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

\[ a \cdot x + b \]

\[ x \]
Batch Normalization

$ax + b$

$x$
Batch Normalization

\[ a x + b \]

\[ x \]
Effect of changing input distribution

- Careful initialization
- Small learning rates
- Rectifiers
Internal covariate shift

- Layer input distributions change during training

\[ \ell = F_2(F_1(u, \Theta_1), \Theta_2) \]

- Change in internal activation distribution requires domain adaptation
Normalize each activation:

\[ x \mapsto \frac{x - \mathbb{E}[x]}{\sqrt{\text{Var}[x]}} \]
Mini-batch mean:

$$\mu_B \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i$$

Mini-batch variance:

$$\sigma_B^2 \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2$$

Normalize:

$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

Scale and shift:

$$y_i \leftarrow \gamma \hat{x}_i + \beta$$
- Replace batch statistics with population statistics

\[
\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \quad \xrightarrow{} \quad \hat{x} \leftarrow \frac{x - \mathbb{E}[x]}{\sqrt{\text{Var}[x] + \epsilon}}
\]
- MNIST: 3 FC layers + softmax, 100 logistic units per hidden layer
- Distribution of inputs to a typical sigmoid, evolving over 100k steps:
● 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

```python
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)

```python
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)

```python
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)

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

$$
\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K
$$
Step 3) Define model architecture
(Other attributes of Dense layer)

```python
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`

Functions from the `constraints` module allow setting constraints (e.g. non-negativity) on network parameters during optimization.

**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

```python
from keras.constraints import maxnorm
model.add(Dense(64, kernel_constraint=maxnorm(2.)))
```
Step 3) Define model architecture (Dropout Layers)

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

Example:
```python
model.add(Dense(128, activation='relu', use_bias=True))
model.add(Dropout(0.2))
```
Step 3) Define model architecture
(Batch Normalization Layers)

Example:

```python
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: $\mathcal{B} = \{x_1, \ldots, x_m\}$; Parameters to be learned: $\gamma, \beta$

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

- $\mu_B \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i$ \quad // mini-batch mean
- $\sigma_B^2 \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2$ \quad // mini-batch variance
- $\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$ \quad // normalize
- $y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i)$ \quad // scale and shift

**Algorithm 1:** Batch Normalizing Transform, applied to activation $x$ over a mini-batch.
Step 4) Compile model
(Loss functions)

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

Custom loss function

```python
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)
```
Step 4) Compile model (Optimizers)

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

**Available loss functions:**
- SGD
- RMSprop
- Adagrad
- Adadelta
- Adam
- Adamax
- Nadam
- TFOptimizer

**Adagrad**

```python
adagrad = keras.optimizers.Adagrad(lr=0.01, epsilon=1e-08, decay=0.0)
model.compile(optimizer=adagrad, loss=myLoss)
```
Deep Learning

Convolutional Neural Network (CNNs)

Slides are partially based on Book, Deep Learning

by Bengio, Goodfellow, and Aaron Courville, 2015
Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.
Convolution

This operation is called convolution.

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

The convolution operation is typically denoted with an asterisk:

\[ s(t) = (x \ast w)(t) \]
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

\[ s[i, j] = (I \ast K)[i, j] = \sum_{m} \sum_{n} I[i - m, j - n]K[m, n] \]

Cross-correlation,

\[ s[i, j] = (I \ast K)[i, j] = \sum_{m} \sum_{n} I[i + m, j + n]K[m, n] \]

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

Image

Convolved Feature
### Convolutions

**Image**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Convolved Feature**

4 3
Convolutions

Image

Convolved Feature
Convolutions

Image

Convolved Feature
Convolutions

Image

Convolved Feature
Convolutions

Image

Convolved Feature
Convolutions

Image

Convolved Feature

1 1 1 0 0
0 1 1 1 0
0 0 1 1 1
0 0 1 1 0
0 1 1 0 0

4 3 4
2 4 3
2 3
Convolutions

Image

Convolved Feature
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 $f(g(x)) = g(f(x))$. 
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)
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.