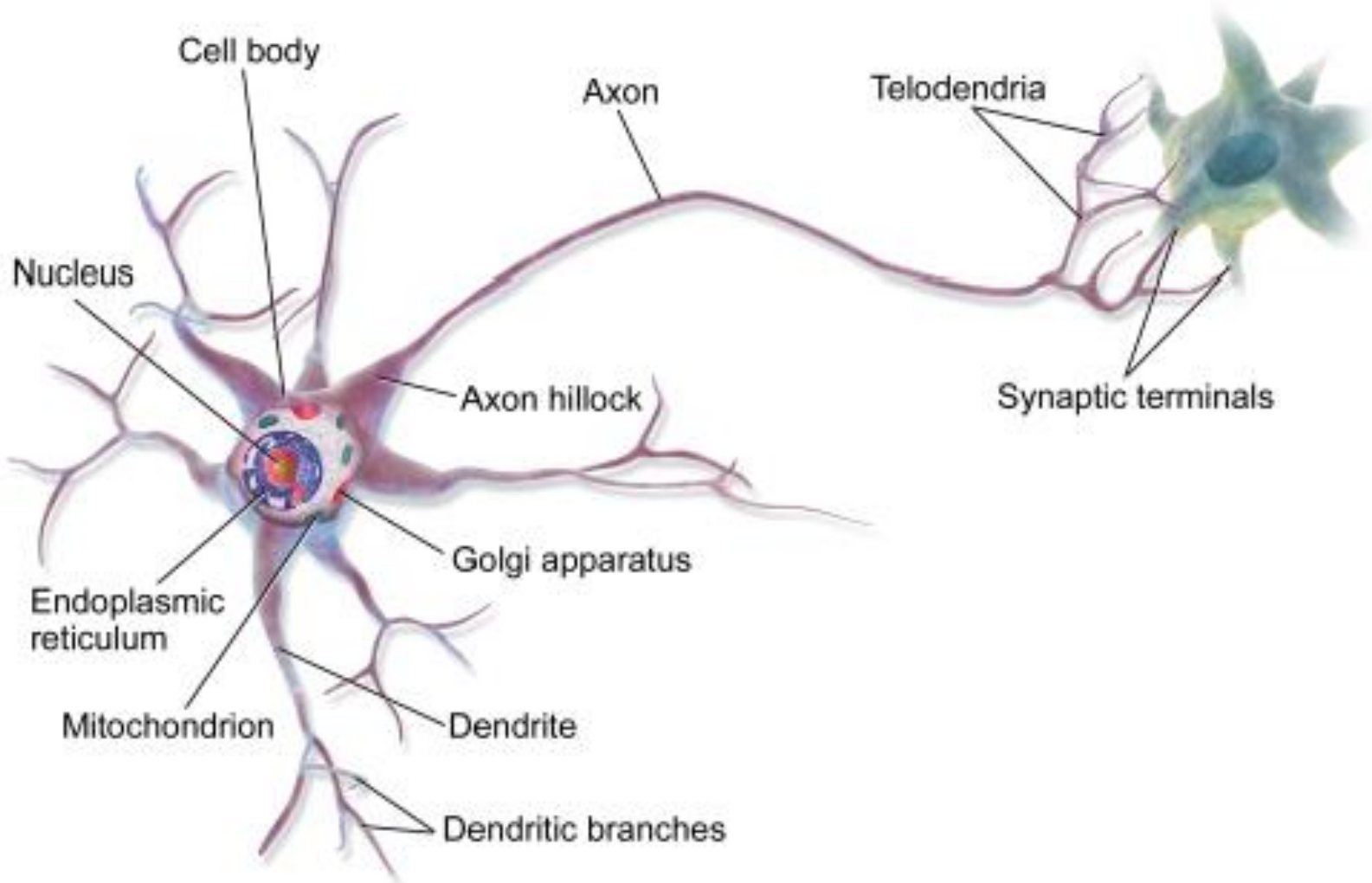
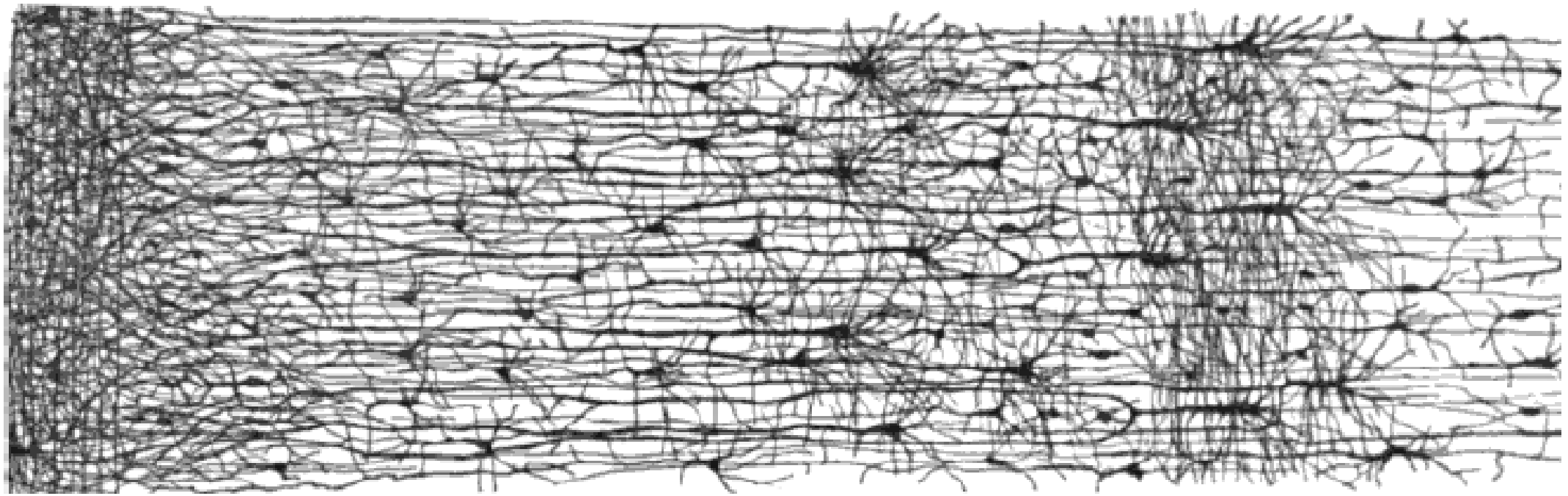
TensorFlow热恋

Yasaka 陈博

神经元

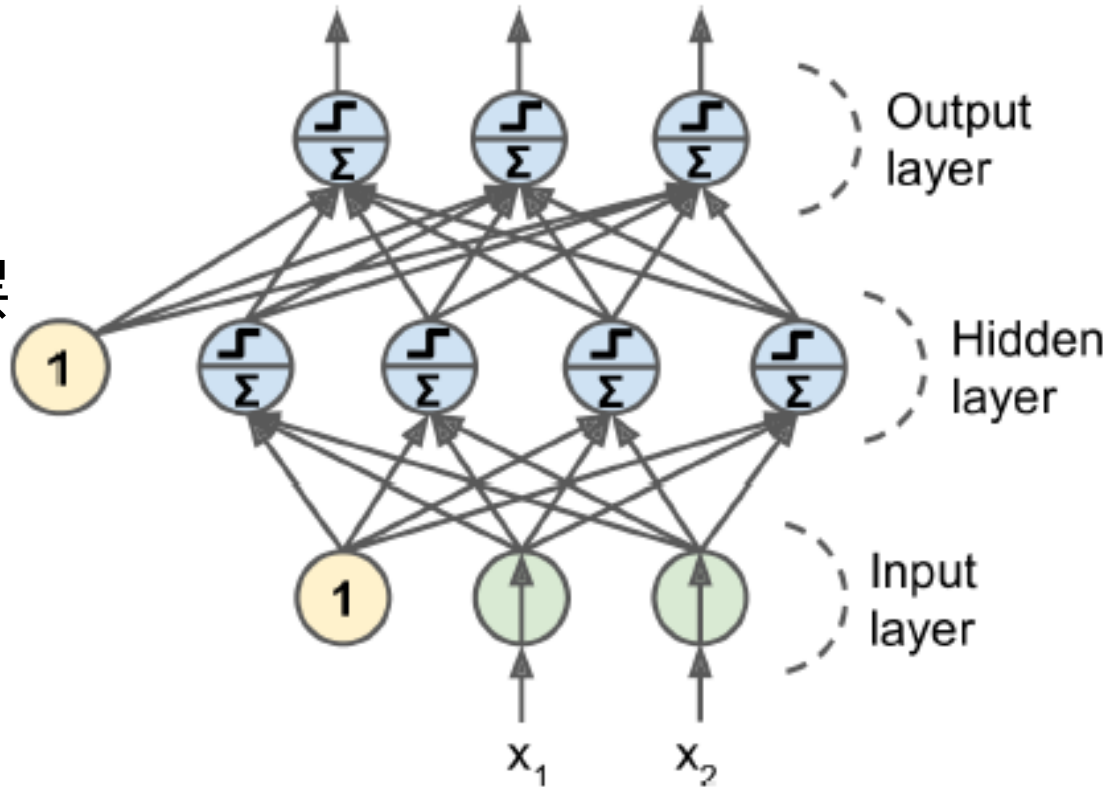

Cell body
Axon
Telodendria
Nucleus
Axon hillock
Synaptic terminals
Golgi apparatus
Endoplasmic reticulum
Mitochondrion
Dendrite
Dendritic branches

多层神经元



Multi-Layer Perceptron

- 多层感知机
- 第一层输入层，最后输出层，中间隐藏层
- 除了输出层，每层神经元包括bias都是
- 全连接到下一层
- ANN人工神经网络有两个或两个以上隐藏层
- 深度神经网络

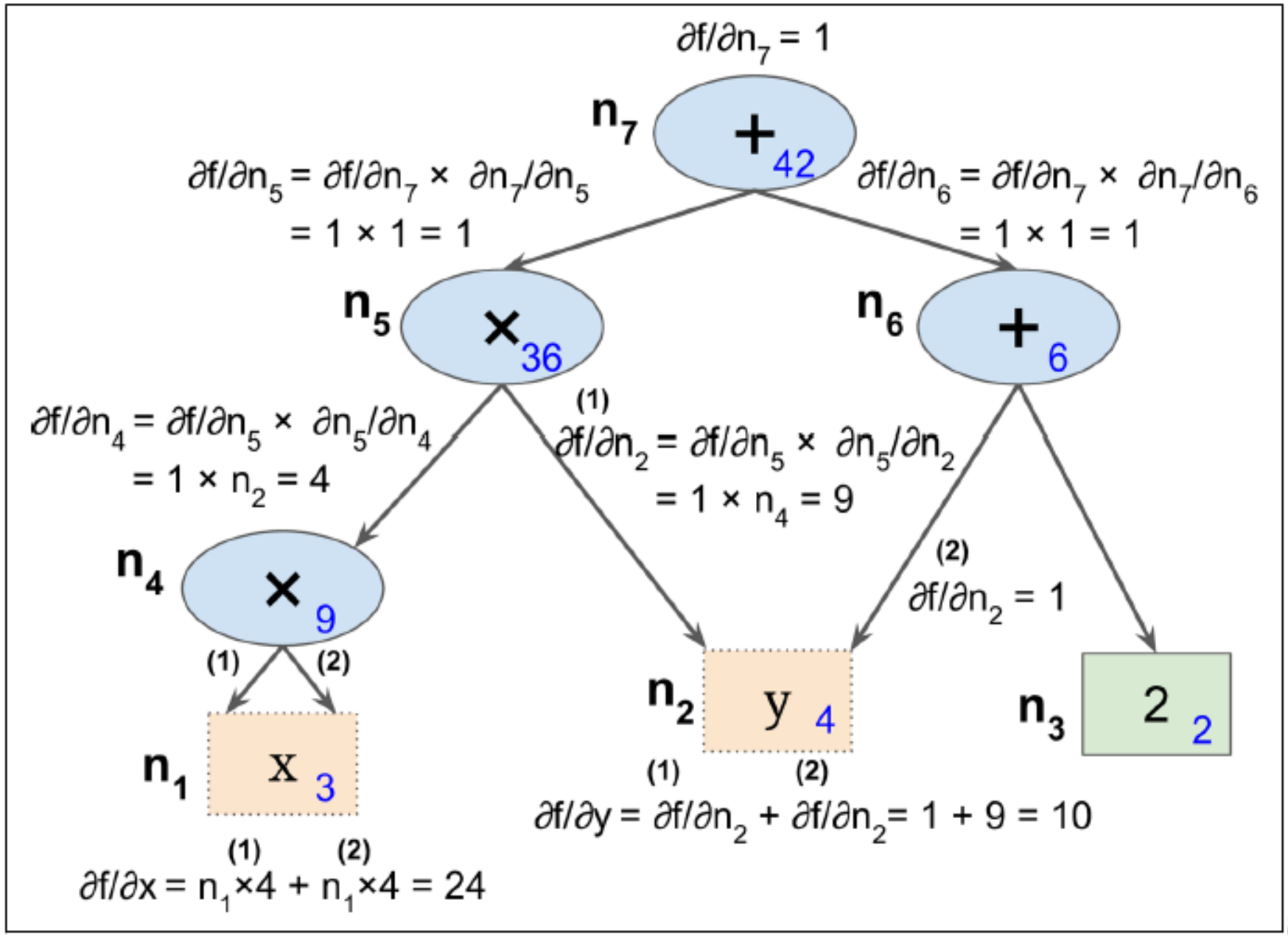


Backpropagation

- 反向传播，就是梯度下降使用reverse-mode autodiff
- 前向传播，就是make predictions，然后计算输出误差，然后计算出每个神经元节点对误差的贡献
- 求贡献就是反向传播是根据前向传播的误差来求梯度
- 然后根据贡献调整原来的权重

Reverse-mode Autodiff

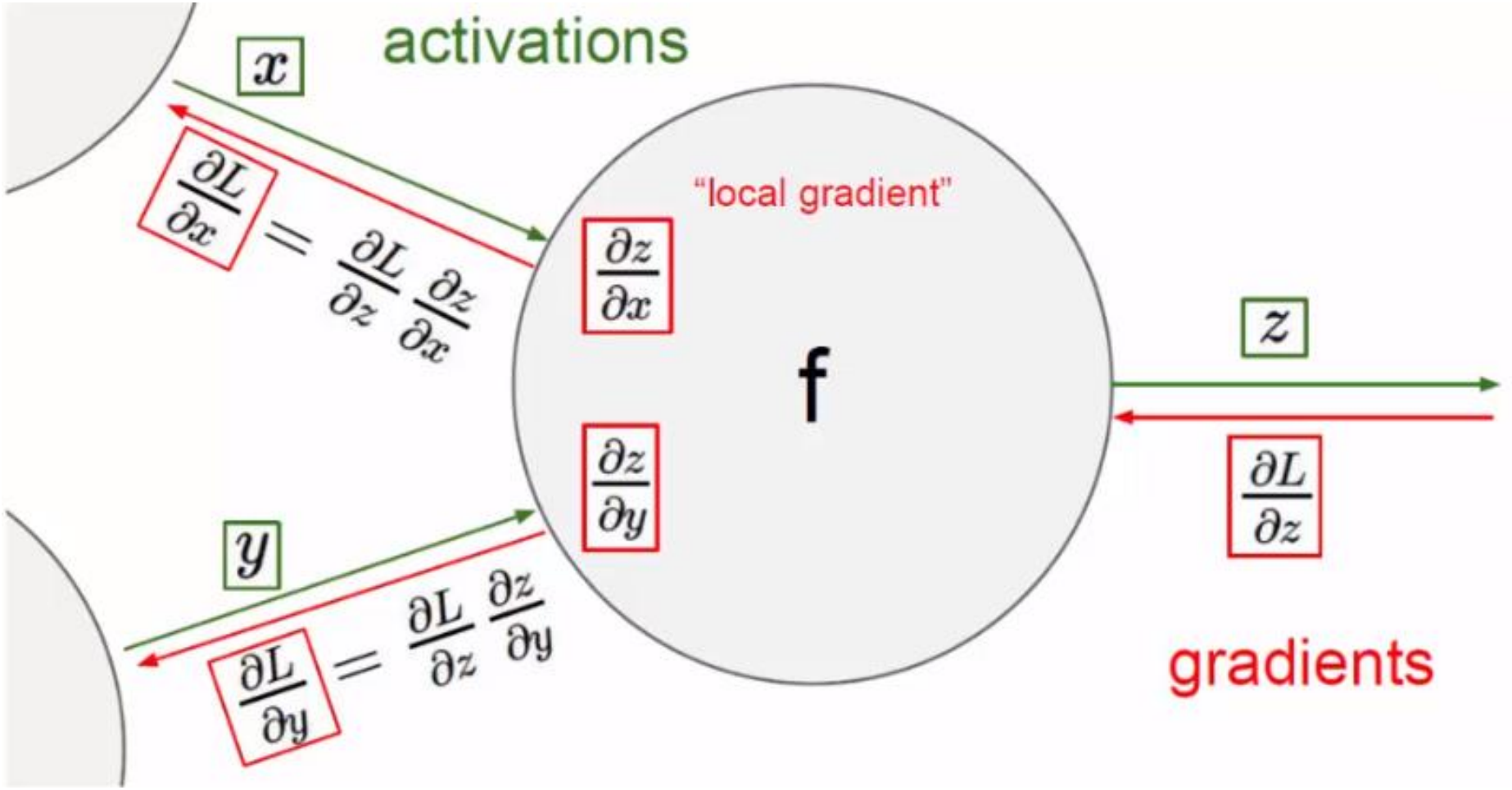
- 反向自动求导是TensorFlow实现的方案，首先，它执行图的前向阶段，从输入到输出，去计算节点值，然后是反向阶段，从输出到输入去计算所有的偏导。
- 下面的图是第二个阶段，在第一个阶段中，从x=3和y=4开始去计算所有的节点值
- f(x,y)=$x^2 * y + y + 2$
- 求解的想法是逐渐的从图上往下计算f(x,y)的偏导，使用每一个连续的节点变量节点，严重依赖链式求导法则！ $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial n_i} \times \frac{\partial n_i}{\partial x}$


∂f/∂n7 = 1
n7
+ 42
∂f/∂n5 = ∂f/∂n7 × ∂n7/∂n5
= 1 × 1 = 1
∂f/∂n6 = ∂f/∂n7 × ∂n7/∂n6
= 1 × 1 = 1
n5
× 36
n6
+ 6
∂f/∂n4 = ∂f/∂n5 × ∂n5/∂n4
= 1 × n2 = 4
(1)
∂f/∂n2 = ∂f/∂n5 × ∂n5/∂n2
= 1 × n4 = 9
n4
× 9
(2)
∂f/∂n2 = 1
(1)
(2)
n2
y 4
n3
2 2
n1
x 3
(1)
(2)
(1)
(2)
∂f/∂y = ∂f/∂n2 + ∂f/∂n2 = 1 + 9 = 10
∂f/∂x = n1×4 + n1×4 = 24

Reverse-mode Autodiff

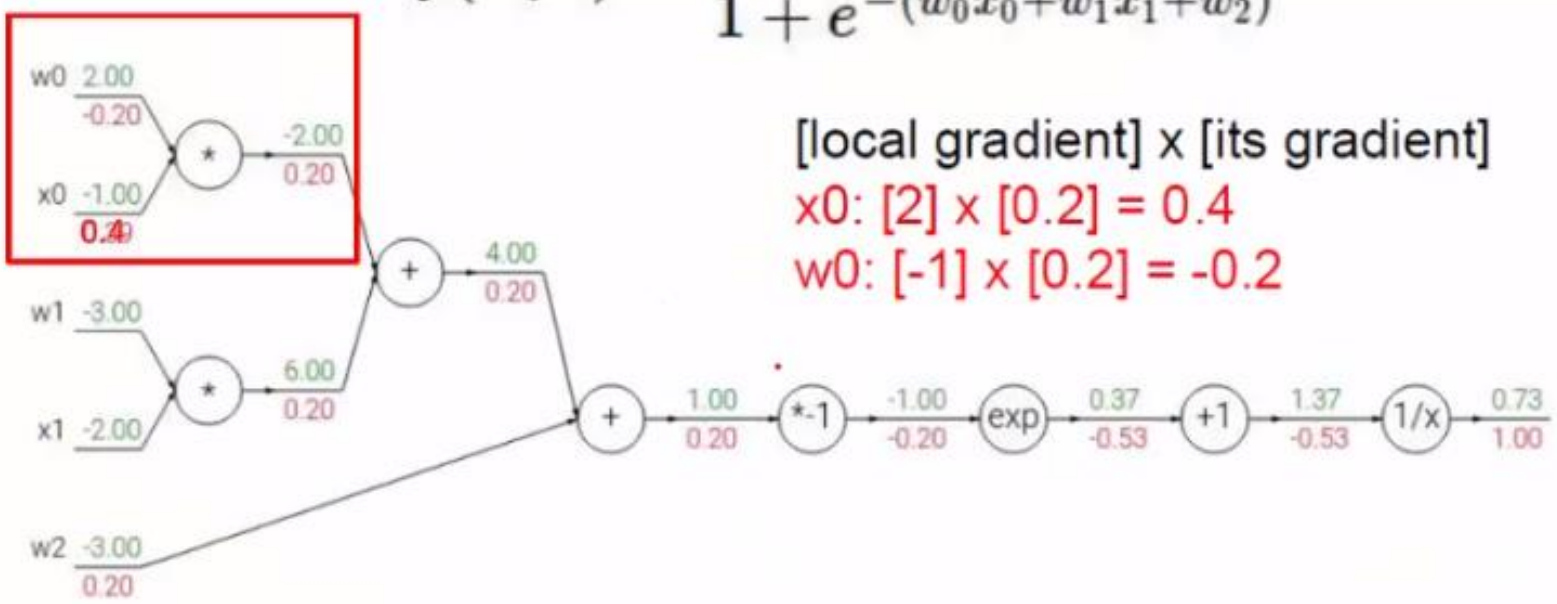
- 因为n7是输出节点，所以f=n7，所以$\frac{\partial \mathrm{f}}{\partial n7} = 1$
- 让我们继续往下走到n5节点，$\frac{\partial \mathrm{f}}{\partial n5} = \frac{\partial \mathrm{f}}{\partial n7} * \frac{\partial n7}{\partial n5}$. 我们已知$\frac{\partial \mathrm{f}}{\partial n7} = 1$，所以我们需要知道$\frac{\partial n7}{\partial n5}$，因为n7=n5+n6，所以我们求得$\frac{\partial n7}{\partial n5}$ =1，所以$\frac{\partial \mathrm{f}}{\partial n5}$ = 1*1=1
- 现在我们继续走到节点n4，$\frac{\partial \mathrm{f}}{\partial n4} = \frac{\partial \mathrm{f}}{\partial n5} * \frac{\partial n5}{\partial n4}$，因为n5=n4*n2，我们求得$\frac{\partial n5}{\partial n4}$ = n2，$\frac{\partial \mathrm{f}}{\partial n4}$ = 1*4
- 沿着图一路向下，我们可以计算出所有节点，就能计算出

Backpropagation



Backpropagation

$$f(w,x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$



[local gradient] x [its gradient]
x0: [2] x [0.2] = 0.4
w0: [-1] x [0.2] = -0.2

$$f(x) = e^x \rightarrow \frac{df}{dx} = e^x$$

$$f_a(x) = ax \rightarrow \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \rightarrow \frac{df}{dx} = -1/x^2$$

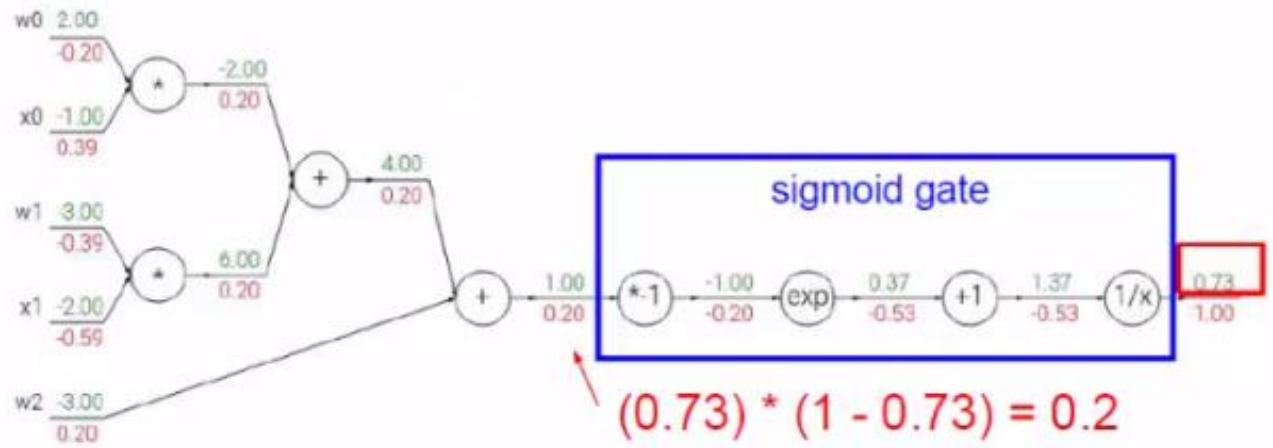
$$f_c(x) = c + x \rightarrow \frac{df}{dx} = 1$$

整体Backpropagation

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

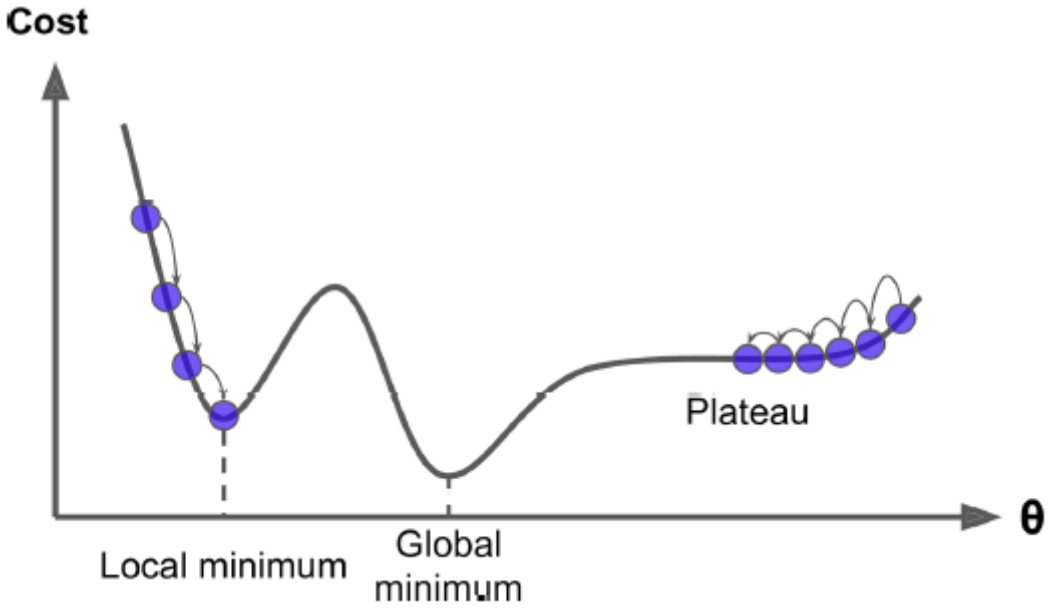
$$\sigma(x) = \frac{1}{1+e^{-x}}$$ sigmoid function

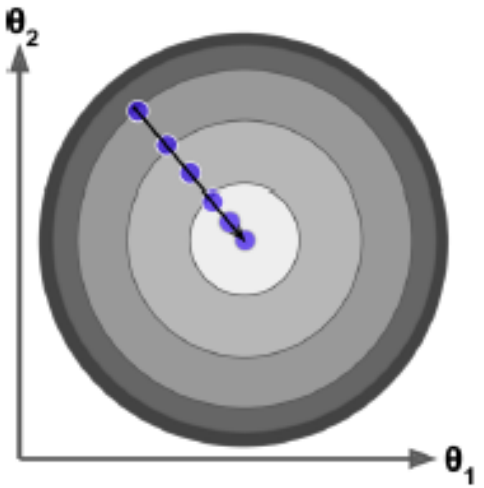
$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) = (1-\sigma(x))\,\sigma(x)$$

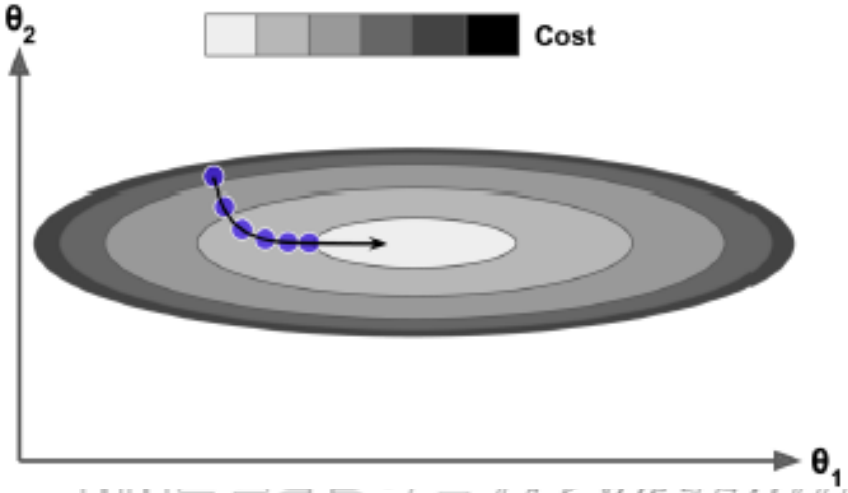


Momentum Optimization

- 下降速度会比普通梯度下降快10倍
- `optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate, momentum=0.9)`
- 普通梯度下降速度就会比较慢
- 而且用了momentum=0.9还可以更容易跳出局部最优解







调优神经网络超参数

- 神经网络有着灵活性，同时这也是算法的主要缺点：需要有许多超参数需要去调节
- 层数、每层神经元数、在每层使用的激活函数、初始化权重的逻辑 等等
- 你怎么知道哪种组合最适合你的任务?
- Grid search
- Cross validation
- 但是需要时间

调优神经网络超参数

- 隐藏层数
- 对于许多问题，你可以开始只用一个隐藏层，就可以获得不错的结果，比如对于复杂的问题我们可以在隐藏层上使用足够多的神经元就行了， 很长一段时间人们满足了就没有去探索深度神经网络
- 但是深度神经网络有更高的参数效率，神经元个数可以指数倍减少，并且训练起来也更快！
- 就好像直接画一个森林会很慢，但是如果画了树枝，复制粘贴树枝成大树，再复制粘贴大树成森林却很快。真实的世界通常是这种层级的结构，DNN就是利用这种优势
- 前面的隐藏层构建低级的结构，组成各种各样形状和方向的线，中间的隐藏层组合低级的结构，譬如方块、圆形，后面的隐藏层和输出层组成更高级的结构，比如面部
- 不仅这种层级的结构帮助DNN收敛更快，同时增加了复用能力到新的数据集，例如，如果你已经训练了一个神经网络去识别面部，你现在想训练一个新的网络去识别发型，你可以复用前面的几层，就是不去随机初始化Weights和biases，你可以把第一个网络里面前面几层的权重值赋给新的网络作为初始化，然后开始训练
- 这样网络不必从原始训练低层网络结构，它只需要训练高层结构，例如，发型
- 对于很多问题，一个到两个隐藏层就是够用的了，MNIST可以达到97%当使用一个隐藏层上百个神经元，达到98%使用两个隐藏层，对于更复杂的问题，你可以逐渐增加隐藏层，直到对于训练集过拟合。
- 非常复杂的任务譬如图像分类和语音识别，需要几十层甚至上百层，但不全是全连接，并且它们需要大量的数据，不过，你很少需要从头训练，非常方便的是复用一些提前训练好的类似业务的经典的网络。那样训练会快很多并且需要不太多的数据

调优神经网络超参数

- 每个隐藏层的神经元个数
- 输入层和输出层的神经元个数很容易确定，根据需求，比如MNIST输入层28*28=784，输出层10
- 通常的做法是每个隐藏层的神经元越来越少，比如第一个隐藏层300个神经元，第二个隐藏层100个神经元，可是，现在更多的是每个隐藏层神经元数量一样，比如都是150个，这样超参数需要调节的就少了，正如前面寻找隐藏层数量一样，可以逐渐增加数量直到过拟合，找到完美的数量更多还是黑科技
- 简单的方式是选择比你实际需要更多的层数和神经元个数，然后使用early stopping去防止过拟合，还有L1、L2正则化技术，还有dropout

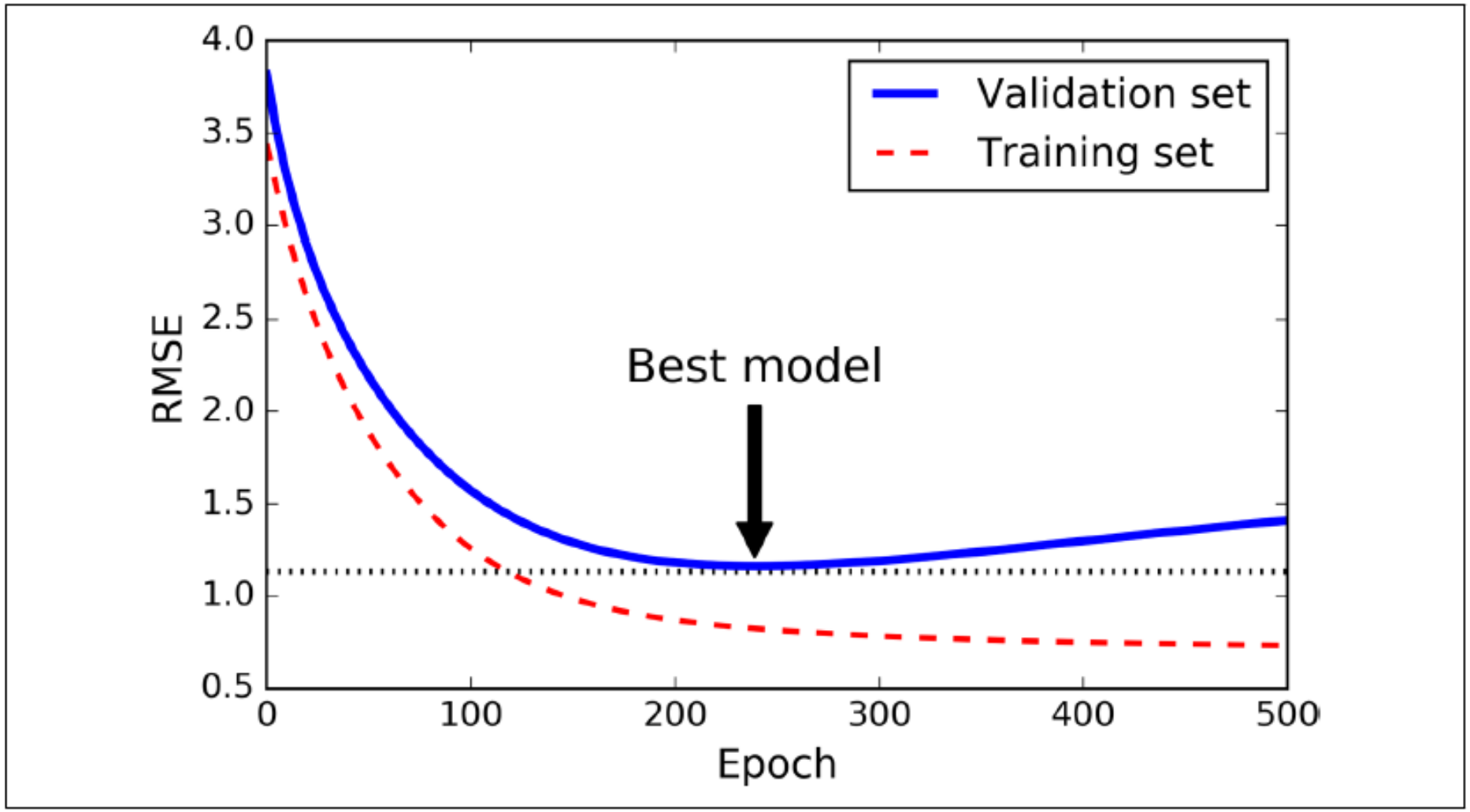
防止过拟合通过Regularization

- 典型的深度学习有成百上千的参数，有时甚至上百万，由于这么多的超参数，网络有不可想象的自由度可以适应大量各种复杂的数据集。但是这个灵活性同时意味着倾向训练集过拟合

Early Stopping

- 去防止在训练集上面过拟合，一个很好的手段是early stopping，当在验证集上面开始下降的时候中断训练
- 一种方式使用TensorFlow去实现，是间隔的比如每50 steps，在验证集上去评估模型，然后保存一下快照如果输出性能优于前面的快照，记住最后一次保存快照时候迭代的steps的数量，当到达step的limit次数的时候，restore最后一次胜出的快照
- 尽管early stopping实际工作做不错，你还是可以得到更好的性能当结合其他正则化技术一起的话

Early Stopping



Early Stopping

```
sgd_reg = SGDRegressor(n_iter=1, warm_start=True, penalty=None,
                       learning_rate="constant", eta0=0.0005)

minimum_val_error = float("inf")
best_epoch = None
best_model = None
for epoch in range(1000):
    sgd_reg.fit(X_train_poly_scaled, y_train)  # continues where it left off
    y_val_predict = sgd_reg.predict(X_val_poly_scaled)
    val_error = mean_squared_error(y_val_predict, y_val)
    if val_error < minimum_val_error:
        minimum_val_error = val_error
        best_epoch = epoch
        best_model = clone(sgd_reg)
```

L1 L2 正则

- 使用L1和L2正则去限制神经网络连接的weights权重
- 一种方式去使用TensorFlow做正则是加合适的正则项到损失函数

```
[...] # construct the neural network
base_loss = tf.reduce_mean(xentropy, name="avg_xentropy")
reg_losses = tf.reduce_sum(tf.abs(weights1)) + tf.reduce_sum(tf.abs(weights2))
loss = tf.add(base_loss, scale * reg_losses, name="loss")
```

L1 L2 正则

- 可是如果有很多层，上面的方式不是很方便，幸运的是，TensorFlow提供了更好的选择，很多函数比如get_variable()或者fully_connected()接受一个 *_regularizer参数，可以传递任何以weights为参数，返回对应正则化损失的函数，l1_regularizer()，l2_regularizer()和l1_l2_regularizer()函数返回这个的函数

```
with arg_scope(
        [fully_connected],
        weights_regularizer=tf.contrib.layers.l1_regularizer(scale=0.01)):
    hidden1 = fully_connected(X, n_hidden1, scope="hidden1")
    hidden2 = fully_connected(hidden1, n_hidden2, scope="hidden2")
    logits = fully_connected(hidden2, n_outputs, activation_fn=None,scope="out")
```

L1 L2 正则

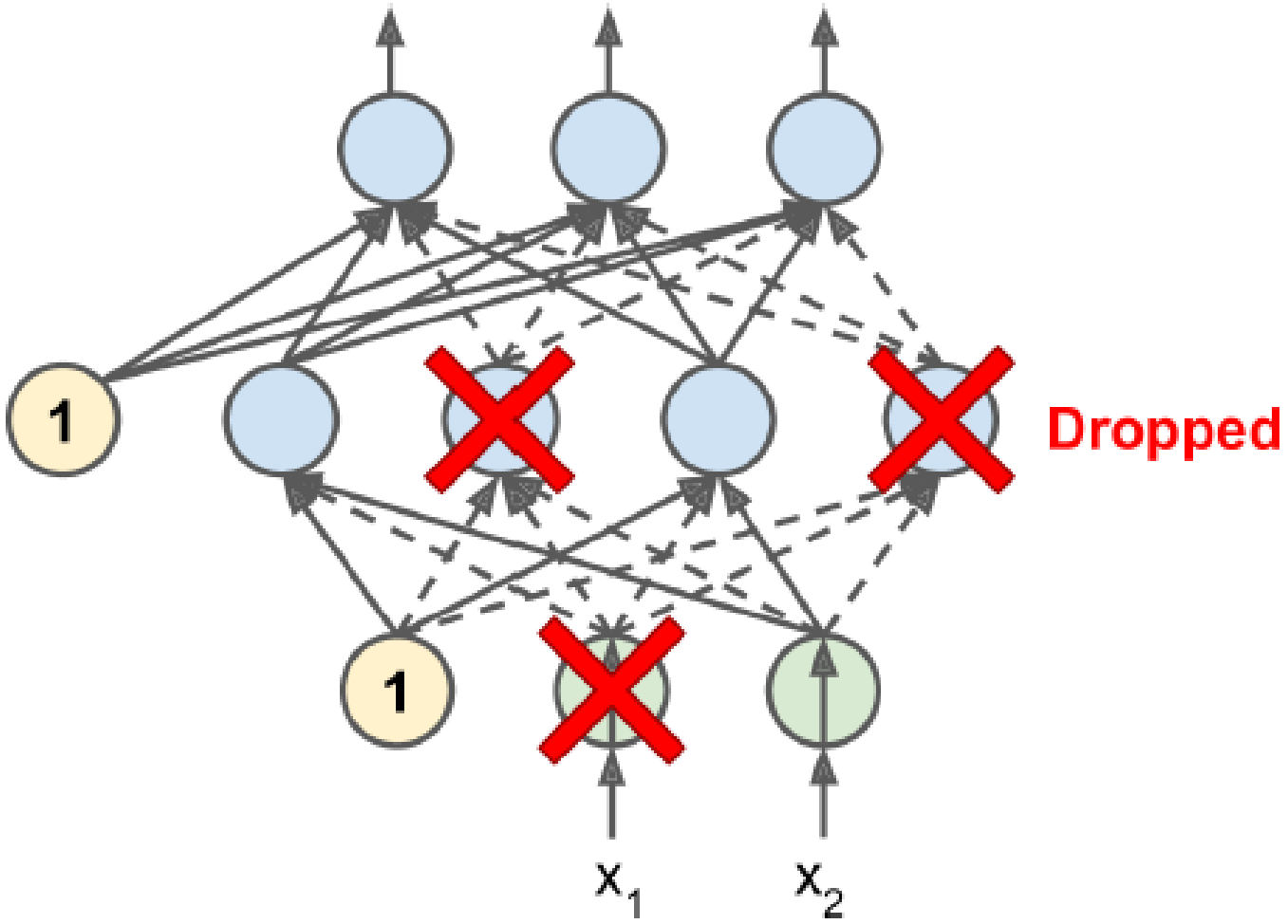
- 上面的代码神经网络有两个隐藏层，一个输出层，同时在图里创建节点给每一层的权重去计算L1正则损失，TensorFlow自动添加这些节点到一个特殊的包含所有正则化损失的集合。你只需要添加这些正则化损失到整体的损失中，不要忘了去添加正则化损失到整体的损失中，否则它们将会被忽略

```
reg_losses = tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)
loss = tf.add_n([base_loss] + reg_losses, name="loss")
```

Dropout

- 在深度学习中，最流行的正则化技术，它被证明非常成功，即使在顶尖水准的神经网络中也可以带来1%到2%的准确度提升，这可能乍听起来不是特别多，但是如果模型已经有了95%的准确率，获得2%的准确率提升意味着降低错误率大概40%，即从5%的错误率降低到3%的错误率！！！
- 在每一次训练step中，每个神经元，包括输入神经元，但是不包括输出神经元，有一个概率被临时的丢掉，意味着它将被忽视在整个这次训练step中，但是有可能下次再被激活

Dropout



Dropout

- keep_prob是保留下来的比例，1-keep_prob是dropout rate
- 当训练的时候，把is_training设置为True，测试的时候，设置为False

```
from tensorflow.contrib.layers import dropout

[...]
is_training = tf.placeholder(tf.bool, shape=(), name='is_training')

keep_prob = 0.5
X_drop = dropout(X, keep_prob, is_training=is_training)

hidden1 = fully_connected(X_drop, n_hidden1, scope="hidden1")
hidden1_drop = dropout(hidden1, keep_prob, is_training=is_training)

hidden2 = fully_connected(hidden1_drop, n_hidden2, scope="hidden2")
hidden2_drop = dropout(hidden2, keep_prob, is_training=is_training)

logits = fully_connected(hidden2_drop, n_outputs, activation_fn=None,
                         scope="outputs")
```

调优神经网络超参数

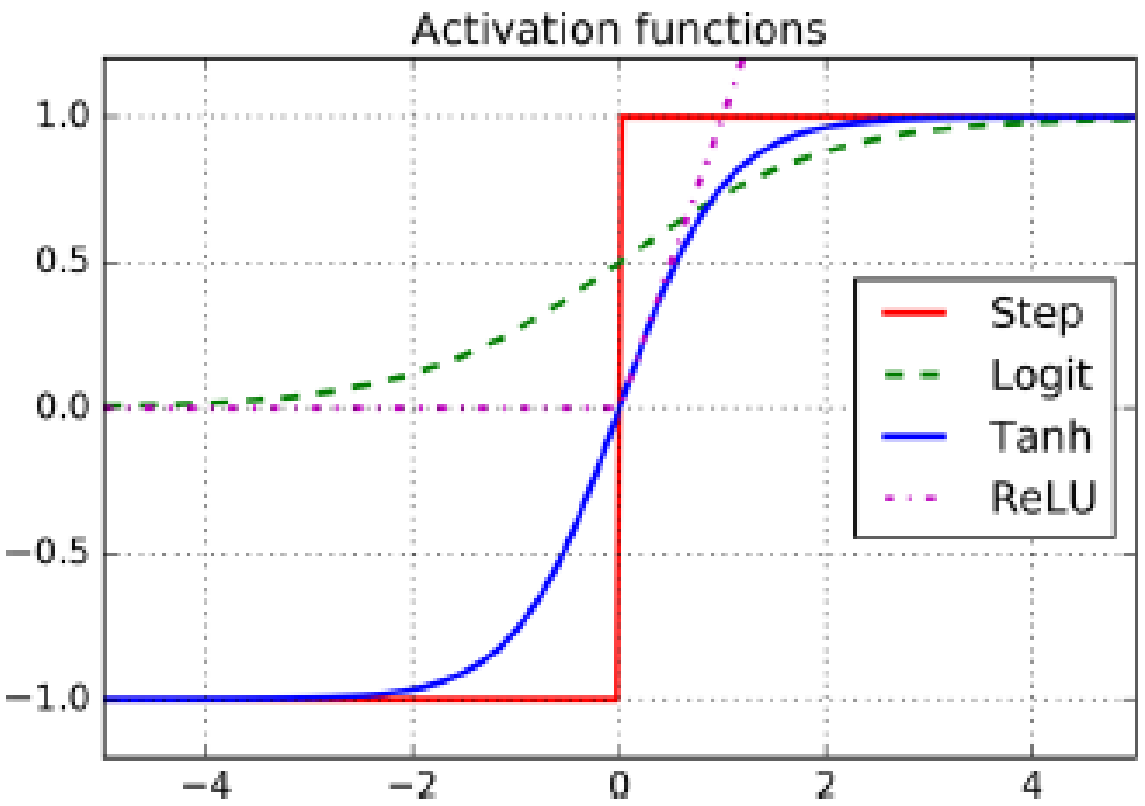
- 激活函数
- 大多数情况下激活函数使用ReLU激活函数，这种激活函数计算更快，并且梯度下降不会卡在plateaus，并且对于大的输入值，它不会饱和，相反对比logistic function和hyperbolic tangent function，将会饱和在1
- 对于输出层，softmax激活函数通常是一个好的选择对于分类任务，因为类别和类别之间是互相排斥的，对于回归任务，根本不使用激活函数

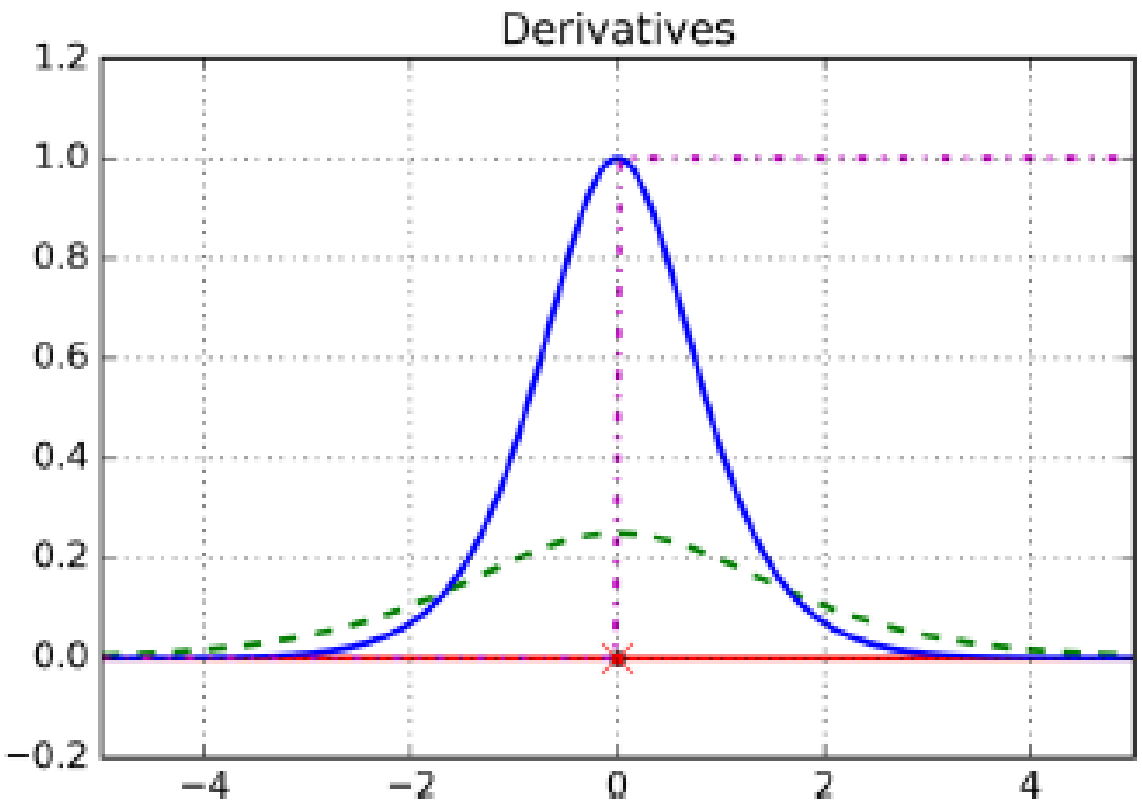
ReLU激活函数

- Rectified Linear Units
- ReLU计算线性函数为非线性，如果大于0就是结果，否则就是0
- 生物神经元的反应看起来其实很像Sigmoid激活函数，所有专家在Sigmoid上卡了很长时间，但是后来发现ReLU才更适合人工神经网络，这是一个模拟生物的误解

$$h_{\mathbf{w},b}(\mathbf{X}) = \max(\mathbf{X} \cdot \mathbf{w} + b, 0)$$

激活函数和导数



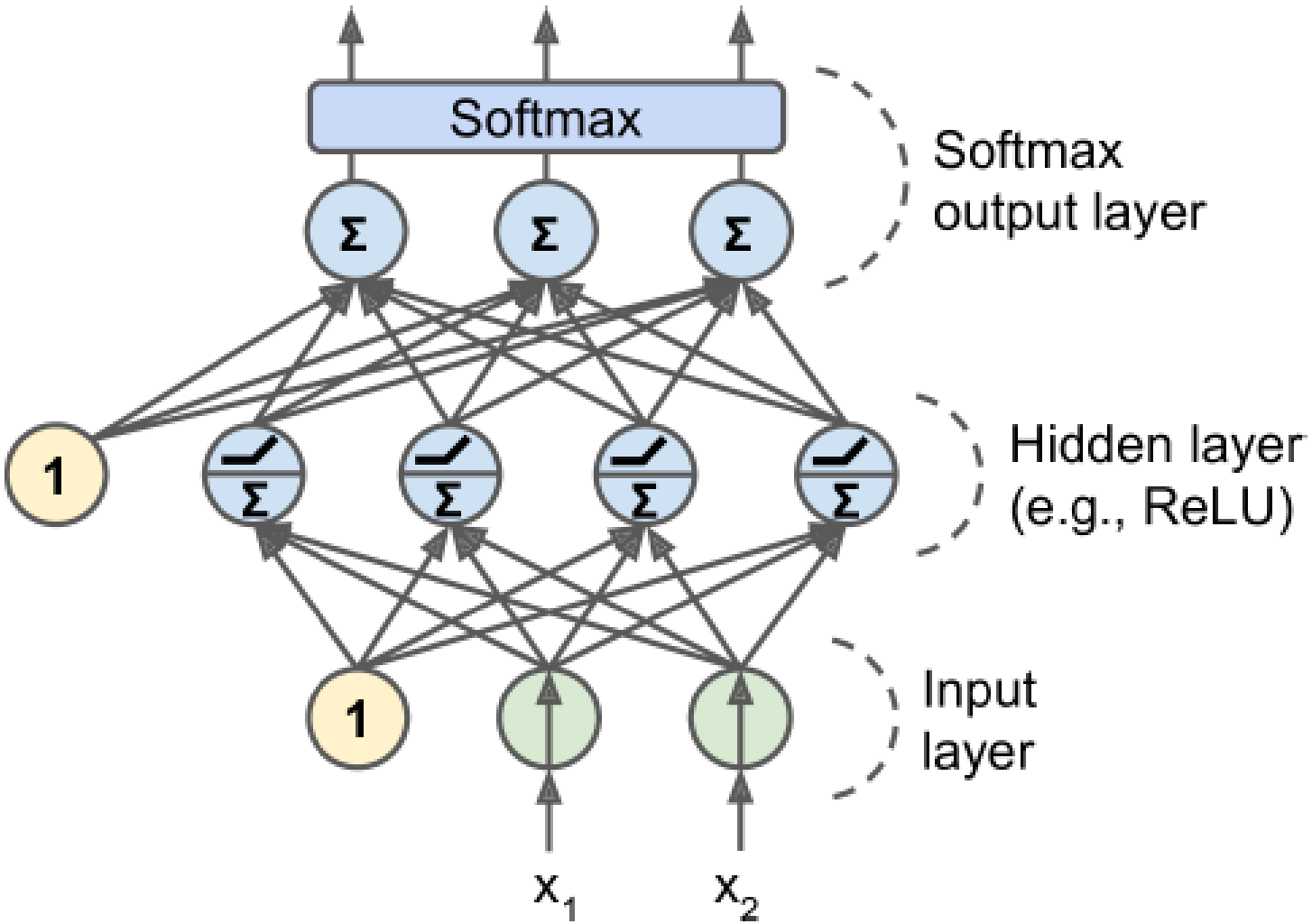


Softmax

- 多层感知机通常用于分类问题，二分类
- 也有很多时候会用于多分类，需要把输出层的激活函数改成共享的softmax函数
- 输出变成用于评估属于哪个类别的概率值

Softmax

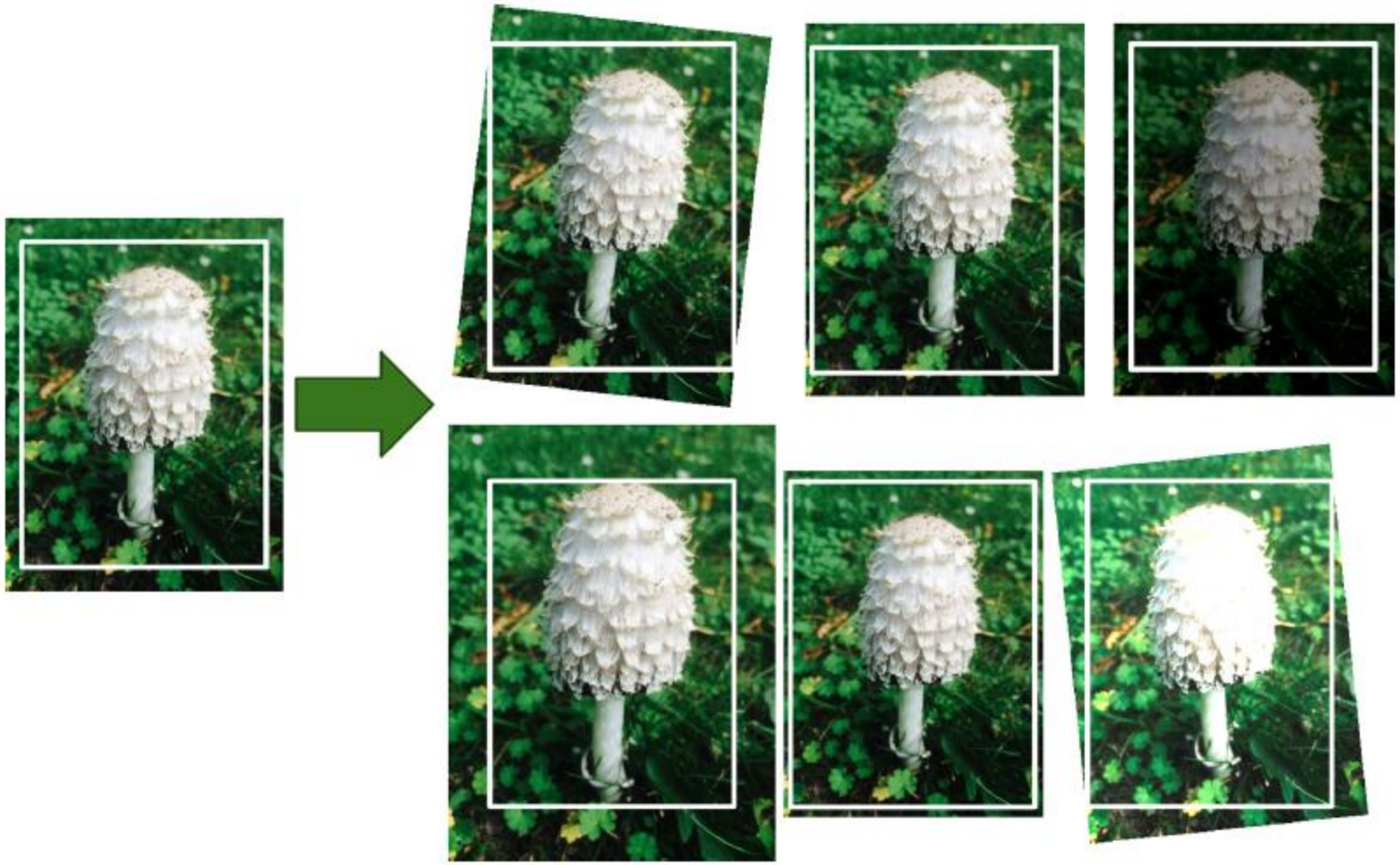
- **feedforw**



数据增大

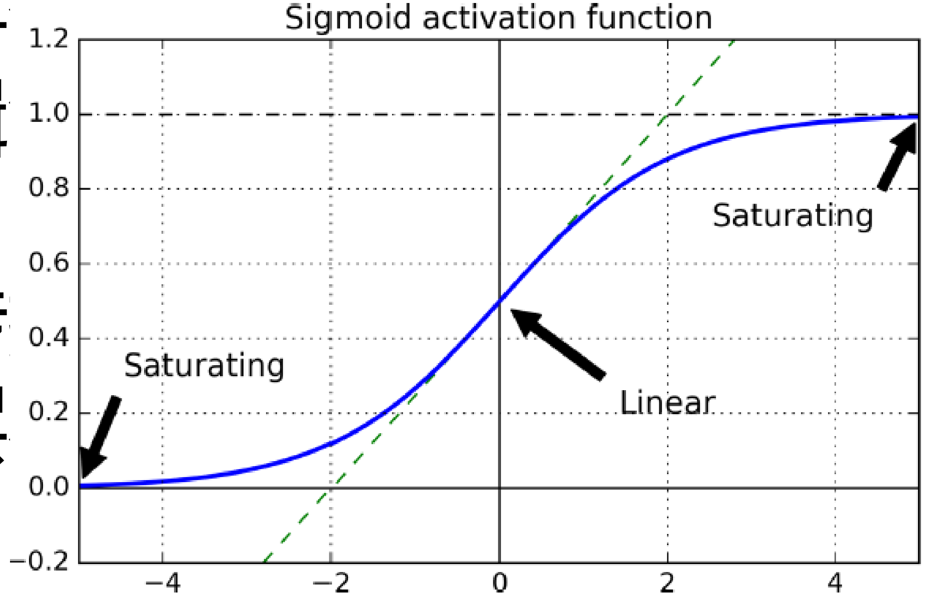
- 从现有的数据产生一些新的训练样本，人工增大训练集，这将减少过拟合
- 例如如果你的模型是分类蘑菇图片，你可以轻微的平移，旋转，改变大小，然后增加这些变化后的图片到训练集，这使得模型可以经受位置，方向，大小的影响，如果你想用模型可以经受光条件的影响，你可以同理产生许多图片用不同的对比度，假设蘑菇对称的，你也可以水平翻转图片
- TensorFlow提供一些图片操作算子，例如transposing(shifting)，rotating，resizing，flipping，cropping，adjusting brightness，contrast，saturation，hue

数据增大



梯度弥散/消失

- Vanishing Gradients
- 在梯度下降中，随着算法反向反馈到前面几层，梯度会越来越小，最终，没有变化，这时或许还没有收敛到比较好的解，这就是梯度消失问题，深度学习遭受不稳定的梯度，不同层学习在不同的速度上
- 如果我们看逻辑…，不管正负，将会饱和在0或1，…反向传播开始，它几乎没有梯度传…会导致只更改高的几层，低的几层



Xavier Initialization

- 我们需要每层输出的方差等于它的输入的方差，并且我们同时需要梯度有相同的方差，当反向传播进入这层时和离开这层时
- 上面理论不能同时保证，除非层有相同的输入连接和输出连接，但是有一个不错的妥协在实际验证中，连接权重被随机初始化，[illegible]输出的连接，也叫fan_in和[illegible] Xaiver initialization

Activation function	Uniform distribution [–r, r]	Normal distribution
Logistic	$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
Hyperbolic tangent	$r = 4\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = 4\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
ReLU (and its variants)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

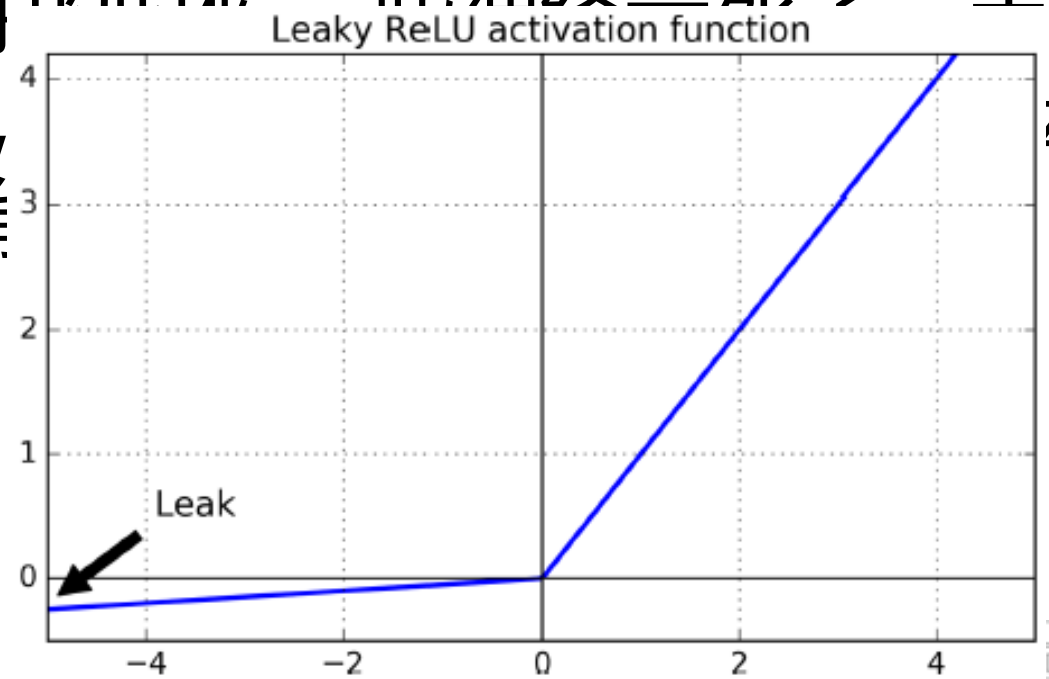
He Initialization

- 初始化策略对应ReLU激活函数或者它的变形叫做He initialization
- fully_connected()函数默认使用Xavier初始化对应uniform distribution
- 我们可以改成He initialization使用variance_scaling_initializer()函数

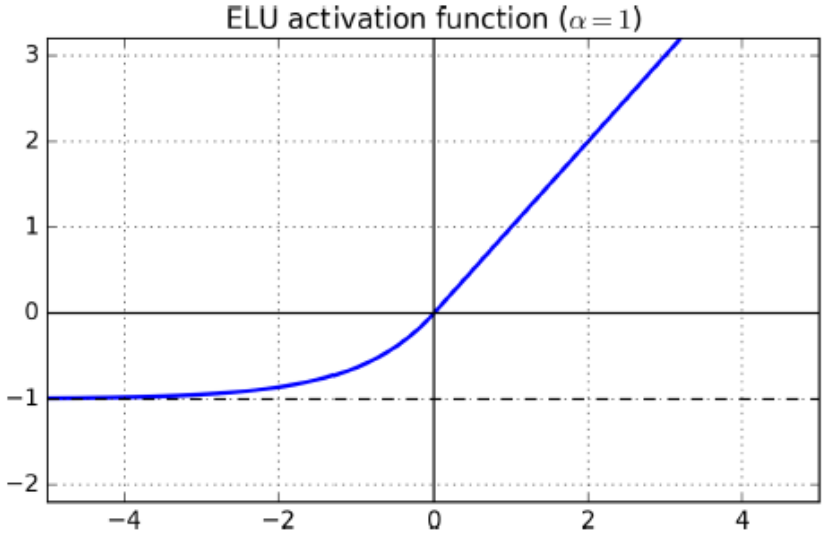
```
he_init = tf.contrib.layers.variance_scaling_initializer()
hidden1 = fully_connected(X, n_hidden1, weights_initializer=he_init, scope="h1")
```

不饱和激活函数

- 选择ReLU更多因为对于正数时候不饱和，同时因为它计算速度快
- 但是不幸的是，ReLU激活函数不完美，有个问题是dying ReLU，在训练的时候，很神经元死了，当它们输出小于0，去解决这个，z，z)，a=0.01，在乎运行性能



不饱和激



- RReLU，Random，a是一个在 的数在训练时，固定的平均值在测试
- PReLU，Parametric，a是一个在训练过程中需要学习的超参数，它会被修改在反向传播中，适合大数据集
- ELU，exponential，计算梯度的速度会慢一些，但是整体因为没有死的神经元，整体收敛快，超参数0.01
- ELU > leaky ReLU > ReLU > tanh > logistic

不饱和激活函数

- ELU直接使用
- Leaky_relu需要自己实现

```
hidden1 = fully_connected(X, n_hidden1, activation_fn=tf.nn.elu)
```

```
def leaky_relu(z, name=None):
    return tf.maximum(0.01 * z, z, name=name)

hidden1 = fully_connected(X, n_hidden1, activation_fn=leaky_relu)
```

Batch Normalization

- 尽管使用He initialization和ELU可以很大程度减少梯度弥散问题在一开始训练，但是不保证在训练的后面不会发生
- Batch Normalization可以更好解决问题
- 它应用于每层激活函数之前，就是做均值和方差归一化，对于每一批次数据
- 并且还做放大缩小，平移

1. $\mu_B = \frac{1}{m_B}\sum_{i=1}^{m_B}\mathbf{x}^{(i)}$
2. $\sigma_B^{\,2} = \frac{1}{m_B}\sum_{i=1}^{m_B}\left(\mathbf{x}^{(i)} - \mu_B\right)^2$
3. $\widehat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{\sigma_B^{\,2} + \epsilon}}$
4. $\mathbf{z}^{(i)} = \gamma\widehat{\mathbf{x}}^{(i)} + \beta$

Batch Normalization

- 在测试时，使用全部训练集的均值、方差，在Batch Normalization层，总共4个超参数被学习，scale，offset，mean，standard deviation
- 利用Batch Normalization即使是用饱和的激活函数，梯度弥散问题也可以很好减轻
- 利用Batch Normalization对权重初始化就不那么敏感了
- 利用Batch Normalization可以使用大一点的学习率来提速了
- 利用Batch Normalization同样的准确率，4.9% top-5错误率，速度减少14倍
- 利用Batch Normalization也是一种正则化，就必要使用dropout了或其他技术了

Batch Normalization

- weight
的收敛
止过拟
regular
复杂度
失函数
函数的

```
import tensorflow as tf
from tensorflow.contrib.layers import batch_norm

n_inputs = 28 * 28
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")

is_training = tf.placeholder(tf.bool, shape=(), name='is_training')
bn_params = {
    'is_training': is_training,
    'decay': 0.99,
    'updates_collections': None
}

hidden1 = fully_connected(X, n_hidden1, scope="hidden1",
                          normalizer_fn=batch_norm, normalizer_params=bn_params)
hidden2 = fully_connected(hidden1, n_hidden2, scope="hidden2",
                          normalizer_fn=batch_norm, normalizer_params=bn_params)
logits = fully_connected(hidden2, n_outputs, activation_fn=None,scope="outputs",
                         normalizer_fn=batch_norm, normalizer_params=bn_params)
```

你所说
的是防
则项（
模型的
度对损
型损失

Batch Normalization

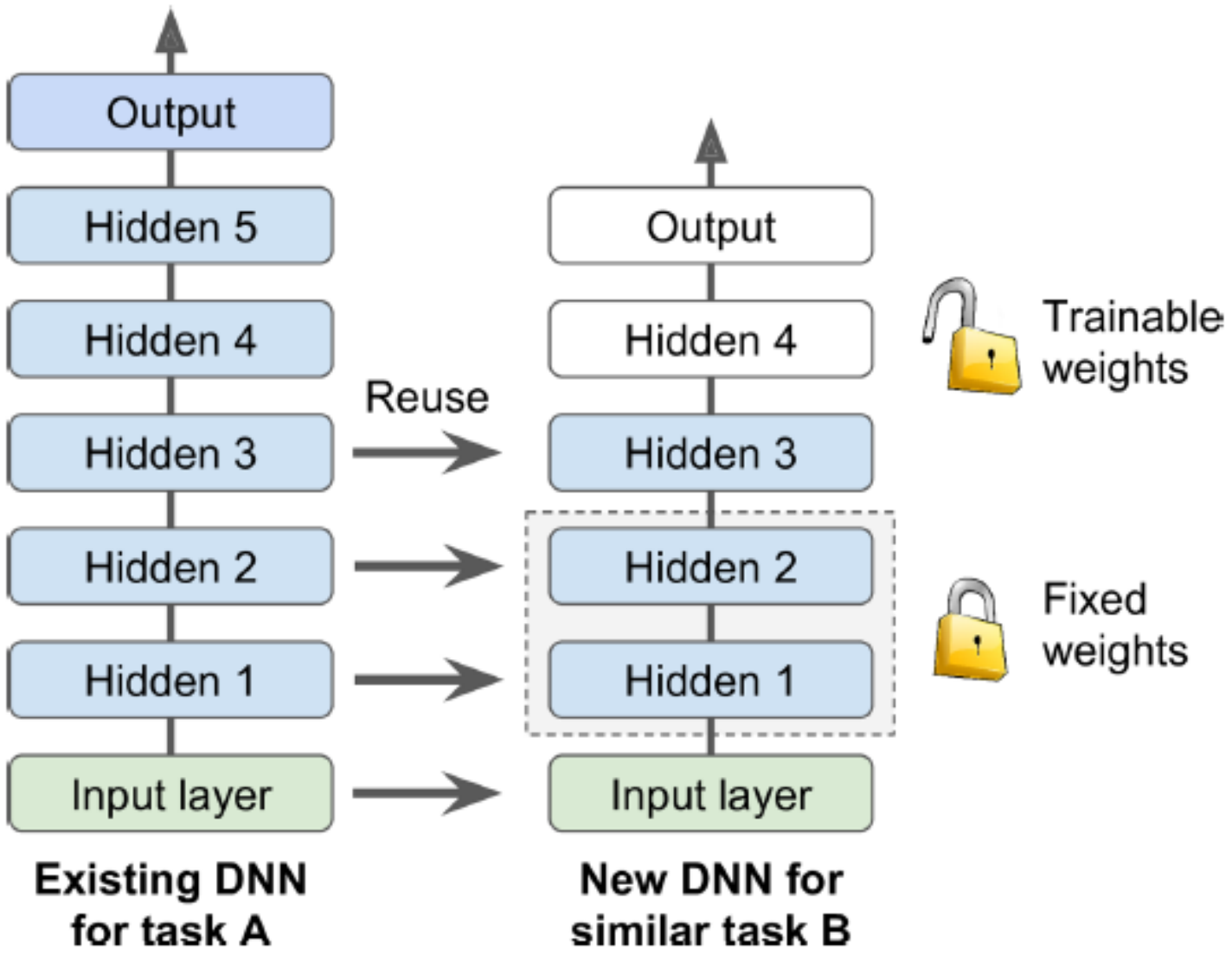
```
[...]

with tf.contrib.framework.arg_scope(
        [fully_connected],
        normalizer_fn=batch_norm,
        normalizer_params=bn_params):
    hidden1 = fully_connected(X, n_hidden1, scope="hidden1")
    hidden2 = fully_connected(hidden1, n_hidden2, scope="hidden2")
    logits = fully_connected(hidden2, n_outputs, scope="outputs",
                             activation_fn=None)
```

Batch Normalization

```
with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        [...]
        for X_batch, y_batch in zip(X_batches, y_batches):
            sess.run(training_op,
                     feed_dict={is_training: True, X: X_batch, y: y_batch})
        accuracy_score = accuracy.eval(
            feed_dict={is_training: False, X: X_test_scaled, y: y_test}))
        print(accuracy_score)
```

复用预先训练好的层

- Transfer learning



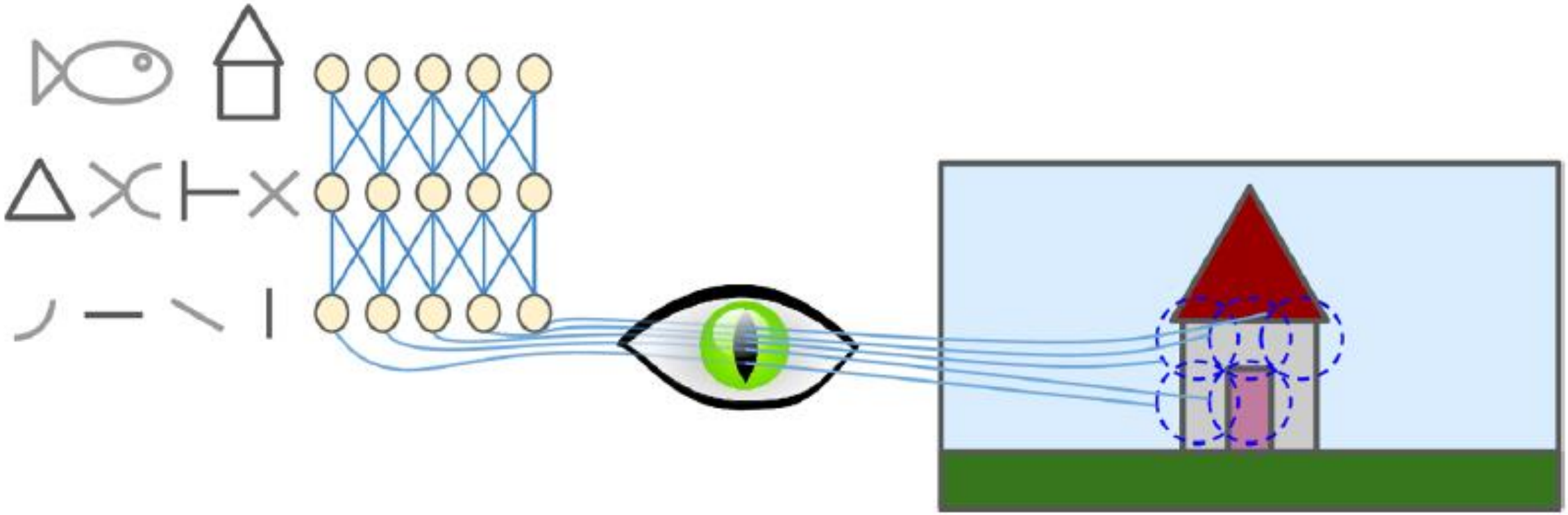
复用TensorFlow模型

```
[...] # construct the original model

with tf.Session() as sess:
    saver.restore(sess, "./my_original_model.ckpt")
    [...] # Train it on your new task

 [...] # build new model with the same definition as before for hidden layers 1-3

 init = tf.global_variables_initializer()

 reuse_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
                                scope="hidden[123]")
 reuse_vars_dict = dict([(var.name, var.name) for var in reuse_vars])
 original_saver = tf.Saver(reuse_vars_dict) # saver to restore the original model

 new_saver = tf.Saver() # saver to save the new model

 with tf.Session() as sess:
     sess.run(init)
     original_saver.restore("./my_original_model.ckpt") # restore layers 1 to 3
     [...] # train the new model
     new_saver.save("./my_new_model.ckpt") # save the whole model
```

卷积神经网络

- Convolutional neural networks
- 视觉皮层、感受野，一些神经元看线，一些神经元看线的方向，一些神经元有更大的感受野，组合底层的图案
- 1998年Yann LeCun等人推出了LeNet-5架构，广泛用于手写体数字识别，包含全连接层和sigmoid激活函数，还有卷积层和池化层

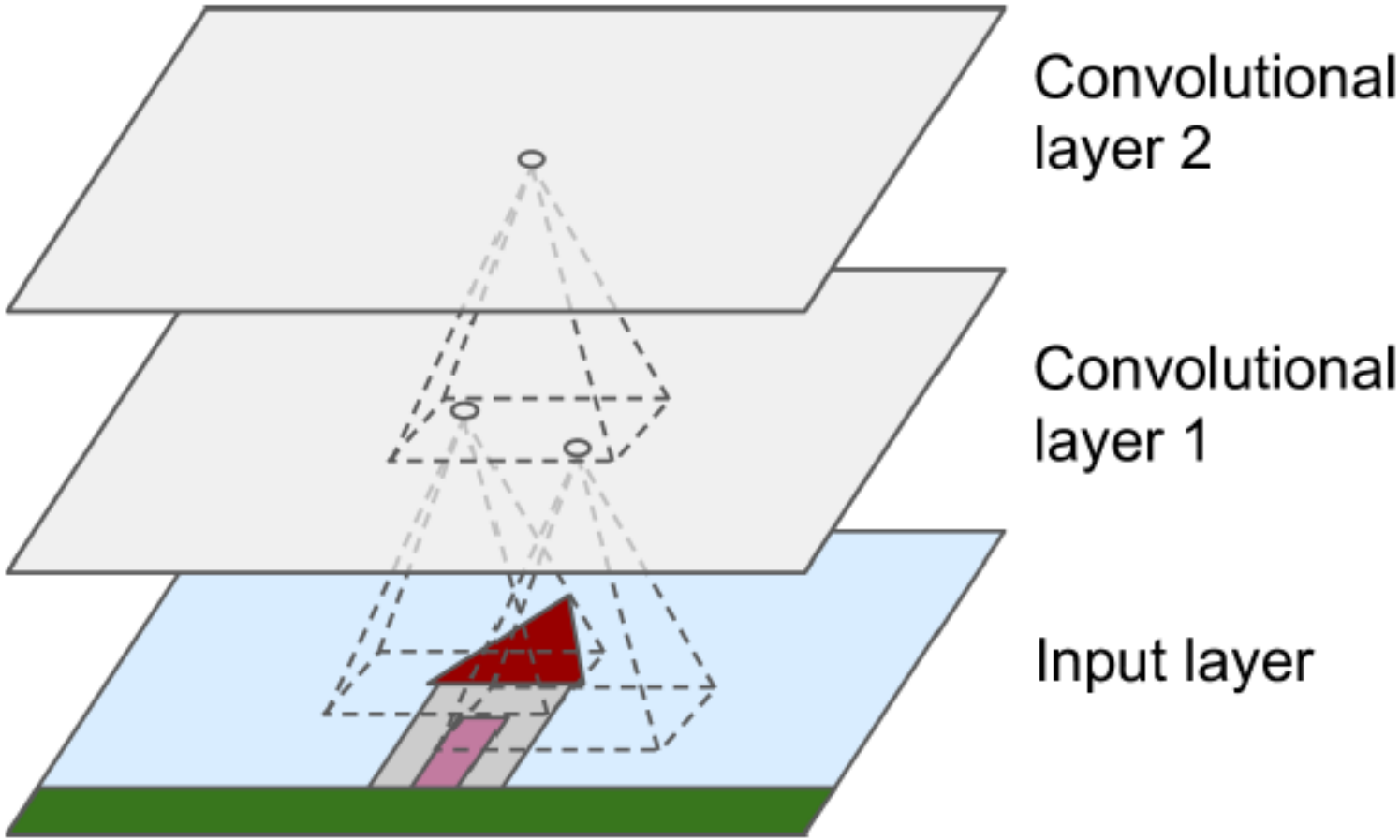
感受野



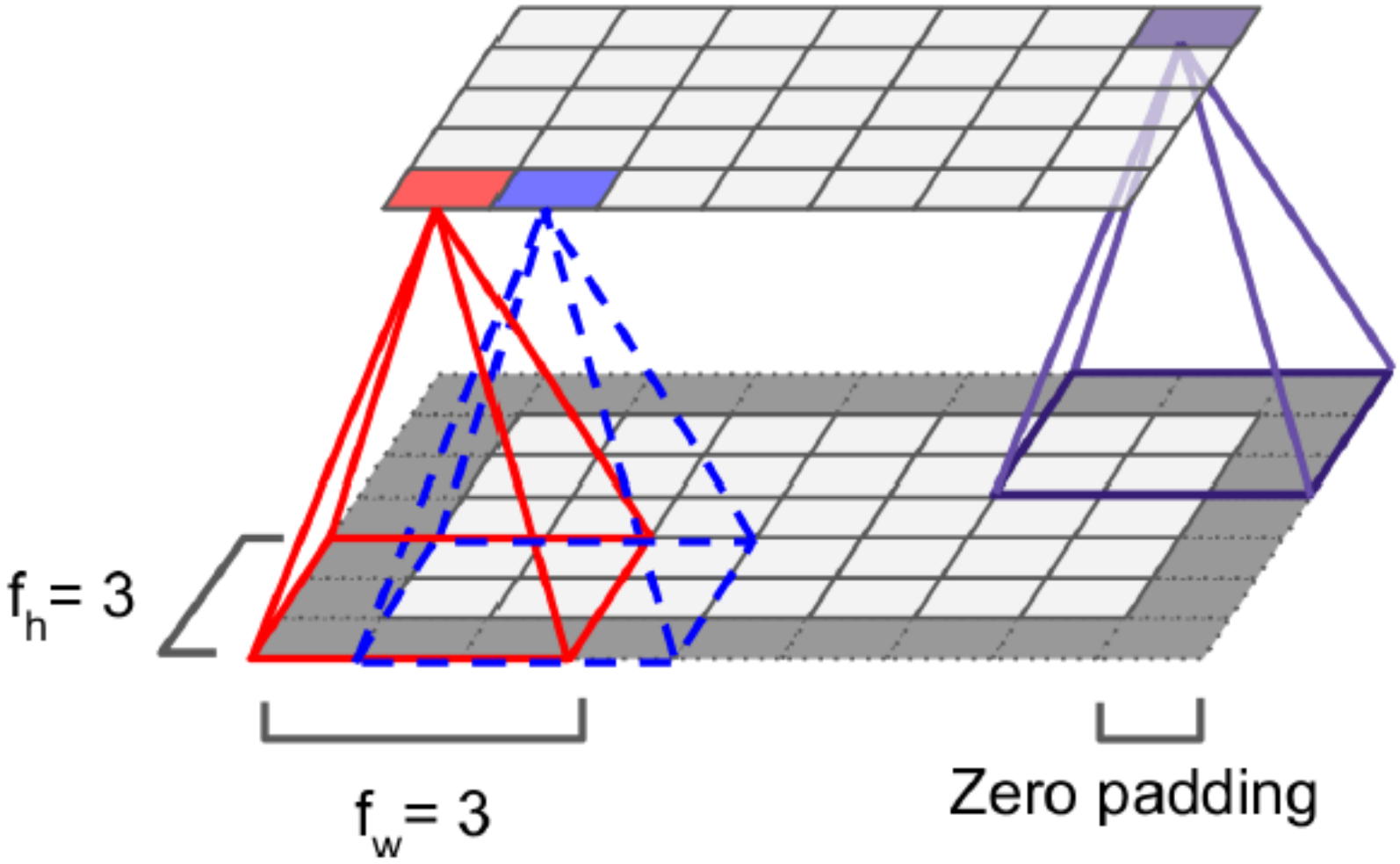
卷积层

- CNN里面最重要的构建单元就是卷积层
- 神经元在第一个卷积层不是连接输入图片的每一个像素，只是连接它们感受野的像素，以此类推，第二个卷积层的每一个神经元仅连接位于第一个卷积层的一个小方块的神经元
- 以前我们做MNIST的时候，把图像变成1D的，现在直接用2D

卷积层



Zero Padding



卷积的计算

- 假设有一个5*5的图像，使用一个3*3的filter进行卷积，想得到一个3*3的Feature Map

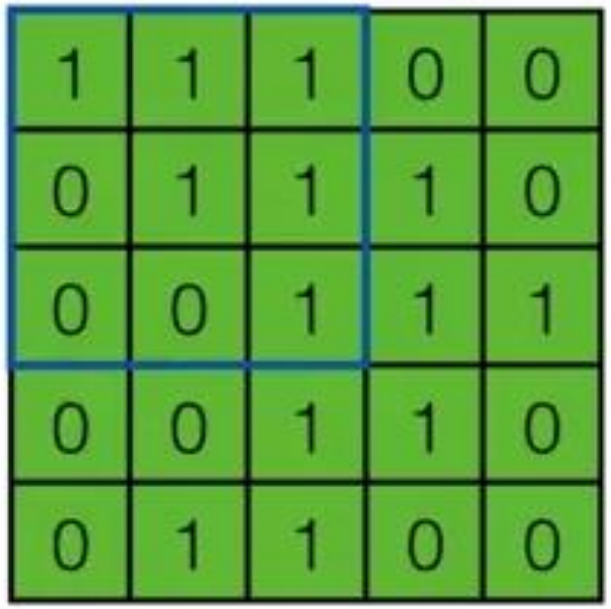
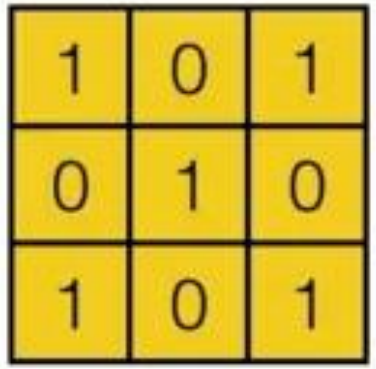
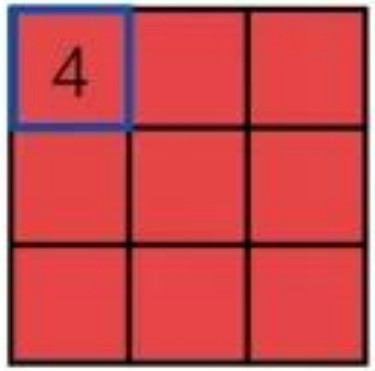


image 5*5



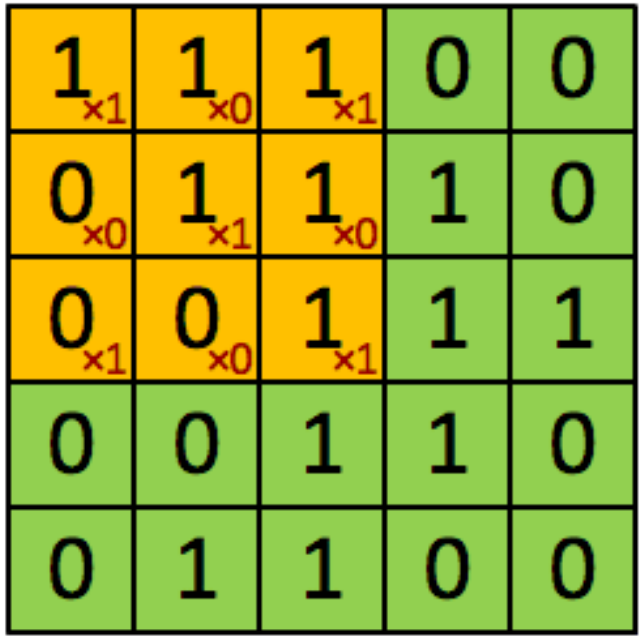
bias=0

filter 3*3



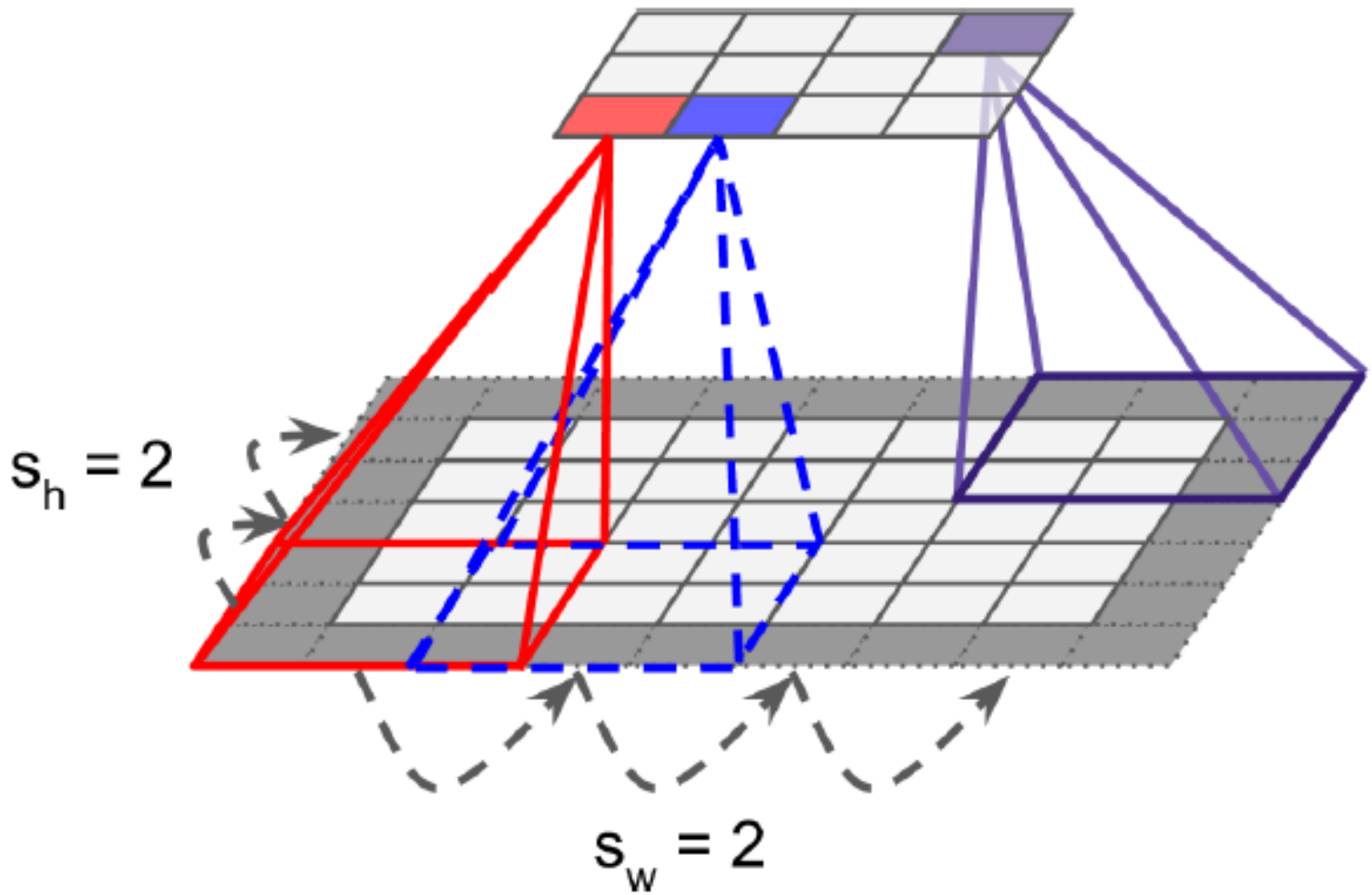
feature map 3*3

卷积的计算



stride

- 步伐



Stride等于2

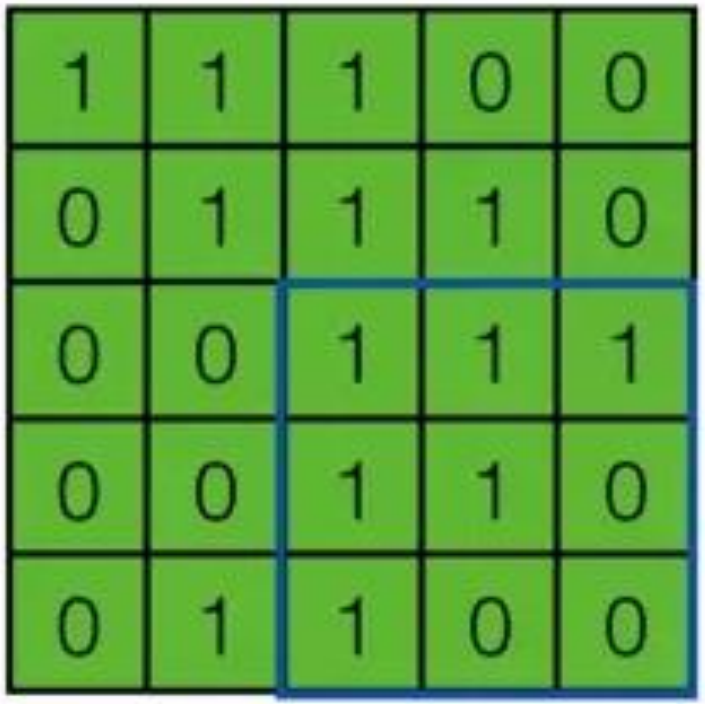
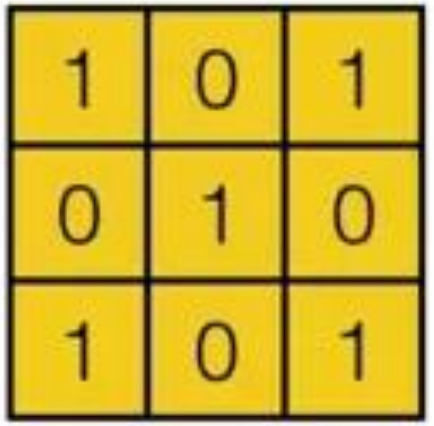
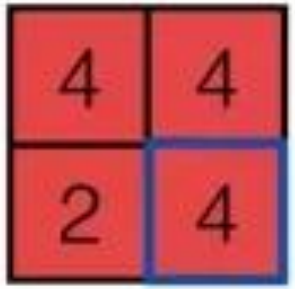


image 5*5



bias=0

filter 3*3

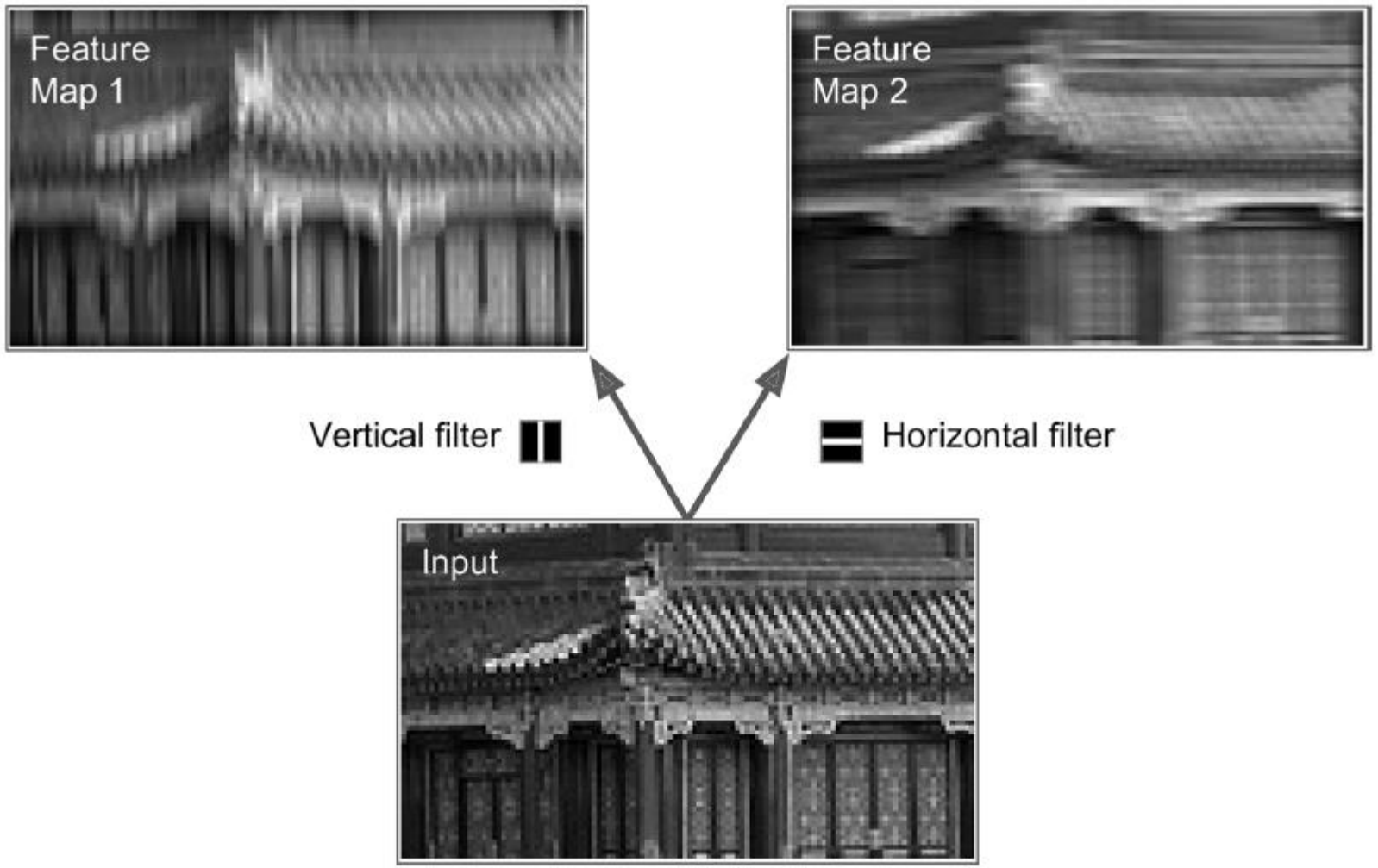


feature map 2*2

Filter卷积核

- Convolution kernels
- Vertical line filter 中间列为1，周围列为0
- Horizontal line filter 中间行为1，周围行为0
- 7*7 matrix

Feature Map

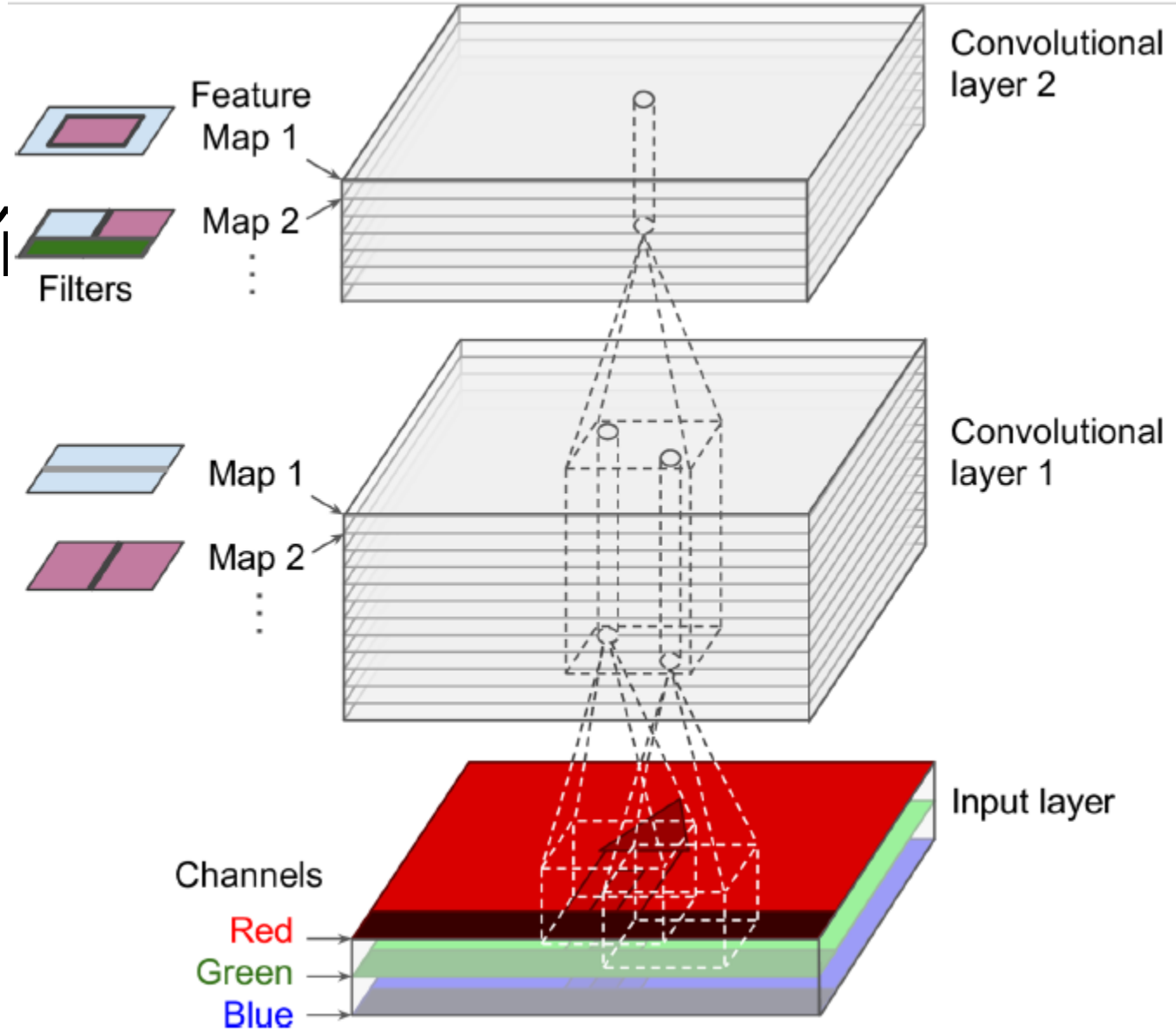

Feature
Map 1
Feature
Map 2
Vertical filter
Horizontal filter
Input

3D

- 在一个特征图里面，所有的神经元共享一样的参数（weights bias），权值共享
- 不同的特征图有不同的参数

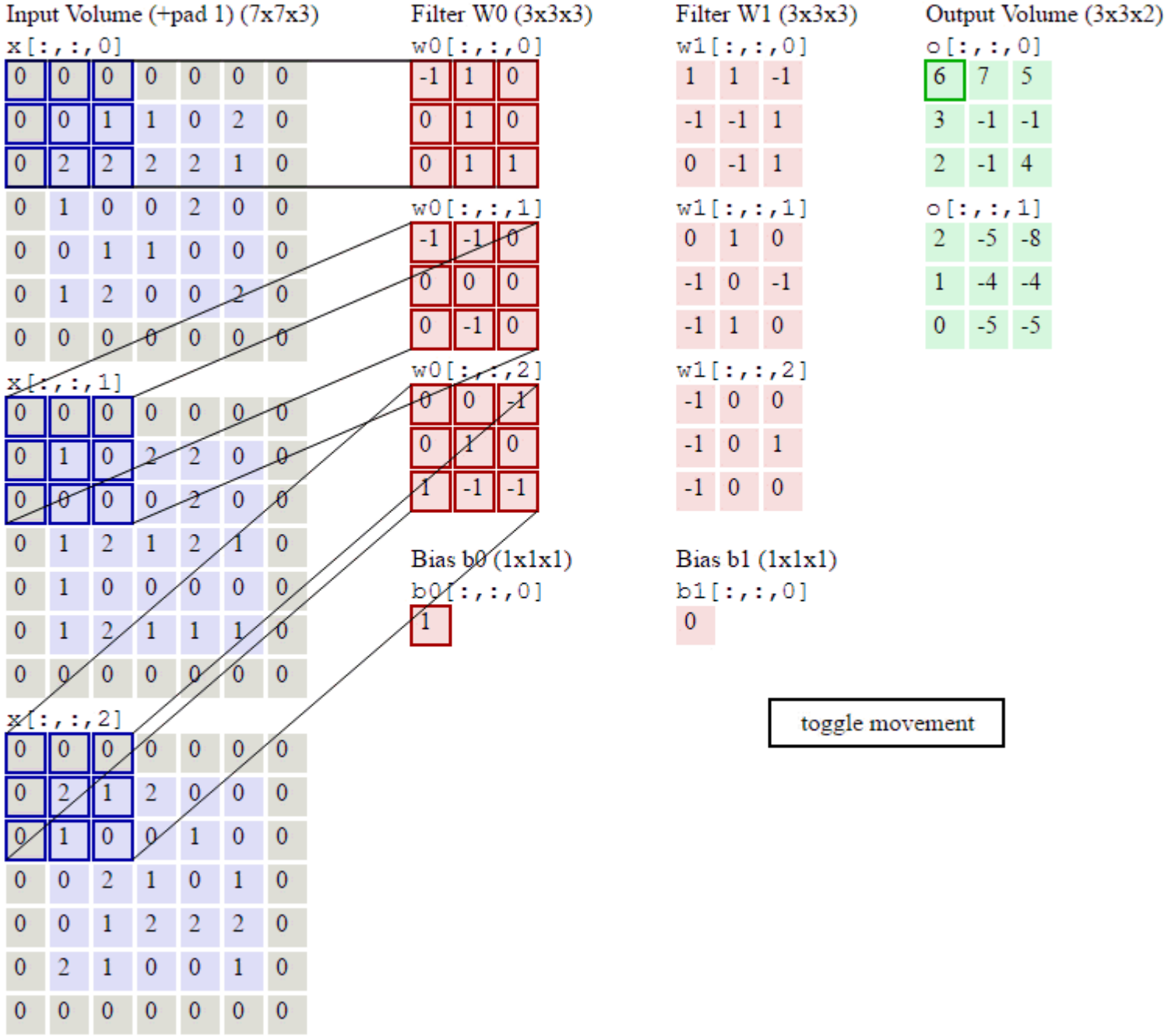
3D

- RGB
- 灰度图一
- 卫星图像



$$a_{i,j} = f(\sum_{d=0}^{D-1}\sum_{m=0}^{F-1}\sum_{n=0}^{F-1} w_{d,m,n}x_{d,i+m,j+n} + w_b)$$

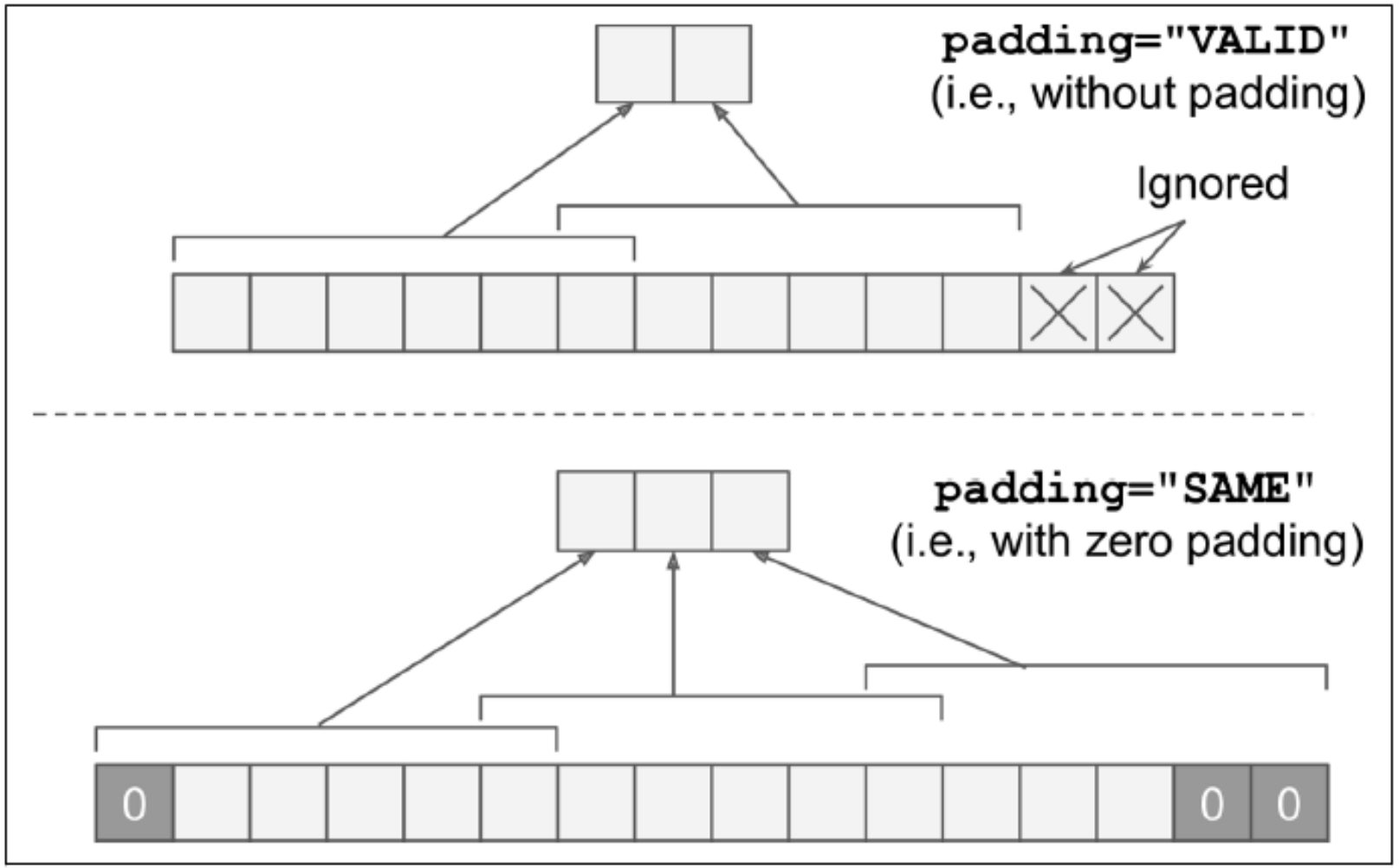
- D是深度；F是filter
- Wd,m,n表示filter的
- 第n列权重；
- ad,l,j表示图像的第
- 第j列像素；



Padding模式

- VALID
- 不适用zero padding，有可能会忽略图片右侧或底下，这个得看stride的设置
- SAME
- 必要会加zero padding，这种情况下，输出神经元个数等于输入神经元个数除以步长 ceil(13/5)=3

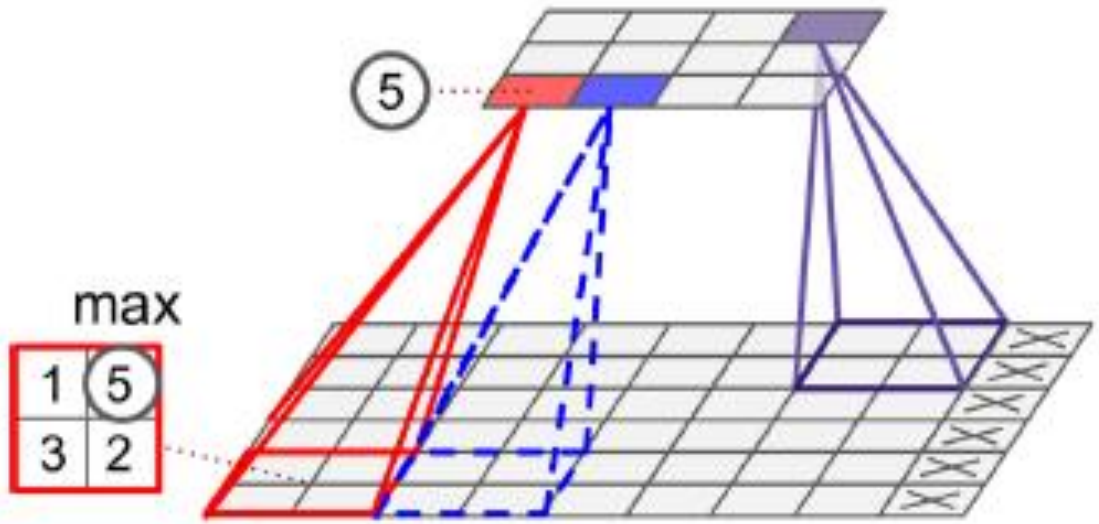
Padding模式

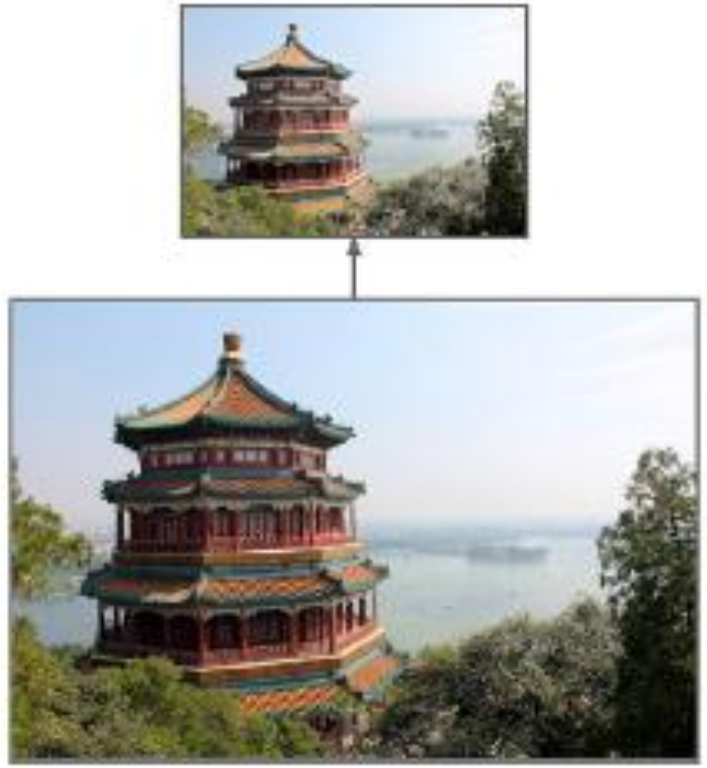

padding="VALID"
(i.e., without padding)
Ignored
padding="SAME"
(i.e., with zero padding)
0
0
0

池化Pooling

- 目标就是降采样subsample，shrink，减少计算负荷，内存使用，参数数量（也可防止过拟合）
- 减少输入图片大小也使得神经网络可以经受一点图片平移，不受位置的影响
- 正如卷积神经网络一样，在池化层中的每个神经元被连接到上面一层输出的神经元，只对应一小块感受野的区域。我们必须定义大小，步长，padding类型
- 池化神经元没有权重值，它只是聚合输入根据取最大或者是求均值
- 2*2的池化核，步长为2，没有填充，只有最大值往下传递

池化Pooling





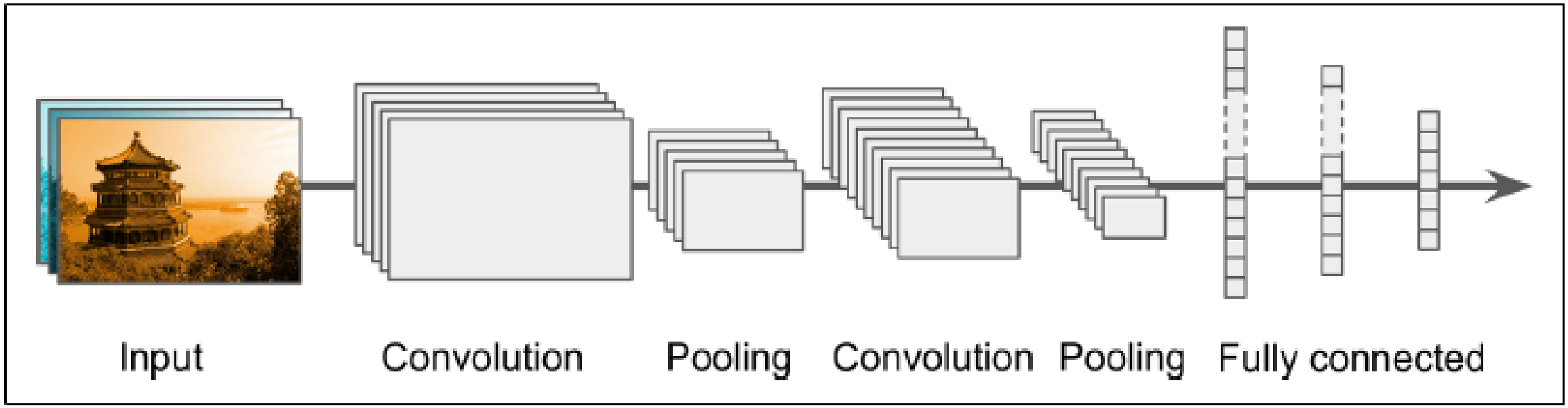
池化Pooling

- 长和宽两倍小，面积4倍小，丢掉75%的输入值
- 一般情况下，池化层工作于每一个独立的输入通道，所以输出的深度和输入的深度相同

CNN架构

- 典型的CNN架构堆列一些卷积层
- 一般一个卷积层后跟ReLU层，然后是一个池化层，然后另一些个卷积层+ReLU层，然后另一个池化层，通过网络传递的图片越来越小，但是也越来越深，例如更多的特征图！
- 最后常规的前向反馈神经网络被添加，由一些全连接的层+ReLU层组成，最后是输出层预测，例如一个softmax层输出预测的类概率
- 一个常见的误区是使用卷积核过大，你可以使用和9*9的核同样效果的两个3*3的核，好处是会有更少的参数需要被计

CNN架构



Top 5 error

- top-five错误率是测试图片系统判断前5个类别预测都没有包含正确答案的数量
- 5年间，由于好的CNN模型的诞生，错误率从26%降到了3%

经典模型

- LeNet-5架构，1998年
- AlexNet，2012年
- GoogLeNet，2014年
- ResNet，2015年

LeNet-5

- 最知名的CNN架构，Yann LeCun在1998年创建，广泛应用于MNIST手写体识别
- Padding使得28*28的图变成32*32
- 使用均值池化
- C3层只连接3或者4个S2
- 最后一层是欧式距离测量
- 它的对应权值向量距离
- 现在一般用交叉熵更好，
- 更多，提供大的梯度，因此收敛更快

Layer	Type	Maps	Size	Kernel size	Stride	Activation
Out	Fully Connected	–	10	–	–	RBF
F6	Fully Connected	–	84	–	–	tanh
C5	Convolution	120	1 × 1	5 × 5	1	tanh
S4	Avg Pooling	16	5 × 5	2 × 2	2	tanh
C3	Convolution	16	10 × 10	5 × 5	1	tanh
S2	Avg Pooling	6	14 × 14	2 × 2	2	tanh
C1	Convolution	6	28 × 28	5 × 5	1	tanh
In	Input	1	32 × 32	–	–	–