Going deeper
For a fixed parameter budget deeper is better

Figure 1: Binary classification using a shallow model with 20 hidden units (solid line) and a deep model with two layers of 10 units each (dashed line). The right panel shows a close-up of the left panel. Filled markers indicate errors made by the shallow model.

*On the number of linear regions of deep neural networks; Montufar et al., 2014*
Dropout

- First "deep" regularization technique
- Remove units at random during the forward pass on each sample
- Put them all back during test

Dropout: A Simple Way to Prevent Neural Networks from Overfitting, Srivastava et al., JMLR 2014
Dropout

Interpretation

- Reduces the network dependency to individual neurons and distributes representation
- More redundant representation of data

Ensemble interpretation

- Equivalent to training a large ensemble of shared-parameters, binary-masked models
- Each model is only trained on a single data point
- A network with dropout can be interpreted as an ensemble of $2^N$ models with heavy weight sharing (Goodfellow et al., 2013)
Dropout

- One has to decide on which units/layers to use dropout, and with what probability $p$ units are dropped.
- During training, for each sample, as many Bernoulli variables as units are sampled independently to select units to remove.
- To keep the means of the inputs to layers unchanged, the initial version of dropout was multiplying activations by $p$ during test.
- The standard variant is the "inverted dropout": multiply activations by $\frac{1}{1-p}$ during training and keep the network untouched during test.
Dropout

Overfitting noise

MLP with 3 hidden layers and noisy labels

- train, no dropout
- validation, no dropout

Slide credit: C. Ollion & O. Grisel, M2DS Deep Learning
Dropout

A bit of Dropout

MLP with 3 hidden layers and noisy labels

- train, no dropout
- validation, no dropout
- train, dropout p=0.2
- validation, dropout p=0.2

Slide credit: C. Ollion & O. Grisel, M2DS Deep Learning
Dropout

Too much: underfitting

MLP with 3 hidden layers and noisy labels

- train, no dropout
- validation, no dropout
- train, dropout p=0.8
- validation, dropout p=0.8

Slide credit: C. Ollion & O. Grisel, M2DS Deep Learning
Features learned on MNIST with one hidden layer autoencoders having 256 rectified linear units

*Dropout: A Simple Way to Prevent Neural Networks from Overfitting, Srivastava et al., JMLR 2014*
>>> x = Variable(torch.Tensor(3, 9).fill_(1.0), requires_grad = True)
>>> x.data
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
[torch.FloatTensor of size 3x9]

>>> dropout = nn.Dropout(p = 0.75)
>>> y = dropout(x)
>>> y.data
4 0 4 4 4 0 4 0 0
4 0 0 0 0 0 0 0 0
0 0 0 0 4 0 4 0 4
[torch.FloatTensor of size 3x9]

>>> l = y.norm(2, 1).sum()
>>> l.backward()
>>> x.grad.data
1.7889 0.0000 1.7889 1.7889 0.0000 0.0000 1.7889 0.0000 0.0000
0.0000 0.0000 0.0000 1.7889 0.0000 0.0000 0.0000 2.3094 0.0000
0.0000 0.0000 0.0000 0.0000 2.3094 0.0000 0.0000 0.0000 2.3094
[torch.FloatTensor of size 3x9]
Dropout

For a given network

```python
model = nn.Sequential(nn.Linear(10, 100), nn.ReLU(),
                      nn.Linear(100, 50), nn.ReLU(),
                      nn.Linear(50, 2));
```
For a given network

```python
model = nn.Sequential(nn.Linear(10, 100), nn.ReLU(),
                      nn.Linear(100, 50), nn.ReLU(),
                      nn.Linear(50, 2));
```

we can simply add dropout layers

```python
model = nn.Sequential(nn.Linear(10, 100), nn.ReLU(),
                      nn.Dropout(),
                      nn.Linear(100, 50), nn.ReLU(),
                      nn.Dropout(),
                      nn.Linear(50, 2));
```
Dropout

A model using dropout has to be set in "train" or "test" mode
Dropout

A model using dropout has to be set in "train" or "test" mode

The method \texttt{nn.Module.train(mode)} recursively sets the flag training to all sub-modules.

```python
>>> dropout = nn.Dropout()
>>> model = nn.Sequential(nn.Linear(3, 10), dropout, nn.Linear(10, 3))
>>> dropout.training
True
>>> model.train(False)
Sequential (
(0): Linear (3 -> 10) (1): Dropout (p = 0.5) (2): Linear (10 -> 3)
)
>>> dropout.training
False
```
Spatial Dropout

As pointed out by Tompson et al. (2015), units in a 2d activation map are generally locally correlated, and dropout has virtually no effect.

They proposed SpatialDropout, which drops channels instead of individual units.
Spatial Dropout

```python
>>> dropout2d = nn.Dropout2d()
>>> x = Variable(Tensor(2, 3, 2, 2).fill_(1.0))
>>> dropout2d(x)
Variable containing:
(0 ,0 ,..,)=
 0
 0
 0

(0 ,1 ,..,)=
 0
 0
 0

(0 ,2 ,..,)=
 2
 2
 2

(1 ,0 ,..,)=
 2
 2
 2

(1 ,1 ,..,)=
 0
 0
 0

(1 ,2 ,..,)=
```
Batch normalization

We saw that maintaining proper statistics of the activations and derivatives was a critical issue to allow the training of deep architectures.

It is the main motivation behind weight initialization rules (we'll cover them later).
Batch normalization

We saw that maintaining proper statistics of the activations and derivatives was a critical issue to allow the training of deep architectures.

It is the main motivation behind weight initialization rules (we'll cover them later).

A different approach consists of explicitly forcing the activation statistics during the forward pass by re-normalizing them.

**Batch normalization** proposed by Ioffe and Szegedy (2015) was the first method introducing this idea.
Batch normalization

Normalize activations in each mini-batch before activation function: speeds up and stabilizes training (less dependent on init)

Batch normalization forces the activation first and second order moments, so that the following layers do not need to adapt to their drift.
Batch normalization

Normalize activations in each mini-batch before activation function: speeds up and stabilizes training (less dependent on init)

Input: Values of $x$ over a mini-batch: $\mathcal{B} = \{x_1 \ldots m\}$; Parameters to be learned: $\gamma$, $\beta$

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

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

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

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

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

Batch normalization: Accelerating deep network training by reducing internal covariate shift, Ioffe and Szegedy, ICML 2015
Batch normalization

During training batch normalization shifts and rescales according to the mean and variance estimated on the batch.

\[
\text{Input: Values of } x \text{ over a mini-batch: } B = \{x_1 \ldots m\}; \\
\text{Parameters to be learned: } \gamma, \beta \\
\text{Output: } \{y_i = \text{BN}_{\gamma,\beta}(x_i)\}
\]

\[
\mu_B \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i \quad \text{// mini-batch mean}
\]

\[
\sigma_B^2 \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2 \quad \text{// mini-batch variance}
\]

\[
\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \quad \text{// normalize}
\]

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

As for dropout, the model behaves differently during train and test.
Batch normalization

At **inference time**, use average and standard deviation computed on the **whole dataset** instead of batch.

Widely used in **ConvNets**, but requires the mini-batch to be large enough to compute statistics in the minibatch.
Batch normalization

As dropout, batch normalization is implemented as a separate module torch.BatchNorm1d that processes the input components separately.

```python
>>> x = torch.Tensor(10000, 3).normal_()
>>> x = x * torch.Tensor([2, 5, 10]) + torch.Tensor([-10, 25, 3])
>>> x = Variable(x)
>>> x.data.mean(0)
  -9.9898
  24.9165
  2.8945
  [torch.FloatTensor of size 3]

>>> x.data.std(0)
  2.0006
  5.0146
  9.9501
  [torch.FloatTensor of size 3]
```
Batch normalization

Since the module has internal variables to keep statistics, it must be provided with the sample dimension at creation.

```python
>>> bn = nn.BatchNorm1d(3)
>>> bn.bias.data = torch.Tensor([2, 4, 8])
>>> bn.weight.data = torch.Tensor([1, 2, 3])
>>> y = bn(x)
>>> y.data.mean(0)

2.0000
4.0000
8.0000
[torch.FloatTensor of size 3]

>>> y.data.std(0)

1.0000
2.0001
3.0001
[torch.FloatTensor of size 3]
```
Batch normalization

BatchNorm2d example

```python
>>> x = Variable(torch.randn(20, 100, 35, 45))
>>> bn2d = nn.BatchNorm2d(100)
>>> y = bn2d(x)
>>> x.size()

torch.Size([20, 100, 35, 45])
```
Batch normalization

Results on ImageNet LSVRC 2012:

![Graph showing validation accuracy vs. number of training steps for different models.](image)

<table>
<thead>
<tr>
<th>Model</th>
<th>Steps to 72.2%</th>
<th>Max accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inception</td>
<td>$3.1 \cdot 10^5$</td>
<td>72.2%</td>
</tr>
<tr>
<td>BN-Baseline</td>
<td>$13.3 \cdot 10^3$</td>
<td>72.7%</td>
</tr>
<tr>
<td>BN-x5</td>
<td>$2.1 \cdot 10^3$</td>
<td>73.0%</td>
</tr>
<tr>
<td>BN-x30</td>
<td>$2.7 \cdot 10^3$</td>
<td>74.8%</td>
</tr>
<tr>
<td>BN-x5-Sigmoid</td>
<td></td>
<td>69.8%</td>
</tr>
</tbody>
</table>

Figure 2: Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.

Figure 3: For Inception and the batch-normalized variants, the number of training steps required to reach the maximum accuracy of inception (72.2%), and the maximum accuracy achieved by the network.

**Batch normalization: Accelerating deep network training by reducing internal covariate shift**, Ioffe and Szegedy, *ICML 2015*
Batch normalization

Results on ImageNet LSVRC 2012:

- learning rate can be greater
- dropout and local normalization are not necessary
- $L^2$ regularization influence should be reduced

Batch normalization: Accelerating deep network training by reducing internal covariate shift, Ioffe and Szegedy, ICML 2015
Batch normalization

Deep MLP on a 2d "disc" toy example, with naive Gaussian weight initialization, cross-entropy, standard SGD, $\eta = 0.1$.

def create_model(with_batchnorm, nc = 32, depth = 16):
    modules = []

    modules.append(nn.Linear(2, nc))
    if with_batchnorm: modules.append(nn.BatchNorm1d(nc))
    modules.append(nn.ReLU())

    for d in range(depth):
        modules.append(nn.Linear(nc, nc))
        if with_batchnorm: modules.append(nn.BatchNorm1d(nc))
        modules.append(nn.ReLU())

    modules.append(nn.Linear(nc, 2))

    return nn.Sequential(*modules)
Batch normalization

Slide credit: F. Fleuret, EE-559 Deep Learning
Layer Normalizations

Normalize on the statistics of the layer activations instead of mini-batch.

\[
\mu^l = \frac{1}{H} \sum_{i=1}^{H} a^l_i \quad \sigma^l = \sqrt{\frac{1}{H} \sum_{i=1}^{H} (a^l_i - \mu^l)^2}
\]
Layer Normalizations

Normalize on the statistics of the layer activations instead of mini-batch.

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

The algorithm is then similar as Batch Normalization

*Layer Normalization, Ba et al., 2016*
Layer Normalizations

Normalize on the statistics of the layer activations instead of mini-batch.

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

The algorithm is then similar as Batch Normalization.

Suit for RNNs, degrades performance of CNNs.

*Layer Normalization, Ba et al., 2016*
Weight Normalization

Reparametrize weights of neurons, to decouple direction and norm of the weight:

\[ w = \frac{g}{||v||} v \]
Weight Normalization

Reparametrize weights of neurons, to decouple direction and norm of the weight:

\[ w = \frac{g}{||v||} v \]

One new parameter \( g \) to learn per neuron.

Weight Normalization

Reparameterize weights of neurons, to decouple direction and norm of the weight:

\[ \mathbf{w} = \frac{g}{||\mathbf{v}||} \mathbf{v} \]

One new parameter \( g \) to learn per neuron

Careful data-based initialization of \( g \) and neuron bias \( b \) is better (not applicable to RNNs)

Multiple variants

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.
Architectures

torchvision.models provides a collection of reference networks for computer vision, e.g.:

```python
import torchvision
alexnet = torchvision.models.alexnet()
```

The trained models can be obtained by passing `pretrained = True` to the constructor(s). This may involve an heavy download given there size.
LeNet5

10 classes, input 1 x 28 x 28

(features): Sequential (
(0): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
(1): ReLU (inplace)
(2): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
(3): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
(4): ReLU (inplace)
(5): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
)
(classifier): Sequential (
(0): Linear (400 -> 120)
(1): ReLU (inplace)
(2): Linear (120 -> 84)
(3): ReLU (inplace)
(4): Linear (84 -> 10) )
AlexNet

First conv layer: kernel 11x11x3x96 stride 4

Imagenet classification with deep convolutional neural networks, Krizhevsky et al., NIPS 2012
First conv layer: kernel 11x11x3x96 stride 4

- Kernel shape: (11,11,3,96)
- Output shape: (55,55,96)
- Number of parameters: 34,944
- Equivalent MLP parameters: 43.7 x 1e9

*Imagenet classification with deep convolutional neural networks, Krizhevsky et al., NIPS 2012*
INPUT: [227x227x3]
CONV1: [55x55x96] 96 11x11 filters at stride 4, pad 0
MAX POOL1: [27x27x96] 3x3 filters at stride 2
CONV2: [27x27x256] 256 5x5 filters at stride 1, pad 2
MAX POOL2: [13x13x256] 3x3 filters at stride 2
CONV3: [13x13x384] 384 3x3 filters at stride 1, pad 1
CONV4: [13x13x384] 384 3x3 filters at stride 1, pad 1
CONV5: [13x13x256] 256 3x3 filters at stride 1, pad 1
MAX POOL3: [6x6x256] 3x3 filters at stride 2
FC6: [4096] 4096 neurons
FC7: [4096] 4096 neurons
FC8: [1000] 1000 neurons (softmax logits)

Slide credit: C. Ollion & O. Grisel, M2DS Deep Learning
(features): Sequential (
(0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
(1): ReLU (inplace)
(2): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))
(3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
(4): ReLU (inplace)
(5): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))
(6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(7): ReLU (inplace)
(8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(9): ReLU (inplace)
(10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(11): ReLU (inplace)
(12): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))
)

(classifier): Sequential (
(0): Dropout (p = 0.5)
(1): Linear (9216 -> 4096)
(2): ReLU (inplace)
(3): Dropout (p = 0.5)
(4): Linear (4096 -> 4096)
(5): ReLU (inplace)
(6): Linear (4096 -> 1000)
)
Hierarchical representation

Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]
VGG-16

Very deep convolutional networks for large-scale image recognition, Simonyan and Zisserman, NIPS 2014
### Memory and Parameters

<table>
<thead>
<tr>
<th>Activation maps</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT: [224x224x3] = 150K</td>
<td>0</td>
</tr>
<tr>
<td>CONV3-64: [224x224x64] = 3.2M</td>
<td>(3x3x3)x64 = 1,728</td>
</tr>
<tr>
<td>CONV3-64: [224x224x64] = 3.2M</td>
<td>(3x3x64)x64 = 36,864</td>
</tr>
<tr>
<td>POOL2: [112x112x64] = 800K</td>
<td>0</td>
</tr>
<tr>
<td>CONV3-128: [112x112x128] = 1.6M</td>
<td>(3x3x64)x128 = 73,728</td>
</tr>
<tr>
<td>CONV3-128: [112x112x128] = 1.6M</td>
<td>(3x3x128)x128 = 147,456</td>
</tr>
<tr>
<td>POOL2: [56x56x128] = 400K</td>
<td>0</td>
</tr>
<tr>
<td>CONV3-256: [56x56x256] = 800K</td>
<td>(3x3x128)x256 = 294,912</td>
</tr>
<tr>
<td>CONV3-256: [56x56x256] = 800K</td>
<td>(3x3x256)x256 = 589,824</td>
</tr>
<tr>
<td>CONV3-256: [56x56x256] = 800K</td>
<td>(3x3x256)x256 = 589,824</td>
</tr>
<tr>
<td>POOL2: [28x28x256] = 200K</td>
<td>0</td>
</tr>
<tr>
<td>CONV3-512: [28x28x512] = 400K</td>
<td>(3x3x256)x512 = 1,179,648</td>
</tr>
<tr>
<td>CONV3-512: [28x28x512] = 400K</td>
<td>(3x3x512)x512 = 2,359,296</td>
</tr>
<tr>
<td>CONV3-512: [28x28x512] = 400K</td>
<td>(3x3x512)x512 = 2,359,296</td>
</tr>
<tr>
<td>POOL2: [14x14x512] = 100K</td>
<td>0</td>
</tr>
<tr>
<td>CONV3-512: [14x14x512] = 100K</td>
<td>(3x3x512)x512 = 2,359,296</td>
</tr>
<tr>
<td>CONV3-512: [14x14x512] = 100K</td>
<td>(3x3x512)x512 = 2,359,296</td>
</tr>
<tr>
<td>CONV3-512: [14x14x512] = 100K</td>
<td>(3x3x512)x512 = 2,359,296</td>
</tr>
<tr>
<td>POOL2: [7x7x512] = 25K</td>
<td>0</td>
</tr>
<tr>
<td>FC: [1x1x4096] = 4096</td>
<td>7x7x512x4096 = 102,760,448</td>
</tr>
<tr>
<td>FC: [1x1x4096] = 4096</td>
<td>4096x4096 = 16,777,216</td>
</tr>
<tr>
<td>FC: [1x1x1000] = 1000</td>
<td>4096x1000 = 4,096,000</td>
</tr>
</tbody>
</table>

**TOTAL activations:** 2.4M x 4 bytes ~ 93MB / image (x2 for backward)
## Memory and Parameters

<table>
<thead>
<tr>
<th>Activation maps</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT:</td>
<td>224x224x3</td>
</tr>
<tr>
<td>CONV3-64:</td>
<td>[224x224x64] = 3.2M</td>
</tr>
<tr>
<td>CONV3-64:</td>
<td>[224x224x64] = 3.2M</td>
</tr>
<tr>
<td>POOL2:</td>
<td>[112x112x64] = 800K 0</td>
</tr>
<tr>
<td>CONV3-128:</td>
<td>[112x112x128] = 1.6M</td>
</tr>
<tr>
<td>CONV3-128:</td>
<td>[112x112x128] = 1.6M</td>
</tr>
<tr>
<td>POOL2:</td>
<td>[56x56x128] = 400K 0</td>
</tr>
<tr>
<td>CONV3-256:</td>
<td>[56x56x256] = 800K 0</td>
</tr>
<tr>
<td>CONV3-256:</td>
<td>[56x56x256] = 800K 0</td>
</tr>
<tr>
<td>CONV3-256:</td>
<td>[56x56x256] = 800K 0</td>
</tr>
<tr>
<td>POOL2:</td>
<td>[28x28x256] = 200K 0</td>
</tr>
<tr>
<td>CONV3-512:</td>
<td>[28x28x512] = 400K 0</td>
</tr>
<tr>
<td>CONV3-512:</td>
<td>[28x28x512] = 400K 0</td>
</tr>
<tr>
<td>CONV3-512:</td>
<td>[28x28x512] = 400K 0</td>
</tr>
<tr>
<td>POOL2:</td>
<td>[14x14x512] = 100K 0</td>
</tr>
<tr>
<td>CONV3-512:</td>
<td>[14x14x512] = 100K 0</td>
</tr>
<tr>
<td>CONV3-512:</td>
<td>[14x14x512] = 100K 0</td>
</tr>
<tr>
<td>POOL2:</td>
<td>[7x7x512] = 25K 0</td>
</tr>
<tr>
<td>FC:</td>
<td>[1x1x4096] = 4096 7x7x512x4096 = 102,760,448</td>
</tr>
<tr>
<td>FC:</td>
<td>[1x1x4096] = 4096 4096x4096 = 16,777,216</td>
</tr>
<tr>
<td>FC:</td>
<td>[1x1x1000] = 1000 4096x1000 = 4,096,000</td>
</tr>
</tbody>
</table>

**TOTAL activations: 24.1M x 4 bytes \(\sim\) 93MB / image (x2 for backward)
(0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(1): ReLU (inplace)
(2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(3): ReLU (inplace)
(4): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
(5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(6): ReLU (inplace)
(7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(8): ReLU (inplace)
(9): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
(10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(11): ReLU (inplace)
(12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(13): ReLU (inplace)
(14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(15): ReLU (inplace)
(16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(17): ReLU (inplace)
(18): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
(19): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(20): ReLU (inplace)
(21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(22): ReLU (inplace)
(23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
VGG-19

... 

(30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(31): ReLU (inplace)
(32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(33): ReLU (inplace)
(34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(35): ReLU (inplace)
(36): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))

(classifier): Sequential ( (0): Linear (25088 -> 4096)
(1): ReLU (inplace)
(2): Dropout (p = 0.5)
(3): Linear (4096 -> 4096)
(4): ReLU (inplace)
(5): Dropout (p = 0.5) (6): Linear (4096 -> 1000) 
)
GoogLeNet / Inception

Szegedy et al. (2015) also introduce the idea of "auxiliary classifiers" to help the propagation of the gradient in the early layers.

This is motivated by the reasonable performance of shallow networks that indicates early layers already encode informative and invariant features.
GoogLeNet / Inception

The resulting GoogLeNet has 12 times less parameters than AlexNet and is more accurate on ILSVRC14 (Szegedy et al., 2015).

It was later extended with batch-normalization (Ioffe and Szegedy, 2015) and pass-through a la resnet (Szegedy et al., 2016)
GoogLeNet / Inception

<table>
<thead>
<tr>
<th>type</th>
<th>patch size/strides</th>
<th>output size</th>
<th>depth</th>
<th>#1×1×1</th>
<th>#3×3×3 reduce</th>
<th>#3×3×3</th>
<th>#5×5×5 reduce</th>
<th>#5×5×5</th>
<th>pool proj</th>
<th>params</th>
<th>ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>convolution</td>
<td>7×7/2</td>
<td>112×112×64</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2.7K</td>
<td>34M</td>
</tr>
<tr>
<td>max pool</td>
<td>3×3/2</td>
<td>56×56×64</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>convolution</td>
<td>3×3/1</td>
<td>56×56×192</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>117K</td>
<td>359M</td>
</tr>
<tr>
<td>max pool</td>
<td>3×3/2</td>
<td>28×28×192</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inception (3a)</td>
<td>28×28×256</td>
<td>2</td>
<td>64</td>
<td>96</td>
<td>128</td>
<td>16</td>
<td>32</td>
<td>32</td>
<td>159K</td>
<td>128M</td>
<td></td>
</tr>
<tr>
<td>inception (3b)</td>
<td>28×28×480</td>
<td>2</td>
<td>128</td>
<td>128</td>
<td>192</td>
<td>32</td>
<td>96</td>
<td>64</td>
<td>391K</td>
<td>304M</td>
<td></td>
</tr>
<tr>
<td>max pool</td>
<td>3×3/2</td>
<td>14×14×480</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inception (4a)</td>
<td>14×14×512</td>
<td>2</td>
<td>192</td>
<td>96</td>
<td>208</td>
<td>16</td>
<td>48</td>
<td>64</td>
<td>364K</td>
<td>73M</td>
<td></td>
</tr>
<tr>
<td>inception (4b)</td>
<td>14×14×512</td>
<td>2</td>
<td>160</td>
<td>112</td>
<td>224</td>
<td>24</td>
<td>64</td>
<td>64</td>
<td>437K</td>
<td>88K</td>
<td></td>
</tr>
<tr>
<td>inception (4c)</td>
<td>14×14×512</td>
<td>2</td>
<td>128</td>
<td>128</td>
<td>256</td>
<td>24</td>
<td>64</td>
<td>64</td>
<td>463K</td>
<td>100M</td>
<td></td>
</tr>
<tr>
<td>inception (4d)</td>
<td>14×14×528</td>
<td>2</td>
<td>112</td>
<td>164</td>
<td>288</td>
<td>32</td>
<td>64</td>
<td>64</td>
<td>391K</td>
<td>110M</td>
<td></td>
</tr>
<tr>
<td>inception (4e)</td>
<td>14×14×832</td>
<td>2</td>
<td>256</td>
<td>160</td>
<td>320</td>
<td>32</td>
<td>128</td>
<td>128</td>
<td>840K</td>
<td>170M</td>
<td></td>
</tr>
<tr>
<td>max pool</td>
<td>3×3/2</td>
<td>7×7×832</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inception (5a)</td>
<td>7×7×832</td>
<td>2</td>
<td>256</td>
<td>160</td>
<td>320</td>
<td>32</td>
<td>128</td>
<td>128</td>
<td>1072K</td>
<td>54M</td>
<td></td>
</tr>
<tr>
<td>inception (5b)</td>
<td>7×7×1024</td>
<td>2</td>
<td>384</td>
<td>192</td>
<td>384</td>
<td>48</td>
<td>128</td>
<td>128</td>
<td>1388K</td>
<td>71M</td>
<td></td>
</tr>
<tr>
<td>avg pool</td>
<td>7×7/1</td>
<td>1×1×1024</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dropout (49%)</td>
<td>1×1×1024</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>linear</td>
<td>1×1×1000</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1000K</td>
<td>1M</td>
<td></td>
</tr>
<tr>
<td>softmax</td>
<td>1×1×1000</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fun features:
- Only 5 million params!
(Removes FC layers completely)

Compared to AlexNet:
- 12X less params
- 2x more compute
- 6.67% (vs. 16.4%)
A saturation point

If we continue stacking more layers on a CNN:

![Graphs showing training and test error over iterations for 20-layer and 56-layer CNNs.](image)
A saturation point

If we continue stacking more layers on a CNN:

Deeper models are harder to optimize
ResNet

A block learns the residual w.r.t. identity

Figure 2. Residual learning: a building block.

Deep residual learning for image recognition, K. He et al., CVPR 2016.
ResNet

A block learns the residual w.r.t. identity

- Good optimization properties

*Deep residual learning for image recognition, K. He et al., CVPR 2016.*
ResNet

Even deeper models:

34, 50, 101, 152 layers

Deep residual learning for image recognition, K. He et al., CVPR 2016.
ResNet

5.25% top-5 error vs 7.1%

ResNet50 Compared to VGG:

Superior accuracy in all vision tasks

Deep residual learning for image recognition, K. He et al., CVPR 2016.
ResNet

5.25% top-5 error vs 7.1%

Less parameters
25M vs 138M

ResNet50 Compared to VGG:

Superior accuracy in all vision tasks

Deep residual learning for image recognition, K. He et al., CVPR 2016.
ResNet

5.25% top-5 error vs 7.1%

Less parameters
25M vs 138M

Computational complexity
3.8B Flops vs 15.3B Flops

ResNet50 Compared to VGG:

Superior accuracy in all vision tasks

*Deep residual learning for image recognition, K. He et al., CVPR 2016.*
ResNet

5.25% top-5 error vs 7.1%

Less parameters
25M vs 138M

Computational complexity
3.8B Flops vs 15.3B Flops

Fully Convolutional until the last layer

ResNet50 Compared to VGG:
Superior accuracy in all vision tasks

Deep residual learning for image recognition, K. He et al., CVPR 2016.
ResNet

Performance on ImageNet

Figure 4. Training on ImageNet. Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers. In this plot, the residual networks have no extra parameter compared to their plain counterparts.
ResNet

The output of a residual network can be understood as an ensemble, which explains in part its stability.
Table 1. Architectures for ImageNet. Building blocks are shown in brackets (see also Fig. 5), with the numbers of blocks stacked. Downsampling is performed by conv3.1, conv4.1, and conv5.1 with a stride of 2.
### Results

<table>
<thead>
<tr>
<th>method</th>
<th>top-5 err. (test)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VGG [41] (ILSVRC’14)</td>
<td>7.32</td>
</tr>
<tr>
<td>GoogLeNet [44] (ILSVRC’14)</td>
<td>6.66</td>
</tr>
<tr>
<td>VGG [41] (v5)</td>
<td>6.8</td>
</tr>
<tr>
<td>PReLU-net [13]</td>
<td>4.94</td>
</tr>
<tr>
<td>BN-inception [16]</td>
<td>4.82</td>
</tr>
<tr>
<td><strong>ResNet (ILSVRC’15)</strong></td>
<td><strong>3.57</strong></td>
</tr>
</tbody>
</table>

Table 5. Error rates (%) of *ensembles*. The top-5 error is on the test set of ImageNet and reported by the test server.
ResNet

In PyTorch:

```python
def make_resnet_block(num_feature_maps, kernel_size = 3):
    return nn.Sequential(
        nn.Conv2d(num_feature_maps, num_feature_maps, 
                  kernel_size = kernel_size, 
                  padding = (kernel_size - 1) // 2),
        nn.BatchNorm2d(num_feature_maps),
        nn.ReLU(inplace = True),
        nn.Conv2d(num_feature_maps, num_feature_maps, 
                  kernel_size = kernel_size, 
                  padding = (kernel_size - 1) // 2),
        nn.BatchNorm2d(num_feature_maps),
    )
```
ResNet

In PyTorch:

def __init__(self, num_residual_blocks, num_feature_maps):
    ...
    self.resnet_blocks = nn.ModuleList()
    for k in range(nb_residual_blocks):
        self.resnet_blocks.append(make_resnet_block(num_feature_maps, 3))
    ...

def forward(self,x):
    ...
    for b in self.resnet_blocks:
        x = x + b(x)
    ...
    return x
Deeper is better

ImageNet experiments

Inception-V4 / -ResNet-V2

Deep, modular and state-of-the-art
Achieves 3.1% top-5 classification error on imagenet

*Inception-v4, inception-resnet and the impact of residual connections on learning, C. Szegedy et al., 2016*

Slide credit: C. Ollion & O. Grisel, M2DS Deep Learning
Resnet variants: Stochastic Depth Networks

- DropOut at layer level
- Allows training up to 1K layers

Fig. 2. The linear decay of $p_\ell$ illustrated on a ResNet with stochastic depth for $p_0 = 1$ and $p_L = 0.5$. Conceptually, we treat the input to the first ResBlock as $H_0$, which is always active.

Deep Networks with Stochastic Depth, Huang et al., ECCV 2016
Resnet variants: DenseNet

- Copying feature maps to upper layers via skip-connections
- Better reuse of parameters and redundancy avoidance
Inception-V4 / -ResNet-V2

More building blocks engineering...

Inception-v4, inception-resnet and the impact of residual connections on learning, C. Szegedy et al., 2016

Slide credit: C. Ollion & O. Grisel, M2DS Deep Learning
Inception-V4 / -ResNet-V2

More building blocks engineering...

- Active area or research
- See also DenseNets, Wide ResNets, Fractal ResNets, ResNeXts, Pyramidal ResNets...

*Inception-v4, inception-resnet and the impact of residual connections on learning, C. Szegedy et al., 2016*
Comparison of models

Top 1-accuracy, performance and size on ImageNet

An Analysis of Deep Neural Network Models for Practical Applications, Canziani et al., 2016
Comparison of models

Forward pass time and power consumption

An Analysis of Deep Neural Network Models for Practical Applications, Canziani et al., 2016
Comparison of models

GoogLeNet (2014)  
VGG-VD-16 (2014)  
VGG-M (2013)  
AlexNet (2012)  

ResNet 50 (2015)  
ResNet 152 (2015)  

16 convolutional layers  
50 convolutional layers  
152 convolutional layers  


Comparison of models

3 x more accurate in 3 years

101 ResNet Layers same size/speed as 16 VGG-VD layers
Comparison of models

Number of parameters is about the same
Comparison of models

5 x slower
Recap

- Convolutions and convolutional layers
  - activation functions
  - pooling
  - dilated convolutions

- Regularization for deep neural networks:
  - DropOut
  - Batch Normalization

- Popular neural network architectures:
  - LeNet5, AlexNet, VGG, Inception/GoogLeNet, ResNet, DenseNet
Under the hood of neural networks