_i$$

$$\frac{\partial L}{\partial \beta} = \sum_{i=1}^m \frac{\partial L}{\partial y_i}$$

相关代码实现如下：

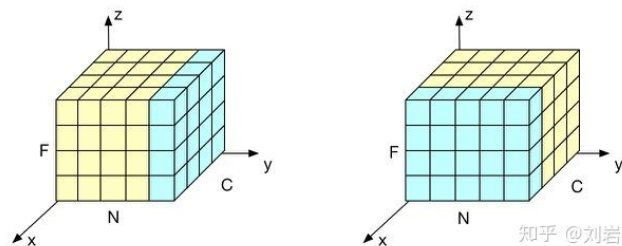
```

gamma, x_std, beta, x, batch_mean, batch_var, eps = cache
N = x.shape[0]
# 先计算变化因子，好计算一点
dgamma = np.sum(dout * x_std, axis = 0)
dbeta = np.sum(dout, axis = 0)
# 再计算对x的梯度
a = 1 / np.sqrt(batch_var + eps)
dx_hat = dout * gamma
# dvar = np.sum(dx_hat * (x - batch_mean) * (-0.5) * (a ** 3), axis = 0)
# dmean = np.sum(-dx_hat * a, axis = 0) #+ dvar * (-2 / N) * np.sum(x - batch_mean, axis = 0) #后面这项为0
dx = dx_hat * a + np.sum(dx_hat * (x - batch_mean) * (-0.5) * (a ** 3), axis = 0) * 2 * (x - batch_mean) / N + np.sum(-dx_hat * a, axis = 0) /

```

补充: Layer Normalization (LN)

BN 用于 RNN 等动态网络和 batch size 较小的时候效果不好。Layer Normalization (LN) 的提出有效的解决 BN 的这两个问题。LN 和 BN 不同点是归一化的维度是互相垂直的，如下图所示。在图中 N 表示样本轴 C 表示通道轴，F 是每个通道的特征数量。BN 如右侧所示，它是取不同样本的同一个通道的特征做归一化；LN 则是如左侧所示，它取的是同一个样本的不同通道做归一化。



其正向传播公式如下：

$$\mu^l = \frac{1}{H} \sum_{i=1}^H a_i^l \quad \sigma^l = \sqrt{\frac{1}{H} \sum_{i=1}^H (a_i^l - \mu^l)^2}$$

$$\hat{\mathbf{a}}^l = \frac{\mathbf{a}^l - \mu^l}{\sqrt{(\sigma^l)^2 + \epsilon}}$$

$$\mathbf{h} = f\left(\frac{\mathbf{g}}{\sqrt{\sigma^2 + \epsilon}} \odot (\mathbf{a} - \mu) + \mathbf{b}\right)$$

相应代码实现如下：

```
x = x.T # (D, N)
_mean = np.mean(x, axis=0) # (N,)
_var = np.var(x, axis=0) # (N,)
x_hat = (x - _mean) / (np.sqrt(_var + eps)) # (D, N)
x_hat = x_hat.T # (N, D)
out = x_hat * gamma + beta
cache = (gamma, x_hat, x, _mean, _var, eps)
```

用同样方法可推导其反向传播过程如下：

$$\begin{aligned} \frac{\partial L}{\partial \sigma} &= \sum_{i=1}^n \frac{\partial L}{\partial h_i} \cdot \hat{a}_i \\ \frac{\partial L}{\partial \beta} &= \sum_{i=1}^n \frac{\partial L}{\partial h_i} \\ \frac{\partial L}{\partial \hat{a}_i} &= \frac{\partial L}{\partial g_{h_i}} \cdot \gamma \\ \frac{\partial L}{\partial \sigma^2} &= \sum_{i=1}^n \frac{\partial L}{\partial \hat{a}_i} (\hat{a}_i - \mu) \left(-\frac{1}{2}\right) (\sigma^2 + \epsilon)^{-\frac{3}{2}} \\ \frac{\partial L}{\partial \mu} &= \left(\sum_{i=1}^n \frac{\partial L}{\partial \hat{a}_i} \cdot \frac{-1}{\sqrt{\sigma^2 + \epsilon}} \right) + \frac{\partial L}{\partial \sigma^2} \left(\frac{\sum_{i=1}^n -2(\hat{a}_i - \mu)}{n} \right) \\ \frac{\partial L}{\partial \hat{a}_i} &= \frac{\partial L}{\partial \hat{a}_i} \cdot \frac{1}{\sqrt{\sigma^2 + \epsilon}} + \frac{\partial L}{\partial \sigma^2} \cdot \frac{2(\hat{a}_i - \mu)}{n} + \frac{\partial L}{\partial \mu} \cdot \frac{1}{n} \end{aligned}$$

相应代码实现如下：


```

gamma, x_hat, x, _mean, _var, eps = cache
N = x_hat.shape[1]
dgamma = np.sum(dout * x_hat, axis = 0)
dbeta = np.sum(dout, axis = 0)
dx_hat = (dout * gamma).T
a = np.sqrt(_var + eps)
# 先计算方差
dvar = np.sum(- 0.5 * (x - _mean) * dx_hat / a ** 3, axis=0)
dmean = np.sum(- dx_hat / a, axis=0) + dvar * np.sum(-2 * (x - _mean), axis=0) / N
dx = dx_hat / a + dmean / N + 2 * dvar * (x - _mean) / N

dx = dx.T

```

总结:

- 1、BN 层使得网络的训练对网络参数初始化变得不那么敏感
- 2、BN 层适合大的 batch size，因为大的批量使得我们得均值和方差估计得更准确。
- 3、LN 是和 BN 非常近似的一种归一化方法，不同的是 BN 取的是不同样本的同一个特征，而 LN 取的是同一个样本的不同特征。在 BN 和 LN 都能使用的场景中，BN 的效果一般优于 LN，原因是基于不同数据，同一特征得到的归一化特征更不容易损失信息。

● Dropout:

为了防止神经网络过拟合数据，可以采用 dropout 方法。其主要思想是：对隐藏层中部分输出或者权重随机置为 0。

在标准 dropout 正则化中，通过按保留（未丢弃）的节点的分数进行归一化来消除每一层的偏差。换言之，每个中间激活值 h 以丢弃概率 p 由随机变量 h' 替换，如下所示：

$$h' = \begin{cases} 0 & \text{概率为 } p \\ \frac{h}{1-p} & \text{其他情况} \end{cases}$$

从而保持期望值保持不变，即 $E[h'] = E[h]$ 。

相应代码实现：

```

mask = ( np.random.rand(*x.shape) < p ) / p
out = x * mask

```

易得其反向传播为：

```
dx = dout * mask
```

总结:

- 1、除了控制权重向量的维数和大小之外，dropout 也是避免过拟合的另一种工具。它们通常是联合使用的。
- 2、dropout 可能会适当地降低网络的表示能力，但是能在一定程度上提高模型的泛化能力。带有 dropout 的模型相当于多个模型的集成。但是如果保留的神经元太少，则会让模型很难拟合数据集。
- 3、dropout 仅在训练期间使用。

● Convolutional Networks:

Q1:实现卷积层的批量归一化

A1: 通过之前的实验不难发现, 批量归一化是训练深度全连接网络时的一种非常有用的技术。原论文 Sergey Ioffe and Christian Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”, ICML 2015. 中提出的, 批量归一化也可以被用于卷积网络, 但我们需要对其进行一点调整, 这个修改将被称为“空间批量归一化”。

通常的应用于全连接层的批量归一化层接受形状 (N, D) 的输入, 并产生形状 (N, D) 的输出。而应用于卷积层时, 批量归一化需要接受形状 (N, C, H, W) 的输入并产生形状 (N, C, H, W) 的输出, 其中 N 维给出了批处理大小, (H, W) 维给出了特征图的空间大小。

如果特征向量是使用卷积生成的, 那么我们期望每个特征通道的统计量在不同图像之间以及同一图像内不同位置之间都是相对一致的。因此, 空间批量归一化通过计算批处理维数 N 和空间维数 H、W 的统计量来计算每个特征通道 (C) 的平均值和方差。其实现可以借助之前实现的批量归一化函数, 相应代码如下:

前向传播:

```
N, C, H, W = x.shape
# (N, H, W, C)
input = x.transpose(0, 2, 3, 1).reshape(-1, C)
temp_out, cache = batchnorm_forward(input, gamma, beta, bn_param)
out = temp_out.reshape(N, H, W, C).transpose(0, 3, 1, 2)
```

反向传播:

```
N, C, H, W = dout.shape
dout = dout.transpose(0, 2, 3, 1).reshape(-1, C)
dx, dgamma, dbeta = batchnorm_backward(dout, cache)
dx = dx.reshape(N, H, W, C).transpose(0, 3, 1, 2)
```

Q2:实现卷积层的组归一化

A2: 在之前的部分, 我们提到过 Layer Normalization 是一种替代的标准化技术, 它减轻了 batch Normalization 的批大小限制。然而当使用卷积层时, 层标准化的性能不如批处理标准化。

在全连接层中, 层中的所有隐藏单元都倾向于对最终预测做出类似的贡献, 重新定位和重新缩放对层的求和输入效果很好。然而, 类似贡献的假设不再适用于卷积神经网络。因为在卷积神经网络中接收域位于图像边界附近的大量隐藏单元很少被打开, 因此与同一层内的其他隐藏单元的统计数据有很大的不同。

论文 Wu, Yuxin, and Kaiming He. “Group Normalization.” arXiv preprint arXiv:1803.08494 (2018). 提出了一种中间技术。与 Layer Normalization(对每个数据点的整个特性进行标准化)相反, 他们建议将每个数据点特性一致地拆分为组, 然后进行每组每数据点的标准化。其 S 值定义如下:

$$\mathcal{S}_i = \{k \mid k_N = i_N, \lfloor \frac{k_C}{C/G} \rfloor = \lfloor \frac{i_C}{C/G} \rfloor\}.$$

它与 i_N 类似, 差别在于对通道分组处理。G 是预先定义的分组数, 默认为 32, $\lfloor \cdot \rfloor$ 为向下取整符号, 在 (H, W, C/G) 三个维度上计算均值和方差, GN 针对每个组计算 μ σ , 针对每个通道学习 γ β 。相关代码如下:

正向传播:


```

N, C, H, W = x.shape
size = (N * G, C // G * H * W)
x = x.reshape(size).T
gamma = gamma.reshape(1, C, 1, 1)
beta = beta.reshape(1, C, 1, 1)
# similar to batch normalization
mu = x.mean(axis=0)
var = x.var(axis=0) + eps
std = np.sqrt(var)
z = (x - mu) / std
z = z.T.reshape(N, C, H, W)
out = gamma * z + beta
# save values for backward call
cache = {'std': std, 'gamma': gamma, 'z': z, 'size': size}

```

反向传播:

```

N, C, H, W = dout.shape
size = cache['size']
dbeta = dout.sum(axis=(0, 2, 3), keepdims=True)
dgamma = np.sum(dout * cache['z'], axis=(0, 2, 3), keepdims=True)

# reshape tensors
z = cache['z'].reshape(size).T
M = z.shape[0]
dfdz = dout * cache['gamma']
dfdz = dfdz.reshape(size).T
# copy from batch normalization backward alt
dfdz_sum = np.sum(dfdz, axis=0)
dx = dfdz - dfdz_sum / M - np.sum(dfdz * z, axis=0) * z / M
dx /= cache['std']
dx = dx.T.reshape(N, C, H, W)

```

补充：多种归一化模型：

对于归一化层，目前主要有这几个方法，Batch Normalization、Layer Normalization、Instance Normalization、Group Normalization、Switchable Normalization。如下图所示：

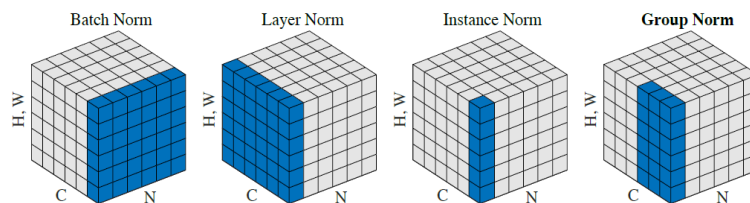


Figure 2. **Normalization methods.** Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

将输入的图片 shape 记为 $[N, C, H, W]$ ，这几个方法特征如下：

- 1、batchNorm 是在 batch 上，对 NHW 做归一化，对小 batchsize 效果不好；
- 2、layerNorm 在通道方向上，对 CHW 归一化，主要对 RNN 作用明显；
- 3、instanceNorm 在图像像素上，对 HW 做归一化，用在风格化迁移；
- 4、GroupNorm 将 channel 分组，然后再做归一化；
- 5、SwitchableNorm 是将 BN、LN、IN 结合，赋予权重，让网络自己去学习归一化层应该使用什么方法。

其中，SN 相应公式如下：

SN has an intuitive expression

$$\hat{h}_{ncij} = \gamma \frac{h_{ncij} - \sum_{k \in \Omega} w_k \mu_k}{\sqrt{\sum_{k \in \Omega} w_k' \sigma_k^2 + \epsilon}} + \beta, \quad (3)$$

$$w_k = \frac{e^{\lambda_k}}{\sum_{z \in \{\text{in}, \text{ln}, \text{bn}\}} e^{\lambda_z}}, \quad k \in \{\text{in}, \text{ln}, \text{bn}\},$$

$$\mu_{\text{in}} = \frac{1}{HW} \sum_{i,j}^{H,W} h_{ncij}, \quad \sigma_{\text{in}}^2 = \frac{1}{HW} \sum_{i,j}^{H,W} (h_{ncij} - \mu_{\text{in}})^2,$$


$$\mu_{\text{ln}} = \frac{1}{C} \sum_{c=1}^C \mu_{\text{in}}, \quad \sigma_{\text{ln}}^2 = \frac{1}{C} \sum_{c=1}^C (\sigma_{\text{in}}^2 + \mu_{\text{in}}^2) - \mu_{\text{ln}}^2,$$

$$\mu_{\text{bn}} = \frac{1}{N} \sum_{n=1}^N \mu_{\text{in}}, \quad \sigma_{\text{bn}}^2 = \frac{1}{N} \sum_{n=1}^N (\sigma_{\text{in}}^2 + \mu_{\text{in}}^2) - \mu_{\text{bn}}^2, \quad (5)$$

● Pytorch:

Q1: 运用所学知识构筑模型，要求在 cifar10 验证集上打到 70%以上正确率。

A1: 使用简单的数据裁剪，放缩作为图像增强方法，模型使用自己构筑的 resnet18，经过调参，在验证集上准确率约为 0.886，在测试集上准确率为 0.875，如下图所示：

CIFAR-10 - Object Recognition in Images				
Identify the subject of 60,000 labeled images				
 Kaggle · 231 teams · 7 years ago				
Overview	Data	Code	Discussion	Leaderboard
				Rules
				Team
				My Submissions
				Late Submission
				...
Your most recent submission				
Name	Submitted	Wait time	Execution time	Score
submission (1).csv	6 days ago	1 seconds	2 seconds	0.87560
Complete				
Jump to your position on the leaderboard				

之后又尝试了使用预训练的 resnet34 模型，在运行到 30 个 epoch 时，验证集上准确率和自构筑的 resnet18 模型持平，且 loss 处于平稳下降阶段，但是之后运行会卡住（可能是内存问题），推测经过足够的运行周期后，该预训练模型能够取得更好的测试集准确率。

● 本章总结:

在经过上一章实验的锻炼后，在本章实验的前 3 部分中，能够较为顺利的实现反向传播，但在第 4 部分卷积层中，随着维度的升高，对反向传播过程中各变量形状变换的推理和理解依旧比较吃力，还需要进一步加强（目前还有一个报错没有解决）。

通过本章节学习，复习巩固了 MLP, Adam, RMsprop, BN, Dropout, Conv 相关知识，学习了 LN, GN, SN 等新的归一化模型，对归一化的理解获得了进一步提高。