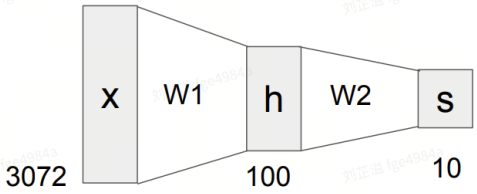
Neural Work

- 广泛的概念，又被称作**全连接网络**或**多层感知机**（MLP）

Neural networks: hierarchical computation

(**Before**) Linear score function: $f = Wx$

(**Now**) 2-layer Neural Network $f = W_2 \max(0, W_1 x)$



$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

Activation function

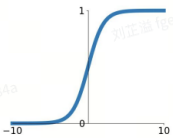
- 如果没有激活函数，神经网络退化成线性分类器

Activation functions

ReLU is a good default choice for most problems

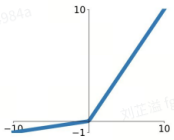
Sigmoid

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



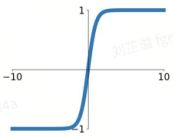
Leaky ReLU

$\max(0.1x, x)$



tanh

$\tanh(x)$

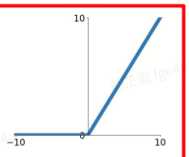


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

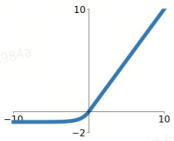
ReLU

$\max(0, x)$



ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



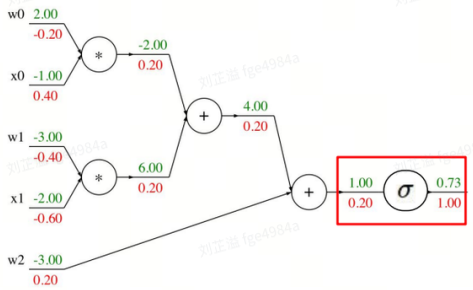
实现过程

- 定义网络
- 前向传播
- 计算梯度
- 更新梯度

反向传播

- **减小工作量，更改中间模型，不影响反向传播过程**
- 使用链式求导法则，进行梯度的更改

Backprop Implementation: "Flat" code



Forward pass: Compute output

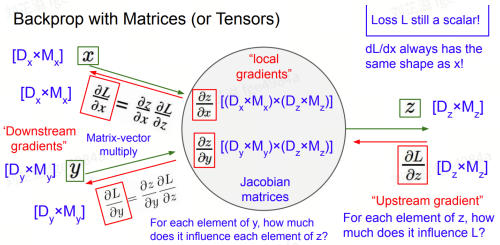
```
def f(w0, x0, w1, x1, w2):
  s0 = w0 * x0
  s1 = w1 * x1
  s2 = s0 + s1
  s3 = s2 + w2
  L = sigmoid(s3)
```

Sigmoid

```
  grad_L = 1.0
  grad_s3 = grad_L * (1 - L) * L
  grad_w2 = grad_s3
  grad_s2 = grad_s3
  grad_s0 = grad_s2
  grad_s1 = grad_s2
  grad_w1 = grad_s1 * x1
  grad_x1 = grad_s1 * w1
  grad_w0 = grad_s0 * x0
  grad_x0 = grad_s0 * w0
```

向量化的反向传播

- 损失是一个标量，输入是向量



[N×D] [N×M] [M×D]

$$\frac{\partial L}{\partial x} = \left(\frac{\partial L}{\partial y}\right) w^T$$

[D×M] [D×N] [N×M]

$$\frac{\partial L}{\partial w} = x^T \left(\frac{\partial L}{\partial y}\right)$$

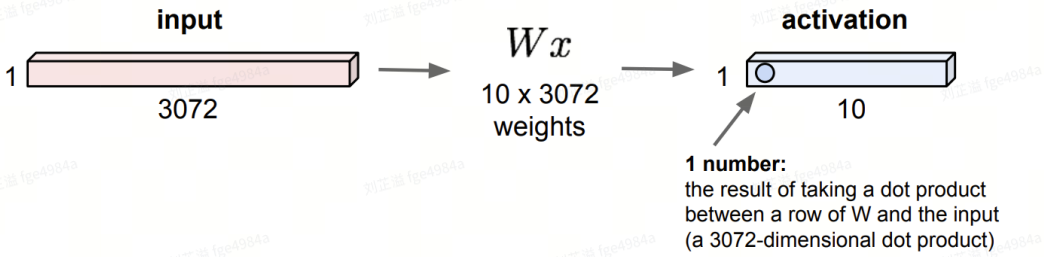
Convolutional Neural Networks

- 卷积神经网络

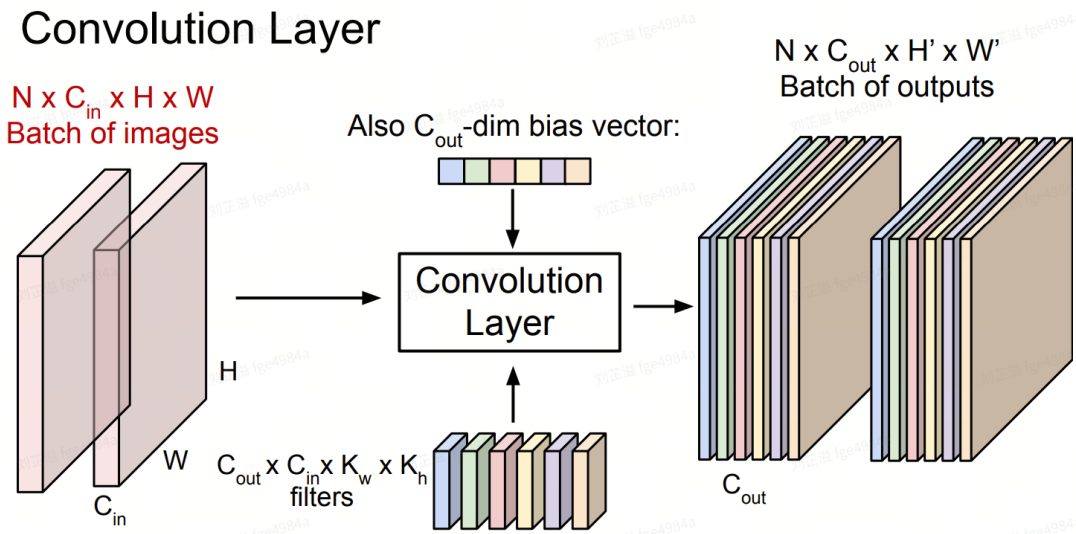
全连接层与卷积层

Fully Connected Layer

32x32x3 image -> stretch to 3072 x 1



- 卷积层使用多个卷积核进行卷积



- 卷积后的大小计算公式：P是padding的值，stride是步长，F是卷积核的大小

(recall:)
(N + 2P - F) / stride + 1

问题

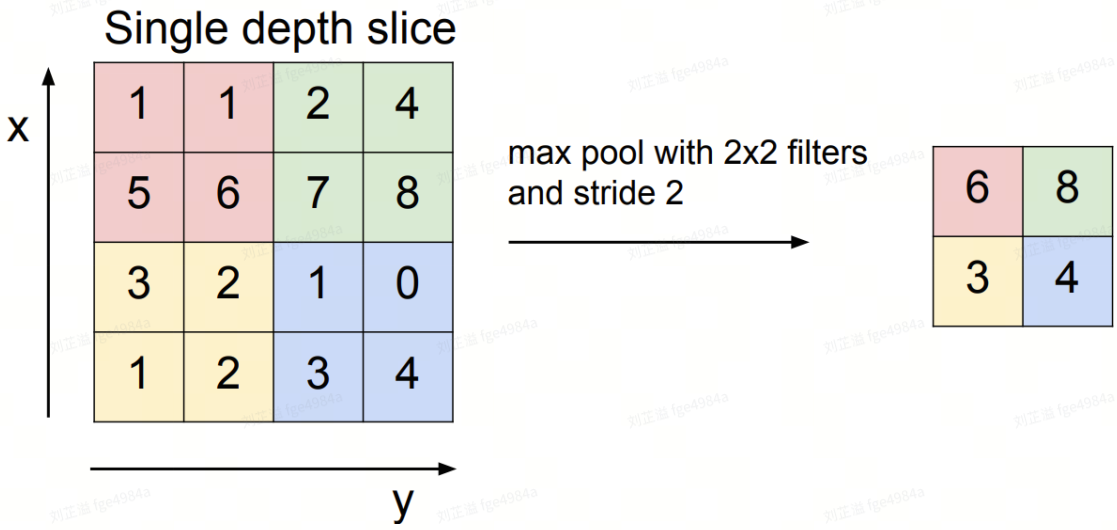
- 需要很多层才能让输出的感受野为整张图片
  - 使用下采样（Downsample）来解决
  - 使用strided convolution

Pooling layer

- 使得每个输出的更小且更易于管理
- 独立地对每个激活图进行操作

MAX POOLING

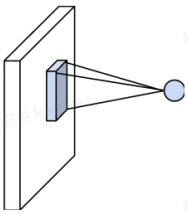
- 最大池化层



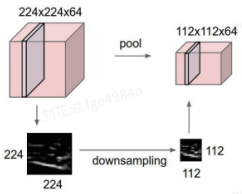
- Historically architectures looked like
  [(CONV-RELU)*N-POOL?]*M-(FC-RELU)*K,SOFTMAX
  where N is usually up to ~5, M is large, 0 <= K <= 2.
- But recent advances such as ResNet/GoogLeNet have challenged this paradigm

Components of CNNs

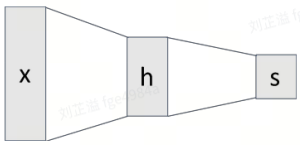
Convolution Layers



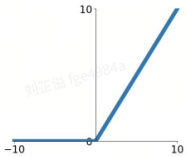
Pooling Layers



Fully-Connected Layers



Activation Function



Normalization

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

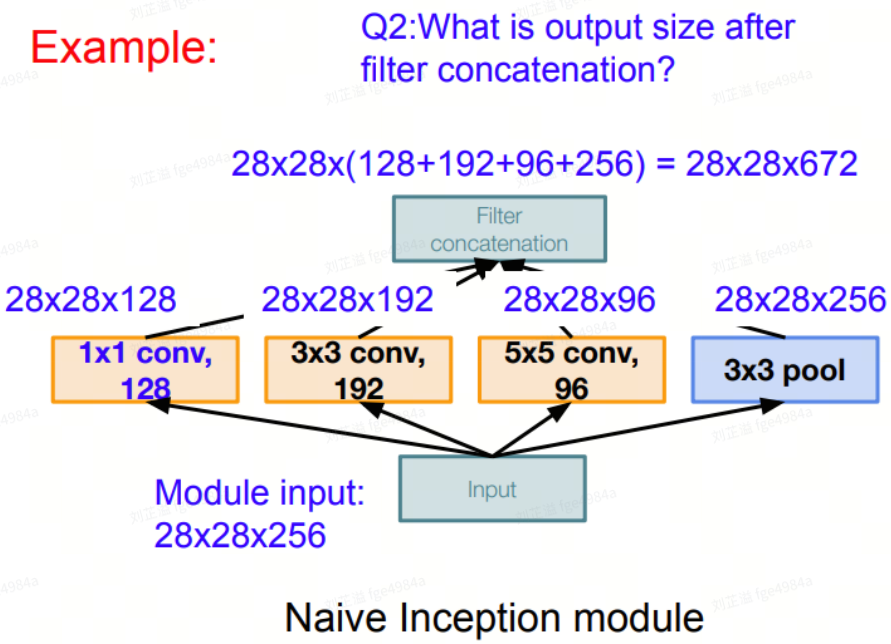
Case

VGGNet

- 使用多个卷积层，卷积核较小
- Stack of three 3x3 conv (stride 1) layers has same effective receptive field as one 7x7 conv layer
- 消耗大量内存；最后的全连接层拥有大量参数

GoogLeNet

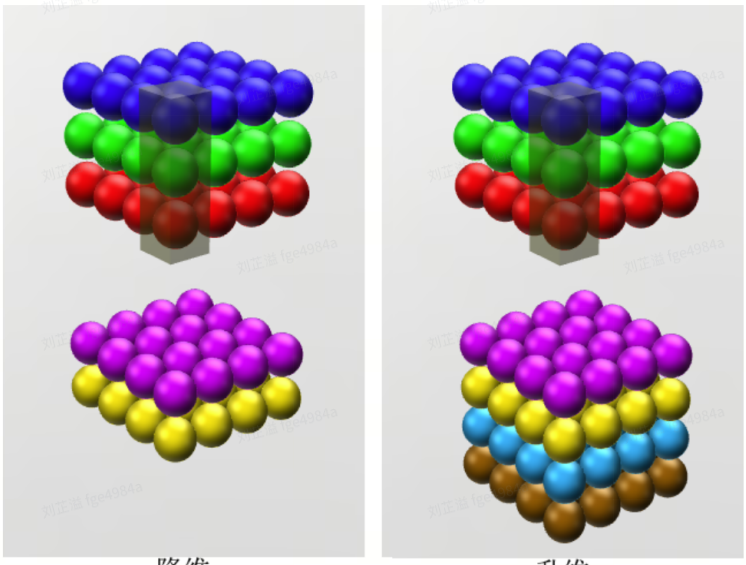
- 并行使用多个卷积层，每个卷积核的大小均不同



Naive Inception module

1*1的CONV

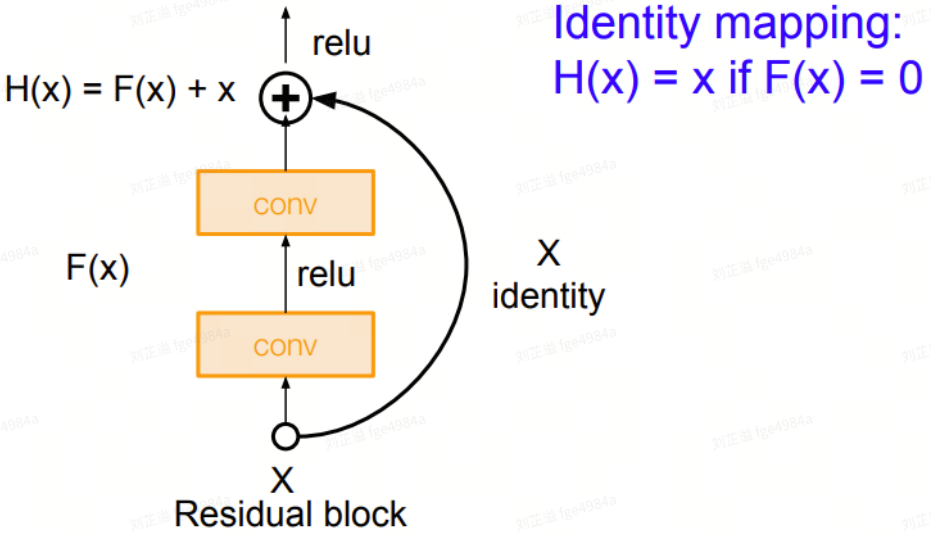
- 对每个像素点，在不同的channels上进行线性组合；保留图片原有的平面结构，调控层的深度
- 输入为6x6x32时，1x1卷积的形式是1x1x32，当只有一个1x1卷积核的时候，此时输出为6x6x1。此时便可以体会到1x1卷积的实质作用：降维。当1x1卷积核的个数小于输入channels数量时，即降维



降维　　升维

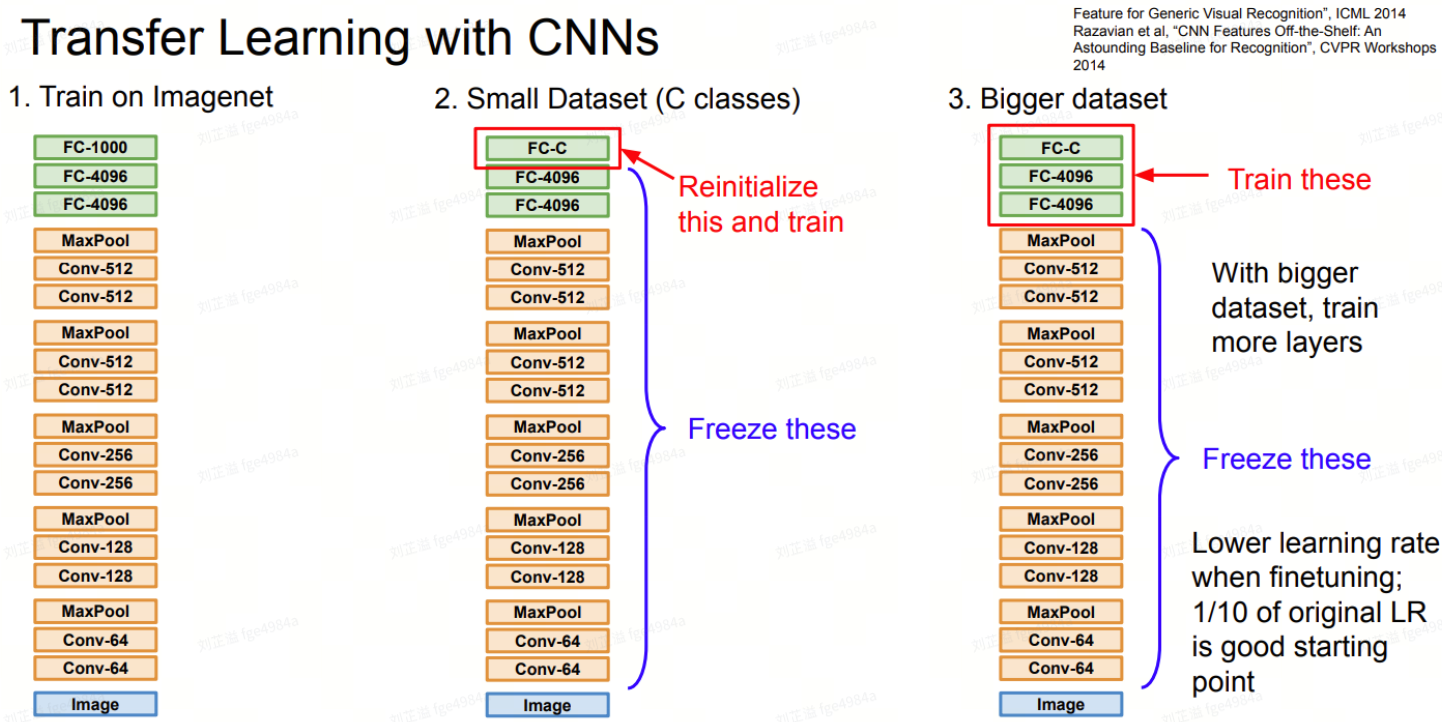
ResNet

- 使用残差块来连接很深的网络



迁移学习

- 先冻住除开连接层外的所有层；训练连接层的参数
- 接着训练更上面的层；

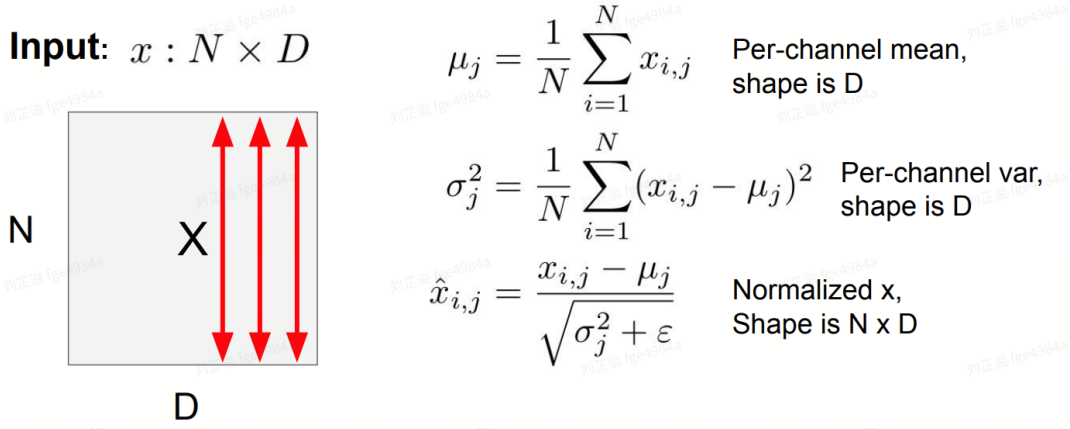


Normalization

- 解决的问题：在机器学习中，协变量可以看作是输入。一般的机器学习算法都要求输入在训练集和测试集上的分布是相似的。如果不满足这个假设，在训练集上学习到的模型在测试集上的表现会比较差。**解决内部协变量偏移问题，就要使得每一个神经层的输入的分布在训练过程中保持一致**

Batch Normalization

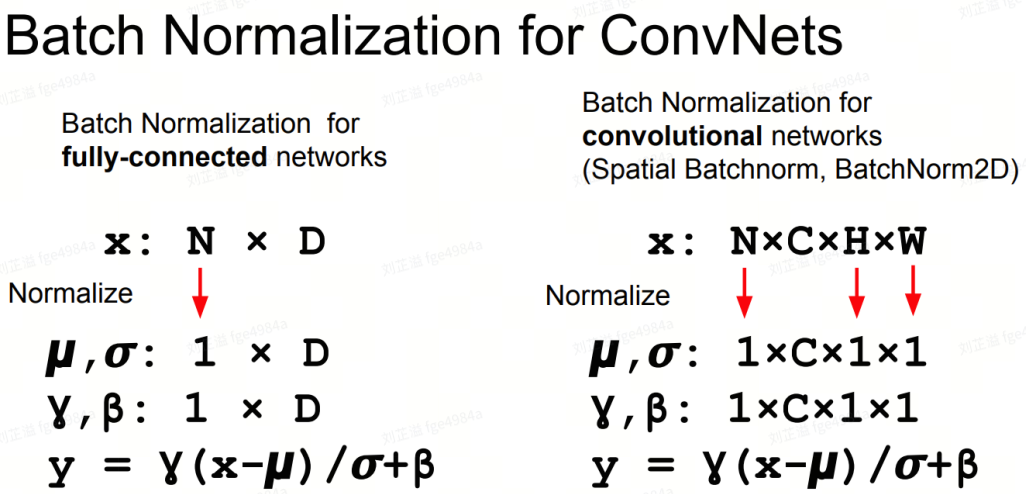
- 考虑某些层的一个批次的激活
- 使得某一维度的均方差为0；**不同训练数据对同一个神经元的归一化**



- 通常放在**全连接层或卷积层的后面；非线性层的前面**

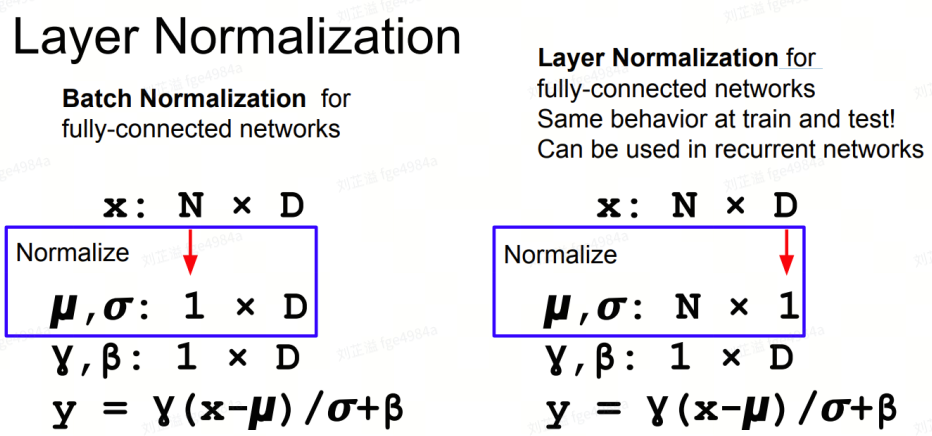
注意：BN层在训练和测试阶段表现不同，可能出现BUG

BN层使用



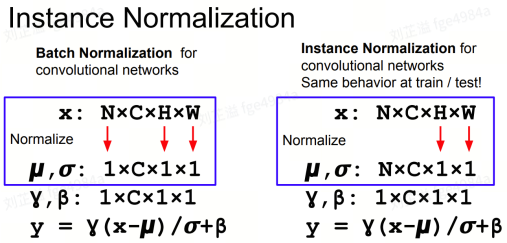
Layer Normalization

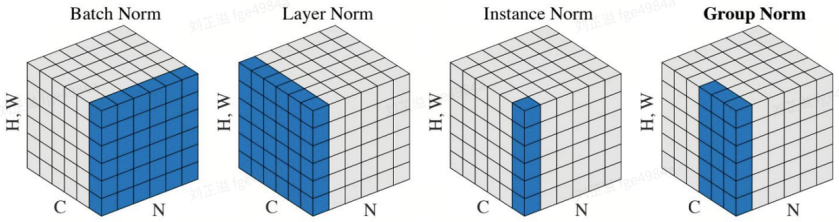
- **单个训练数据对某一层所有神经元之间的归一化**
- 对同一样本的不同特征做归一化



Instance Normalization

- **对输入的特征进行归一化**





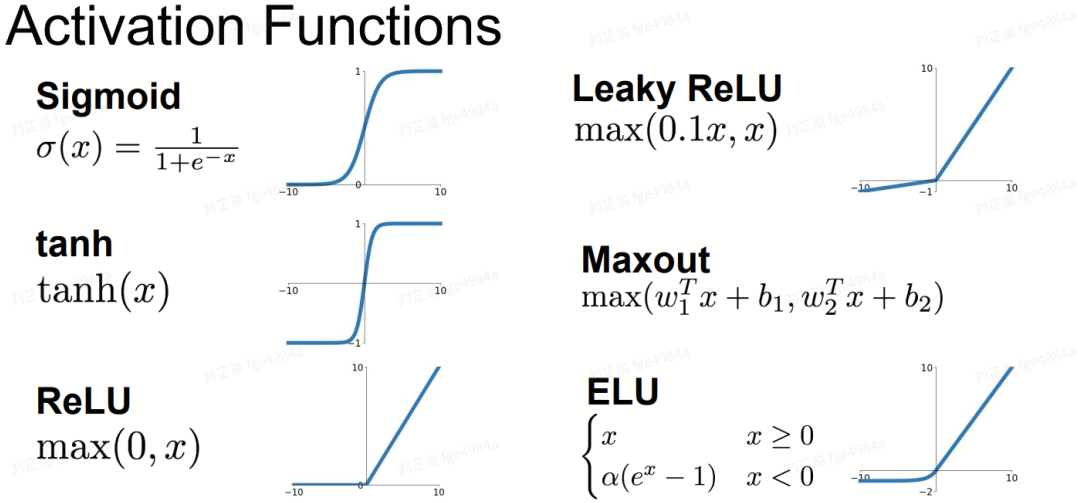
Training Neural Work

Mini-batch SGD

Loop:

1. **Sample** a batch of data
2. **Forward** prop it through the graph (network), get loss
3. **Backprop** to calculate the gradients
4. **Update** the parameters using the gradient

Activation Functions



Sigmoid

- 饱和神经元可能造成梯度消失
- sigmoid的输出不是0为中心的
- exp()的计算代价过高

Tanh

- 0为中心
- 梯度消失问题

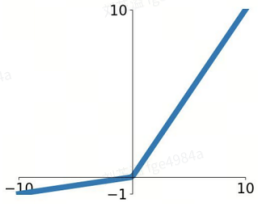
ReLU

- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice
- 不是以0为中心的输出
- 梯度小于0时的处理
  - initialize ReLU neurons with slightly positive biases

Leaky ReLU

Activation Functions

[Maas et al., 2013]
[He et al., 2015]



Leaky ReLU

$$f(x) = \max(0.01x, x)$$

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not "die".**

Parametric Rectifier (PReLU)

$$f(x) = \max(\alpha x, x)$$

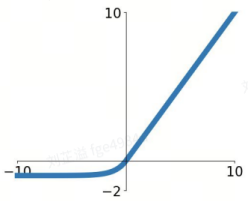
backprop into α (parameter)

ELU

Activation Functions

[Clevert et al., 2015]

Exponential Linear Units (ELU)



$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$

(Alpha default = 1)

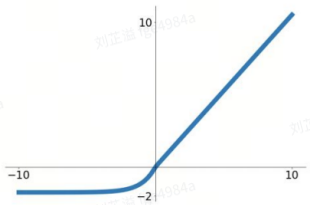
- All benefits of ReLU
- Closer to zero mean outputs
- Negative saturation regime compared with Leaky ReLU adds some robustness to noise
- Computation requires exp()

SELU

Activation Functions

[Klambauer et al. ICLR 2017]

Scaled Exponential Linear Units (SELU)



- Scaled version of ELU that works better for deep networks
- "Self-normalizing" property;
- Can train deep SELU networks without BatchNorm

$$f(x) = \begin{cases} \lambda x & \text{if } x > 0 \\ \lambda\alpha(e^x - 1) & \text{otherwise} \end{cases}$$

$$\alpha = 1.6732632423543772848170429916717$$

$$\lambda = 1.0507009873554804934193349852946$$

Maxout

Maxout "Neuron"

[Goodfellow et al., 2013]

- Does not have the basic form of dot product -> nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

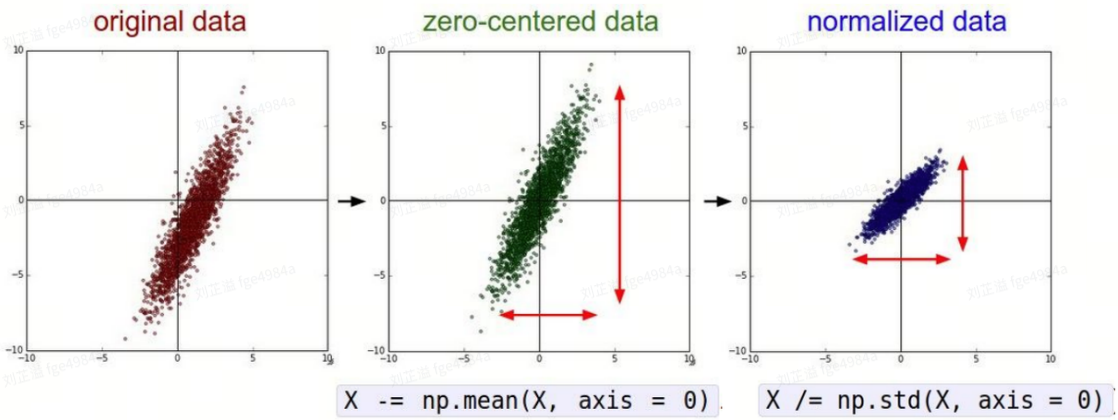
Problem: doubles the number of parameters/neuron :(

In practice

- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU / Maxout / ELU / SELU
  - To squeeze out some marginal gains
- Don't use sigmoid or tanh

Data Preprocessing

Data Preprocessing



In practice for Images

- Just center only

TLDR: In practice for Images: center only

e.g. consider CIFAR-10 example with [32,32,3] images

- Subtract the mean image (e.g. AlexNet)
  (mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet)
  (mean along each channel = 3 numbers)
- Subtract per-channel mean and
  Divide by per-channel std (e.g. ResNet)
  (mean along each channel = 3 numbers)

Not common to do PCA or whitening

Weight Initialization

- Small random numbers

(gaussian with zero mean and 1e-2 standard deviation)

```
W = 0.01 * np.random.randn(Din, Dout)
```

Weight Initialization: "Xavier" Initialization

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

"Xavier" initialization: std = 1/sqrt(Din)

"Just right": Activations are nicely scaled for all layers!

For conv layers, Din is filter_size^2 * input_channels

Let: $y = x_1w_1+x_2w_2+...+x_{Din}w_{Din}$

Assume: $Var(x_1) = Var(x_2) = ... = Var(x_{Din})$

We want: $Var(y) = Var(x_i)$

$Var(y) = Var(x_1w_1+x_2w_2+...+x_{Din}w_{Din})$
$= Din\ Var(x_iw_i)$
$= Din\ Var(x_i)\ Var(w_i)$
[Assume all x_i, w_i are iid]

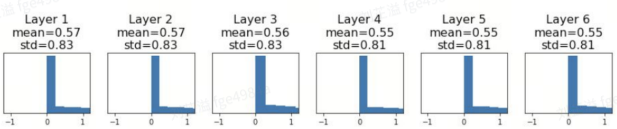
So, $Var(y) = Var(x_i)$ only when $Var(w_i) = 1/Din$

Weight Initialization: Kaiming / MSRA Initialization

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) * np.sqrt(2/Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

ReLU correction: std = sqrt(2 / Din)

"Just right": Activations are nicely scaled for all layers!



He et al, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification", ICCV 2015

Regularization

- 处理神经网络训练过拟合的情况

Add term to loss

$$L = \frac{1}{N}\sum_{i=1}^{N}\sum_{j\neq y_i}\max(0, f(x_i;W)_j - f(x_i;W)_{y_i} + 1) + \boxed{\lambda R(W)}$$

In common use:

L2 regularization	$R(W) = \sum_k \sum_l W_{k,l}^2$ (Weight decay)
L1 regularization	$R(W) = \sum_k \sum_l \lvert W_{k,l}\rvert$
Elastic net (L1 + L2)	$R(W) = \sum_k \sum_l \beta W_{k,l}^2 + \lvert W_{k,l}\rvert$

Dropout

- 随机使得一些层的神经元失效
- 防止特征的共同适应

More common: “Inverted dropout”

```python
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

test time is unchanged!

A common pattern

- 训练时增加随机性
- 测试时平均随机性

Training: Add some kind of randomness

$$y = f_W(x, z)$$

Testing: Average out randomness (sometimes approximate)

$$y = f(x) = E_z\big[f(x, z)\big] = \int p(z) f(x, z) dz$$

数据增强

- 水平翻转
- 随机位置和大小
- 颜色抖动

DropConnect

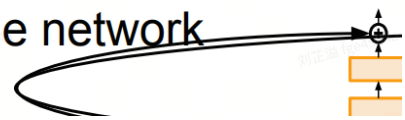
Regularization: DropConnect
Training: Drop connections between neurons (set weights to 0)
Testing: Use all the connections

Fractional Pooling

Regularization: Fractional Pooling
Training: Use randomized pooling regions
Testing: Average predictions from several regions

随机深度模型

Regularization: Stochastic Depth
Training: Skip some layers in the network
Testing: Use all the layer



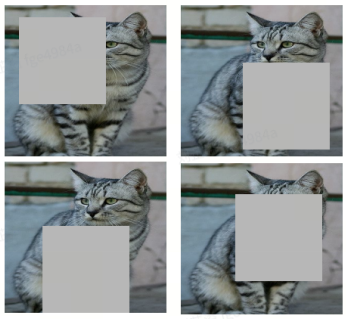
Cutout

Regularization: Cutout

Training: Set random image regions to zero
Testing: Use full image

Examples:
Dropout
Batch Normalization
Data Augmentation
DropConnect
Fractional Max Pooling
Stochastic Depth
Cutout / Random Crop

DeVries and Taylor, "Improved Regularization of Convolutional Neural Networks with Cutout", arXiv 2017

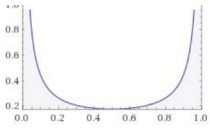


Works very well for small datasets like CIFAR, less common for large datasets like ImageNet

Mixup

Regularization: Mixup

Training: Train on random blends of images
Testing: Use original images



Examples:
Dropout
Batch Normalization
Data Augmentation
DropConnect
Fractional Max Pooling
Stochastic Depth
Cutout / Random Crop
Mixup



CNN

Target label:
cat: 0.4
dog: 0.6

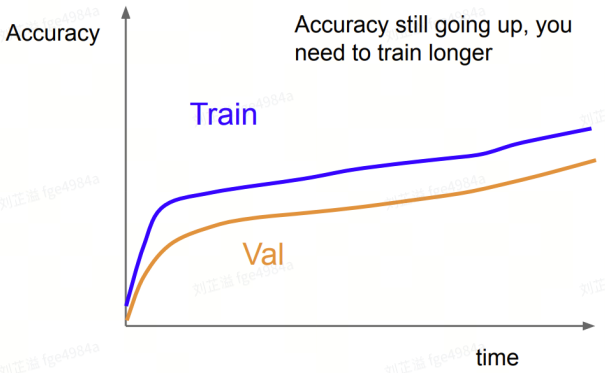
Randomly blend the pixels of pairs of training images, e.g. 40% cat, 60% dog

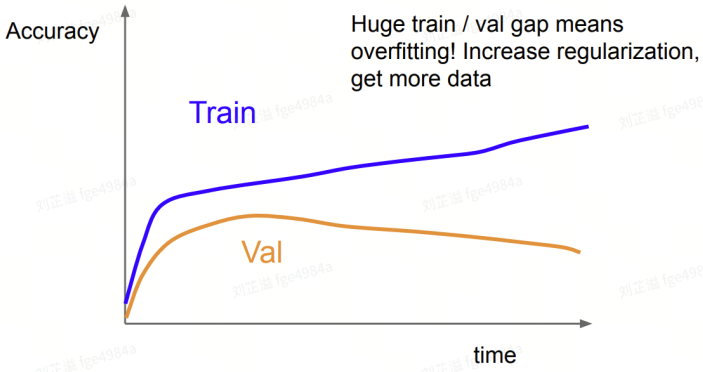
In practice

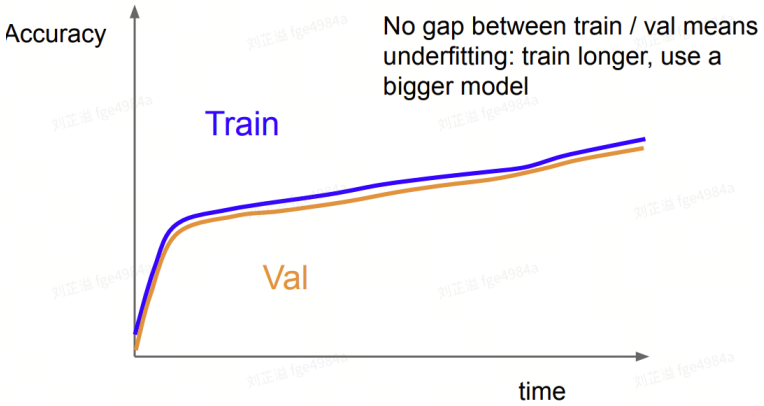
- 大型全连接层，考虑dropout
- BN和数据增强
- 尝试对小数据集进行cutout和mixup

选择超参数

- 检查初始的Loss
- 过拟合少量样本
  - Loss not going down? LR too low, bad initialization
  - Loss explodes to Inf or NaN? LR too high, bad initialization
- 找到使得Loss下降的学习率：**Good learning rates to try: 1e-1, 1e-2, 1e-3, 1e-4**
- 粗略训练1~5轮：**Good weight decay to try: 1e-4, 1e-5, 0**
- 细化网格训练，训练更长时间
- 查看损失和准确率曲线







Summary

- Improve your training error:
  - Optimizers
  - Learning rate schedules
- Improve your test error:
  - Regularization
  - Choosing Hyperparameters

- Activation Functions (use ReLU)
- Data Preprocessing (images: subtract mean)
- Weight Initialization (use Xavier/Kaiming init)
- Batch Normalization (use this!)
- Transfer learning (use this if you can!)

训练前馈神经网络

Training “Feedforward” Neural Networks

1. **One time set up**: activation functions, preprocessing, weight initialization, regularization, gradient checking

2. **Training dynamics**: babysitting the learning process, parameter updates, hyperparameter optimization

3. **Evaluation**: model ensembles, test-time augmentation, transfer learning