Batch Normalization

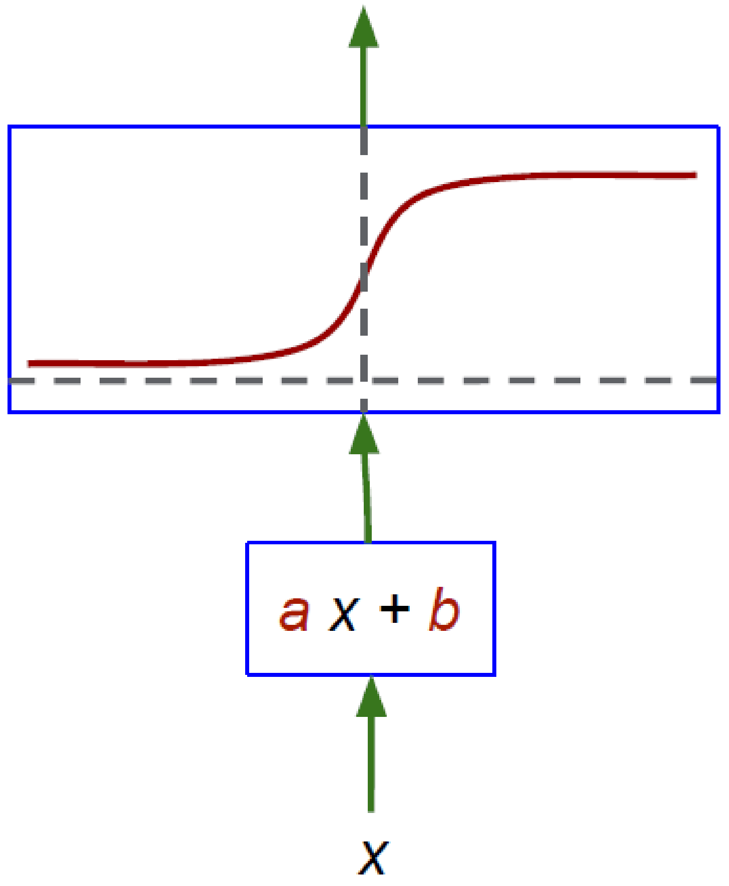
Slide modified from Sergey Ioffe *, with permission*

Slides based on

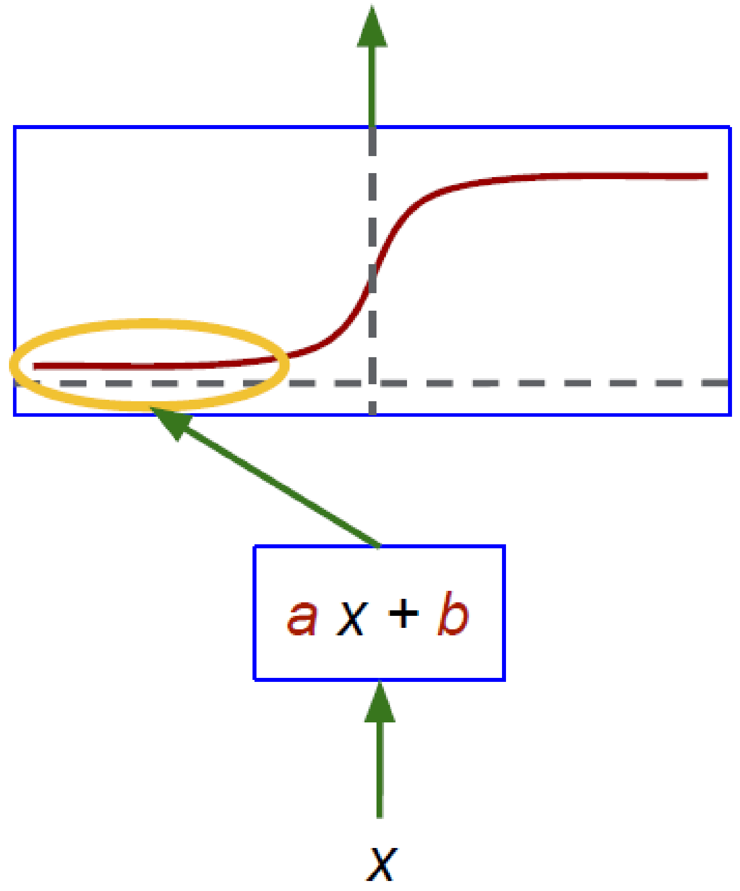
Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

By Sergey Ioffe and Christian Szegedy

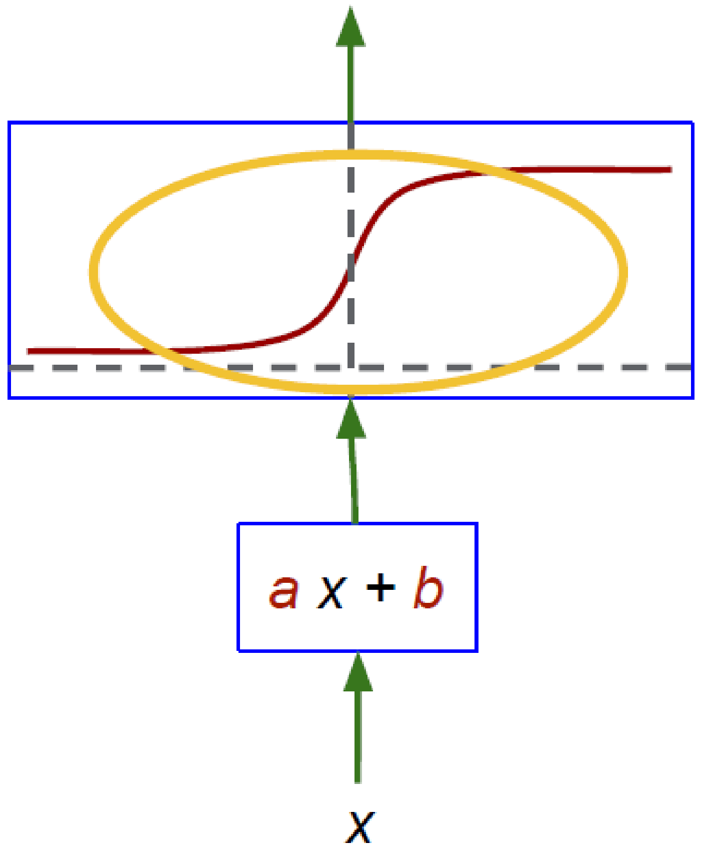
Batch Normalization



Batch Normalization

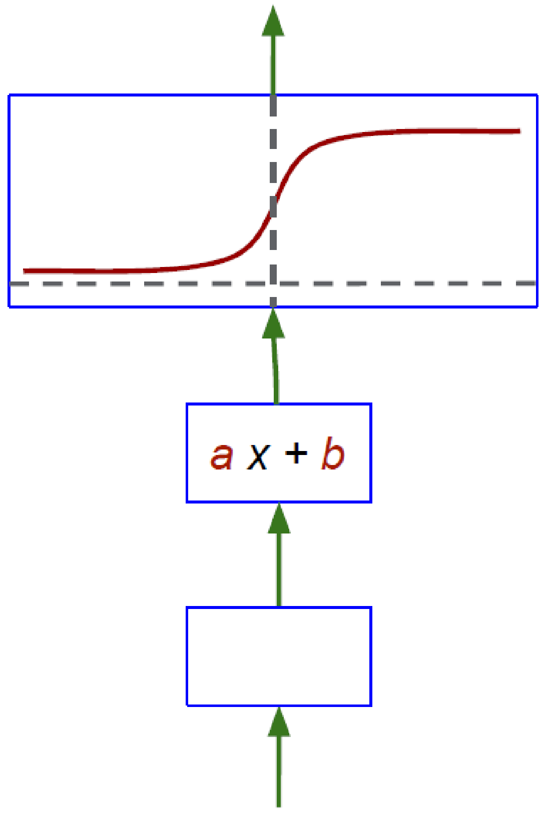


Batch Normalization



Effect of changing input distribution

- Careful initialization
- Small learning rates
- Rectifiers



Internal covariate shift

- Layer input distributions change during training

$$\ell = F_2(\ F_1(\mathrm{u}, \Theta_1),\ \Theta_2)$$

- Change in internal activation distribution requires domain adaptation

- Normalize each activation:

$$x \mapsto \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x]}}$$

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

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

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

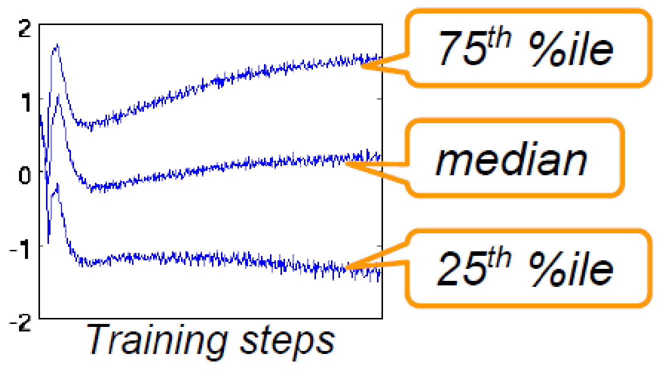
Scale and shift: $$y_i \leftarrow \gamma \widehat{x}_i + \beta$$

- Replace batch statistics with population statistics

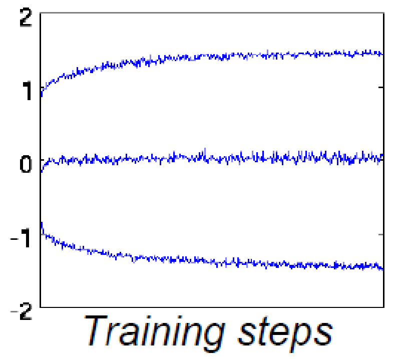
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad \Rightarrow \quad \widehat{x} \leftarrow \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}}$$

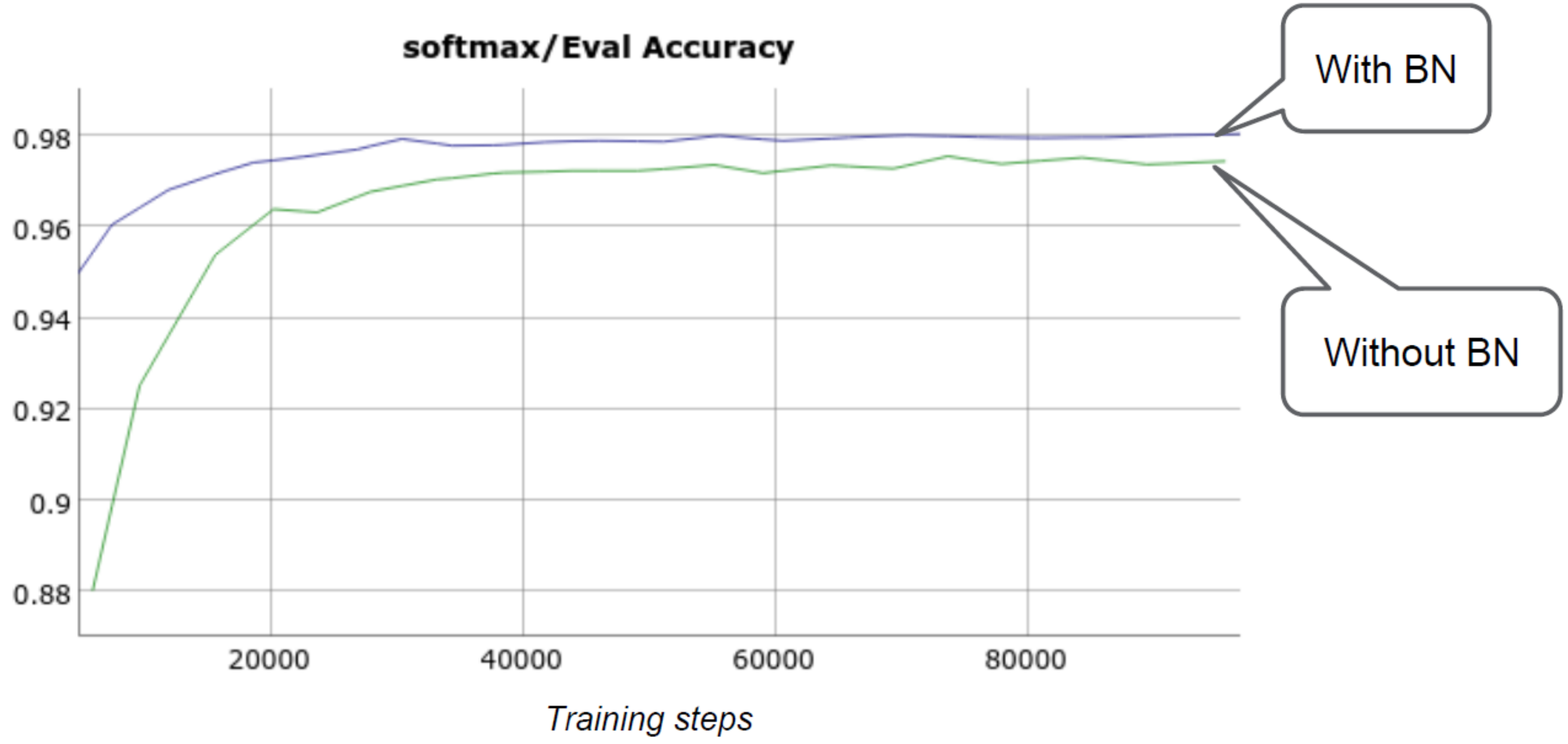
- MNIST: 3 FC layers + softmax, 100 logistic units per hidden layer
- Distribution of inputs to a typical sigmoid, evolving over 100k steps:

Without BN:

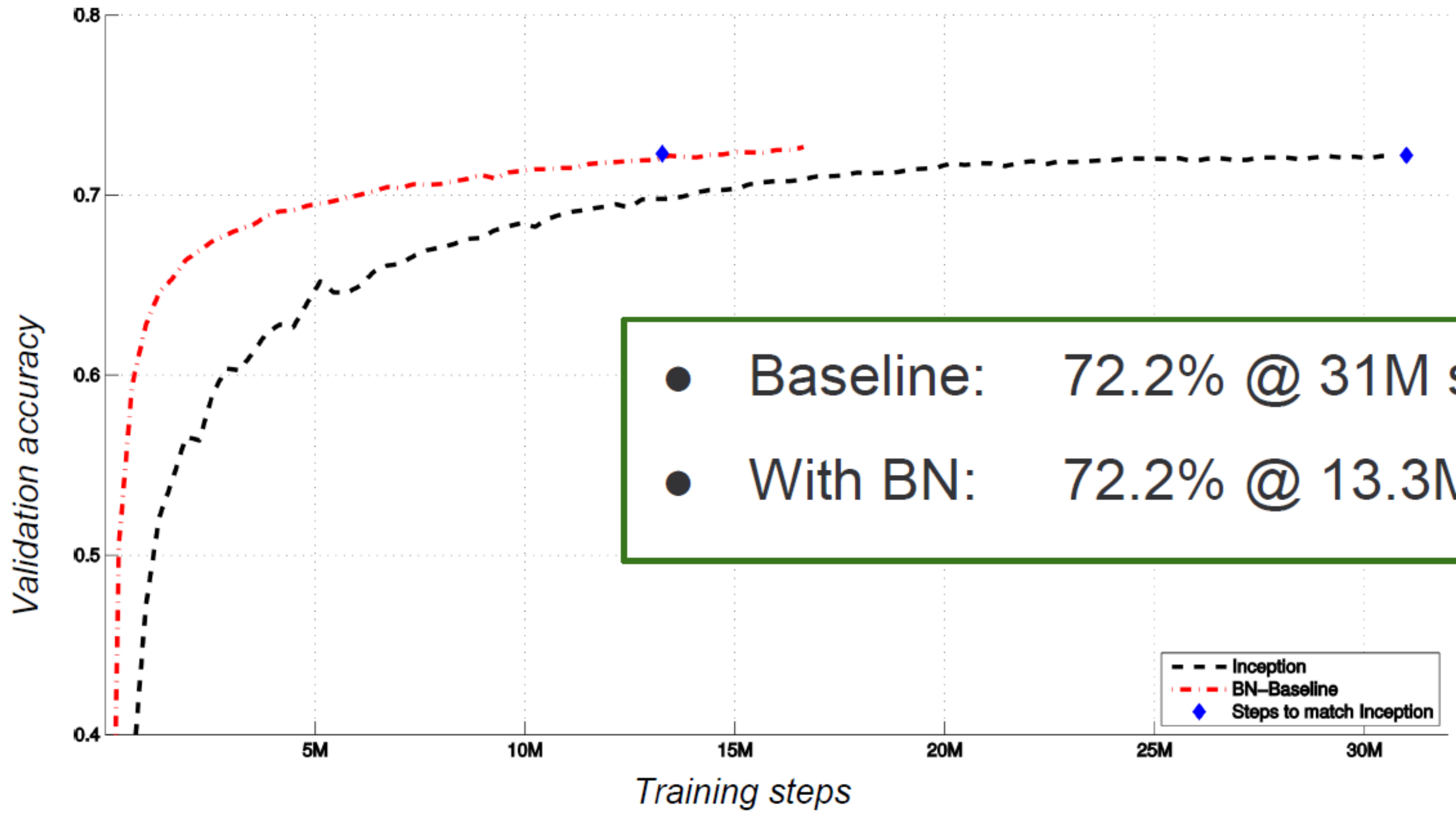


With BN:



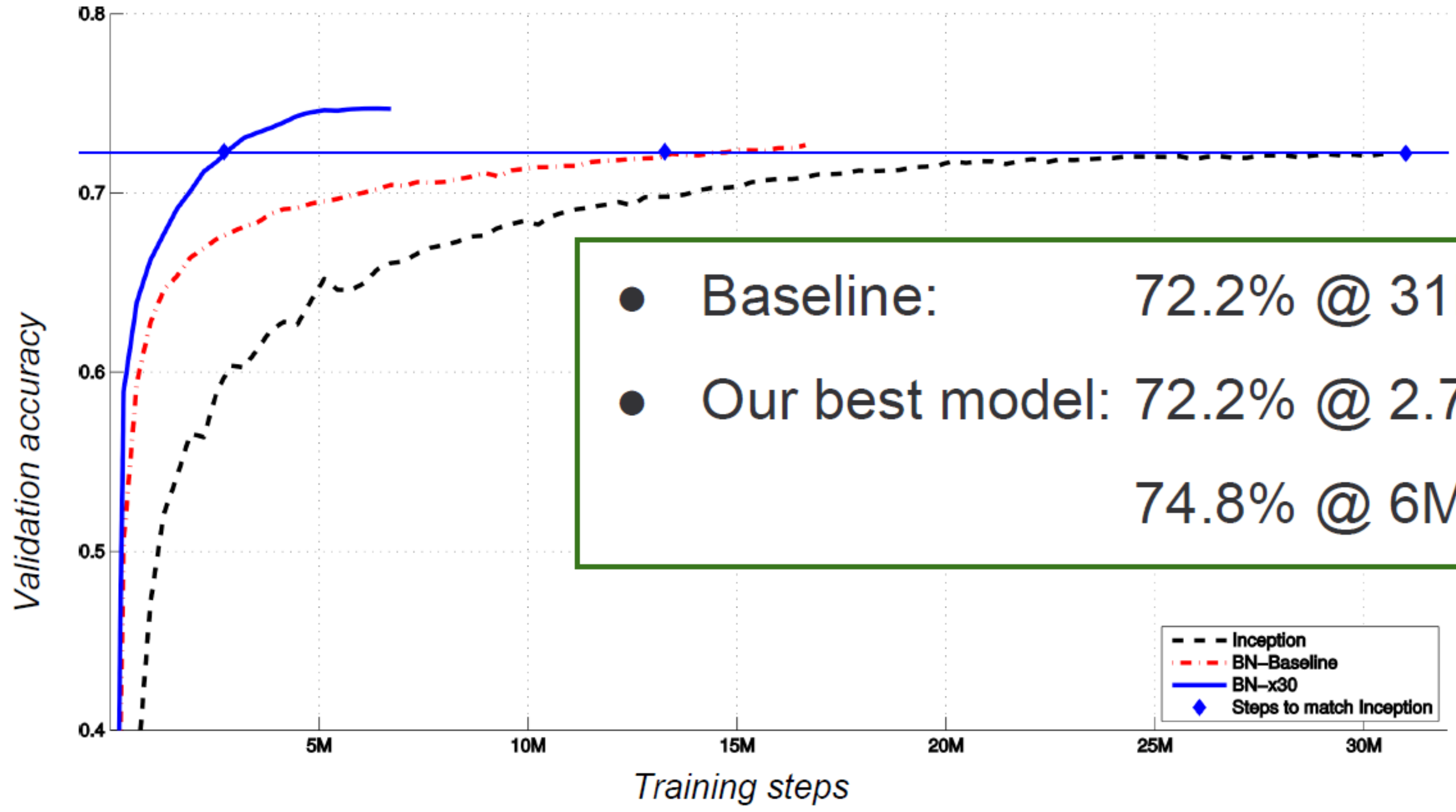

softmax/Eval Accuracy
0.98
0.96
0.94
0.92
0.9
0.88
20000
40000
60000
80000
Training steps
With BN
Without BN

- Inception: deep convolutional ReLU model
- Distributed SGD with momentum
- Batch Normalization applied at every convolutional layer
    - Extra cost (~30%) per training step



- Baseline: 72.2% @ 31M steps
- With BN: 72.2% @ 13.3M steps

- Batch Normalization enables higher learning rate
    - Increased 30x
- Removing dropout improves validation accuracy
    - Batch Normalization as a regularizer?

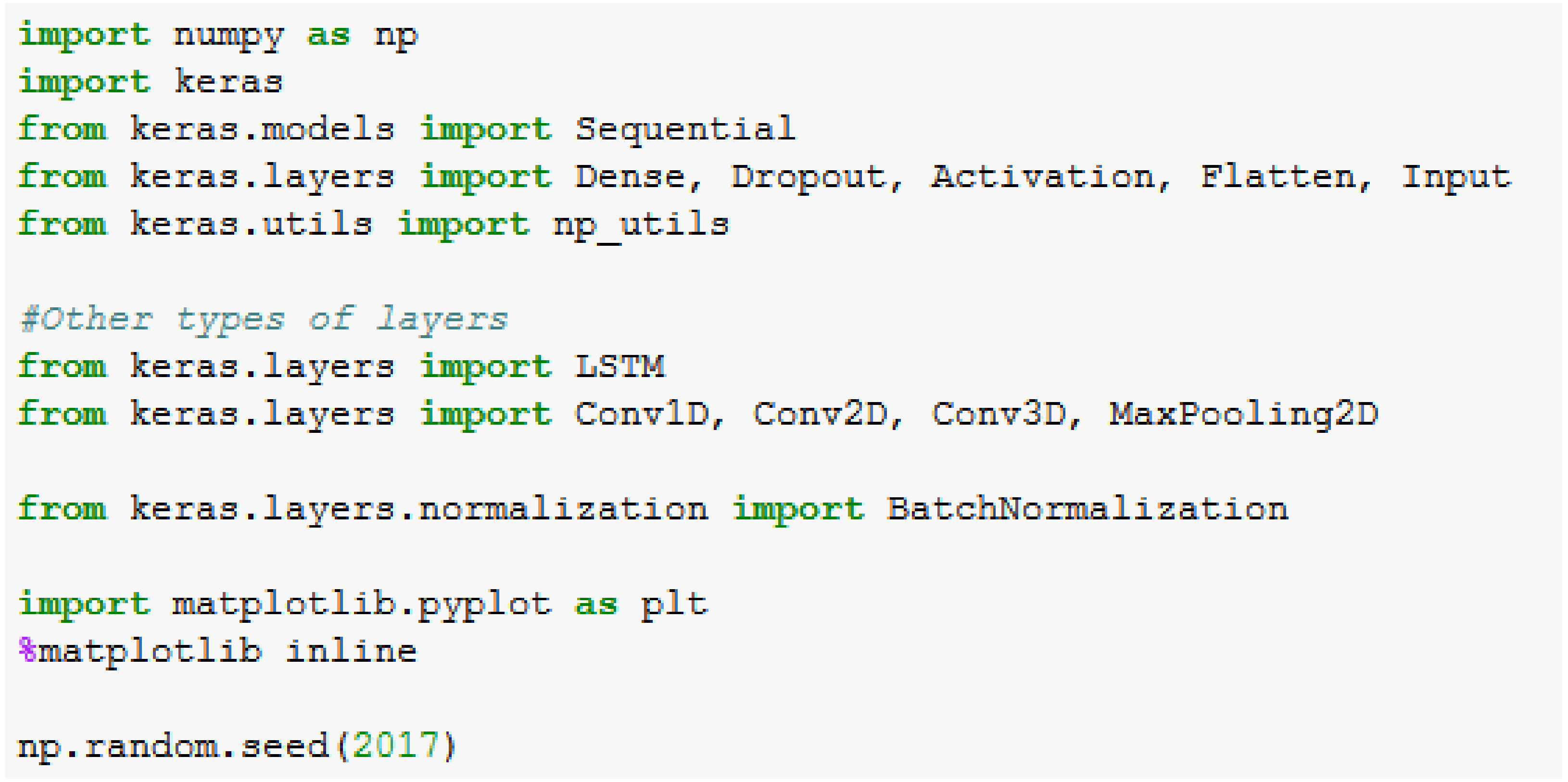


- Baseline: 72.2% @ 31M steps
- Our best model: 72.2% @ 2.7M steps

  74.8% @ 6M steps

- Some slides courtesy of Aref Jafari

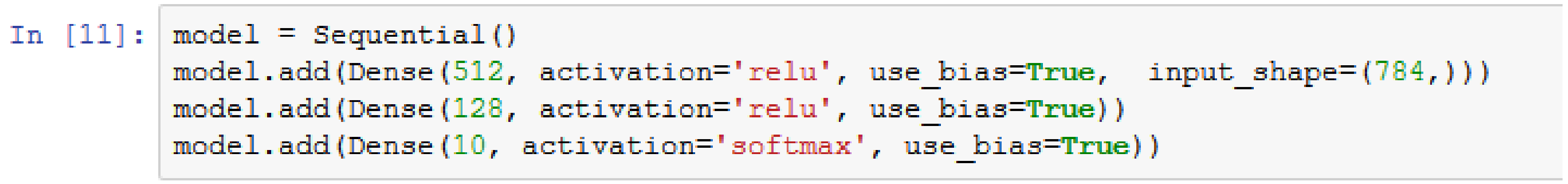
Step 1) Import Libraries



```
import numpy as np
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten, Input
from keras.utils import np_utils

#Other types of layers
from keras.layers import LSTM
from keras.layers import Conv1D, Conv2D, Conv3D, MaxPooling2D

from keras.layers.normalization import BatchNormalization

import matplotlib.pyplot as plt
%matplotlib inline

np.random.seed(2017)
```

Step 3) Define model architecture

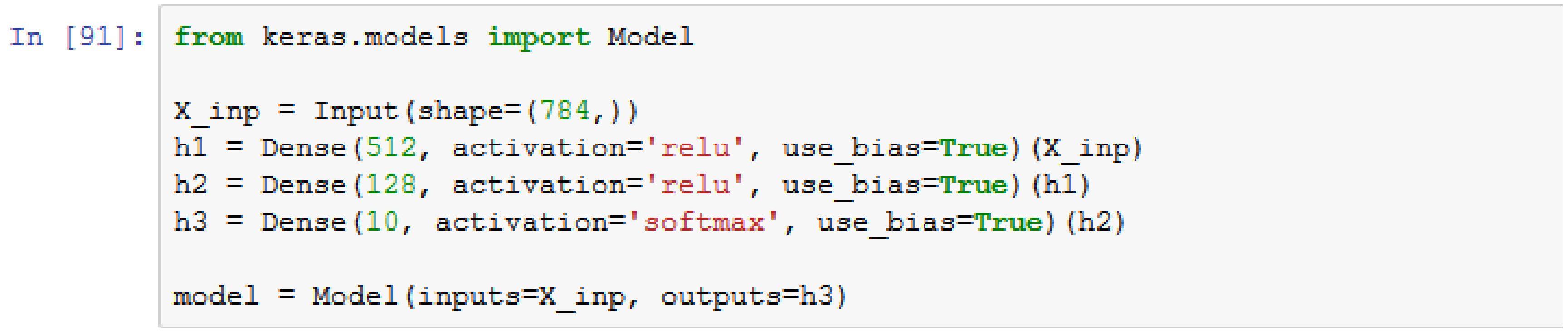
Form 1)



In [11]:

```
model = Sequential()
model.add(Dense(512, activation='relu', use_bias=True,  input_shape=(784,)))
model.add(Dense(128, activation='relu', use_bias=True))
model.add(Dense(10, activation='softmax', use_bias=True))
```

Form 2)



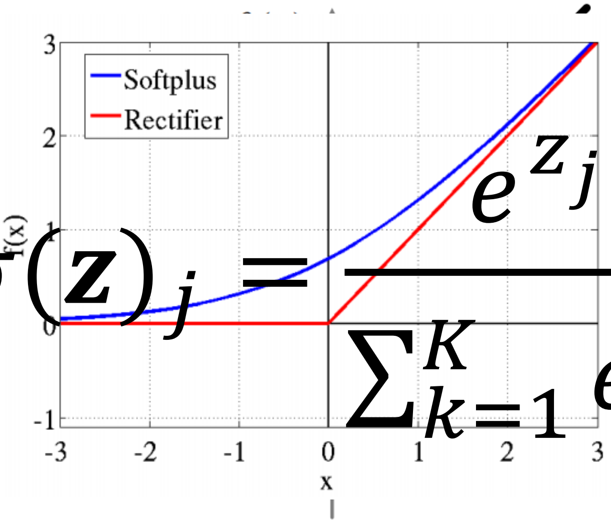
In [91]:

```
from keras.models import Model

X_inp = Input(shape=(784,))
h1 = Dense(512, activation='relu', use_bias=True)(X_inp)
h2 = Dense(128, activation='relu', use_bias=True)(h1)
h3 = Dense(10, activation='softmax', use_bias=True)(h2)

model = Model(inputs=X_inp, outputs=h3)
```

Step 3) Define model architecture (Alternatives for activation)

```
model.add(Dense(128, activation='relu', use_bias=True))
```

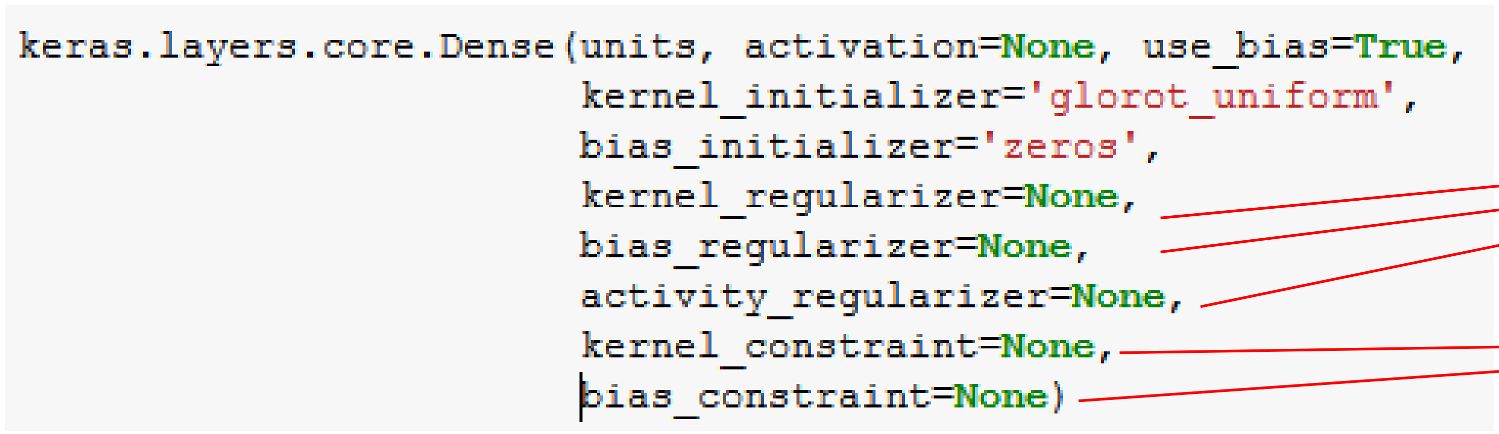
[illegible]



[illegible]

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \; for \; j = 1, \ldots, K$$

Step 3) Define model architecture (Other attributes of Dense layer)

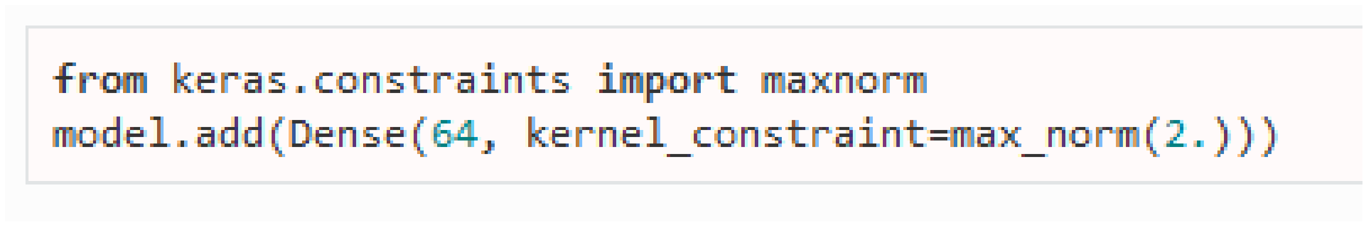


```
keras.layers.core.Dense(units, activation=None, use_bias=True,
                        kernel_initializer='glorot_uniform',
                        bias_initializer='zeros',
                        kernel_regularizer=None,
                        bias_regularizer=None,
                        activity_regularizer=None,
                        kernel_constraint=None,
                        bias_constraint=None)
```

instances of **keras.regularizers.Regularizer** (l1, l2, ...)

Functions from the **constraints** module allow setting constraints (eg. non-negativity) on network parameters during optimization

Example:



```
from keras.constraints import maxnorm
model.add(Dense(64, kernel_constraint=max_norm(2.)))
```

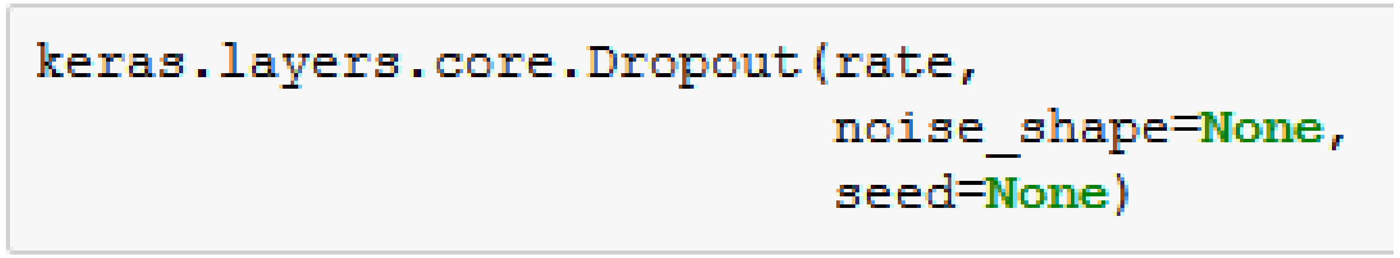
Available constraints

max_norm(max_value=2, axis=0): maximum-norm constraint

non_neg(): non-negativity constraint

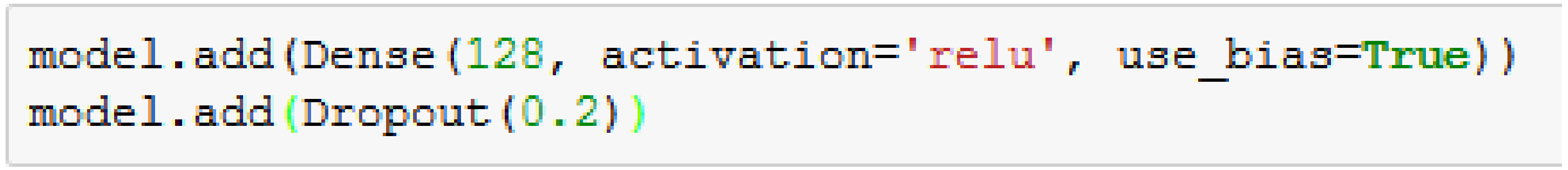
unit_norm(): unit-norm constraint, enforces the matrix to have unit norm along the last axis

Step 3) Define model architecture (Dropout Layers)



```
keras.layers.core.Dropout(rate,
                          noise_shape=None,
                          seed=None)
```

Example:

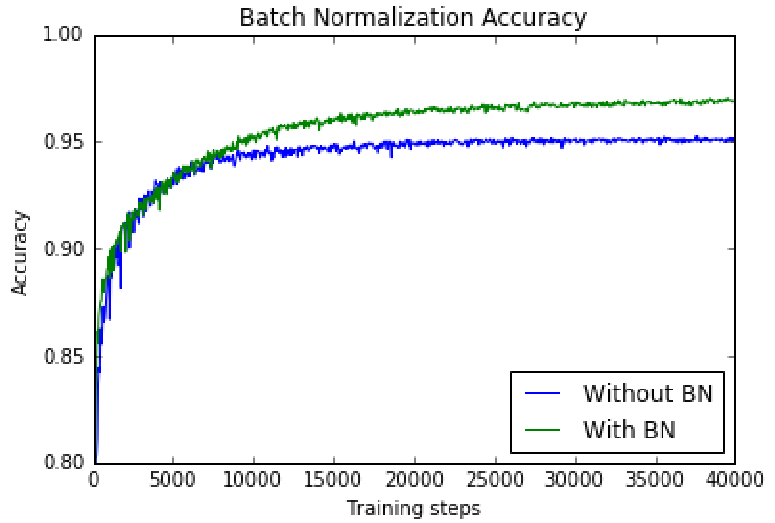


```
model.add(Dense(128, activation='relu', use_bias=True))
model.add(Dropout(0.2))
```

Step 3) Define model architecture (Batch Normalization Layers)

Example:

```
model = Sequential()
model.add(Dense(64, input_dim=14))
model.add(BatchNormalization())
model.add(Activation('tanh'))
model.add(Dropout(0.5))
```



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

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

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

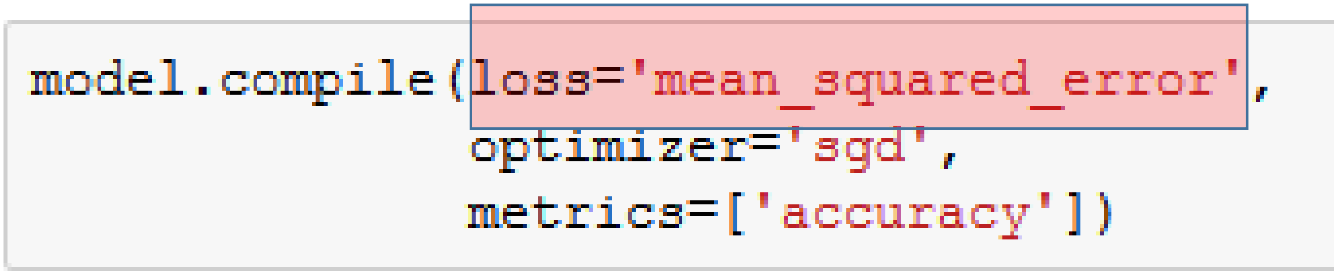
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

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

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

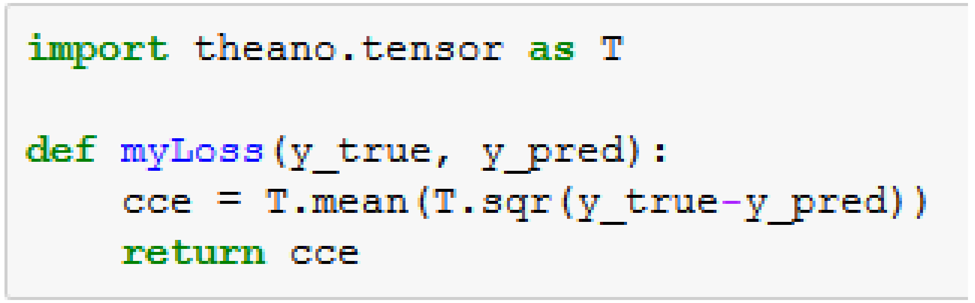
Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Step 4) Compile model (Loss functions)

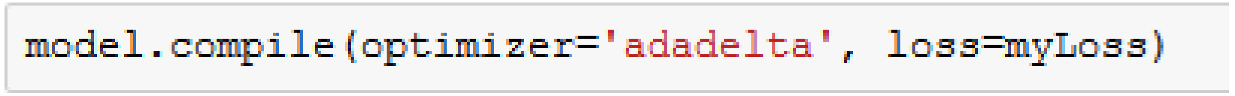


```
model.compile(loss='mean_squared_error',
              optimizer='sgd',
              metrics=['accuracy'])
```

Custom loss function



```
import theano.tensor as T

def myLoss(y_true, y_pred):
    cce = T.mean(T.sqr(y_true-y_pred))
    return cce
```

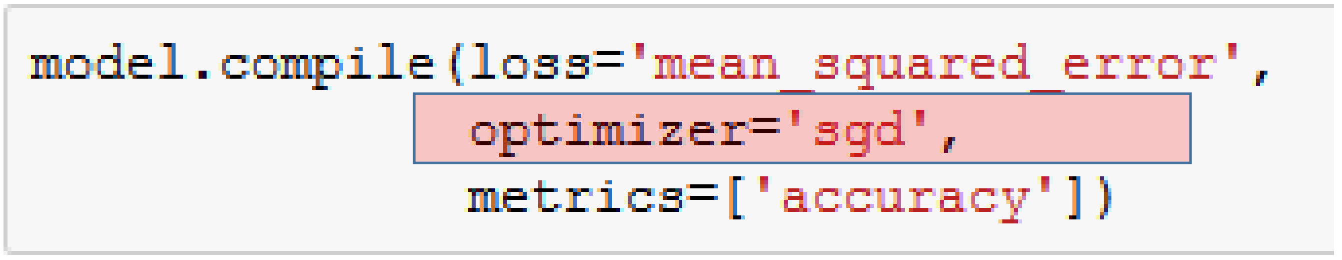


```
model.compile(optimizer='adadelta', loss=myLoss)
```

Available loss functions:

- mean_squared_error
- mean_absolute_error
- mean_absolute_percentage_error
- mean_squared_logarithmic_error
- squared_hinge
- hinge
- categorical_hinge
- logcosh
- categorical_crossentropy
- sparse_categorical_crossentropy
- binary_crossentropy
- kullback_leibler_divergence
- poisson
- cosine_proximity

Step 4) Compile model (Optimizers)

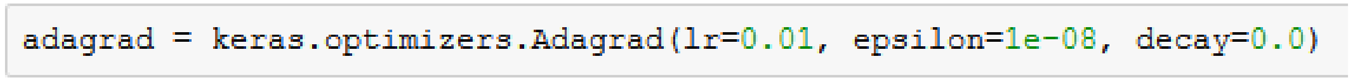


```
model.compile(loss='mean_squared_error',
              optimizer='sgd',
              metrics=['accuracy'])
```

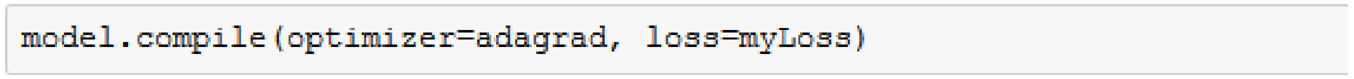
Available loss functions:

- **SGD**
- **RMSprop**
- **Adagrad**
- **Adadelta**
- **Adam**
- **Adamax**
- **Nadam**
- **TFOptimizer**

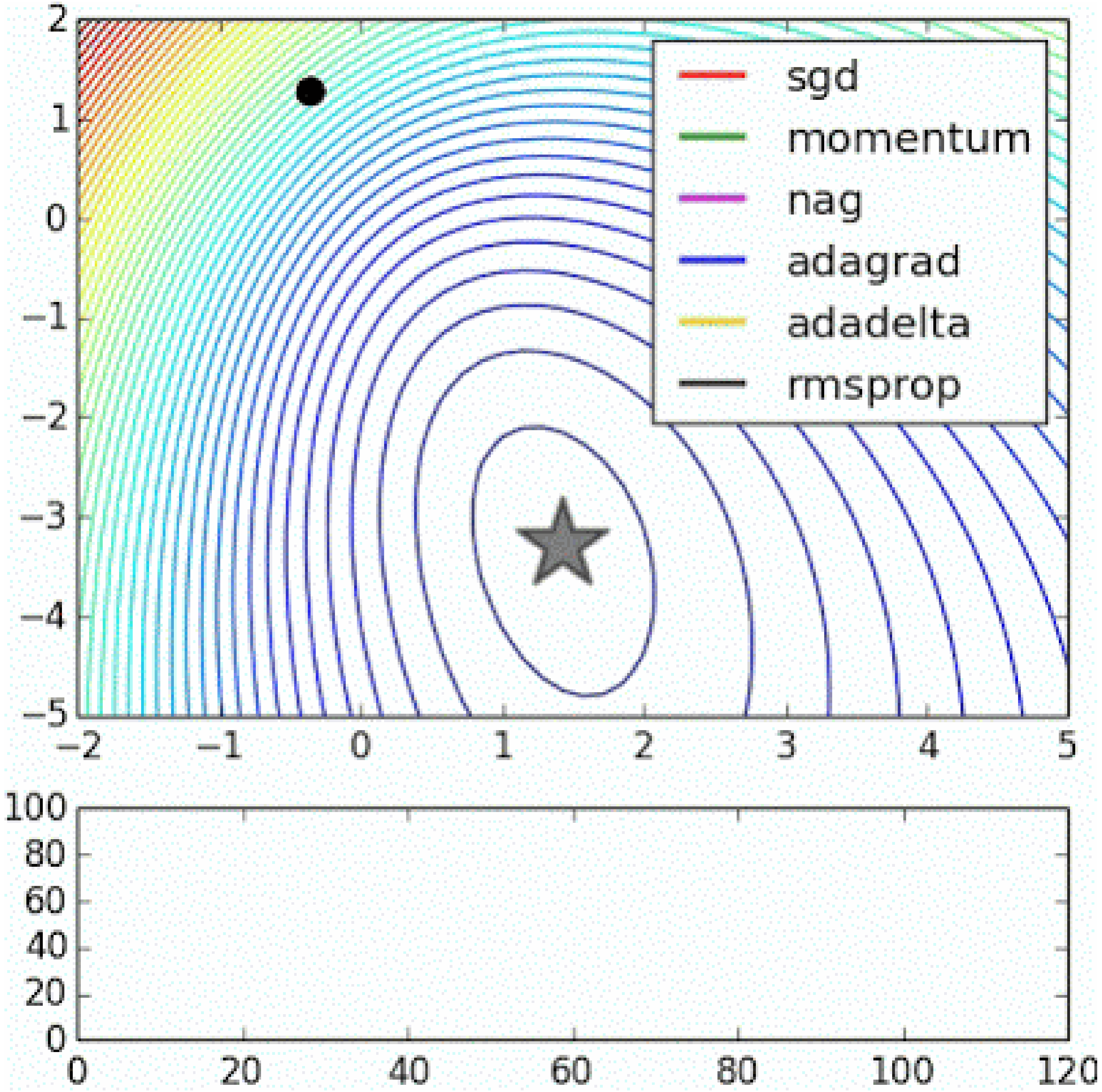
Adagrad



```
adagrad = keras.optimizers.Adagrad(lr=0.01, epsilon=1e-08, decay=0.0)
```



```
model.compile(optimizer=adagrad, loss=myLoss)
```


sgd
momentum
nag
adagrad
adadelta
rmsprop
2
1
0
−1
−2
−3
−4
−5
−2
−1
0
1
2
3
4
5
100
80
60
40
20
0
0
20
40
60
80
100
120

Deep Learning

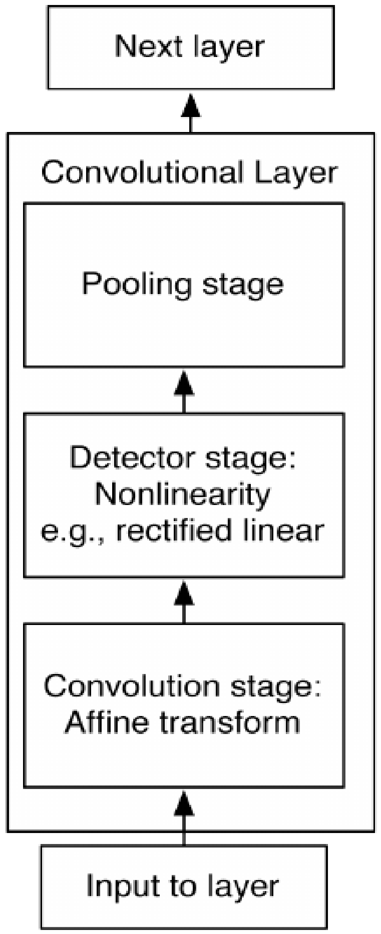
Convolutional Neural Network (CNNs)

Slides are partially based on Book, Deep Learning

by Bengio, Goodfellow, and Aaron Courville, 2015

Convolutional Networks

Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.


Next layer
Convolutional Layer
Pooling stage
Detector stage: Nonlinearity e.g., rectified linear
Convolution stage: Affine transform
Input to layer

Convolution

This operation is called convolution.

$$s(t) = \int x(a)w(t-a)da$$

The convolution operation is typically denoted with an asterisk:

$$s(t) = (x * w)(t)$$

Discrete convolution

If we now assume that *x* and *w* are defined only on integer *t*, we can define the discrete convolution:

$$s[t] = (x * w)(t) = \sum_{a=-\infty}^{\infty} x[a]w[t-a]$$

In practice

we often use convolutions over more than one axis at a time.

$$s[i,j] = (I * K)[i,j] = \sum_m \sum_n I[m,n]K[i-m, j-n]$$

The input is usually a multidimensional array of data.

The kernel is usually a multidimensional array of parameters that should be learned.

we assume that these functions are zero everywhere but the finite set of points for which we store the values.

we can implement the infinite summation as a summation over a finite number of array elements.

convolution and cross-correlation

convolution is commutative

$$s[i,j] = (I * K)[i,j] = \sum_m \sum_n I[i-m, j-n] K[m,n]$$

Cross-correlation,

$$s[i,j] = (I * K)[i,j] = \sum_m \sum_n I[i+m, j+n] K[m,n]$$

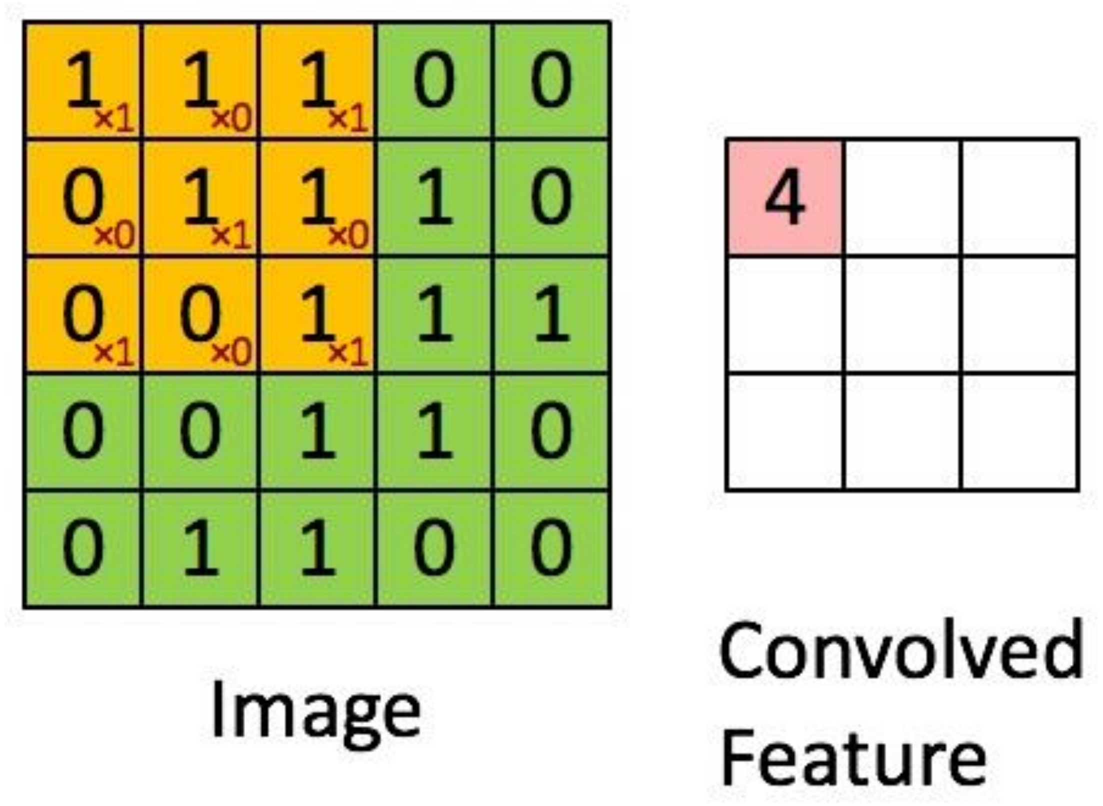
Many machine learning libraries implement cross-correlation but call it convolution.

https://www.youtube.com/watch?v=Ma0YONjMZLI

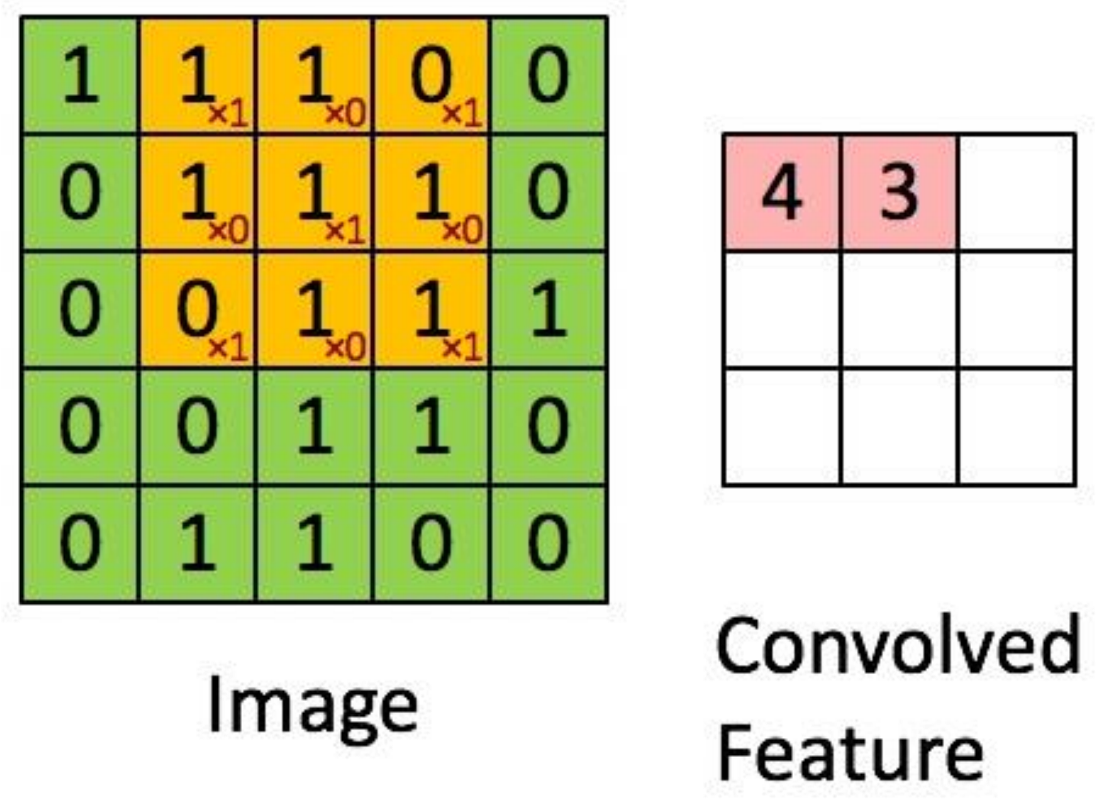
Fig 9.1

Discrete convolution can be viewed as multiplication by a matrix.

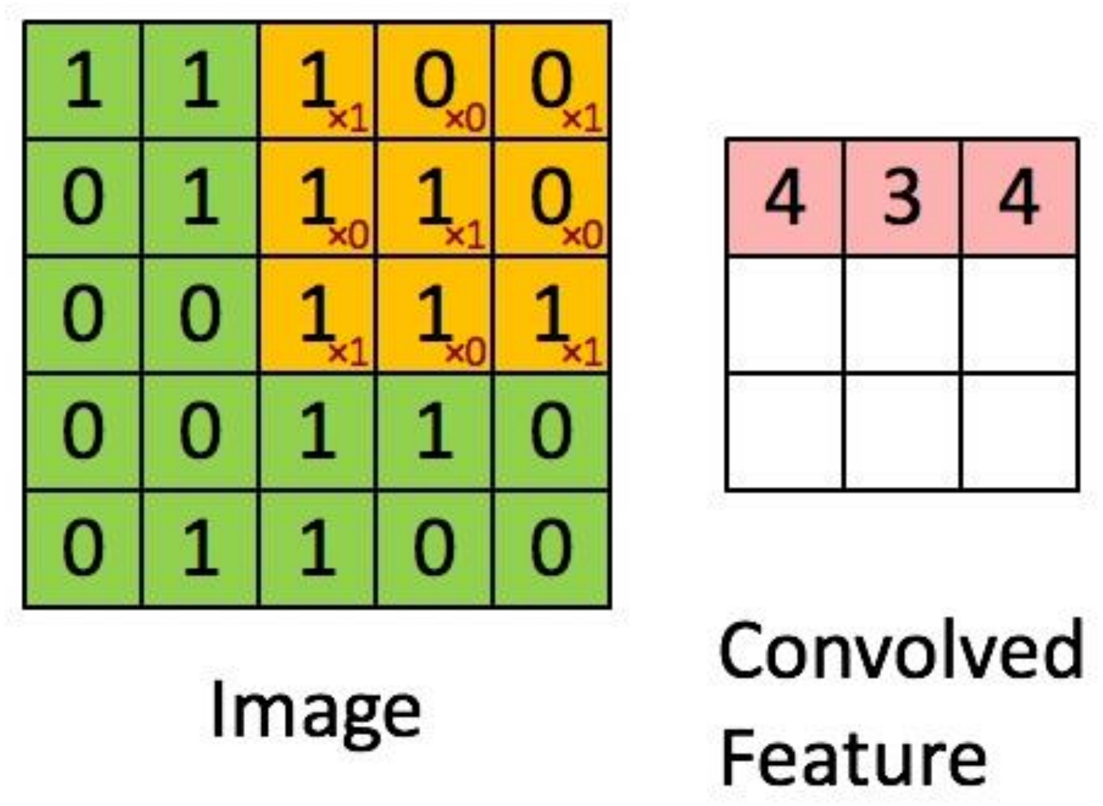
Convolutions



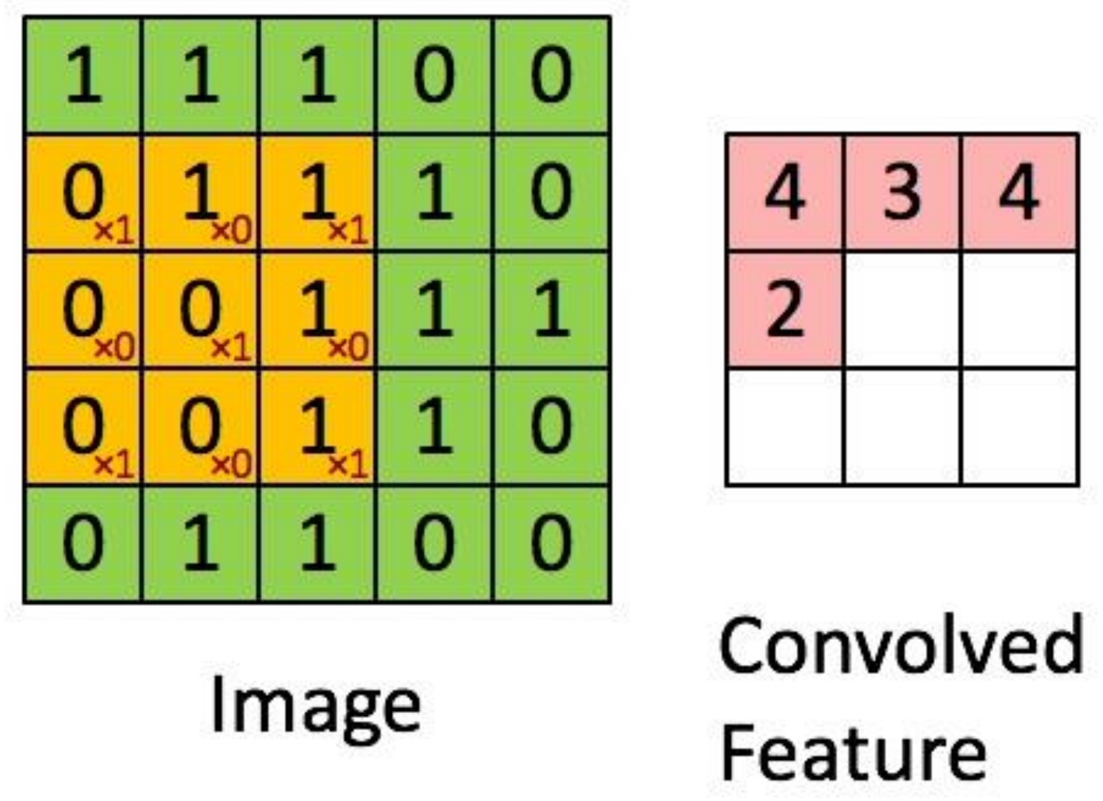
Convolutions



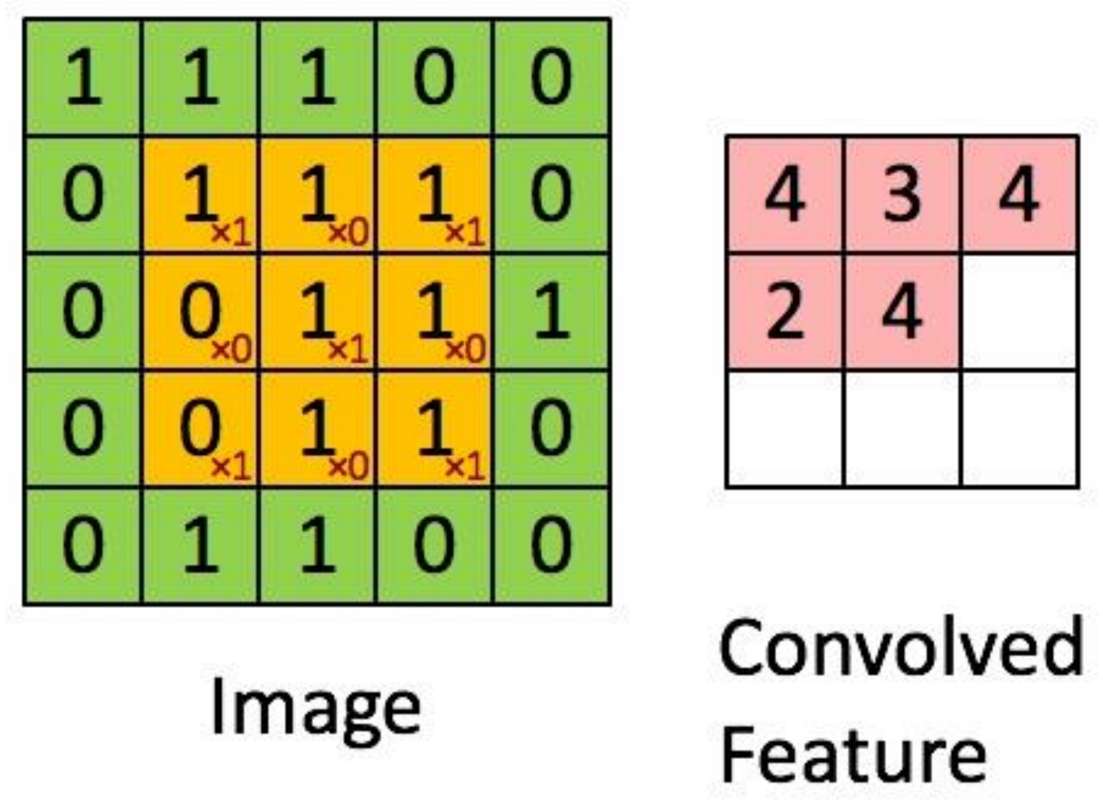
Convolutions



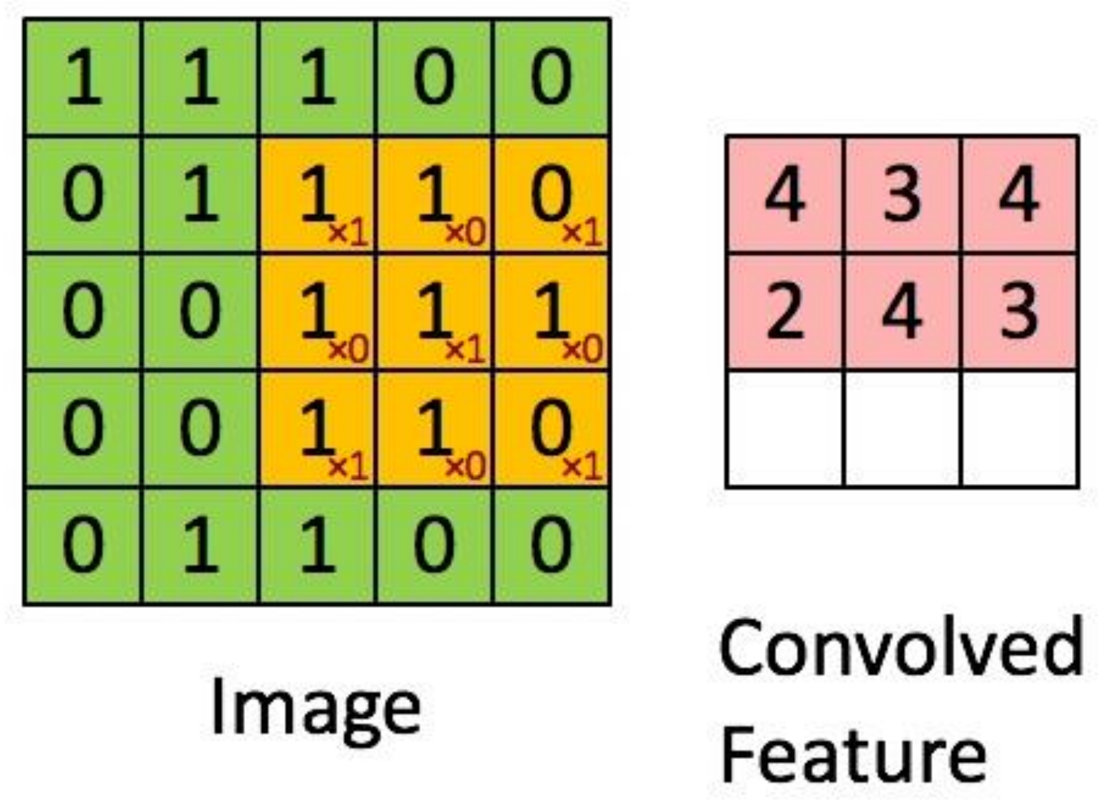
Convolutions



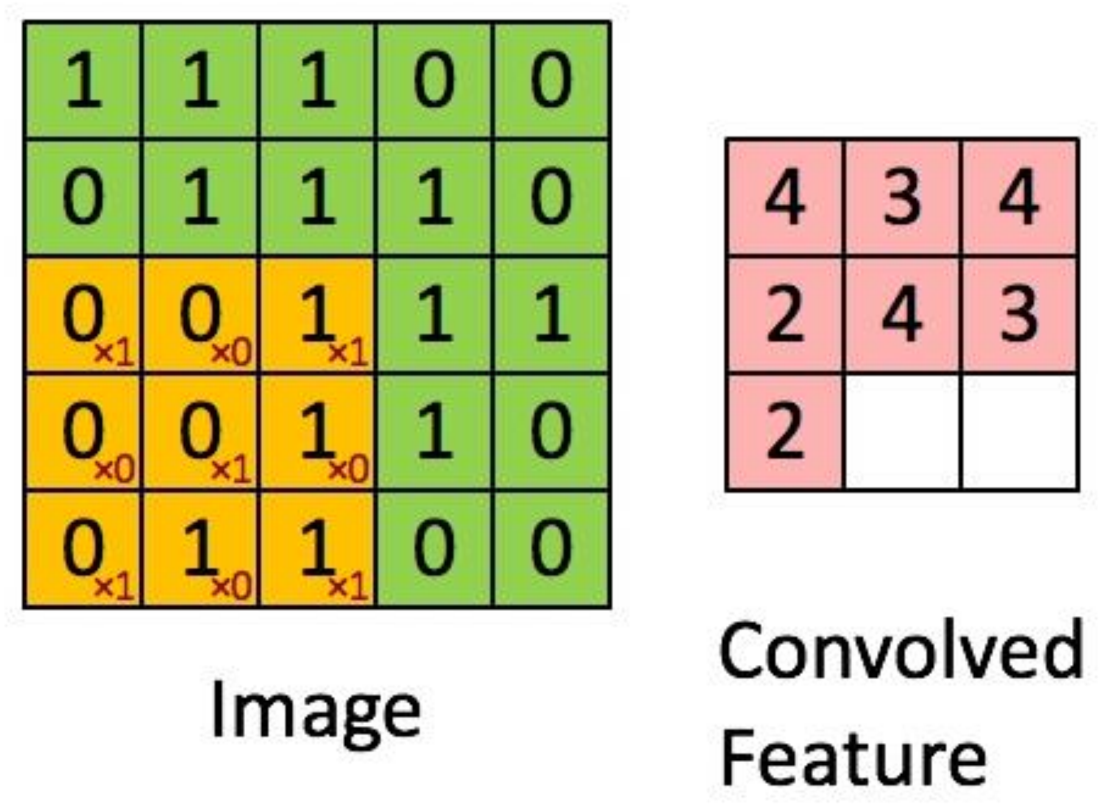
Convolutions



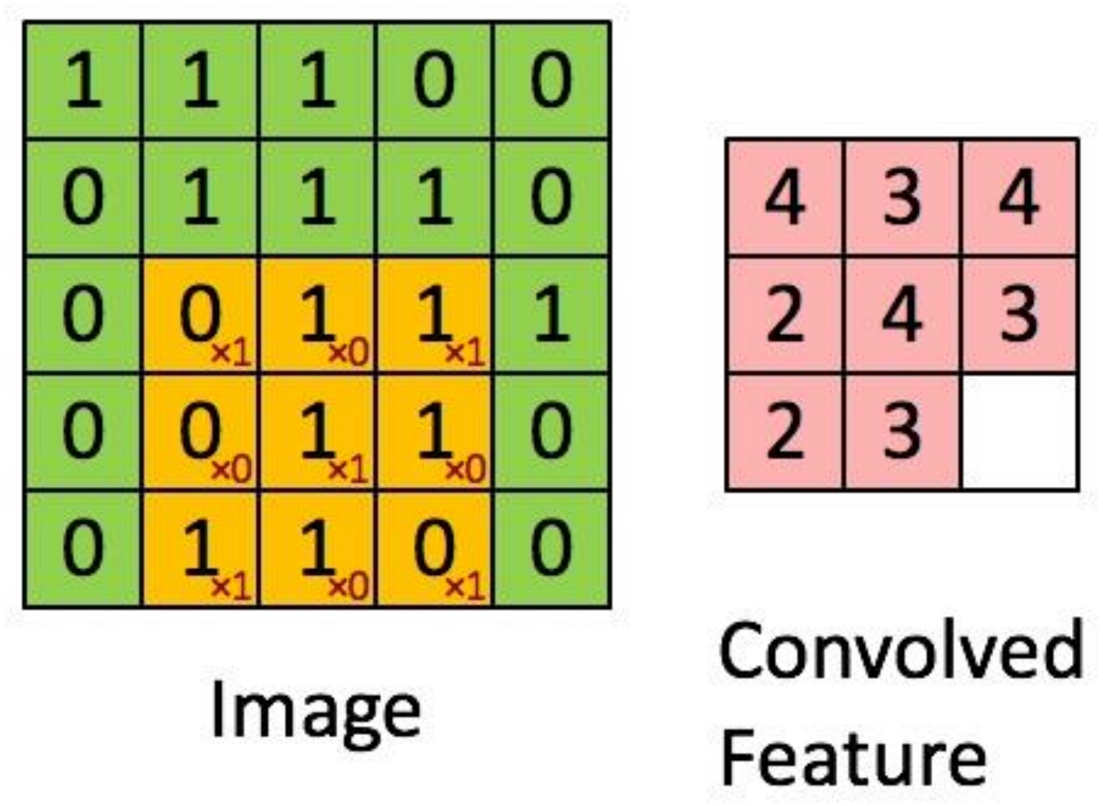
Convolutions



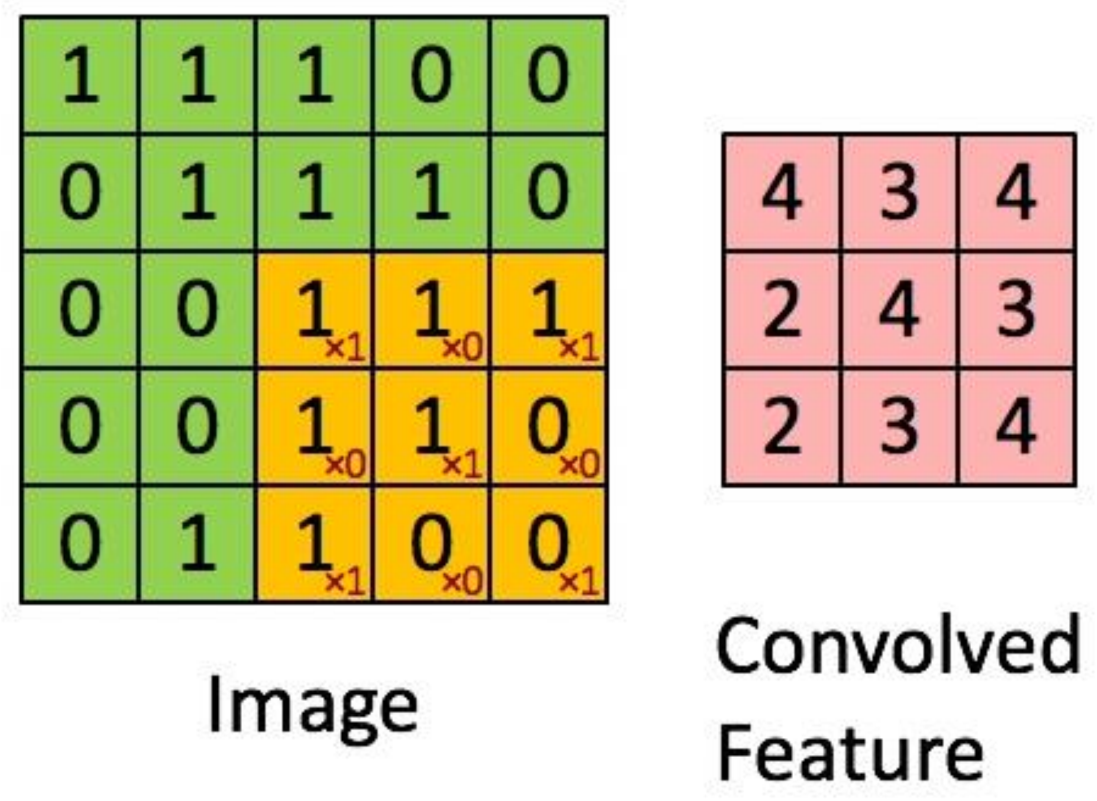
Convolutions



Convolutions



Convolutions



Sparse interactions

In feed forward neural network every output unit interacts with every input unit.

Convolutional networks, typically have sparse connectivity (sparse weights)

This is accomplished by making the kernel smaller than the input

Sparse interactions

When we have m inputs and n outputs, then matrix multiplication requires $m \times n$ parameters. and the algorithms used in practice have $O(m \times n)$ runtime (per example).

limit the number of connections each output may have to k, then requires only $k \times n$ parameters and $O(k \times n)$ runtime.

Parameter sharing

In a traditional neural net, each element of the weight matrix is multiplied by one element of the input. i.e. It is used once when computing the output of a layer.

In CNNs each member of the kernel is used at every position of the input

Instead of learning a separate set of parameters for every location, we learn only one set.

Equivariance

A function $f(x)$ is equivariant to a function g if $f(g(x)) = g(f(x))$.

$$\begin{tikzcd}
X \arrow[r, "g"] \arrow[d, "f"'] & X \arrow[d, "f"] \\
Y \arrow[r, "g"'] & Y
\end{tikzcd}$$

Equivariance

A convolutional layer have equivariance to translation.
For example

$$g(x)[i] = x[i-1]$$

If we apply this transformation to *x*, then apply convolution, the result will be the same as if we applied convolution to *x*, then applied the transformation to the output.

Equivariance

For images, convolution creates a 2-D map of where certain features appear in the input.

Note that convolution is not equivariant to some other transformations, such as changes in the scale or rotation of an image.

Convolutional Networks

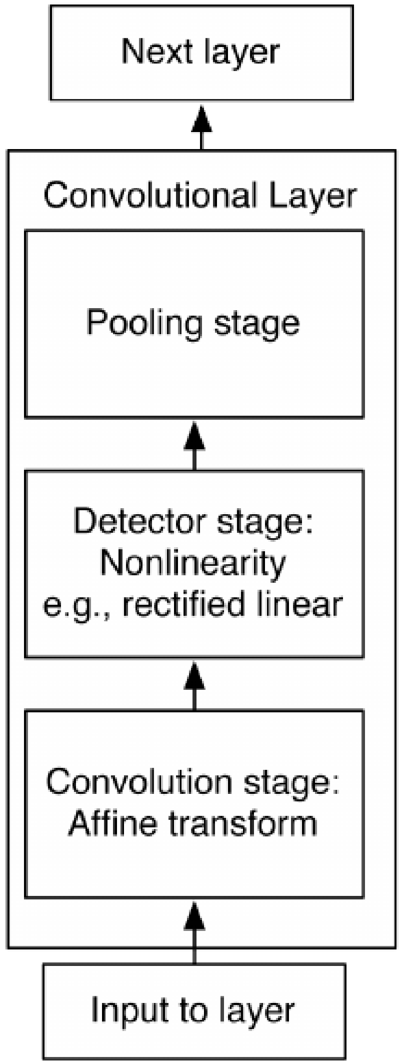
The first stage (Convolution):

The layer performs several convolutions in parallel to produce a set of preactivations.

The second stage (Detector):

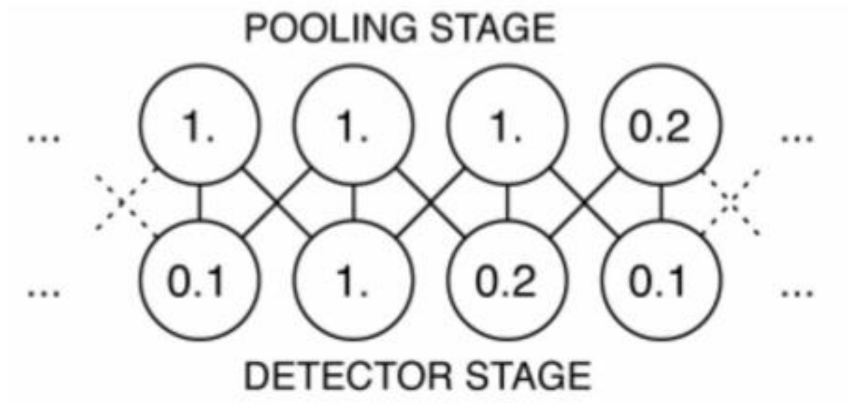
Each preactivation is run through a nonlinear activation function (e.g. rectified linear).

The third stage (Pooling)


Next layer
Convolutional Layer
Pooling stage
Detector stage: Nonlinearity e.g., rectified linear
Convolution stage: Affine transform
Input to layer

Popular Pooling functions

The maximum of a rectangular neighborhood (Max pooling operation)

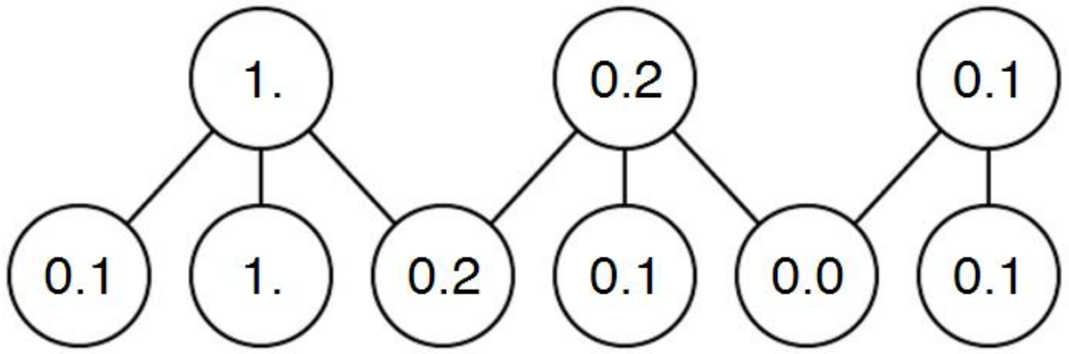


The average of a rectangular neighborhood.

The L2 norm of a rectangular neighborhood.

A weighted average based on the distance from the central pixel.

Pooling with downsampling



Max-pooling with a pool width of 3 and a stride between pools of 2. This reduces the representation size by a factor of 2,which reduces the computational and statistical burden on the next layer.