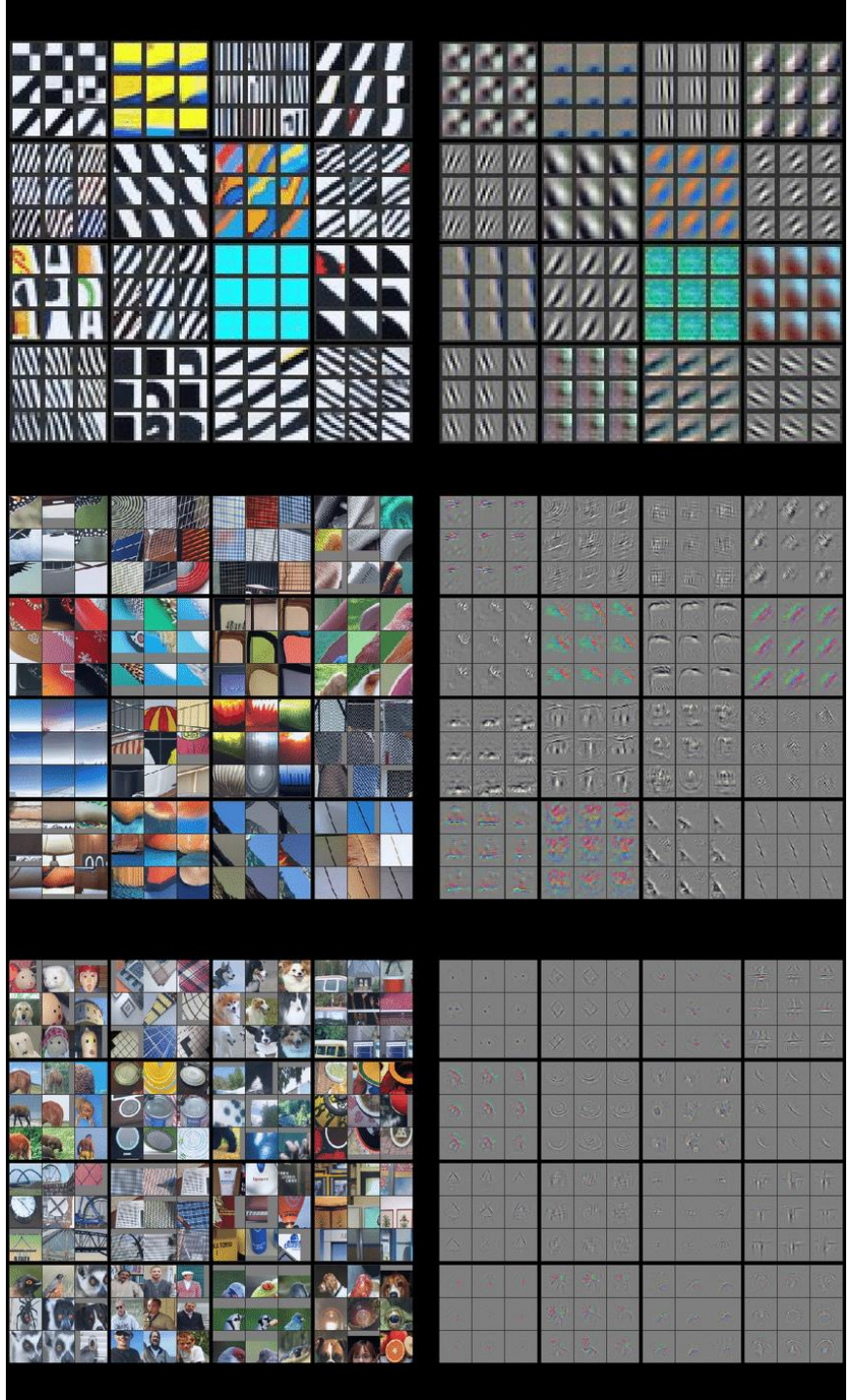
More on CNN

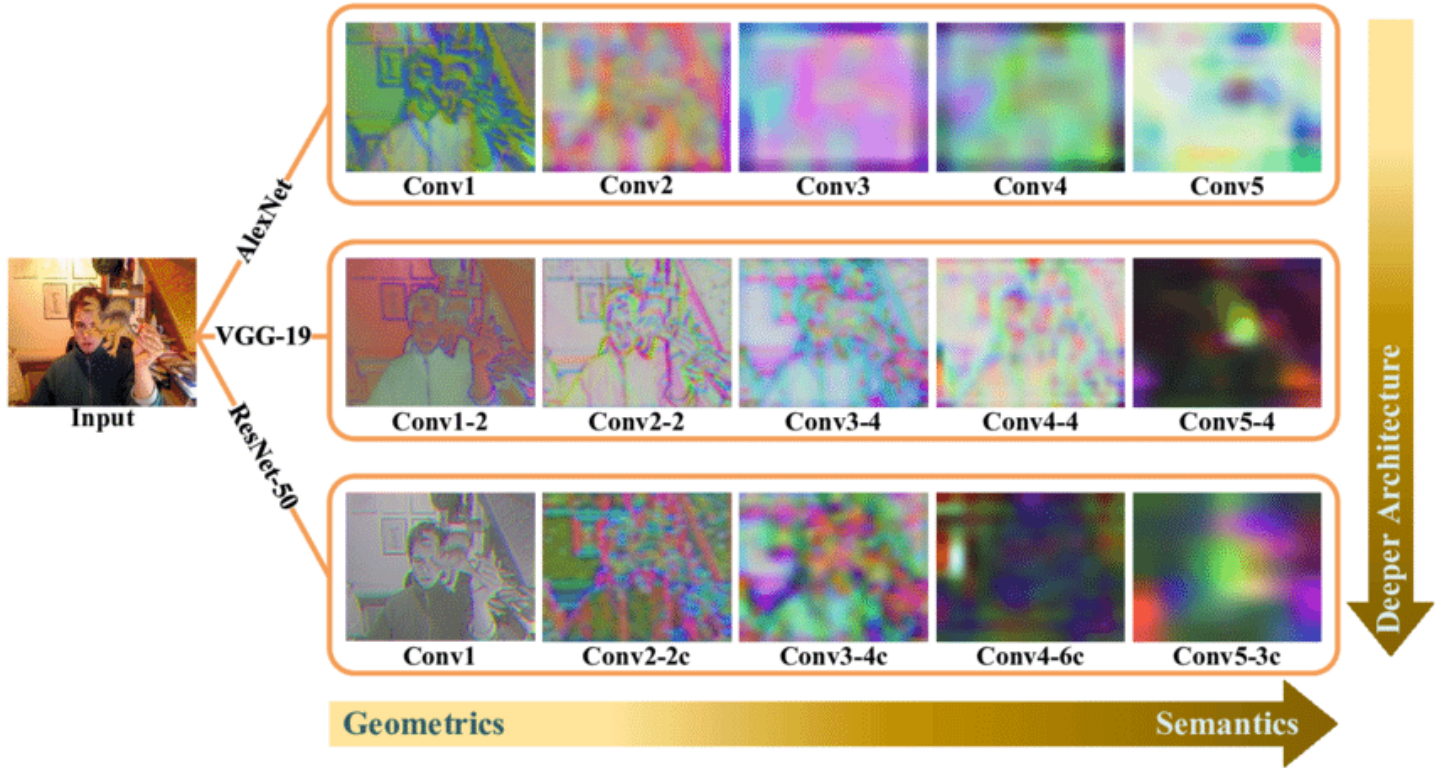
Biplab Banerjee

Thanks to the authors, towards data science and medium

```
class LeNet(Module):
    def __init__(self, numChannels, classes):
        # call the parent constructor
        super(LeNet, self).__init__()

        # initialize first set of CONV => RELU => POOL layers
        self.conv1 = Conv2d(in_channels=numChannels, out_channels=20,
            kernel_size=(5, 5))
        self.relu1 = ReLU()
        self.maxpool1 = MaxPool2d(kernel_size=(2, 2), stride=(2, 2))

        # initialize second set of CONV => RELU => POOL layers
        self.conv2 = Conv2d(in_channels=20, out_channels=50,
            kernel_size=(5, 5))
        self.relu2 = ReLU()
        self.maxpool2 = MaxPool2d(kernel_size=(2, 2), stride=(2, 2))

        # initialize first (and only) set of FC => RELU layers
        self.fc1 = Linear(in_features=800, out_features=500)
        self.relu3 = ReLU()

        # initialize our softmax classifier
        self.fc2 = Linear(in_features=500, out_features=classes)
        self.logSoftmax = LogSoftmax(dim=1)
```

```
    def forward(self, x):
        # pass the input through our first set of CONV => RELU =>
        # POOL layers
        x = self.conv1(x)
        x = self.relu1(x)
        x = self.maxpool1(x)

        # pass the output from the previous layer through the second
        # set of CONV => RELU => POOL layers
        x = self.conv2(x)
        x = self.relu2(x)
        x = self.maxpool2(x)

        # flatten the output from the previous layer and pass it
        # through our only set of FC => RELU layers
        x = flatten(x, 1)
        x = self.fc1(x)
        x = self.relu3(x)

        # pass the output to our softmax classifier to get our output
        # predictions
        x = self.fc2(x)
        output = self.logSoftmax(x)

        # return the output predictions
        return output
```




Input
AlexNet
VGG-19
ResNet-50
Conv1
Conv2
Conv3
Conv4
Conv5
Conv1-2
Conv2-2
Conv3-4
Conv4-4
Conv5-4
Conv1
Conv2-2c
Conv3-4c
Conv4-6c
Conv5-3c
Deeper Architecture
Geometrics
Semantics

Layer		Feature Map	Size	Kernel Size	Stride	Activation
Input	Image	1	227x227x3	-	-	-
1	Convolution	96	55 x 55 x 96	11x11	4	relu
	Max Pooling	96	27 x 27 x 96	3x3	2	relu
2	Convolution	256	27 x 27 x 256	5x5	1	relu
	Max Pooling	256	13 x 13 x 256	3x3	2	relu
3	Convolution	384	13 x 13 x 384	3x3	1	relu
4	Convolution	384	13 x 13 x 384	3x3	1	relu
5	Convolution	256	13 x 13 x 256	3x3	1	relu
	Max Pooling	256	6 x 6 x 256	3x3	2	relu
6	FC	-	9216	-	-	relu
7	FC	-	4096	-	-	relu
8	FC	-	4096	-	-	relu
Output	FC	-	1000	-	-	Softmax

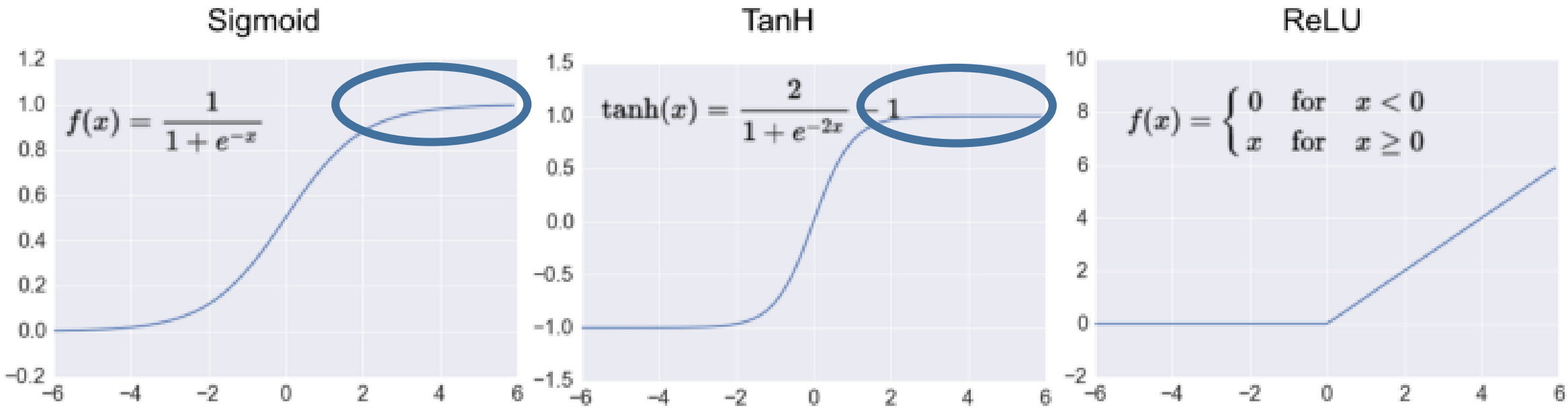
Some insights

- Translation invariance vs equivariance
- Feature hierarchy learning in CNN
- Requires loads of images
- What happens if we do not engineer the model properly?

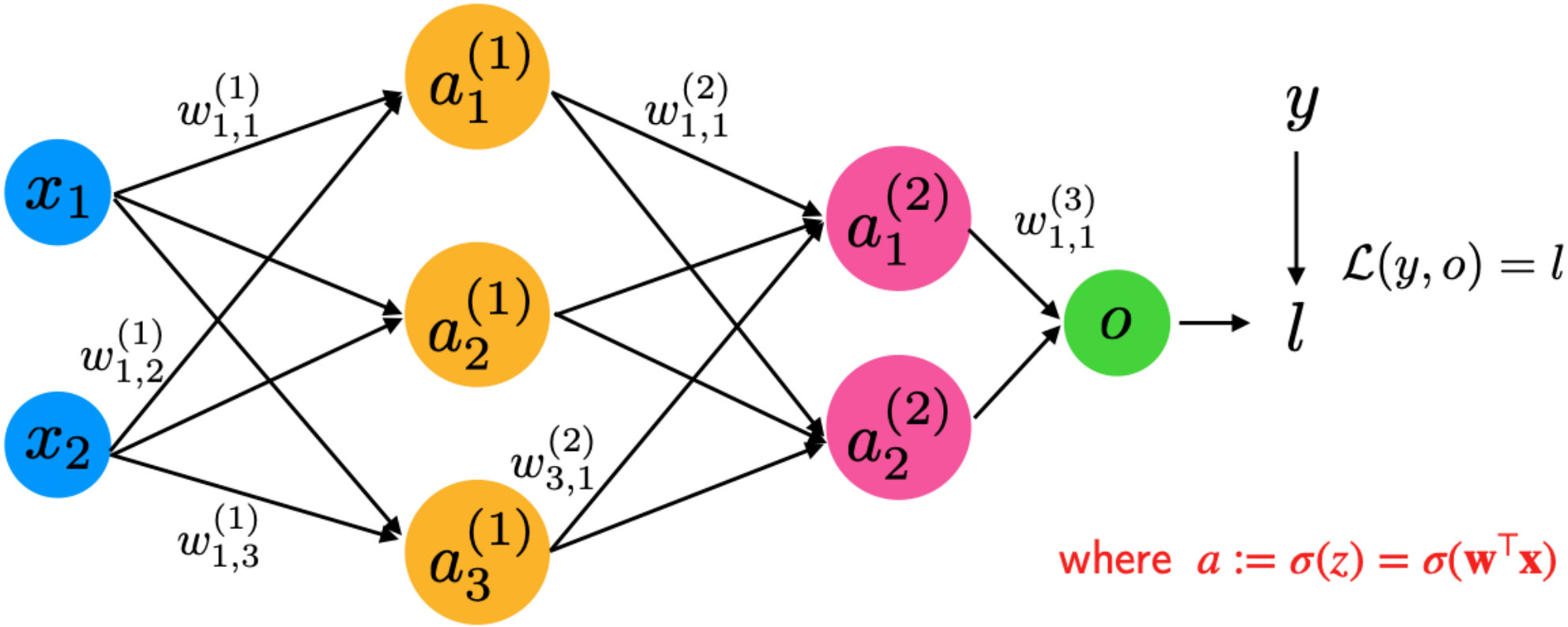
Issues in CNN

- Weight initialization
- Normalization
- Regularization
- Some intuitive CNN architectures

Saturation problem with activations



Vanishing and exploding gradients



where $a := \sigma(z) = \sigma(\mathbf{w}^\top \mathbf{x})$

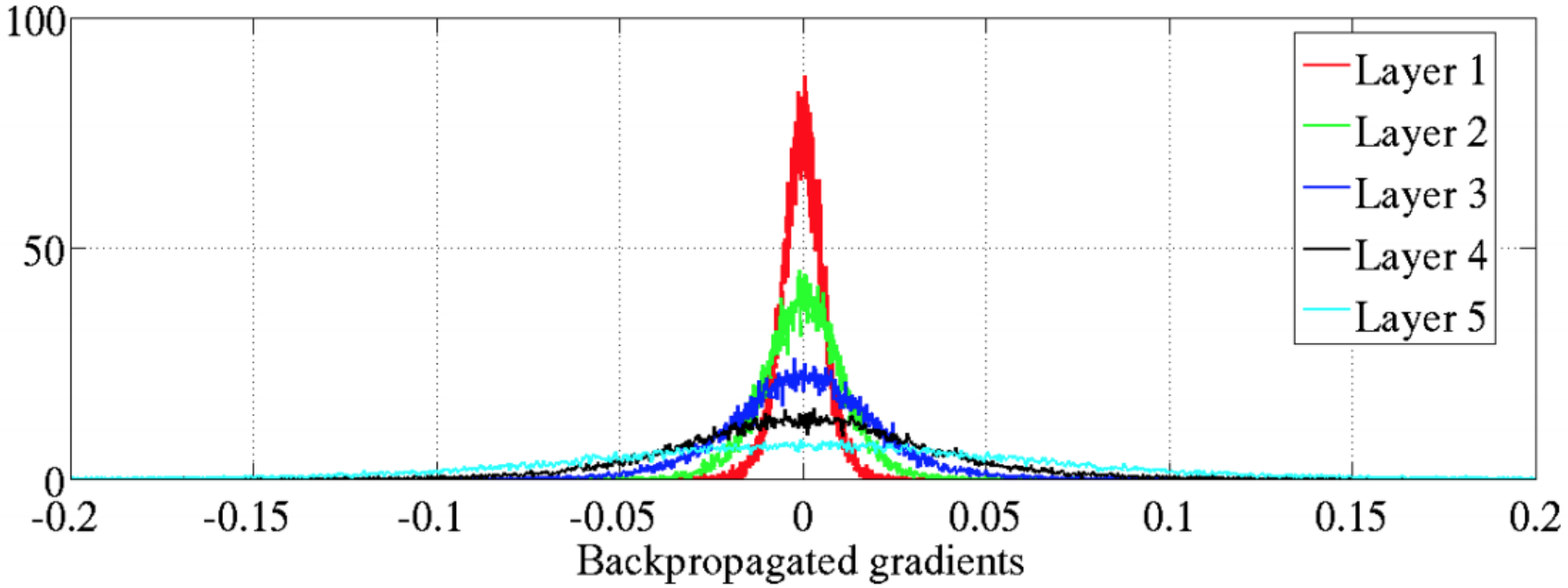
$$\frac{\partial l}{\partial w_{1,1}^{(1)}} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1^{(2)}} \cdot \frac{\partial a_1^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}}$$

Vanishing gradient effects

- The parameters of the higher layers change to a great extent, while the parameters of lower layers barely change (or, do not change at all).

•The model weights could become 0 during training.

•The model learns at a particularly slow pace and the training could stagnate at a very early phase after only a few iterations.



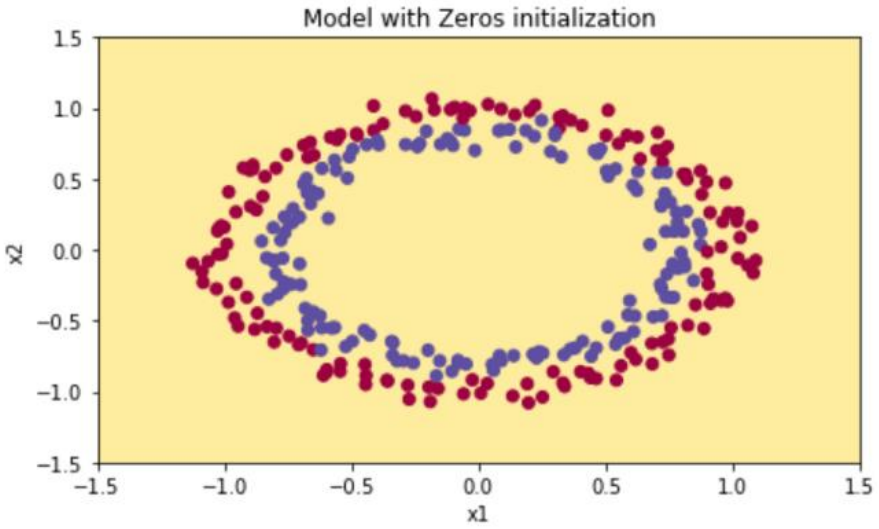
Ways to deal with them

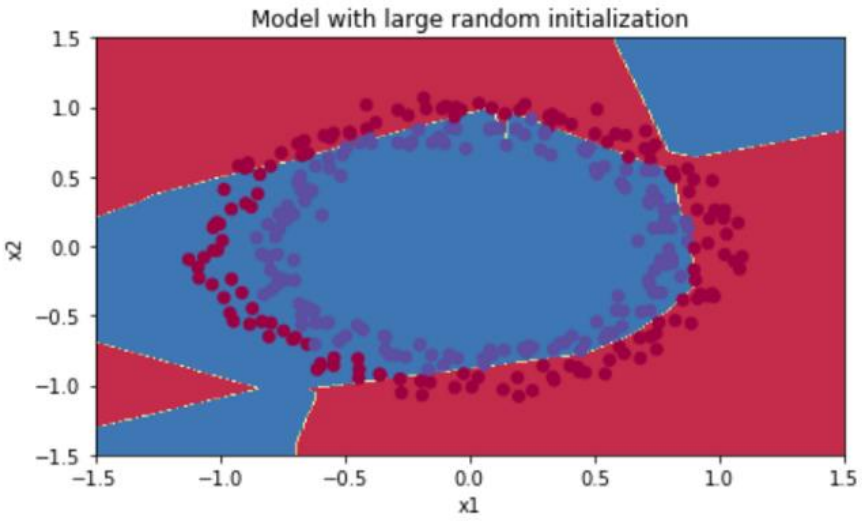
- Vanishing gradients
  - Use proper activations
  - Smaller network (if using sigmoid/tanh type activations)
  - **Proper initialization**

- Exploding gradients
  - Weight clipping
  - **Proper initialization**
  - Model engineering

Problems with standard initializations

- Zero initialization – All neurons learn the same values
- Random initialization – Saturation problem for sigmoid/tanh





Xavier initialization

- Xavier initialization (also known as Glorot initialization) is a strategy designed to maintain a consistent variance of the activations throughout the layers of a neural network. This helps prevent the issues of vanishing or exploding gradients during training. The main idea is to carefully set the initial weights so that the variance of the inputs to any layer remains roughly equal to the variance of the outputs from that layer.

- The variance of the activations $\mathrm{Var}(y)$ remains approximately equal to the variance of the inputs $\mathrm{Var}(x)$.
- The gradients flowing backward have a similar variance at all layers.

Forward pass

- n_{in}: number of input neurons,
- n_{out}: number of output neurons,
- $\mathbf{x} \in \mathbb{R}^{n_{in}}$: input vector,
- $\mathbf{W} \in \mathbb{R}^{n_{out} \times n_{in}}$: weight matrix,
- $\mathbf{y} \in \mathbb{R}^{n_{out}}$: output vector.

The pre-activation output for neuron j is given by:

$$y_j = \sum_{i=1}^{n_{in}} W_{ji}\, x_i.$$

Assumptions:

- The inputs x_i are assumed to be i.i.d. with zero mean and variance $\mathrm{Var}(x_i) = \sigma_x^2$.
- The weights W_{ji} are i.i.d. with zero mean and variance $\mathrm{Var}(W_{ji}) = \sigma_w^2$.
- The inputs and weights are independent.

Computing the Variance:

Since y_j is a sum of independent random variables,

$$\mathrm{Var}(y_j) = \sum_{i=1}^{n_{in}} \mathrm{Var}\left(W_{ji}\, x_i\right).$$

Because W_{ji} and x_i are independent,

$$\mathrm{Var}(W_{ji}\, x_i) = \mathrm{Var}(W_{ji})\,\mathrm{Var}(x_i) = \sigma_w^2\, \sigma_x^2.$$

Thus,

$$\mathrm{Var}(y_j) = n_{in}\, \sigma_w^2\, \sigma_x^2.$$

Goal: We wish for the variance of the output $\mathrm{Var}(y_j)$ to be approximately equal to the variance of the input σ_x^2 (to avoid exploding or vanishing activations). Hence, we set:

$$n_{in}\, \sigma_w^2\, \sigma_x^2 \approx \sigma_x^2.$$

Canceling σ_x^2 (assuming it's nonzero), we get:

$$n_{in}\, \sigma_w^2 \approx 1 \quad \Longrightarrow \quad \sigma_w^2 \approx \frac{1}{n_{in}}.$$

Backward pass

During backpropagation, gradients are propagated in a similar manner. Assume:

- δ_j is the gradient at the output of neuron j,
- The gradient with respect to the input x_i is:

$$\delta_i = \sum_{j=1}^{n_{out}} W_{ji}\, \delta_j.$$

Assume the gradients δ_j have zero mean and variance $\mathrm{Var}(\delta_j) = \sigma_\delta^2$. Then,

$$\mathrm{Var}(\delta_i) = \sum_{j=1}^{n_{out}} \mathrm{Var}\left(W_{ji}\, \delta_j\right) = n_{out}\, \sigma_w^2\, \sigma_\delta^2.$$

For stable gradients (i.e., $\mathrm{Var}(\delta_i) \approx \sigma_\delta^2$), we want:

$$n_{out}\, \sigma_w^2\, \sigma_\delta^2 \approx \sigma_\delta^2,$$

which simplifies to:

$$n_{out}\, \sigma_w^2 \approx 1 \quad \Longrightarrow \quad \sigma_w^2 \approx \frac{1}{n_{out}}.$$

Combining both the passes

To ensure that both the forward activations and the backward gradients maintain their variance, a compromise is reached. Glorot and Bengio proposed averaging the two conditions:

- From the forward pass: $\sigma_w^2 = \frac{1}{n_{in}}$,
- From the backward pass: $\sigma_w^2 = \frac{1}{n_{out}}$.

A balanced choice is to set:

$$\sigma_w^2 = \frac{2}{n_{in} + n_{out}}.$$

This equation is the basis for Xavier initialization when using a normal distribution for the weights.

Weights are drawn from a normal distribution with mean 0 and the variance obtained above:

$$W_{ji} \sim \mathcal{N}\left(0, \frac{2}{n_{in} + n_{out}}\right).$$

Other initialization strategies

He (Kaiming) Initialization

- **Motivation:** Designed specifically for ReLU (and its variants) activations, where only half of the activations are expected to be non-zero. $\mathrm{Var}(\mathrm{ReLU}(y_j)) \approx \frac{1}{2}\mathrm{Var}(y_j)$
- **Method:** For a layer with n_{in} input units, the weights are drawn from a distribution with variance scaled by $\frac{2}{n_{in}}$.
    - **Normal variant:**

$$W \sim \mathcal{N}\left(0, \frac{2}{n_{in}}\right)$$

Other initialization strategies

Orthogonal Initialization

- **Motivation:** Orthogonal matrices preserve the norm of the input under multiplication and can help maintain a stable flow of gradients across layers, especially in very deep networks.
- **Method:** Weights are initialized such that the weight matrix is orthogonal (i.e., $W^T W = I$).
    - **Implementation:** Often involves generating a random matrix (e.g., from a Gaussian distribution) and then using a method like Singular Value Decomposition (SVD) to construct an orthogonal matrix.
- **Key Idea:** By ensuring the weight matrix is orthogonal, the transformation preserves the variance of the activations, reducing the risk of gradient explosion or vanishing.

Other initialization strategies

LSUV (Layer-Sequential Unit Variance) Initialization

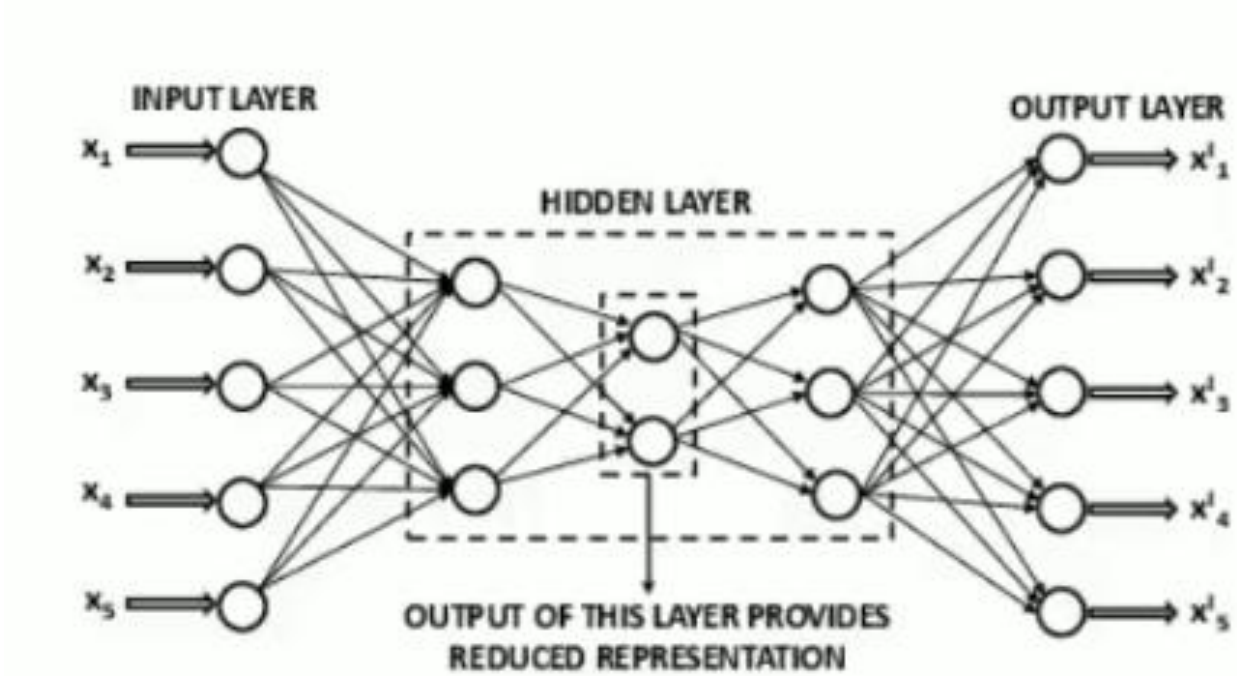
- **Motivation:** Ensure that every layer starts with unit variance activations, which can further stabilize training.
- **Method:**
  - Begin with a rough initialization (often orthogonal or He initialization).
  - Pass a small batch of data through each layer sequentially.
  - Measure the variance of the layer's output.
  - Scale the weights of that layer until the output variance is close to 1.

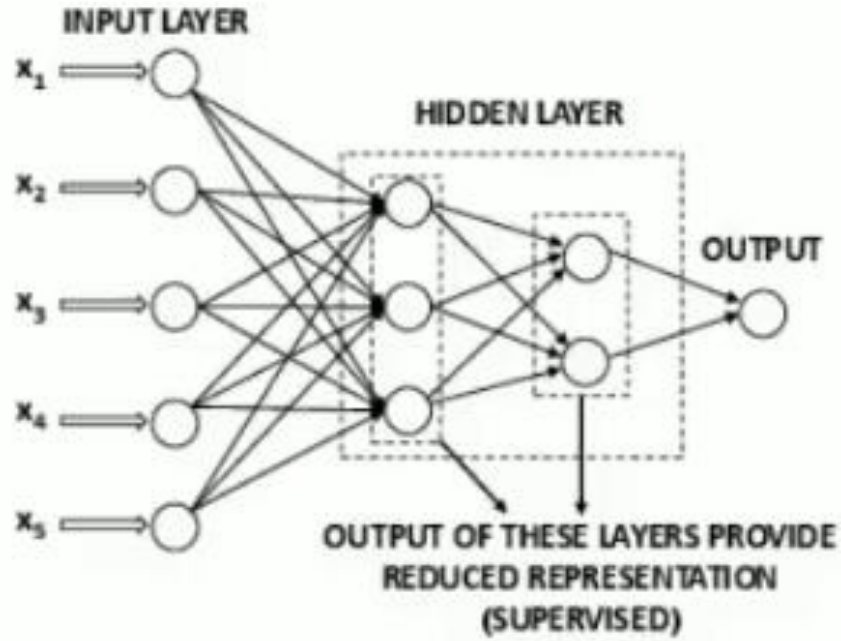
```
from keras.models import Sequential
from keras.layers import Dense, Conv2D, Flatten


#create model
model = Sequential()


#add model layers
model.add(Conv2D(64, kernel_size=3, activation='relu', input_shape=
(28,28,1)))
model.add(Conv2D(32, kernel_size=3, activation='relu'))
model.add(Flatten())
model.add(Dense(10, activation='softmax'))
```

```
W = tf.get_variable("W", shape=[784, 256],
           initializer=tf.contrib.layers.xavier_initializer())
```

pre-training

- *Unsupervised pretraining:* Use training data without labels for initialization.
  - Improves convergence behavior.
  - Regularization effect.
- *Supervised pretraining:* Use training data with labels for initialization.
  - Improves convergence but might overfit.
- Focus on unsupervised pretraining.


INPUT LAYER
HIDDEN LAYER
OUTPUT LAYER
x_1
x_2
x_3
x_4
x_5
x'_1
x'_2
x'_3
x'_4
x'_5
OUTPUT OF THIS LAYER PROVIDES
REDUCED REPRESENTATION


INPUT LAYER
HIDDEN LAYER
OUTPUT
x_1
x_2
x_3
x_4
x_5
OUTPUT OF THESE LAYERS PROVIDE
REDUCED REPRESENTATION
(SUPERVISED)

Suggested reading

Understanding the difficulty of training deep feedforward neural networks

Xavier Glorot **Yoshua Bengio**
DIRO, Université de Montréal, Montréal, Québec, Canada

Data normalization

- Normalized data helps in stable training
- It helps in dealing with **internal covariate shift**
- Since the CNN is trained with BATCH of training samples, lets see the idea of batch gradient descent in back-propagation

Batch gradient descent

$$J_{train}(\theta) = (1/2m)\ \Sigma(\ h_\theta(x^{(i)}) \ - y^{(i)})^2$$

Repeat {

$\theta j = \theta j - (\text{learning rate}/m) * \Sigma(\ h_\theta(x^{(i)}) \ - y^{(i)})x_j^{(i)}$

For every j =0 …n

}

Stochastic gradient descent

$$J_{train}(\theta) = (1/m) \Sigma \text{ Cost}(\theta, (x^{(i)},y^{(i)}))$$

Repeat {

For i=1 to m{

$$\theta_j = \theta_j - (\text{learning rate}) * \Sigma(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

For every j =0 …n

}

}

Mini-batch gradient descent

Repeat {

For i=1,11, 21,…..,91

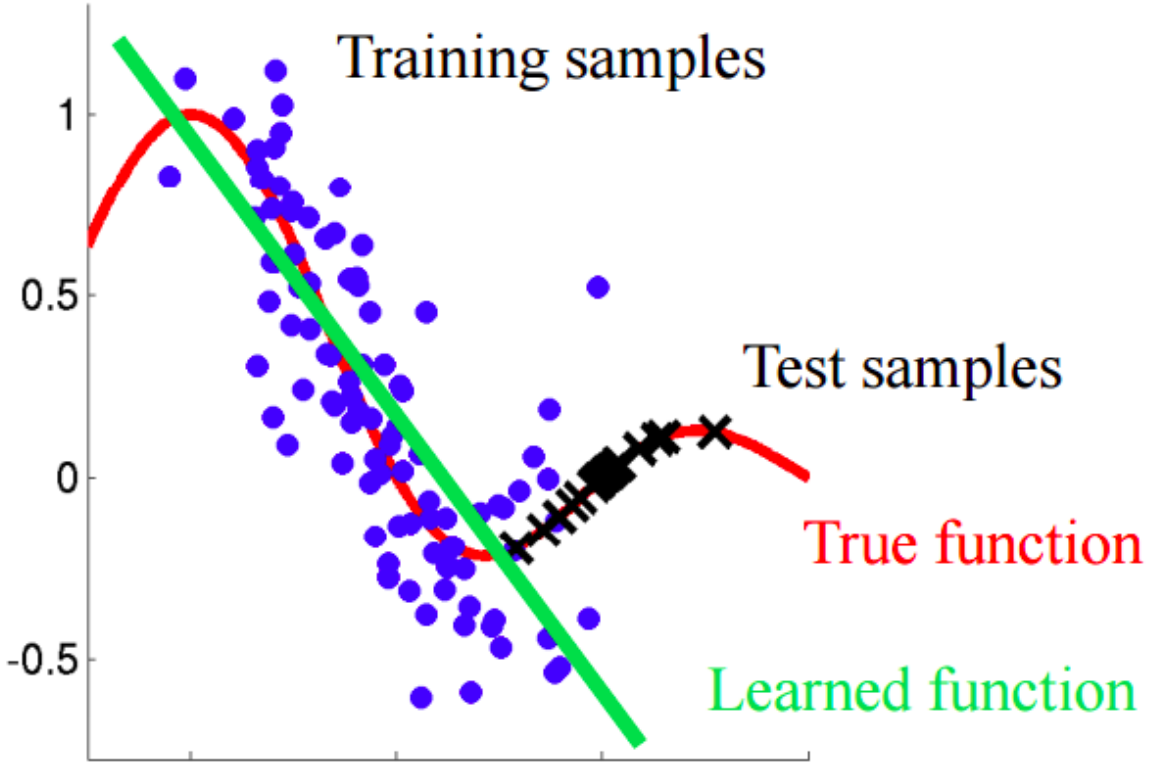
Let Σ be the summation from i to i+9 represented by k.

$\theta_j = \theta_j$ – (learning rate/size of (b)) * Σ($h_\theta(x^{(k)})$ - $y^{(k)})x_j^{(k)}$

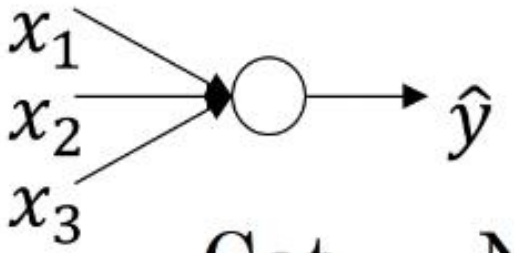
For every j =0 …n

}

The idea of dataset (covariate) shift



“**Internal Covariate Shift** is the change in the distribution of network activations due to the change in network parameters during training.”



Cat
$y = 1$



Non-Cat
$y = 0$



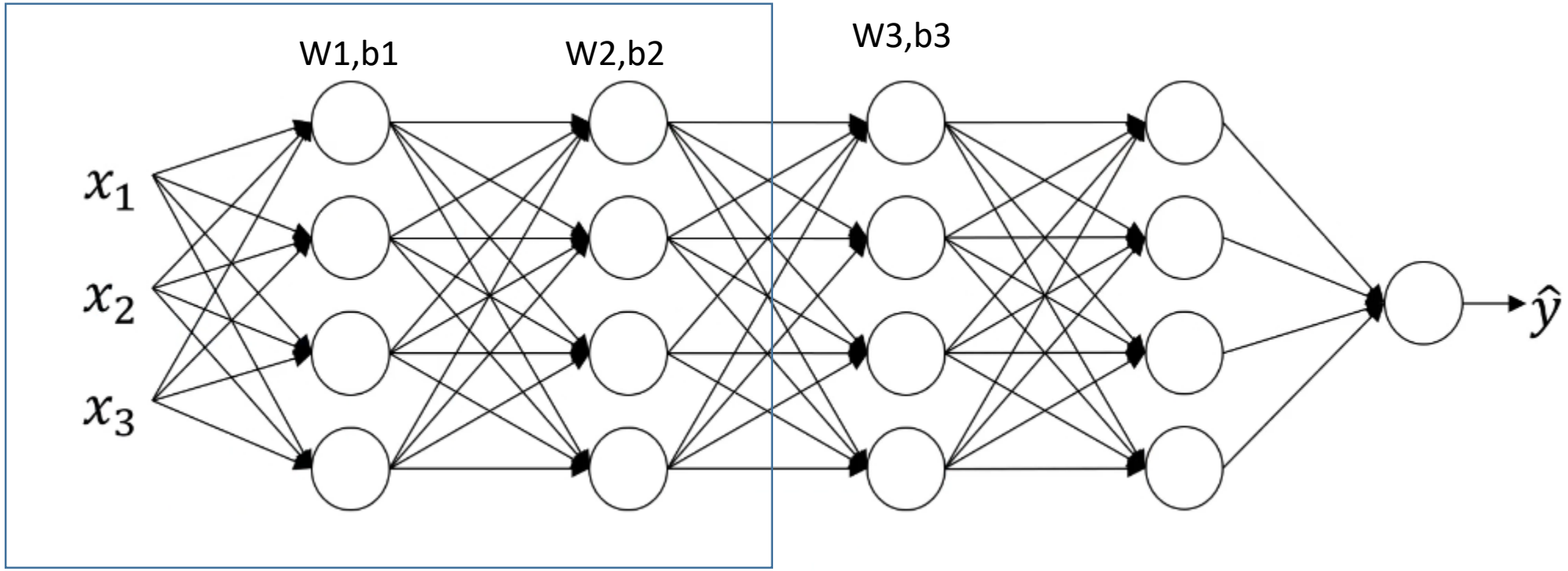
$y = 1$



$y = 0$



What does each layer weight see?



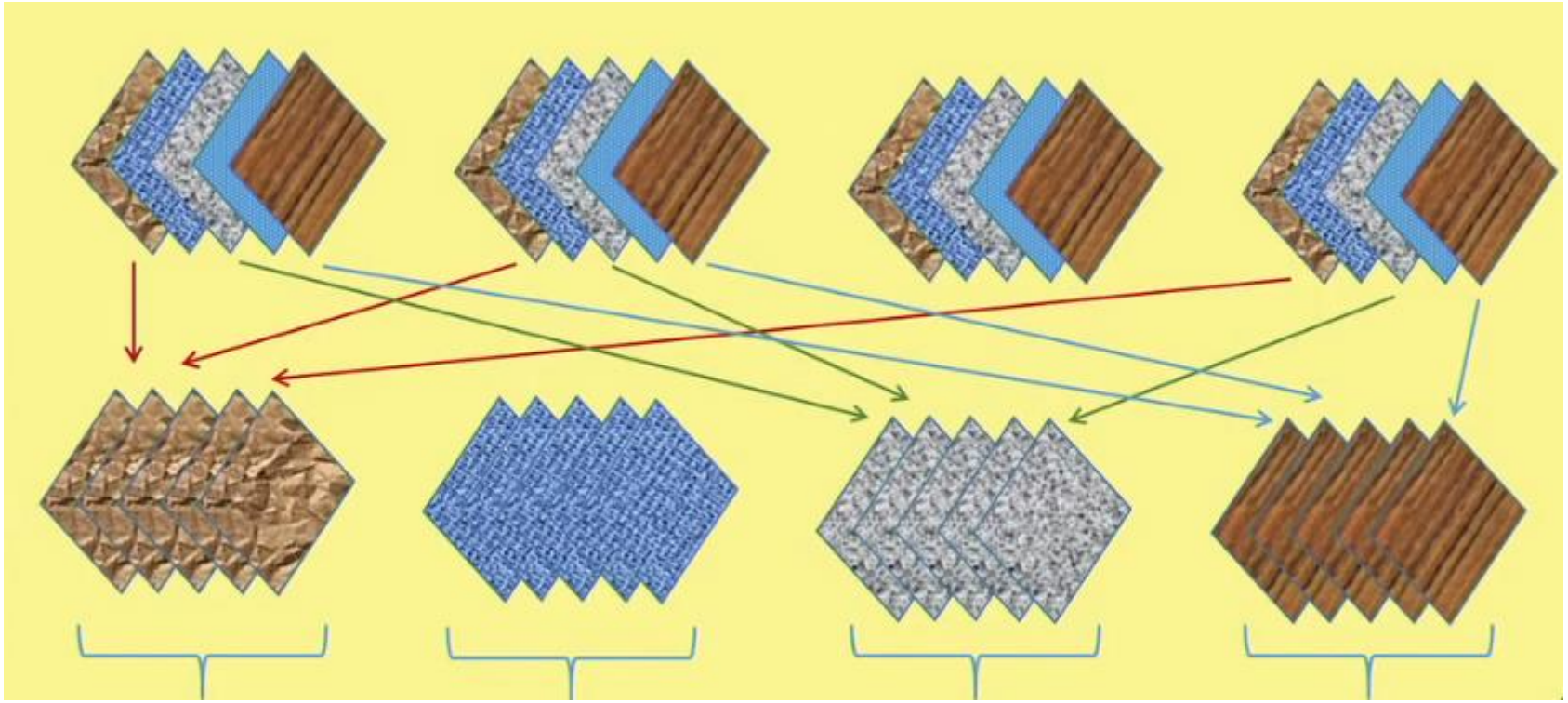
Inputs to layer 3 changes over time as (W1,b1) and (W2,b2) are also changing

Batch-normalization

> *Training Deep Neural Networks is complicated by the fact that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialization, and makes it notoriously hard to train models with saturating nonlinearities.*

Normalizing the inputs to the layer has an effect on the training of the model, dramatically reducing the number of epochs required. It can also have a regularizing effect, reducing generalization error much like the use of activation regularization.

How BN works



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

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

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

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

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

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

$$\frac{\partial \ell}{\partial \widehat{x}_i} = \frac{\partial \ell}{\partial y_i} \cdot \gamma$$

$$\frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot (x_i - \mu_{\mathcal{B}}) \cdot \frac{-1}{2} (\sigma_{\mathcal{B}}^2 + \epsilon)^{-3/2}$$

$$\frac{\partial \ell}{\partial \mu_{\mathcal{B}}} = \left(\sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \right) + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{\sum_{i=1}^{m} -2(x_i - \mu_{\mathcal{B}})}{m}$$

$$\frac{\partial \ell}{\partial x_i} = \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{2(x_i - \mu_{\mathcal{B}})}{m} + \frac{\partial \ell}{\partial \mu_{\mathcal{B}}} \cdot \frac{1}{m}$$

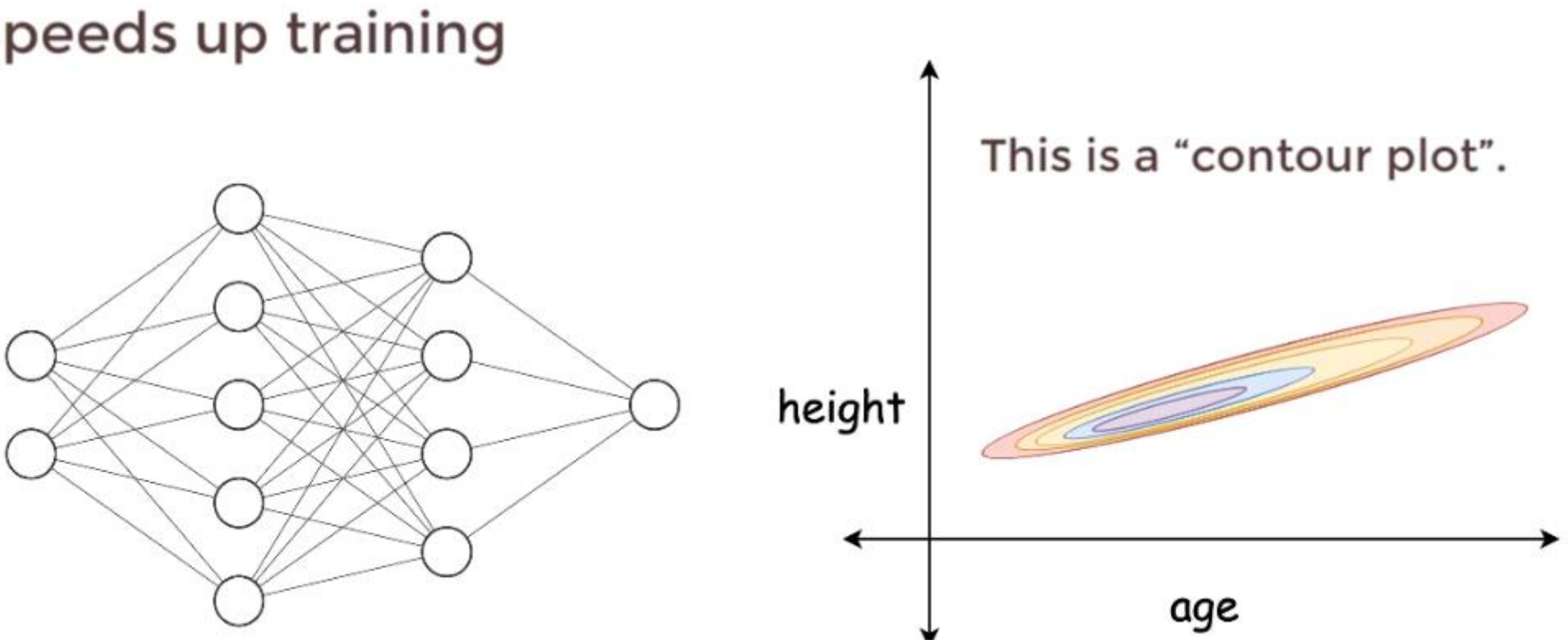
$$\frac{\partial \ell}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i} \cdot \widehat{x}_i$$

$$\frac{\partial \ell}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i}$$

Advantages of BN

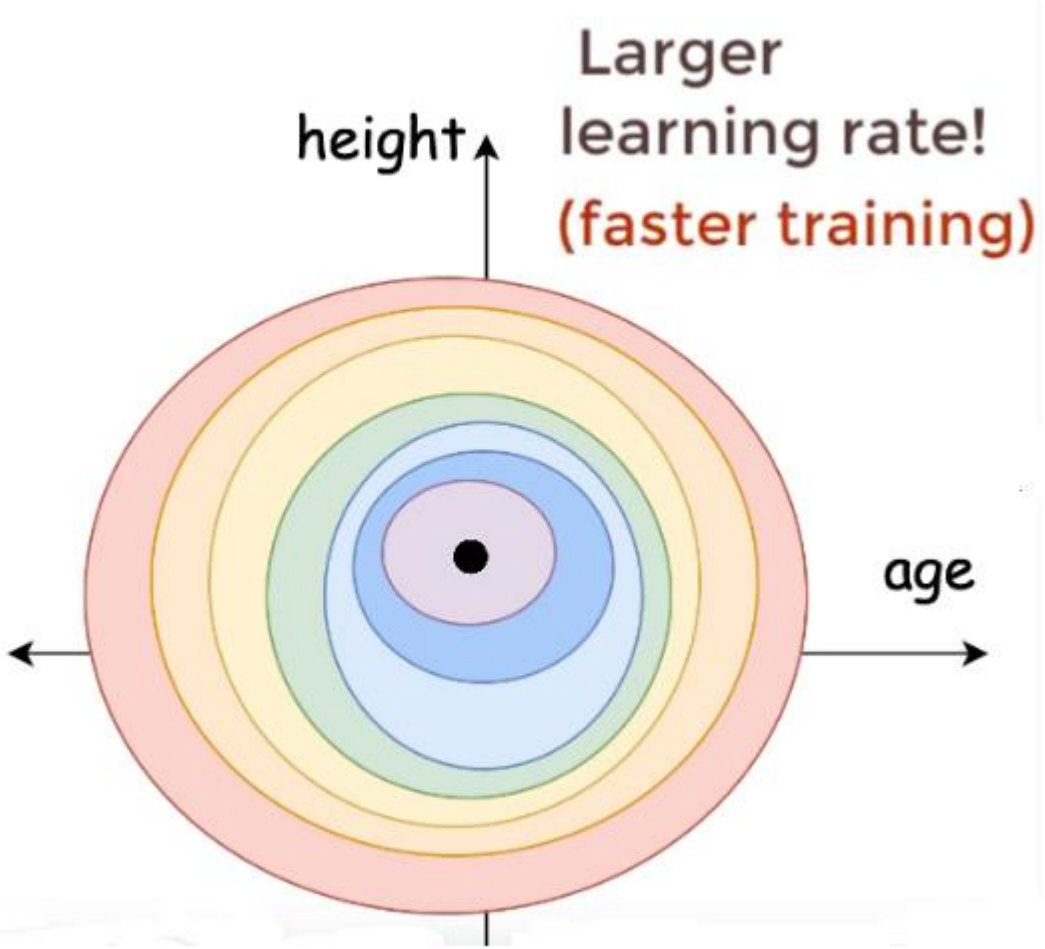
- Speeds up training
- Reduce dependence on weight initialization
- Small regularization effect

Speeds up training



In different batch, let us consider we have young to old people

After standardization

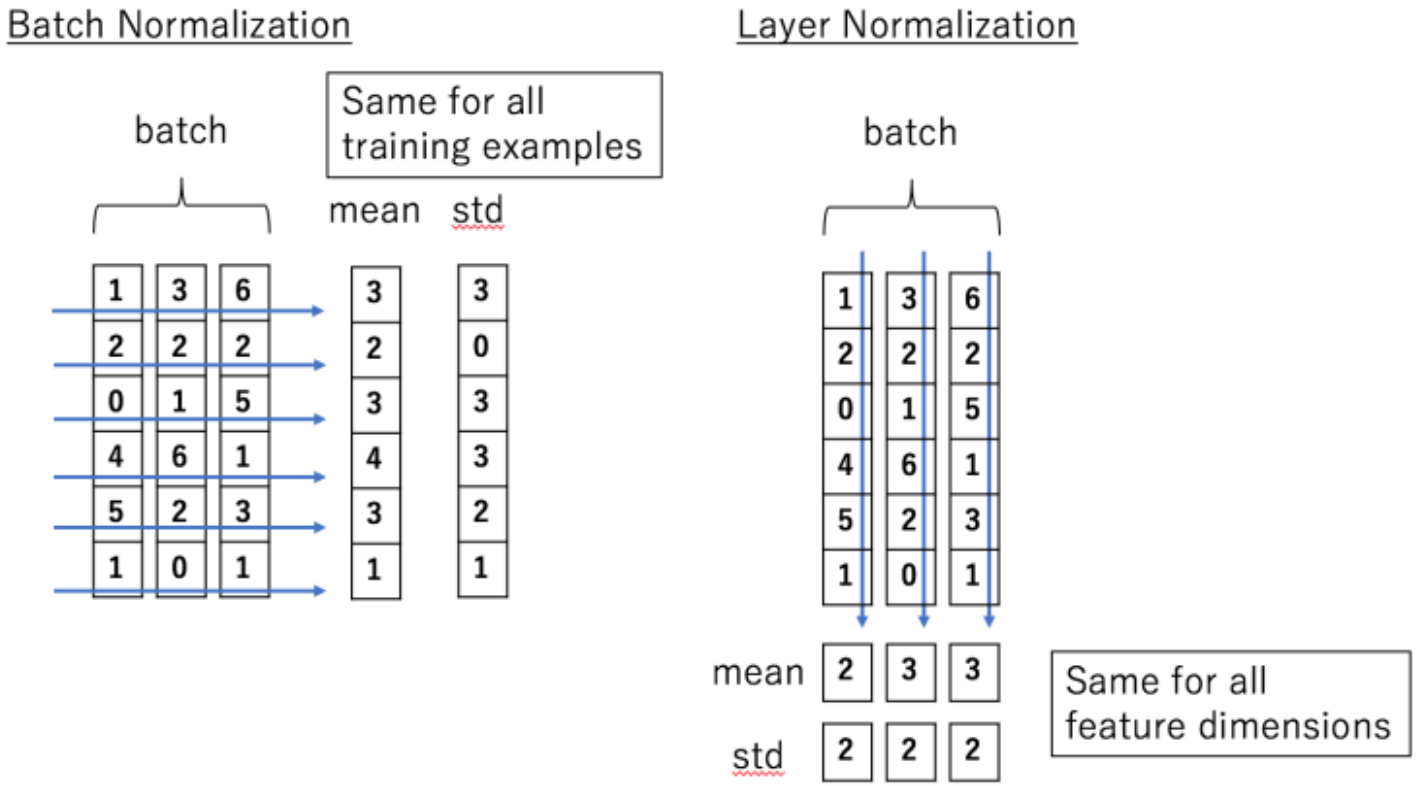


Where can the BN not be used?

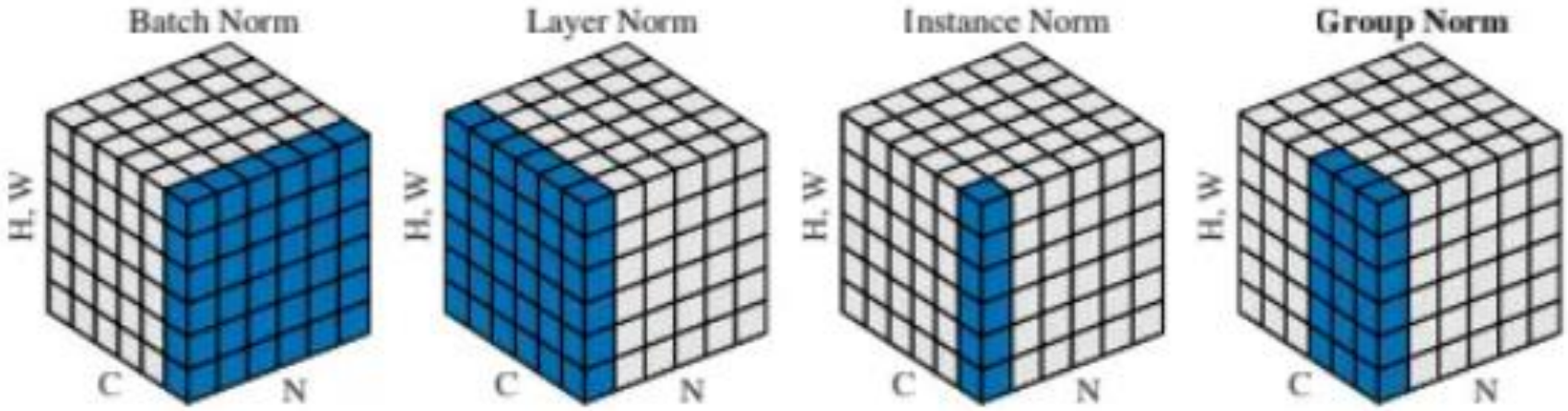
- Small batch size – high noise in calculated batch mean and std!
- Recurrent type models – separate batch-norm at each time-step!

- Solutions?
  - Weight normalization
  - Layer normalization
  - Instance normalization
  - Spectral normalization
  - Group normalization
  - …

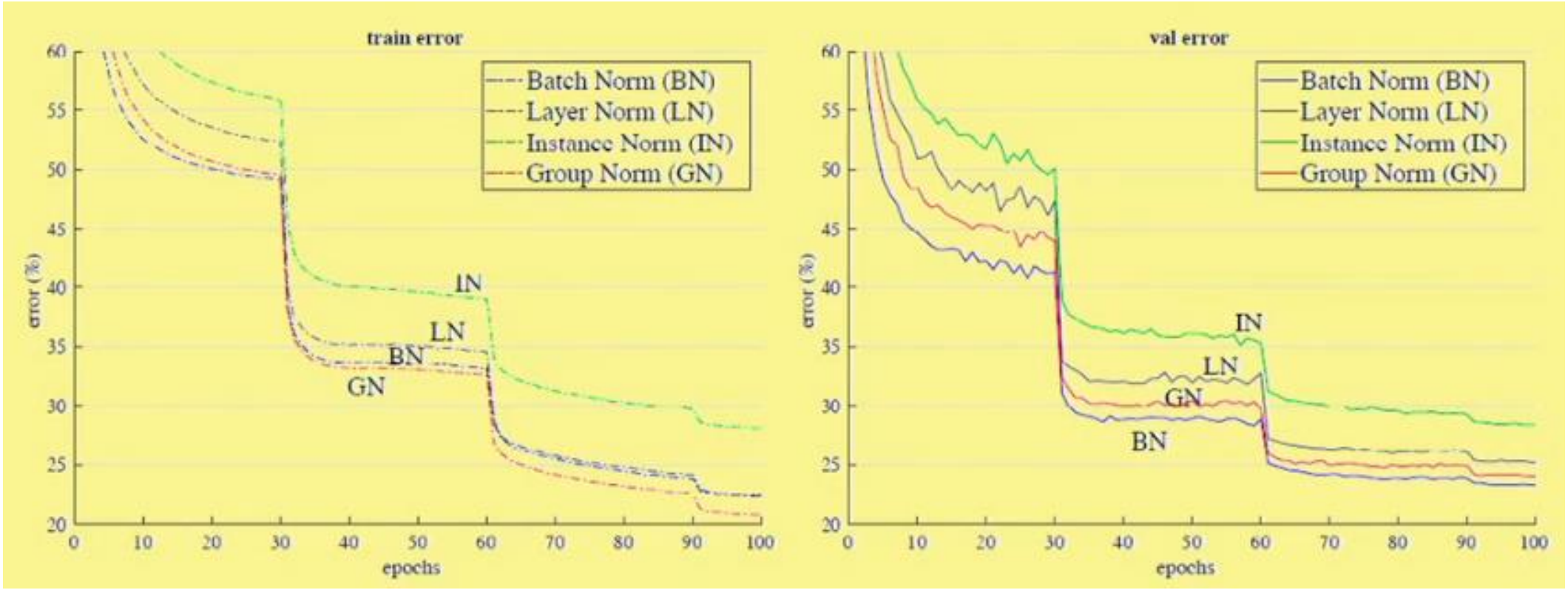
Layer normalization



All together



Comparison of different normalization (ECCV 18’)



Suggested reading

How Does Batch Normalization Help Optimization?

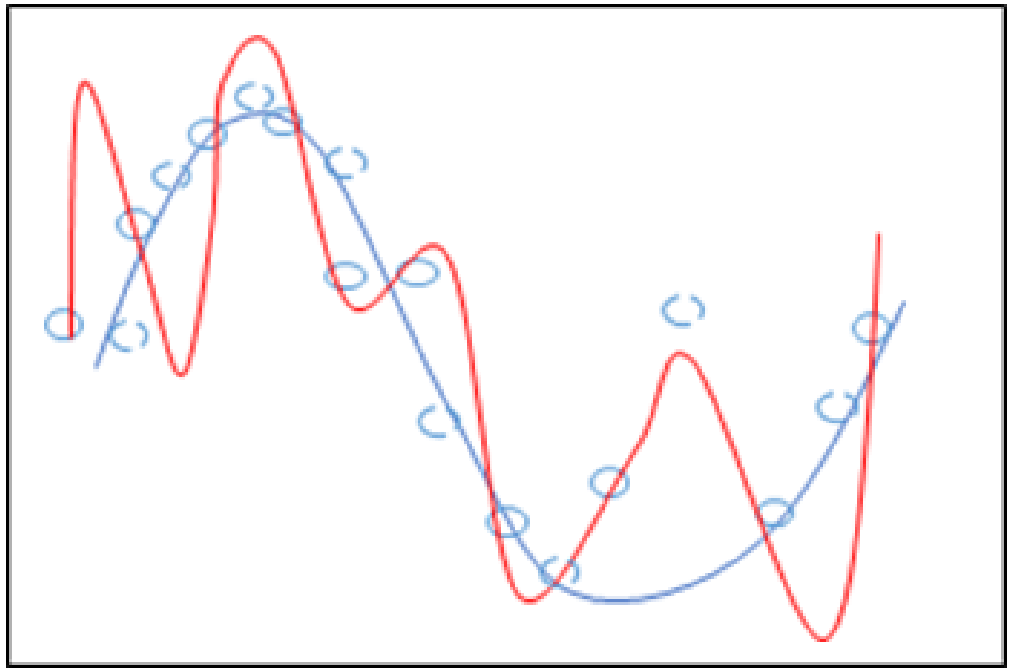
Shibani Santurkar* MIT shibani@mit.edu

Dimitris Tsipras* MIT tsipras@mit.edu

Andrew Ilyas* MIT ailyas@mit.edu

Aleksander Mądry MIT madry@mit.edu

Model regularization

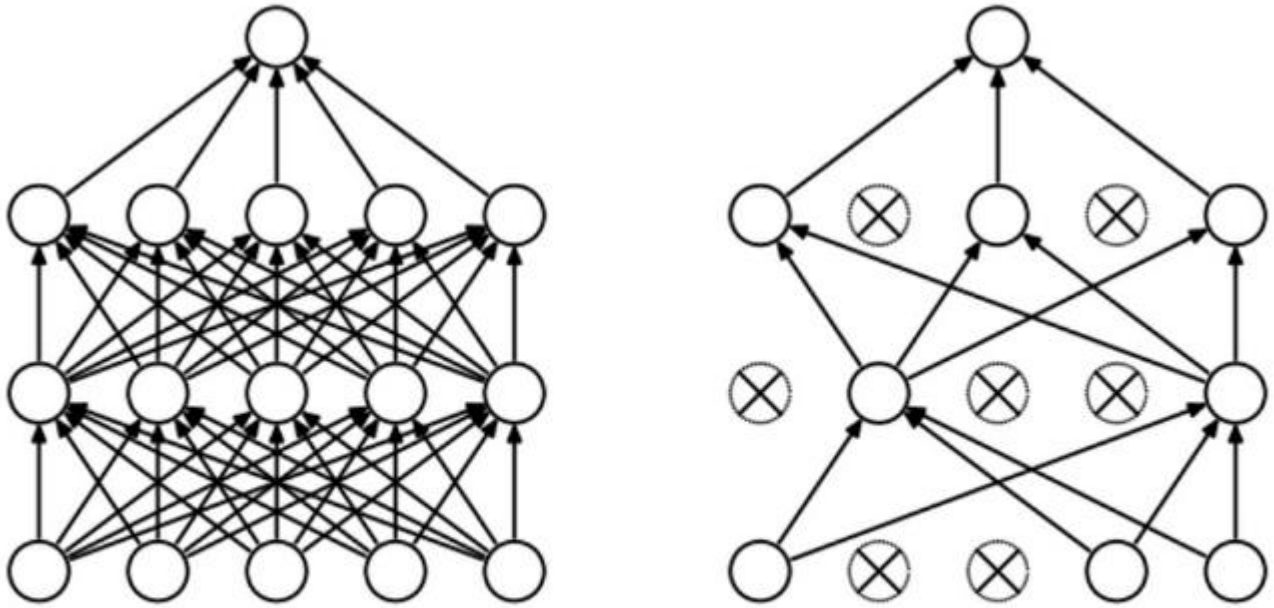


1. Less training data
2. Too many model parameters
3. Model overfitting
4. Regularization – ways to avoid overfitting

Regularization ways

- We have to ensure that the model does not overfit the training data – does not memorize the training samples
- Restrict the values the model parameters can take (lasso, ridge regressions)
- Ensemble model – mixture of experts

Dropout regularization



(a) Standard Neural Net

(b) After applying dropout.

Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting", JMLR 2014

Given

- $h(x) = xW + b$ a linear projection of a d_i-dimensional input x in a d_h-dimensional output space.
- $a(h)$ an activation function

it's possible to model the application of Dropout, in the training phase only, to the given projection as a modified activation function:

$$f(h) = D \odot a(h)$$

Where $D = (X_1, \cdots, X_{d_h})$ is a d_h-dimensional vector of Bernoulli variables X_i.

A Bernoulli random variable has the following probability mass distribution:

$$f(k; p) = \begin{cases} p & \text{if} \quad k = 1 \\ 1 - p & \text{if} \quad k = 0 \end{cases}$$

Where k are the possible outcomes.

$$O_i = X_i a(\sum_{k=1}^{d_i} w_k x_k + b) = \begin{cases} a(\sum_{k=1}^{d_i} w_k x_k + b) & \text{if} \quad X_i = 1 \\ 0 & \text{if} \quad X_i = 0 \end{cases}$$

where $P(X_i = 0) = p$.

Since during train phase a neuron is kept on with probability q, during the testing phase we have to emulate the behavior of the ensemble of networks used in the training phase.

To do this, the authors suggest scaling the activation function by a factor of q during the test phase in order to use the expected output produced in the training phase as the single output required in the test phase. Thus:

Train phase: $O_i = X_i a(\sum_{k=1}^{d_i} w_k x_k + b)$

Test phase: $O_i = qa(\sum_{k=1}^{d_i} w_k x_k + b)$

Inverted dropout

Train phase: $O_i = \frac{1}{q} X_i a(\sum_{k=1}^{d_i} w_k x_k + b)$

Test phase: $O_i = a(\sum_{k=1}^{d_i} w_k x_k + b)$

	$M = 0$	$M = 1$	$M = 3$	$M = 9$
$w_0^\star$	0.19	0.82	0.31	0.35
$w_1^\star$		-1.27	7.99	232.37
$w_2^\star$			-25.43	-5321.83
$w_3^\star$			17.37	48568.31
$w_4^\star$				-231639.30
$w_5^\star$				640042.26
$w_6^\star$				-1061800.52
$w_7^\star$				1042400.18
$w_8^\star$				-557682.99
$w_9^\star$				125201.43

Dropout + L2 normalization

$$\mathcal{L}(y, \hat{y}) = \mathcal{E}(y, F(W; x)) + \frac{\lambda}{2} W^2$$

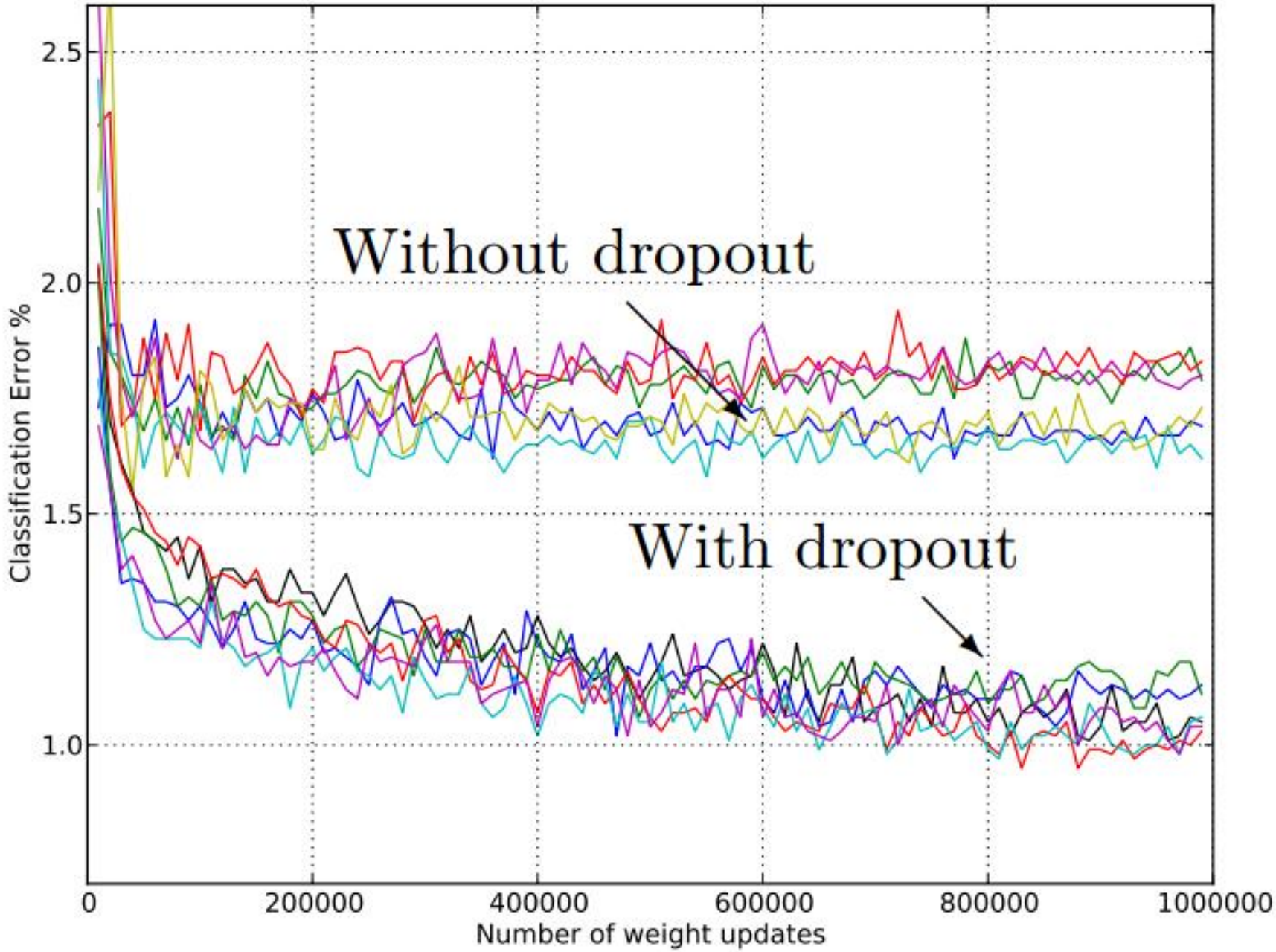
Dropout can’t control the range of values the weight variables can take!

Dropout

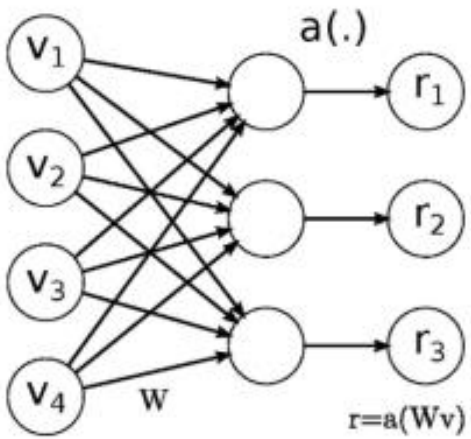
$$w \leftarrow w - \eta \left(\frac{\partial F(W; x)}{\partial w} + \lambda w \right)$$

Inverted Dropout

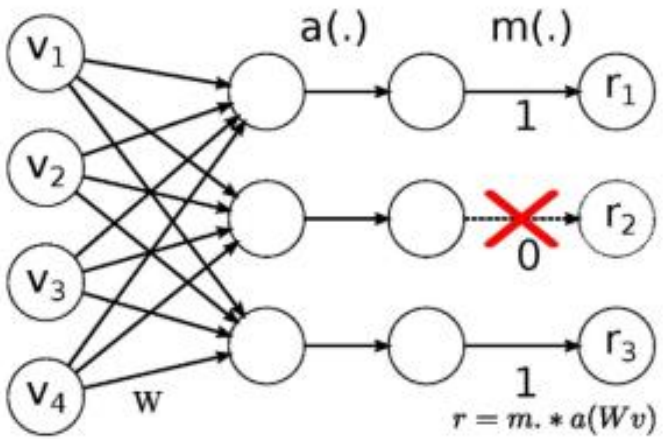
$$w \leftarrow w - \eta \left(\frac{1}{q} \frac{\partial F(W; x)}{\partial w} + \lambda w \right)$$


Without dropout
With dropout
Classification Error %
2.5
2.0
1.5
1.0
0
200000
400000
600000
800000
1000000
Number of weight updates

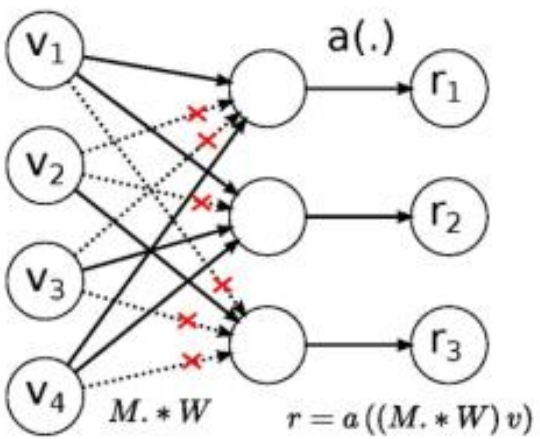
DropConnect



No-Drop Network



DropOut Network



DropConnect Network

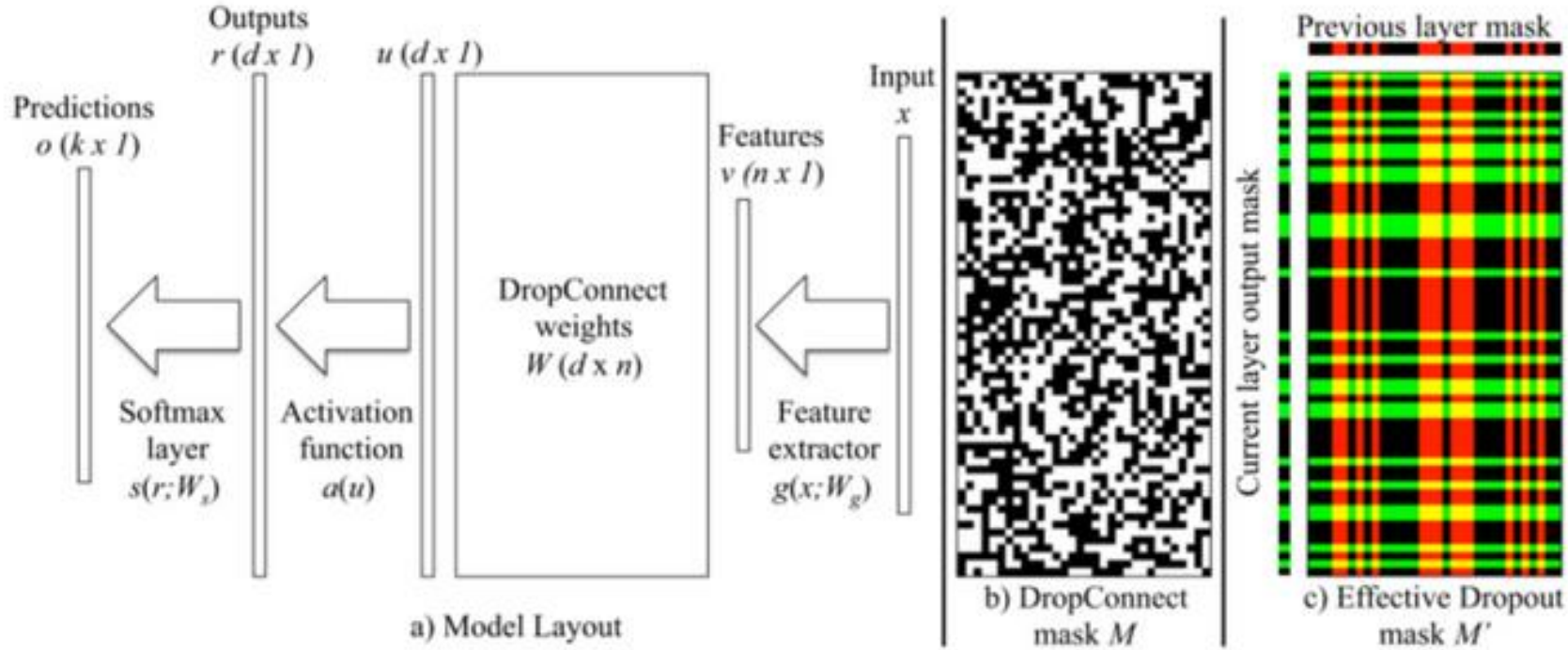
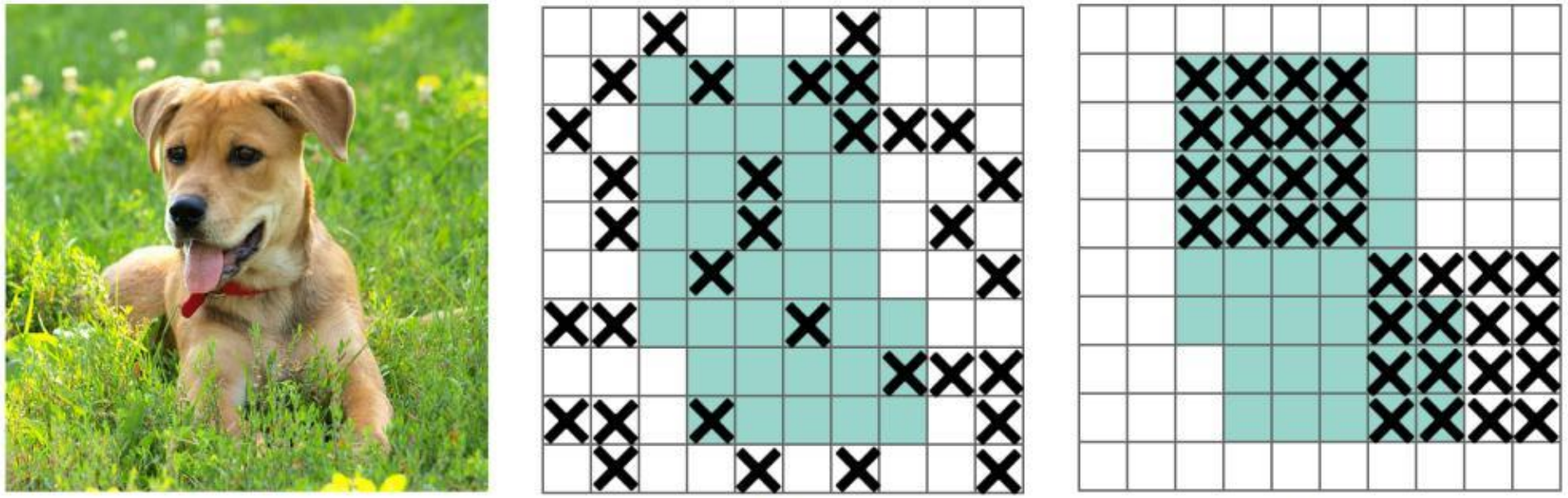


Fig 3. After Wan et al. (2013) (a): An example model layout for a single DropConnect layer. After running feature extractor g() on input x, a random instantiation of the mask M (e.g. (b)), masks out the weight matrix W. The masked weights are multiplied with this feature vector to produce u which is the input to an activation function a and a softmax layer s. For comparison, c) shows an effective weight mask for elements that Dropout uses when applied to the previous layer's output (columns) and this layer's output (rows).

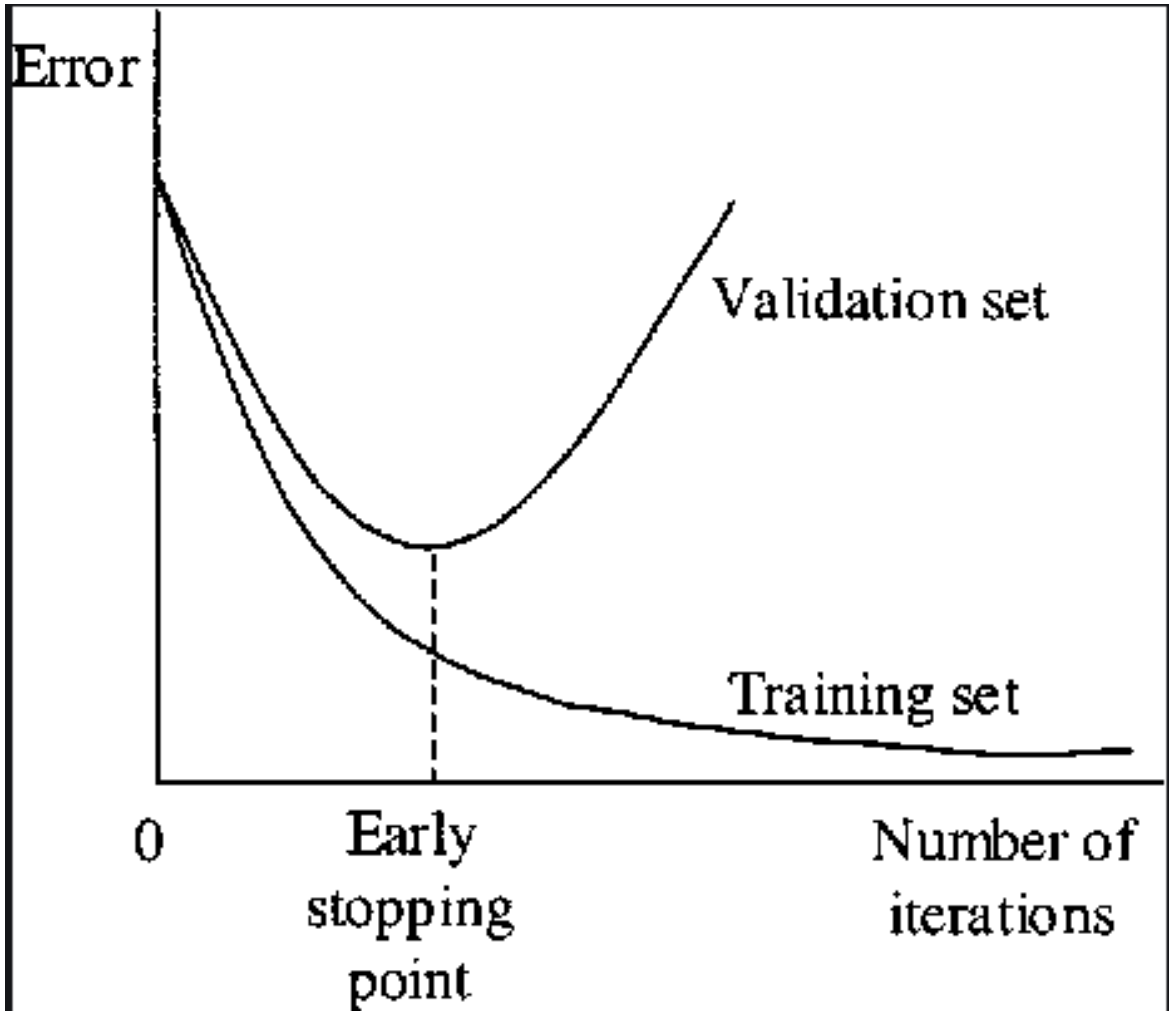
Dropblock



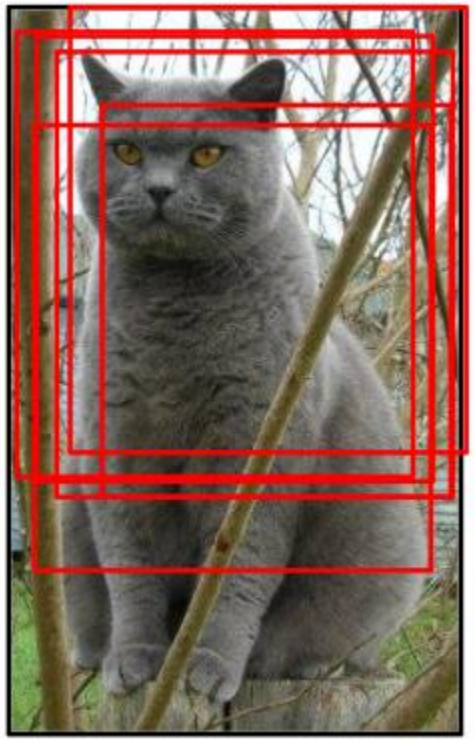
(a) (b) (c)

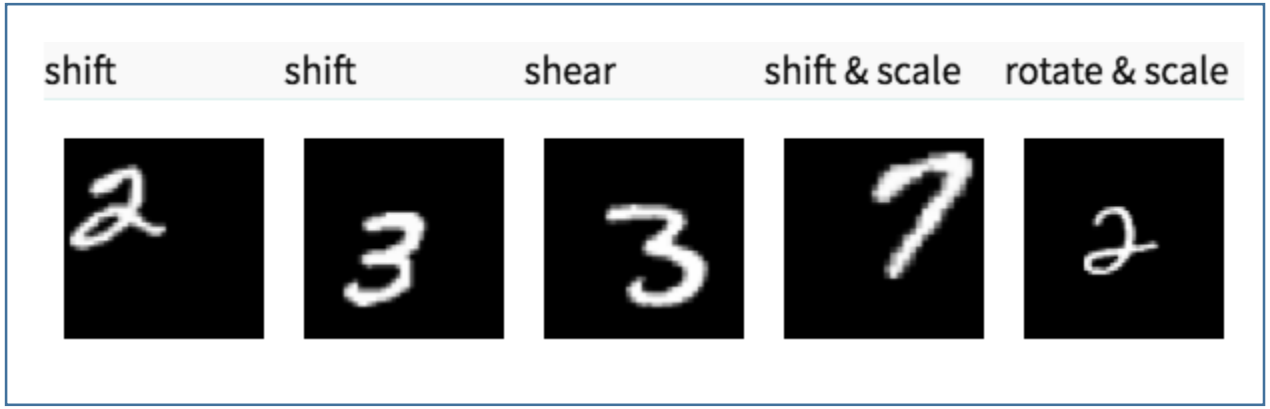
Figure 1: (a) input image to a convolutional neural network. The green regions in (b) and (c) include the activation units which contain semantic information in the input image. Dropping out activations at random is not effective in removing semantic information because nearby activations contain closely related information. Instead, dropping continuous regions can remove certain semantic information (e.g., head or feet) and consequently enforcing remaining units to learn features for classifying input image.

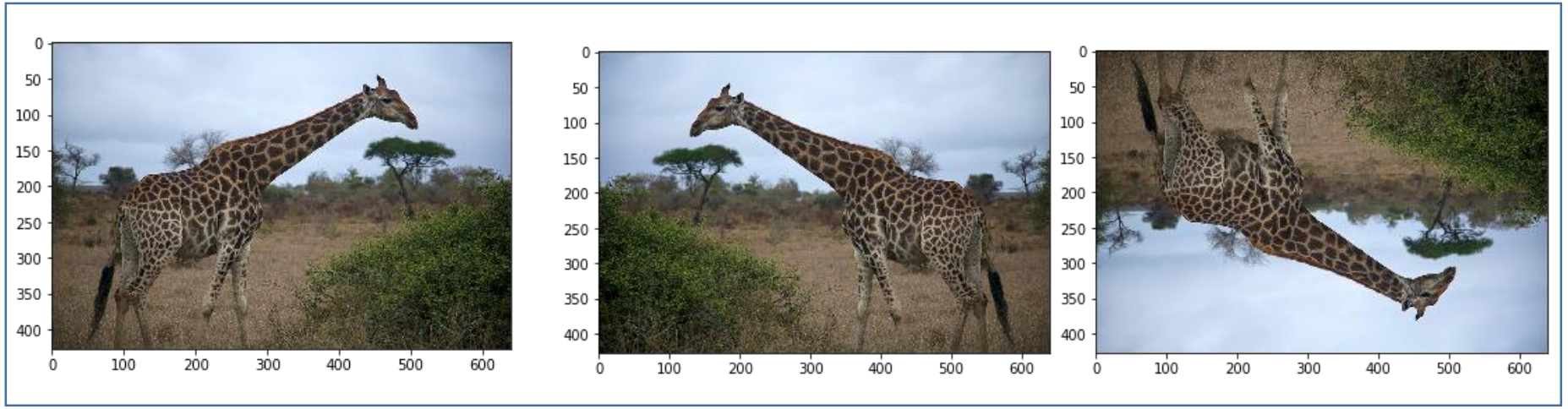
Early stopping – avoid over-training



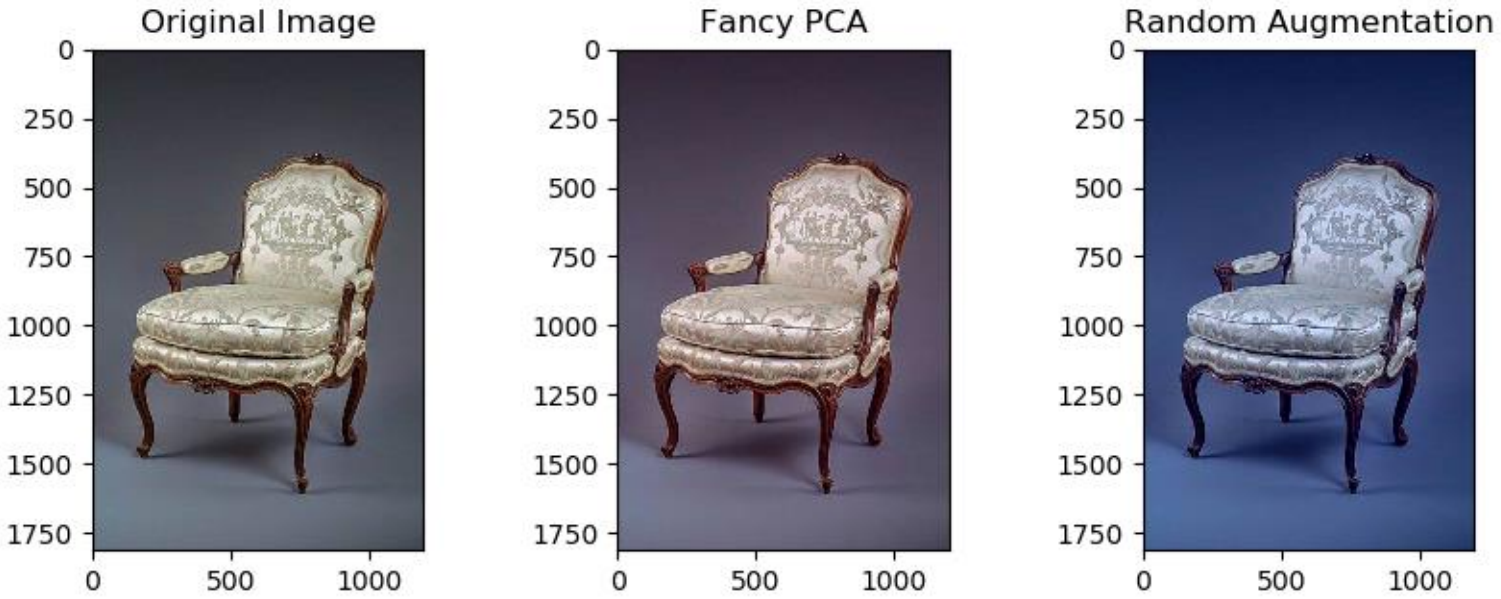
Data augmentation







Fancy PCA



$$I_{xy} = [I^R_{xy}, I^G_{xy}, I^B_{xy}]^T + [p1, p2, p3][\alpha_1\lambda_1, \alpha_2\lambda_2, \alpha_3\lambda_3]^T$$

p denote the Eigenvector, \lambda Denote the eigenvalue \alpha is a random noise

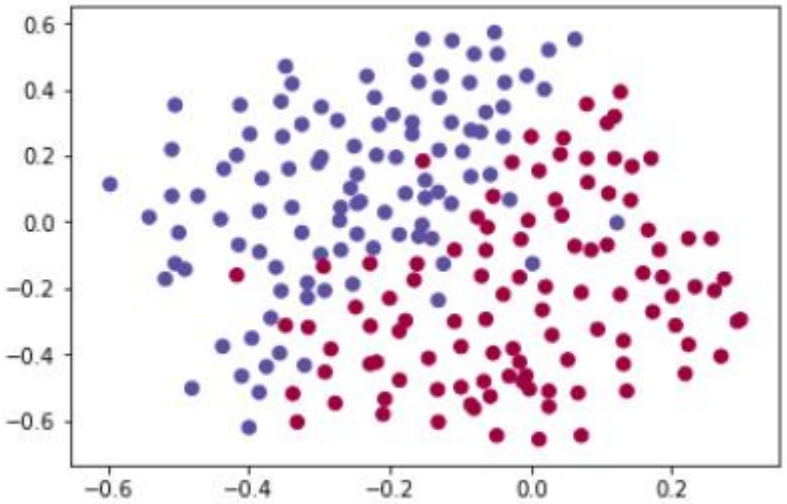
Suggested reading

Improving Deep Learning using Generic Data Augmentation

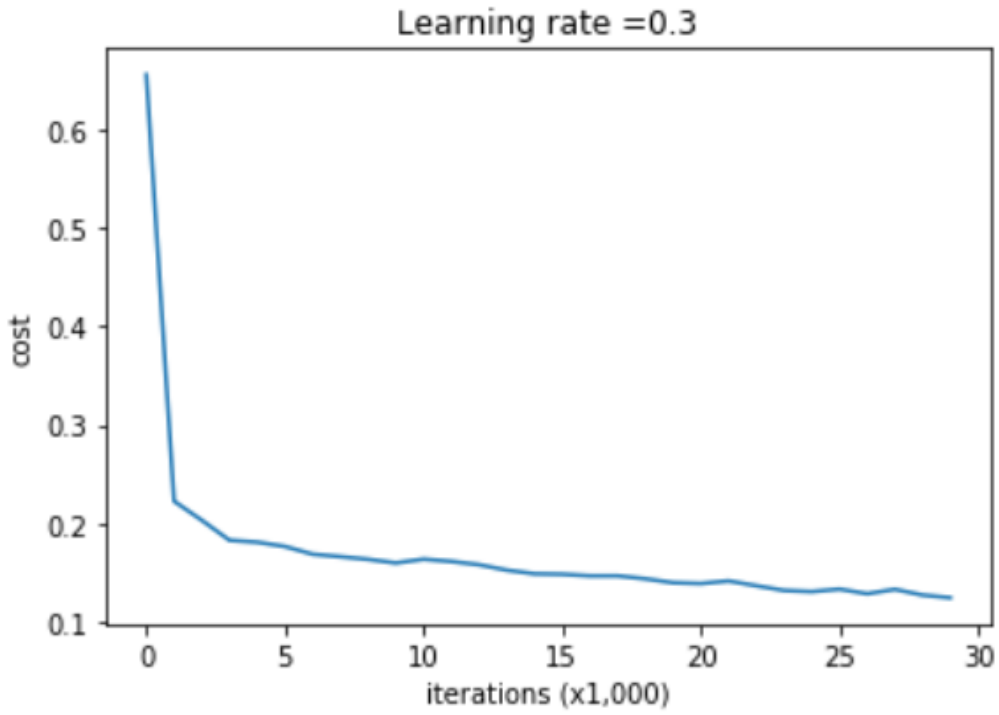
Luke Taylor
University of Cape Town
Department of Computer Science
Cape Town, South Africa
tylchr011@uct.ac.za

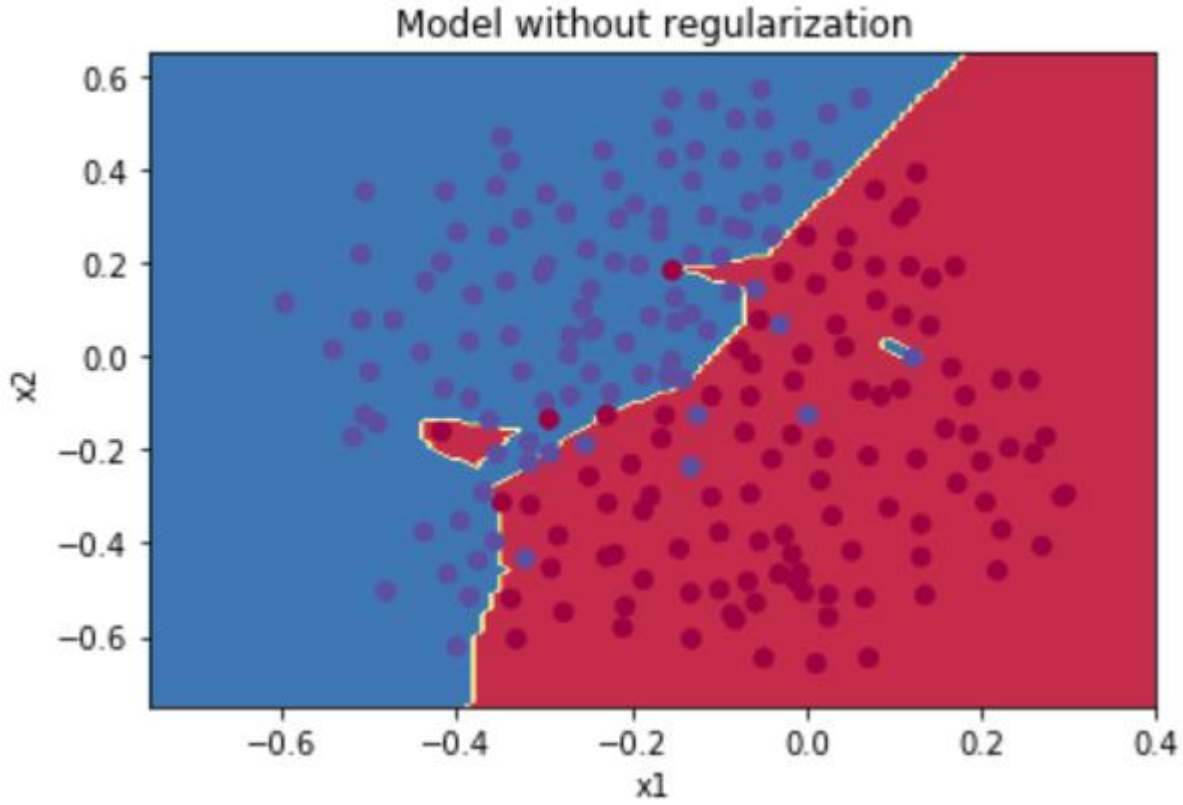
Geoff Nitschke
University of Cape Town
Department of Computer Science
Cape Town, South Africa
gnitschke@cs.uct.ac.za

Some comparison



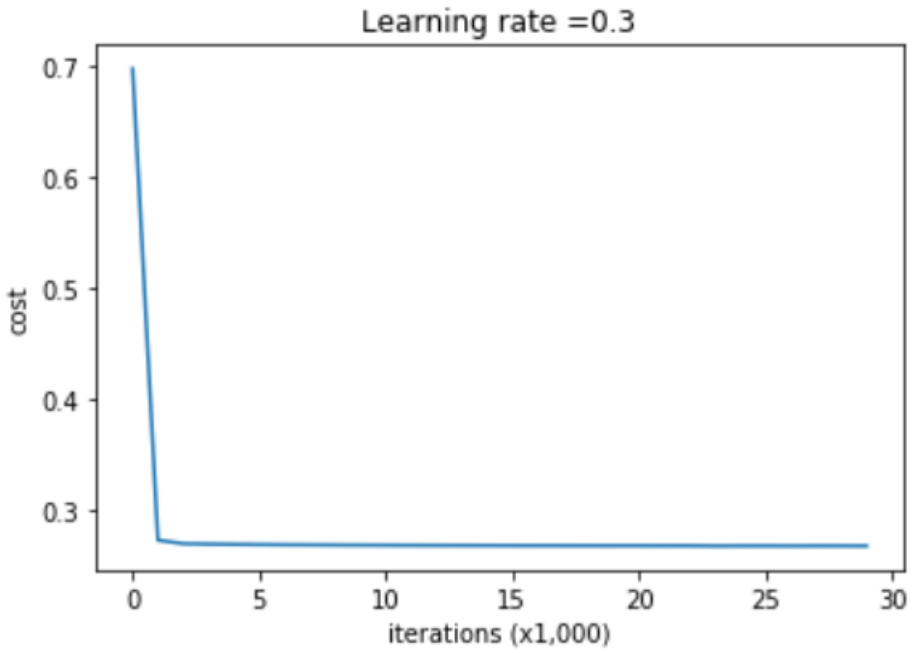
Non-regularized version

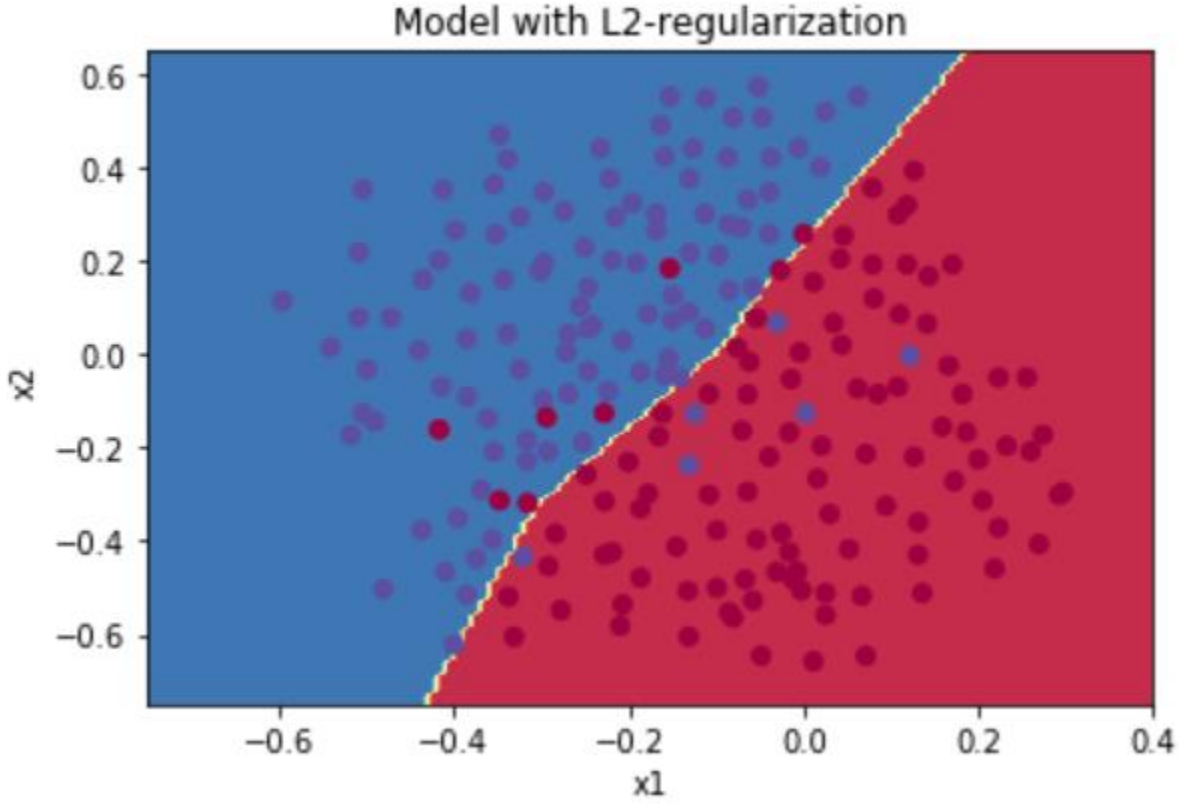





```
On the training set:
Accuracy: 0.9478672985781991
On the test set:
Accuracy: 0.915
```

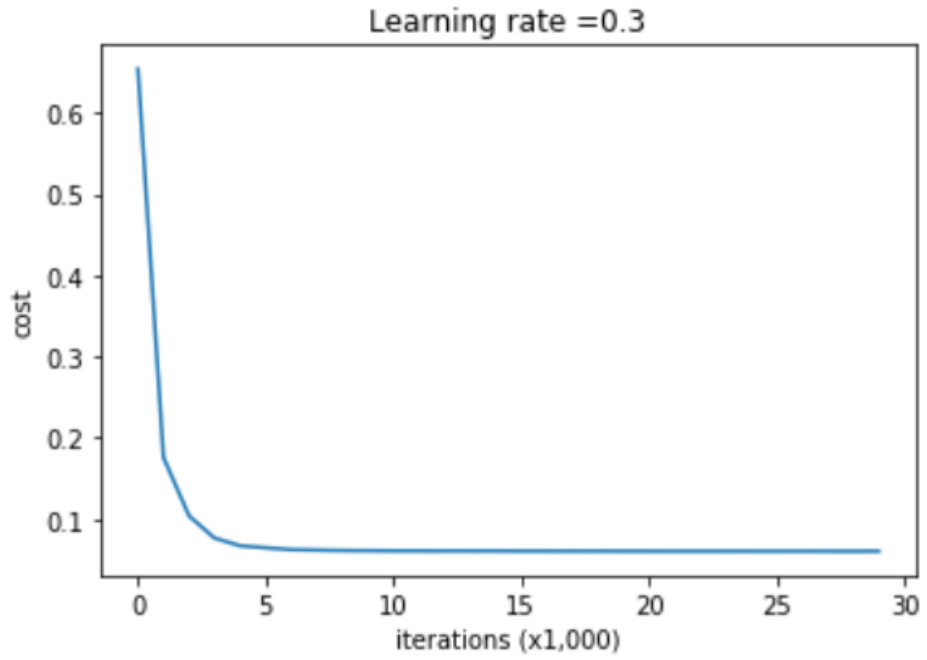
L2 regularization

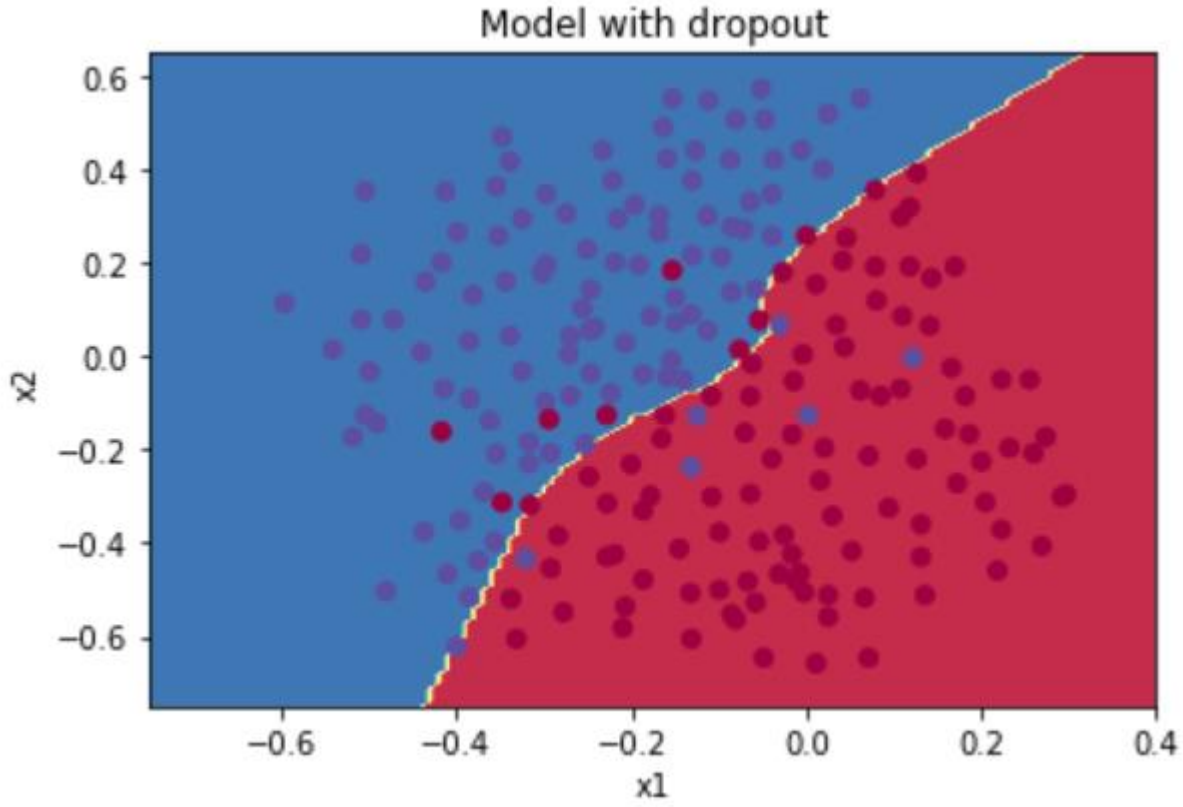





```
On the train set:
Accuracy: 0.9383886255924171
On the test set:
Accuracy: 0.93
```

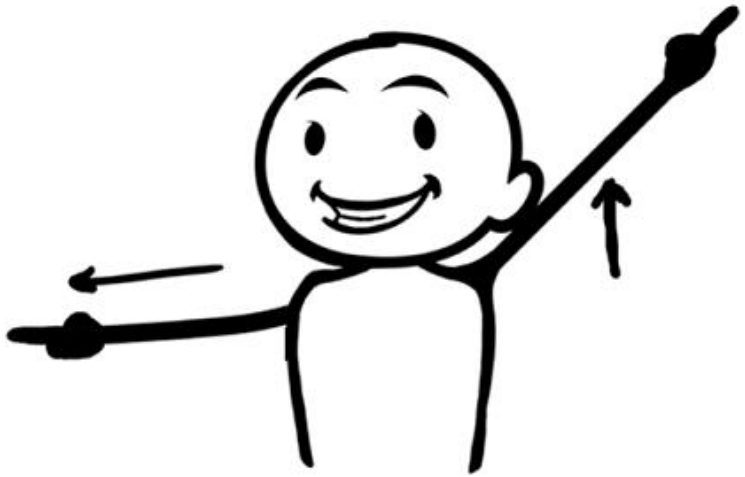
Dropout






```
On the train set:
Accuracy: 0.9289099526066351
On the test set:
Accuracy: 0.95
```

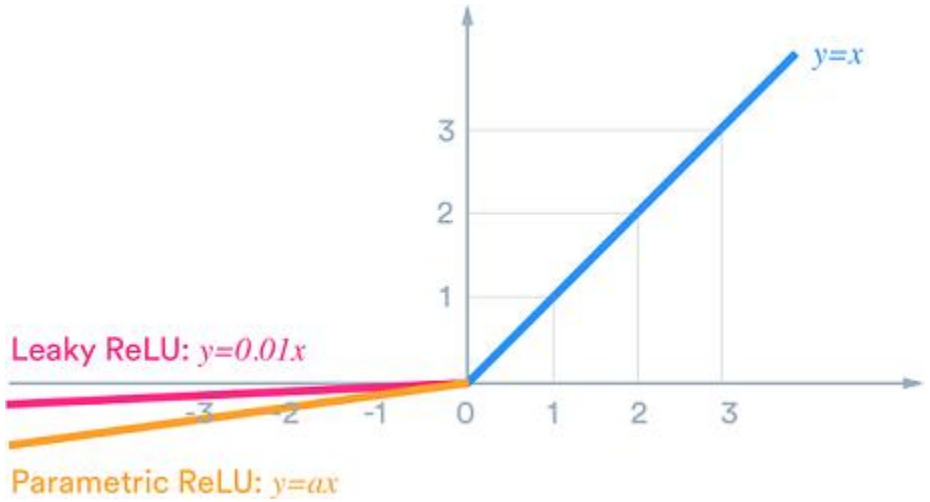
Some notes on the ReLU function



Idea of Dying ReLU

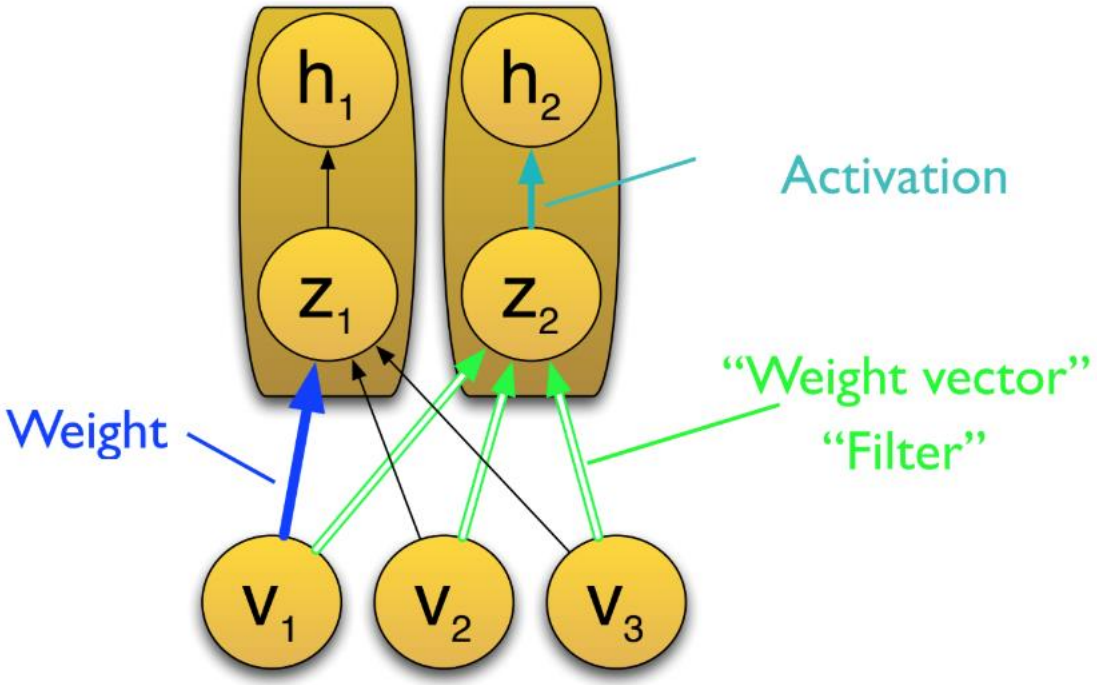
Need to ensure some gradient pass if x<0

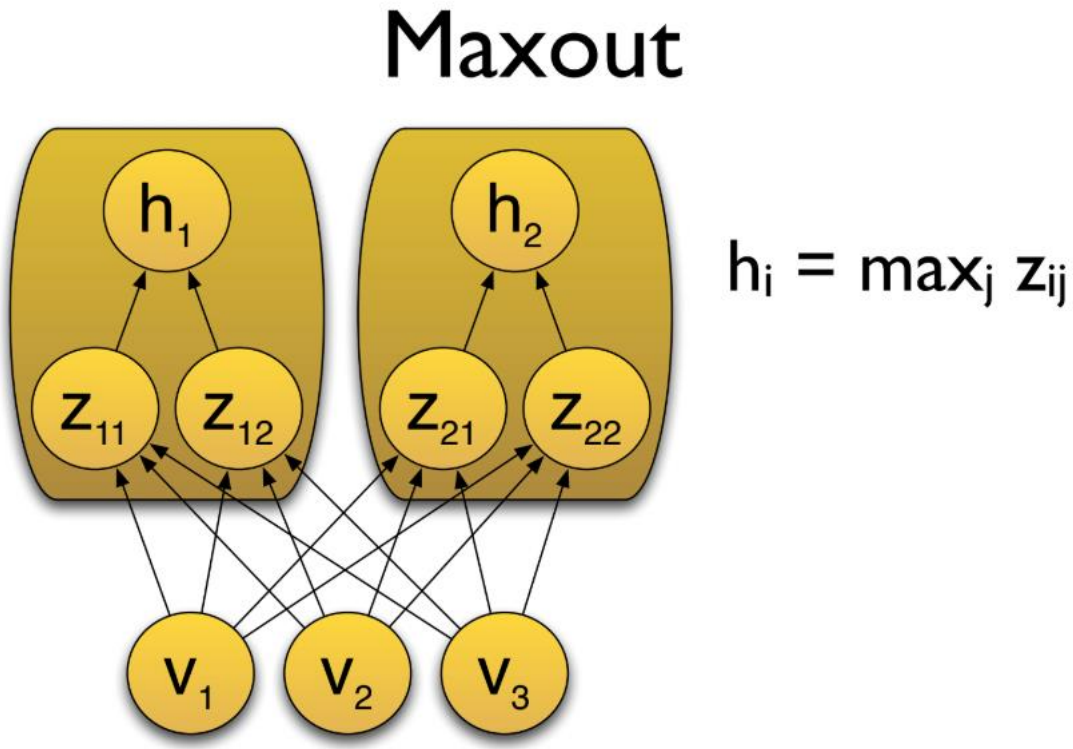
Leaky, parametric, exponential ReLU

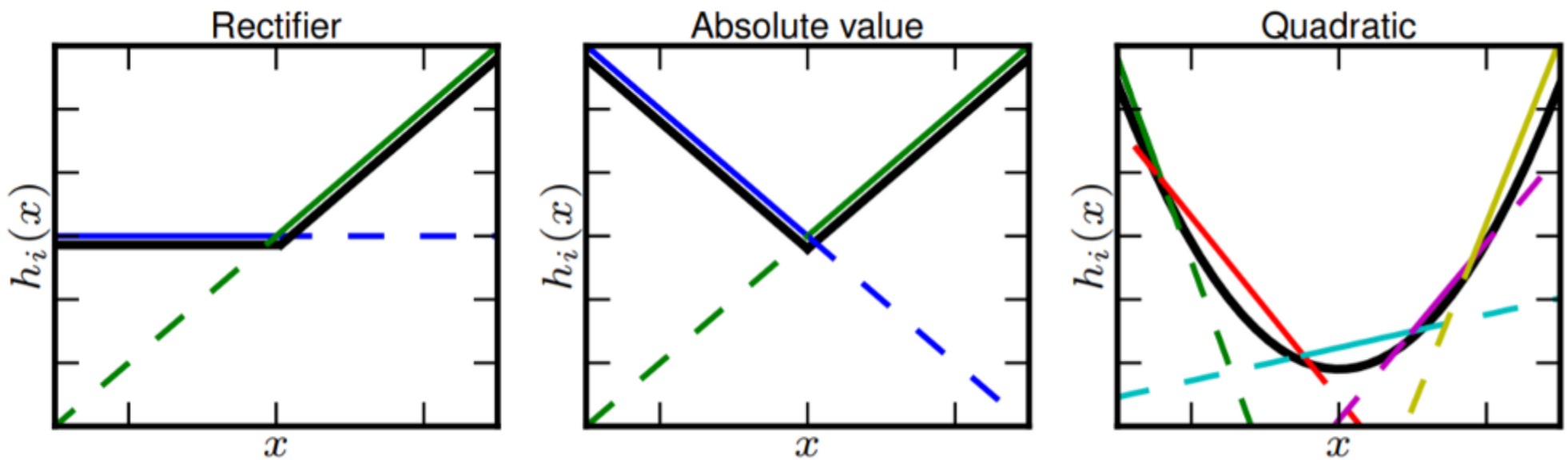


$$y = \begin{cases} \alpha(e^x - 1), & x < 0 \\ x, & x \geq 0 \end{cases}$$

Maxout activation







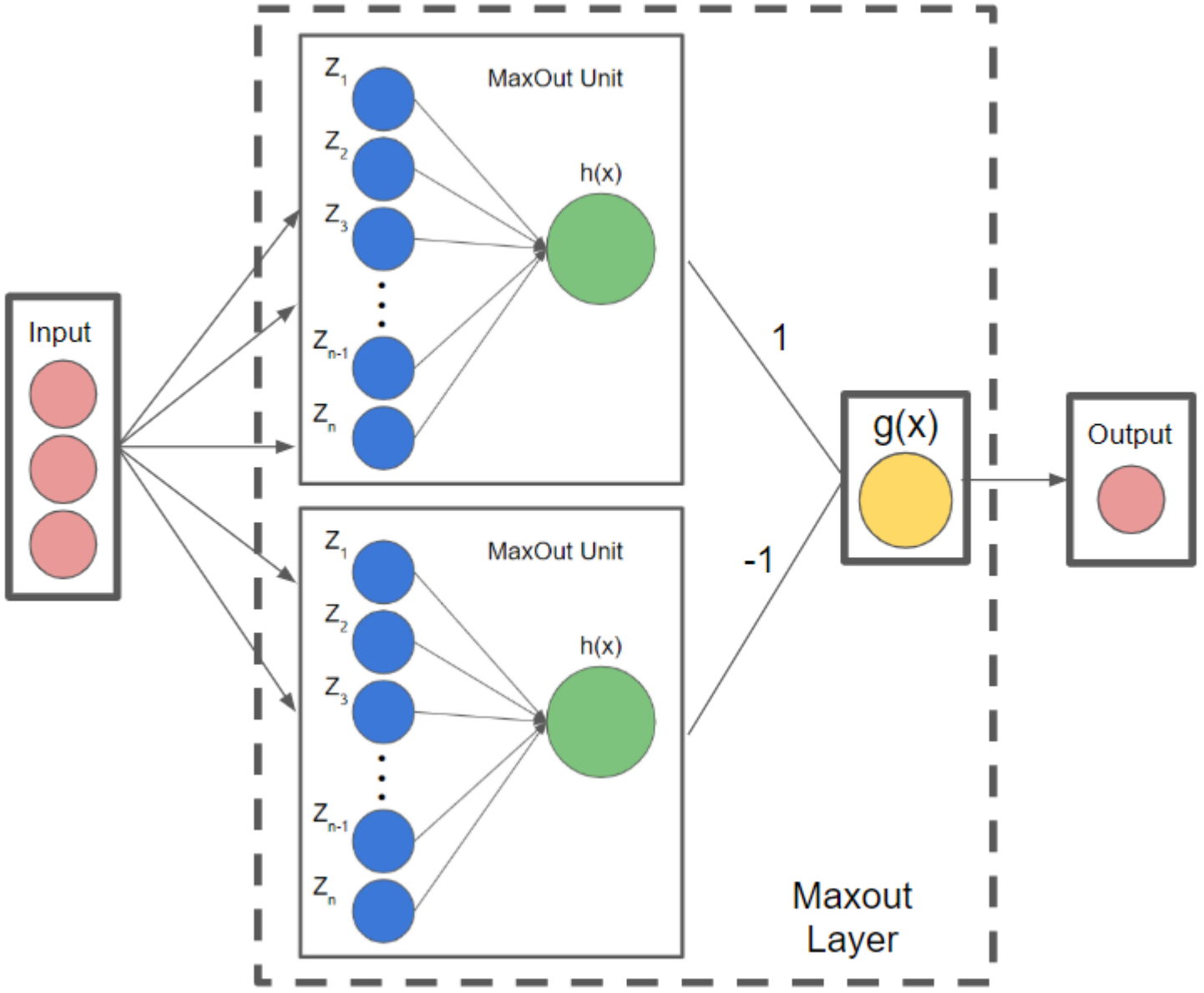
$$ReLU = \max(0, x), \ \text{abs}(x) = \max(x, \ -x)$$

Any continuous PWL function can be expressed as a difference of two convex PWL functions.

Consider, two convex functions h1(x) and h2(x), approximated by two Maxout units. By the above preposition, the function **g(x) is a continuous PWL function.**

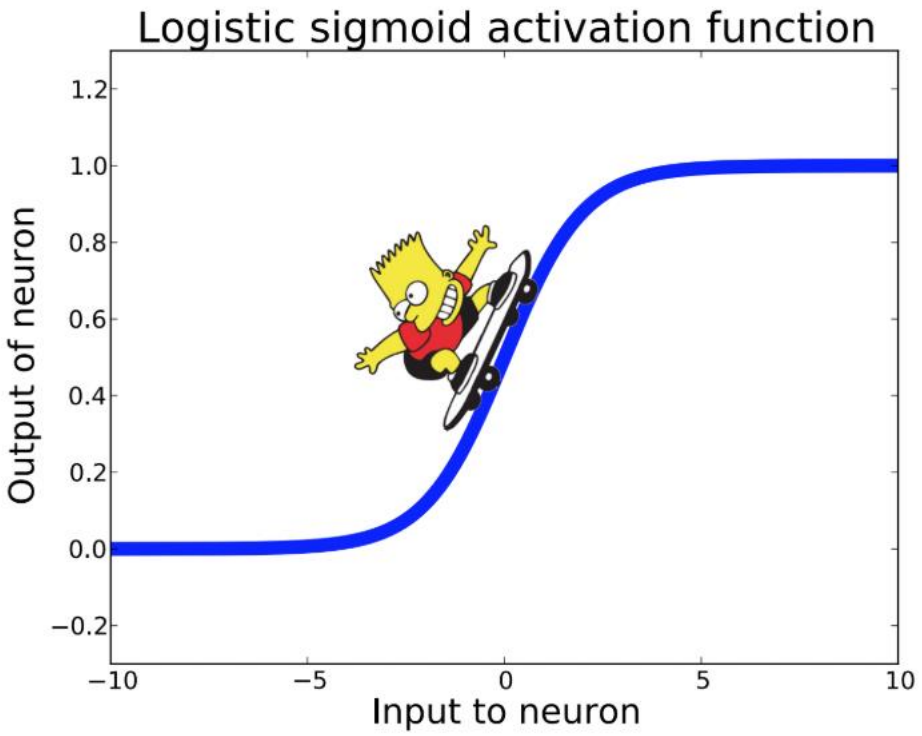
$$g(x) = h_1(x) - h_2(x)$$

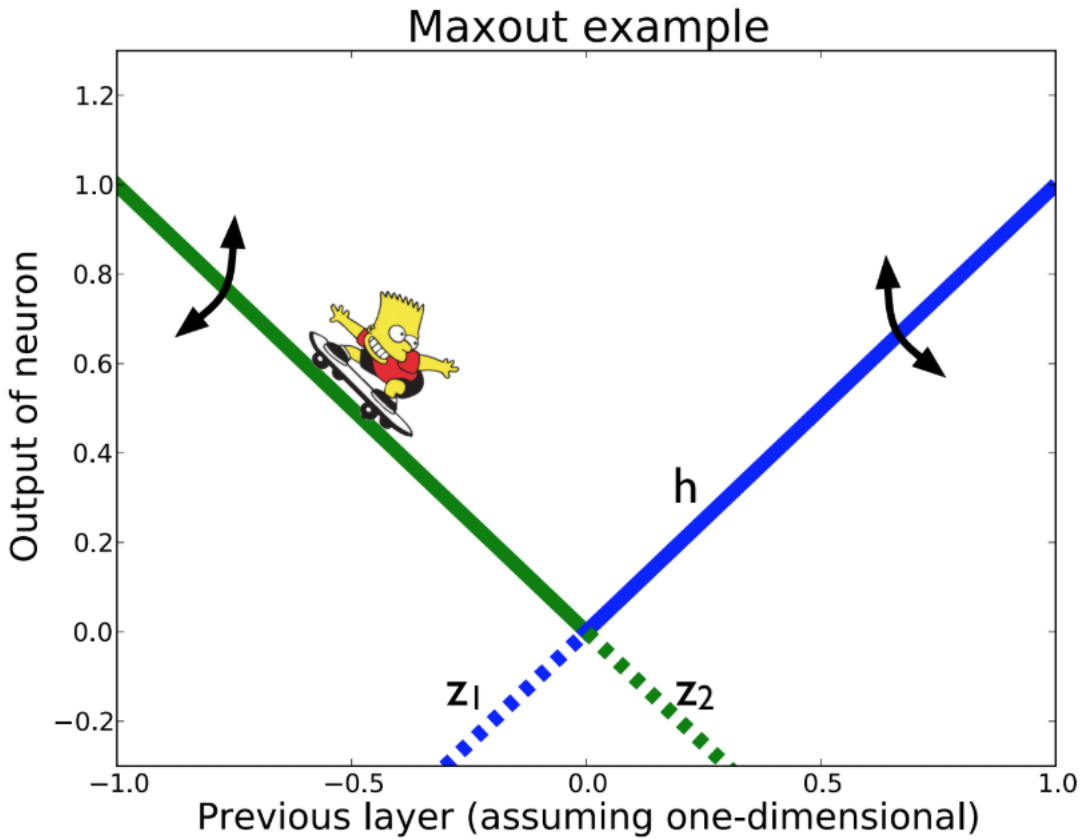
Thus, it is found that a **Maxout layer consisting of two Maxout units can approximate any continuous function *f*(*v*) arbitrarily well.**



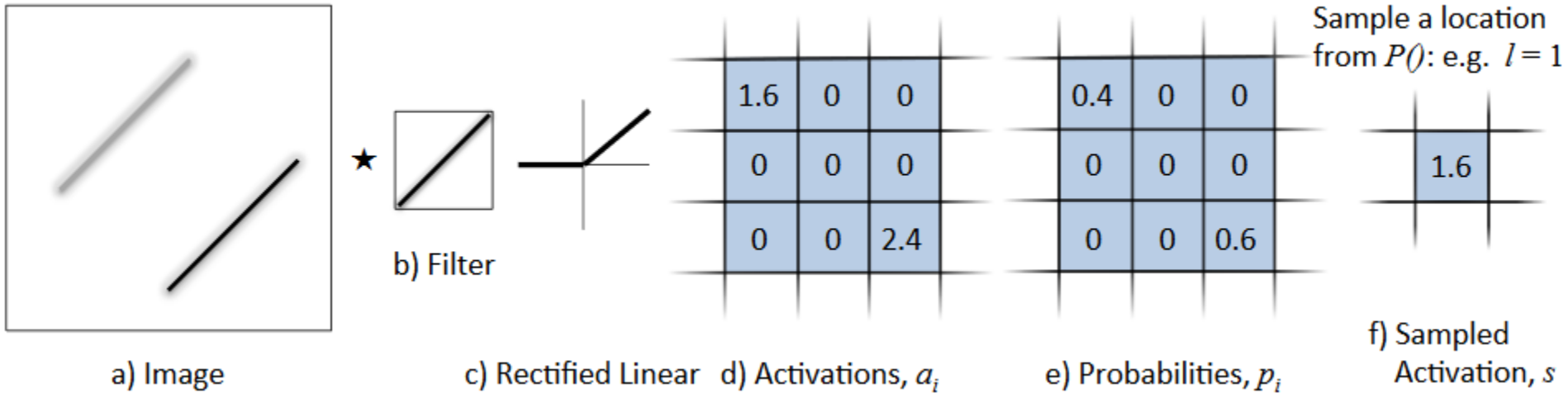
$$g(x) = h_1(x) - h_2(x)$$

Gradients everywhere!





Stochastic pooling



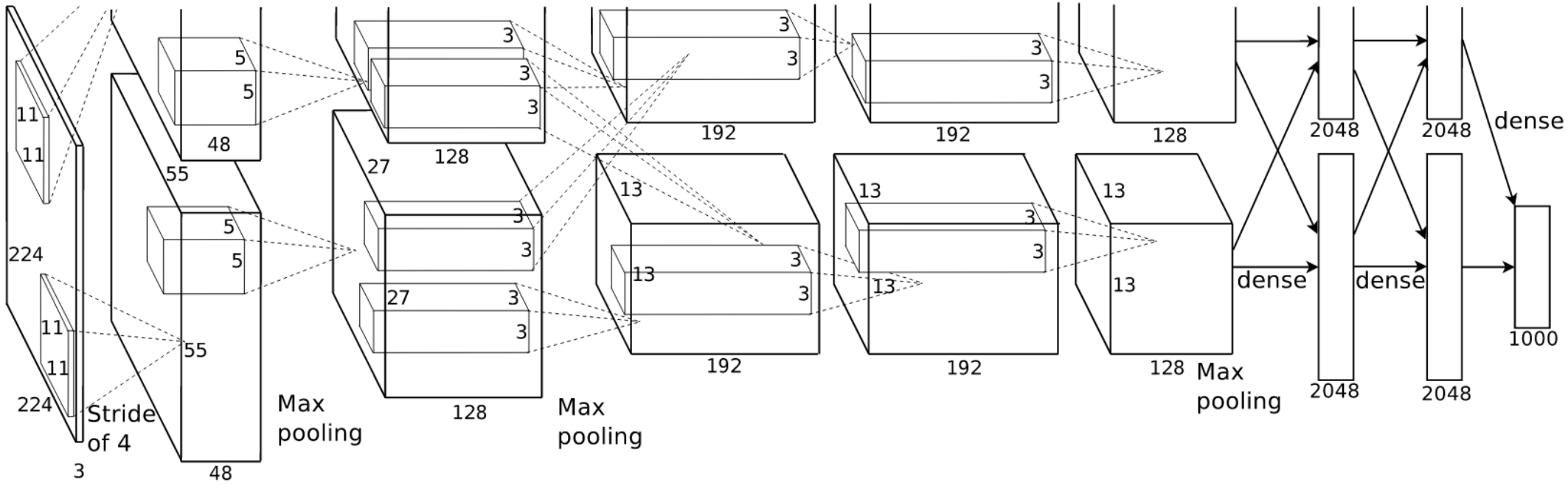
$$p_i = \frac{a_i}{\sum_{k \in R_j} a_k}$$

$$s_j = a_l \quad \text{where } l \sim P(p_1, \ldots, p_{|R_j|})$$

Popular CNN architecture

- Alexnet
- Zfnet
- VGG
- Inception (v1-v4)
- Resnet
- Resnext
- …

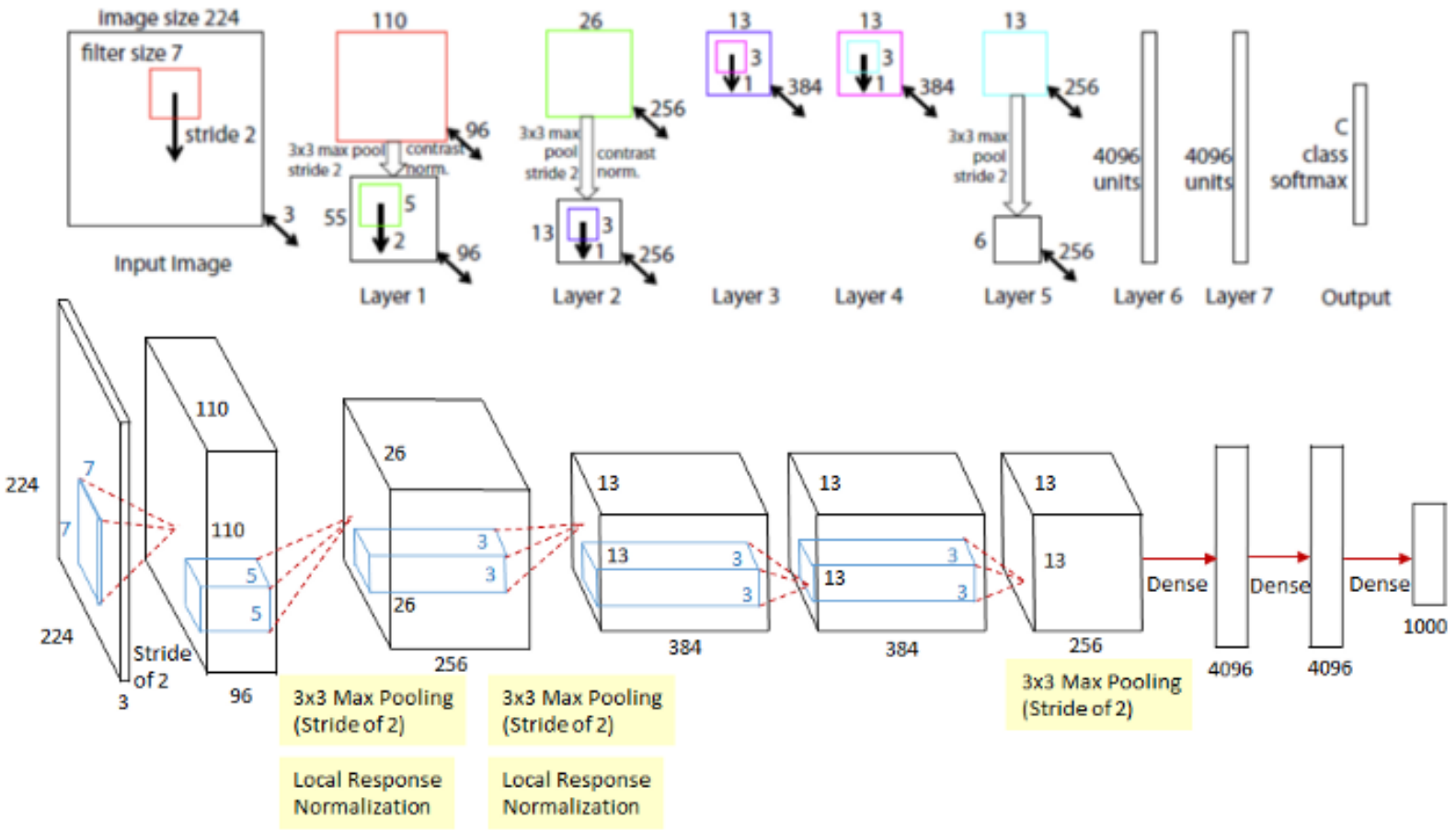
AlexNet


224
224
3
11
11
11
11
Stride
of 4
48
48
55
55
5
5
5
5
Max
pooling
27
27
128
128
3
3
3
3
3
3
Max
pooling
192
192
13
13
13
13
192
192
128
128
13
13
Max
pooling
dense
dense
2048
2048
2048
2048
dense
1000

Highlights

- First deep net in working mode after LeNet
- Eight layers
- ReLU non-linearity
- To avoid overfitting, some of the layers are skipped randomly during training

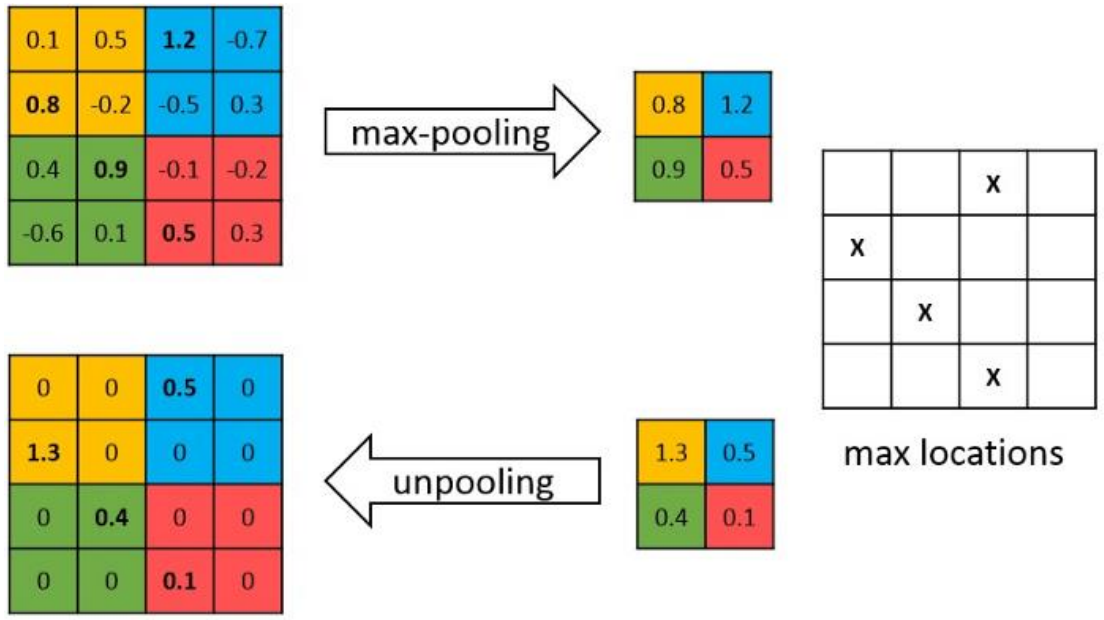
Zfnet



- Reduced stride to 2 from 4

Highlights

- Mainly used for visualization by adding a symmetric decoder
- Used unpool payer for upsampling

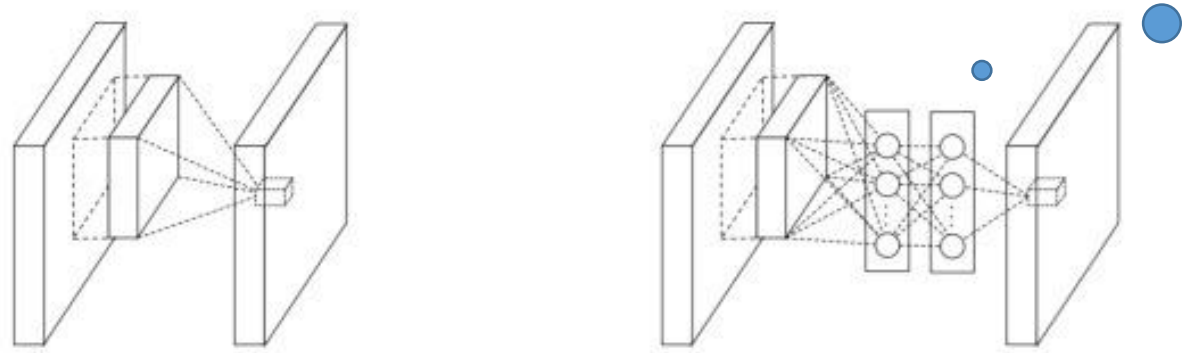


VGGnet


3x3 conv, 64
3x3 conv, 64
pool/2
3x3 conv, 128
3x3 conv, 128
pool/2
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
pool/2
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
pool/2
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
pool/2
fc 4096
fc 4096
fc 4096
Size:224
Size:112
Size:56
Size:28
Size:14
Size:7

Network in network

MLP-conv layer



(a) Linear convolution layer (b) Mlpconv layer

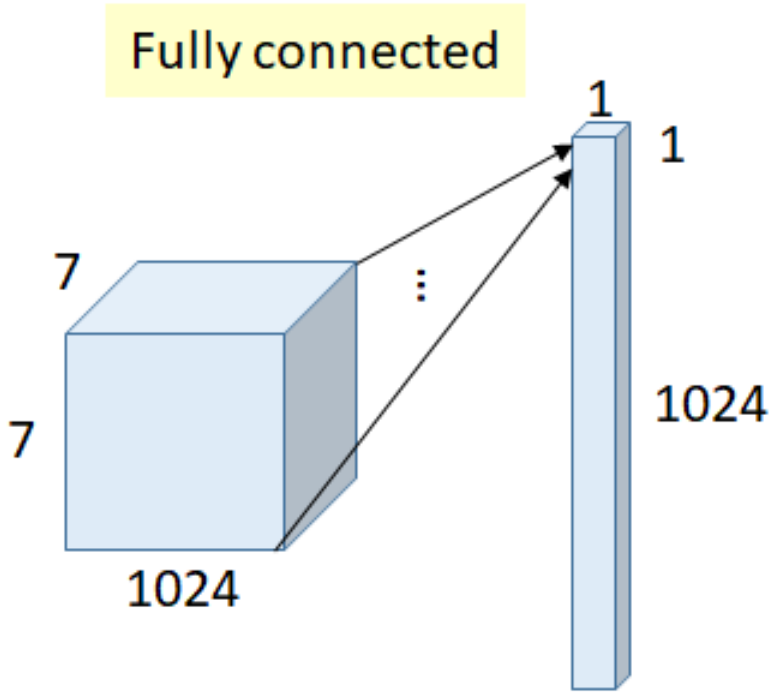
Figure 1: Comparison of linear convolution layer and mlpconv layer. The linear convolution layer includes a linear filter while the mlpconv layer includes a micro network (we choose the multilayer perceptron in this paper). Both layers map the local receptive field to a confidence value of the latent concept.

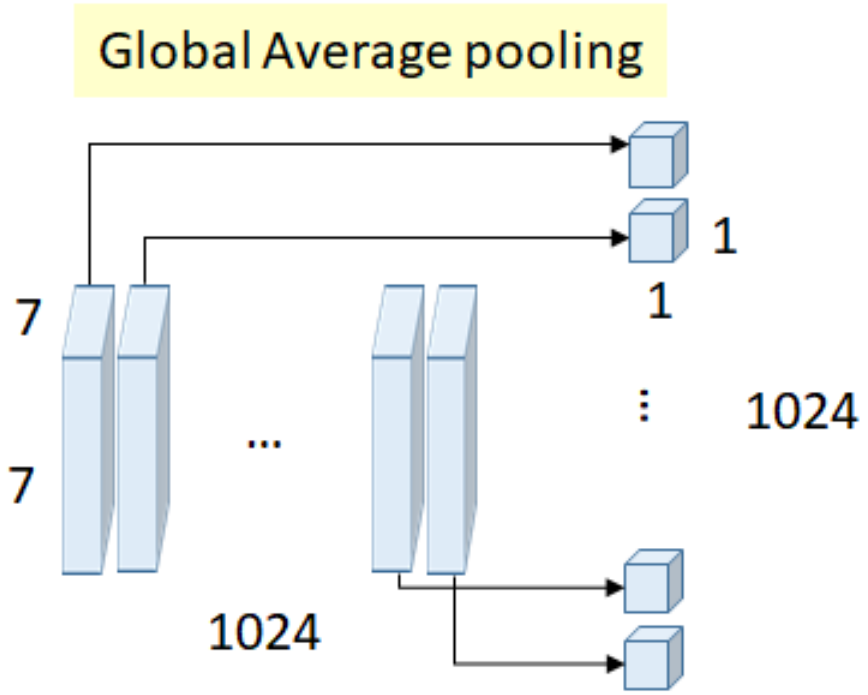
$$f_{i,j,k} = \max(w_k^T x_{i,j}, 0).$$

$$\begin{aligned} f^1_{i,j,k_1} &= \max({w^1_{k_1}}^T x_{i,j} + b_{k_1}, 0). \\ &\vdots \\ f^n_{i,j,k_n} &= \max({w^n_{k_n}}^T f^{n-1}_{i,j} + b_{k_n}, 0). \end{aligned}$$

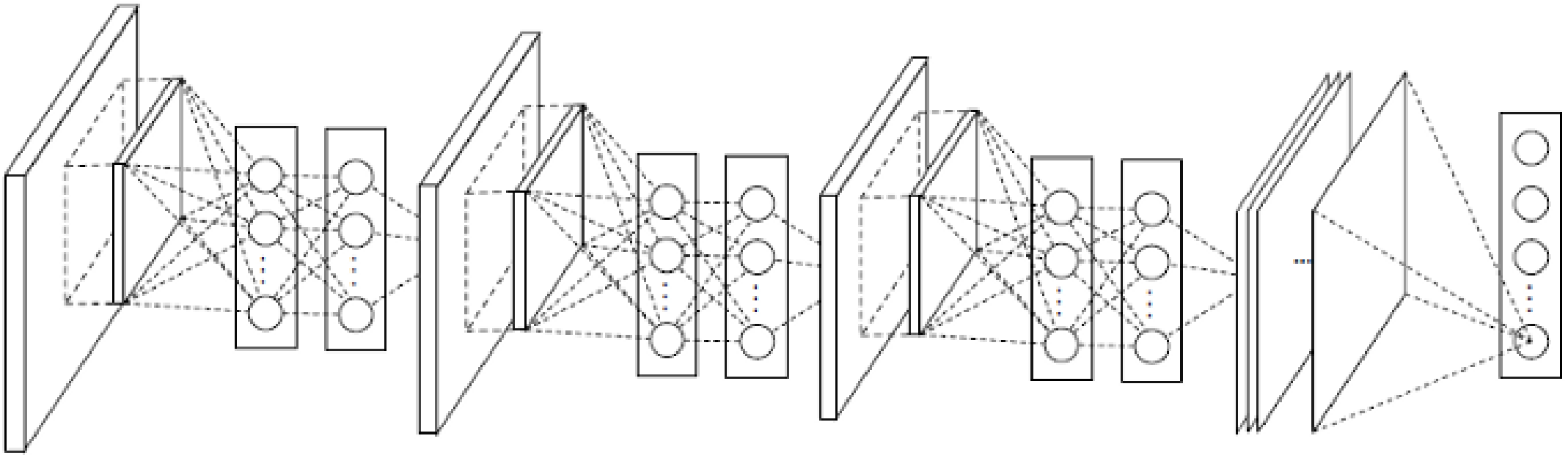
Representations that achieve good abstraction are non-linear function of the input data!

Network in network



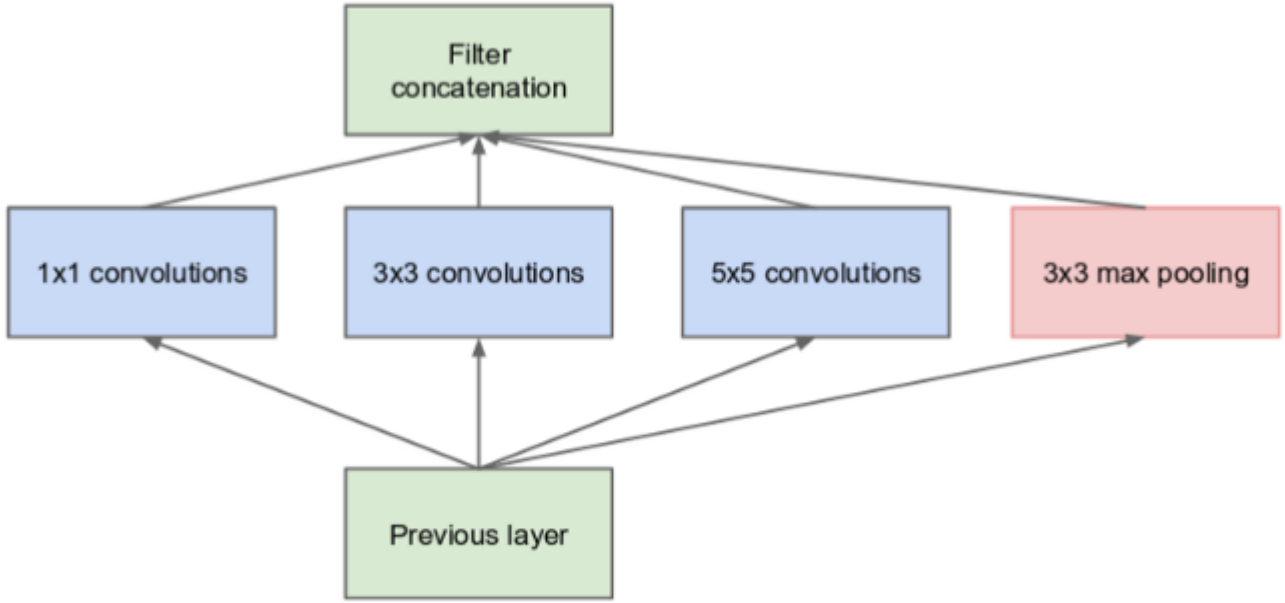


No learnable parameter!
Can work with images of different resolution!

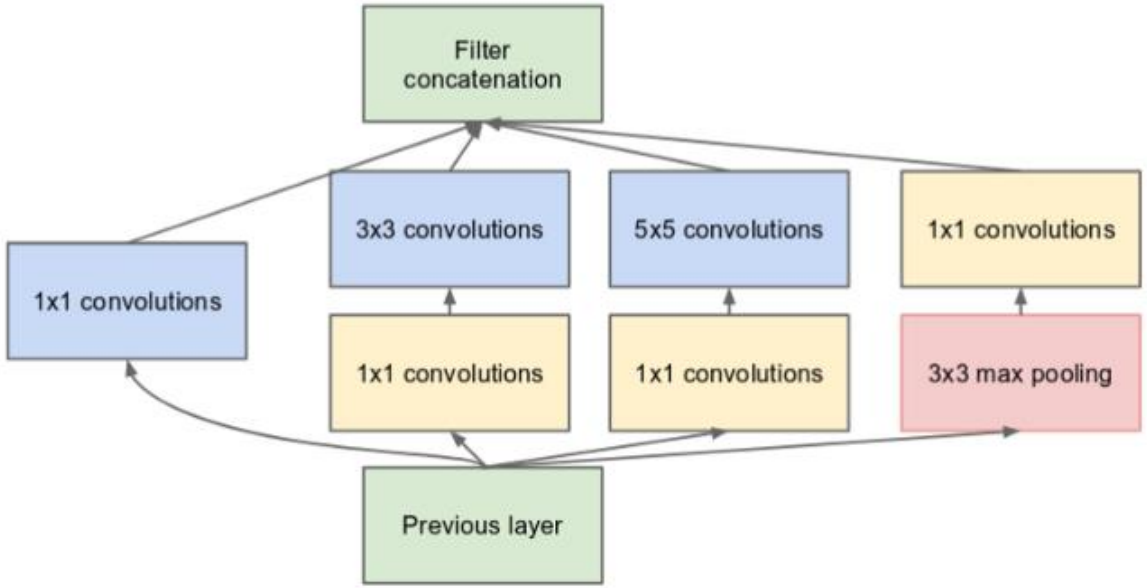


Inception architecture

What is a good kernel size for DOG feature extraction?

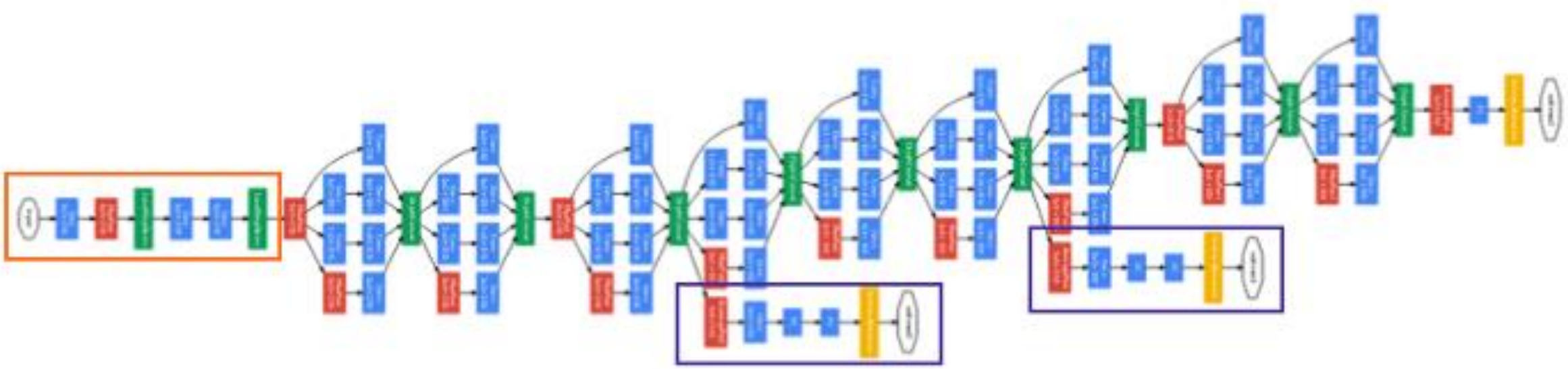


(a) Inception module, naïve version



(b) Inception module with dimension reductions

Inception v1 (Googlenet)



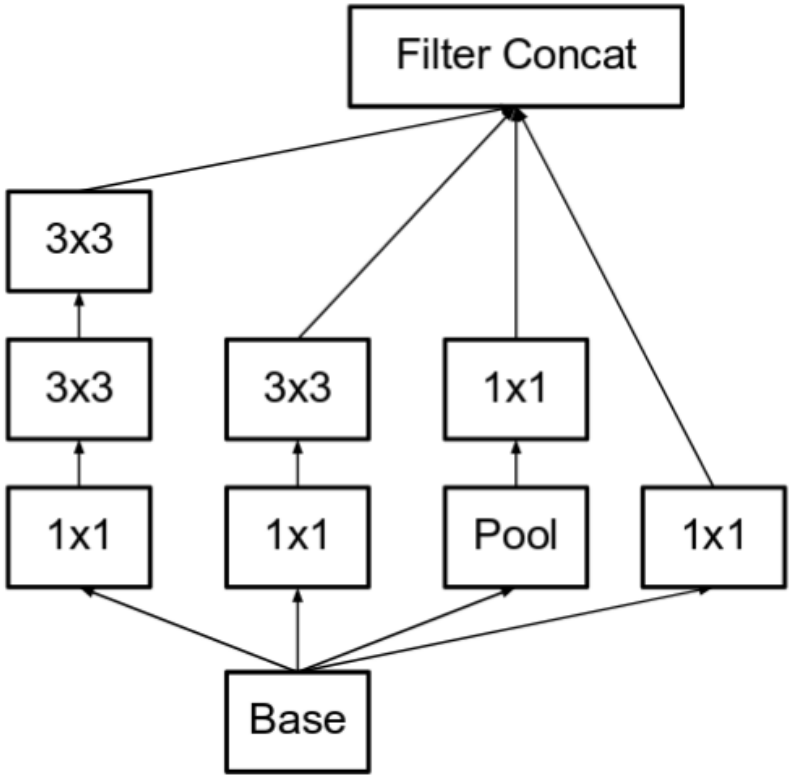
```
# The total loss used by the inception net during training.
total_loss = real_loss + 0.3 * aux_loss_1 + 0.3 * aux_loss_2
```

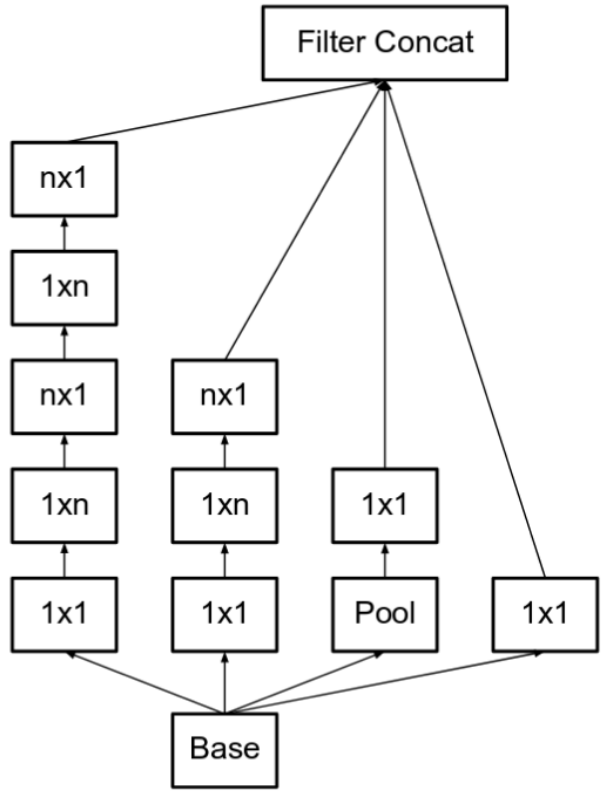
9 inception module, 22 layers, global average pooling at the end, 2 intermediate classifier to take care
Of the ‘dying out’ effect

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

Inception v2

- Filter separability and small filter size

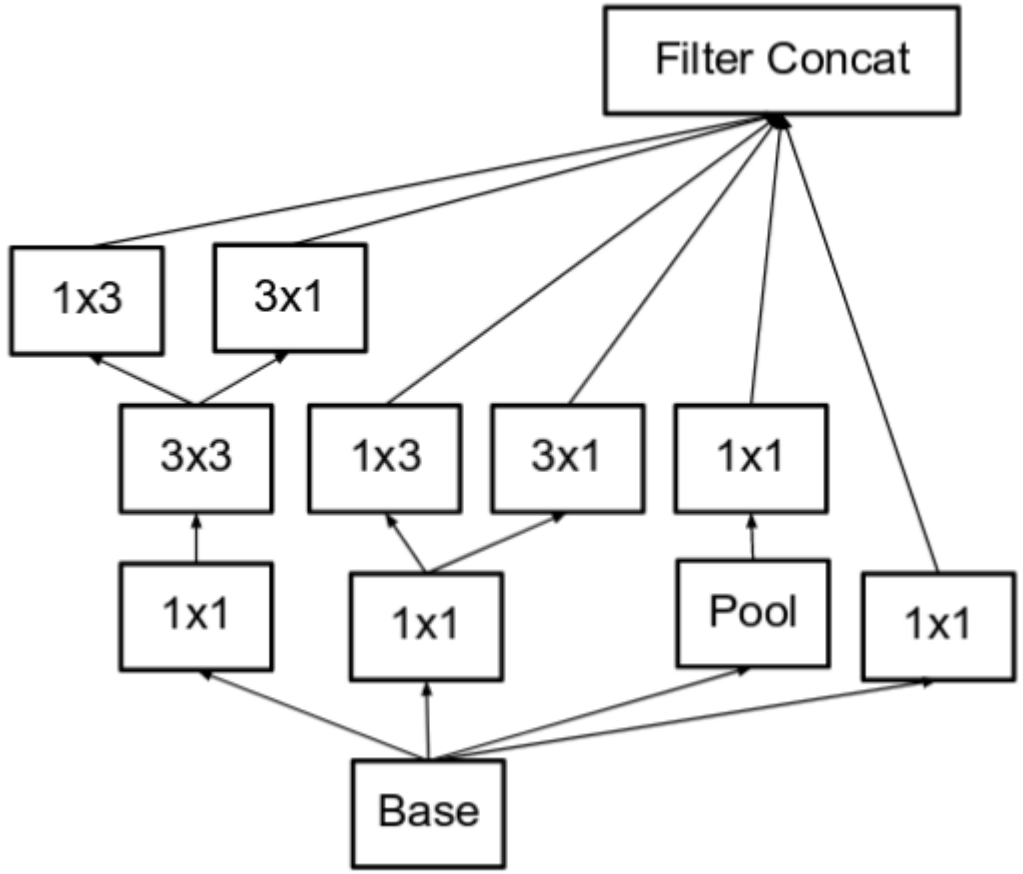




Representational bottleneck

- Wider network than deeper network

- Sudden reduction in depth causes information
- Loss – Rep Bottleneck



type	**patch size/stride** or remarks	**input size**
conv	$3\times3/2$	$299\times299\times3$
conv	$3\times3/1$	$149\times149\times32$
conv padded	$3\times3/1$	$147\times147\times32$
pool	$3\times3/2$	$147\times147\times64$
conv	$3\times3/1$	$73\times73\times64$
conv	$3\times3/2$	$71\times71\times80$
conv	$3\times3/1$	$35\times35\times192$
$3\times$Inception	As in figure 5	$35\times35\times288$
$5\times$Inception	As in figure 6	$17\times17\times768$
$2\times$Inception	As in figure 7	$8\times8\times1280$
pool	8×8	$8\times8\times2048$
linear	logits	$1\times1\times2048$
softmax	classifier	$1\times1\times1000$

Inception v3

- The auxiliary classifiers do not contribute much except during the end of the training phase!

1. RMSProp Optimizer.
2. Factorized 7x7 convolutions.
3. BatchNorm in the Auxillary Classifiers.
4. Label Smoothing (A type of regularizing component added to the loss formula that prevents the network from becoming too confident about a class. Prevents over fitting).

Label smoothing

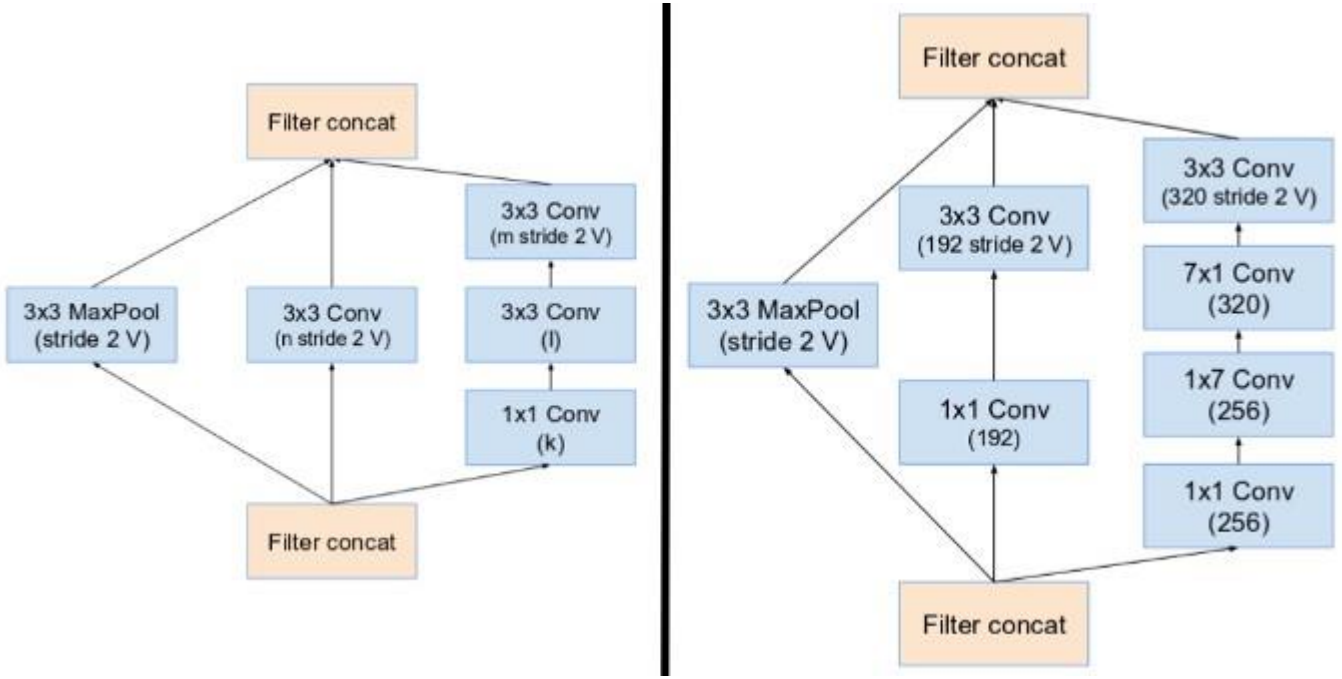
Usually, we have output vectors provided to us as
$y_{label} = [1, 0, 0, 0...0]$.

Softmax output is usually of the form
$y_{out} = [0.87, 0.001, 0.04, 0.1, ...0.03]$.

Maximum likelihood learning with a softmax classifier and hard targets may actually never converge, the softmax can never predict a probability of exactly 0 or exactly 1, so it will continue to learn larger and larger weights, making more extreme predictions. forever

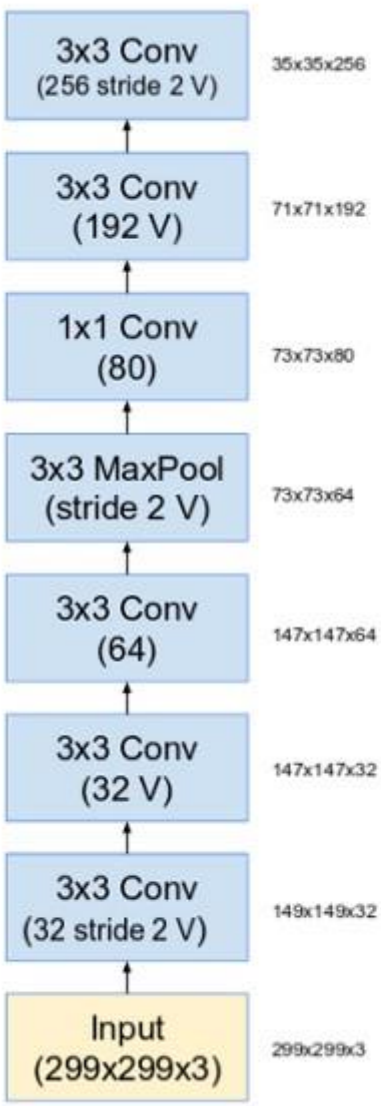
$$y_{label} = [1 - \epsilon, \frac{\epsilon}{K-1}, \frac{\epsilon}{K-1}, \frac{\epsilon}{K-1} \cdots \frac{\epsilon}{K-1}].$$

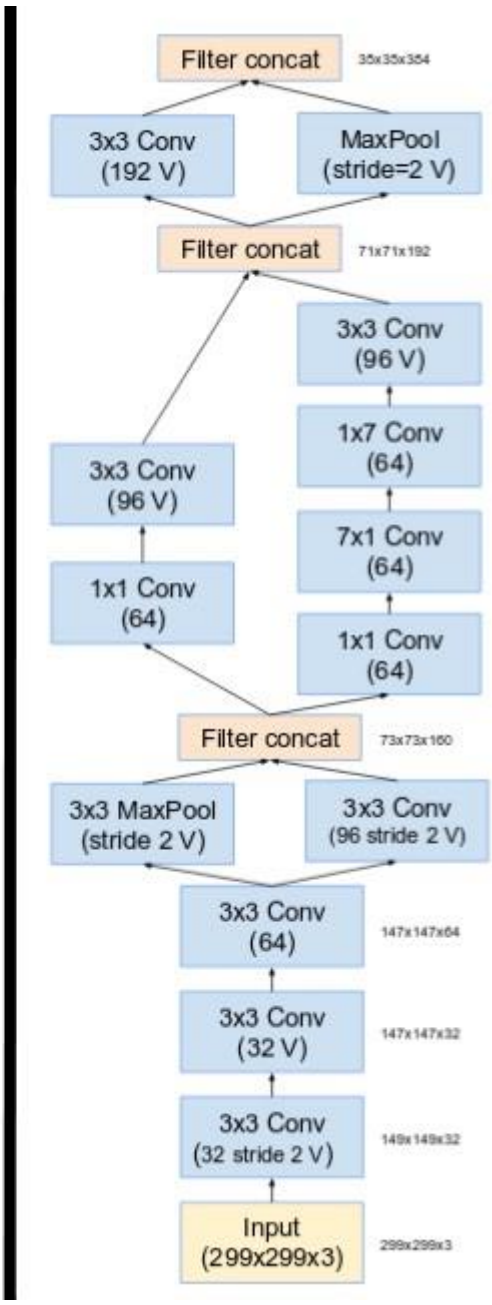
Inception v4

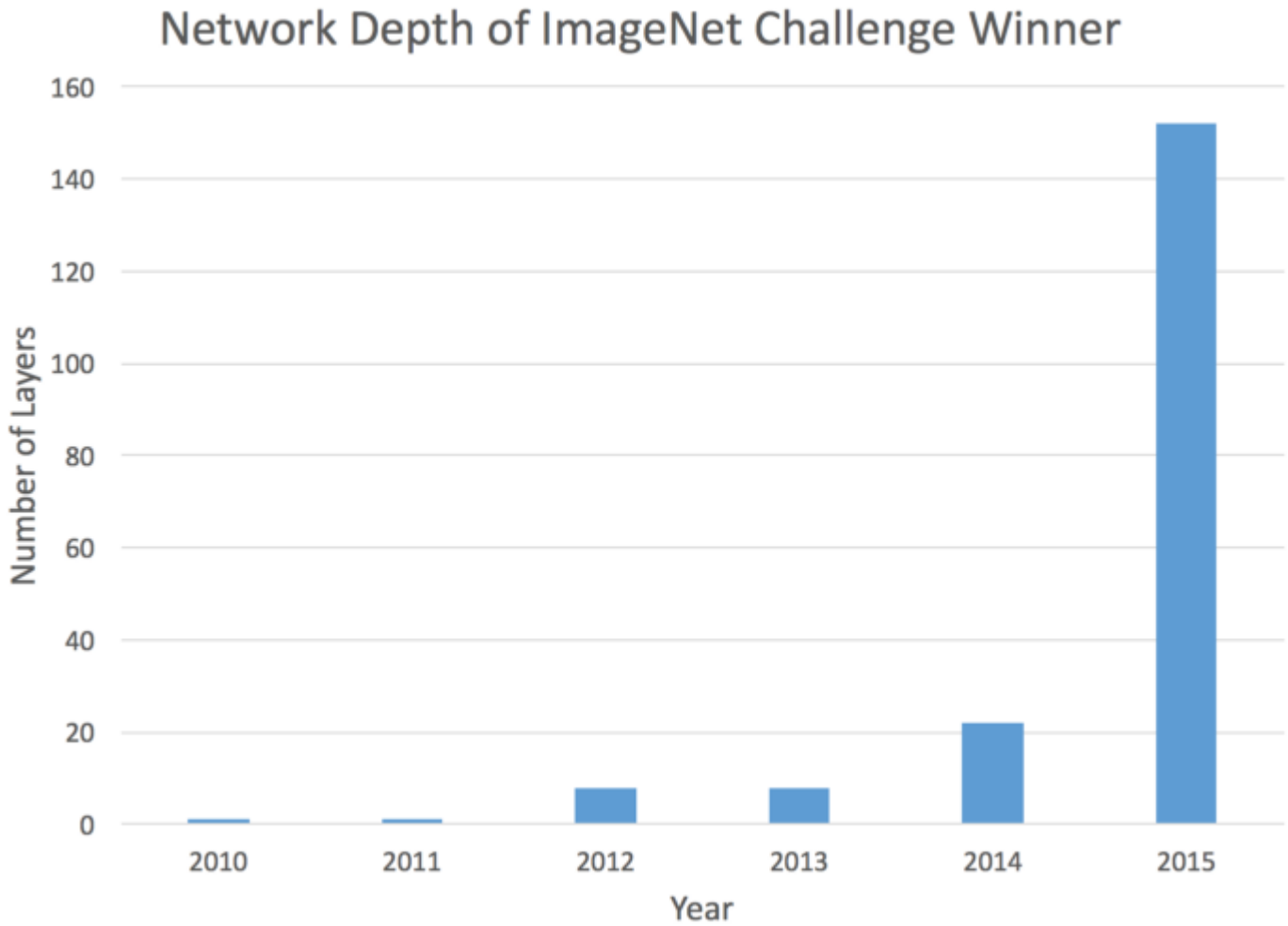


Reduce spatial dimensions!

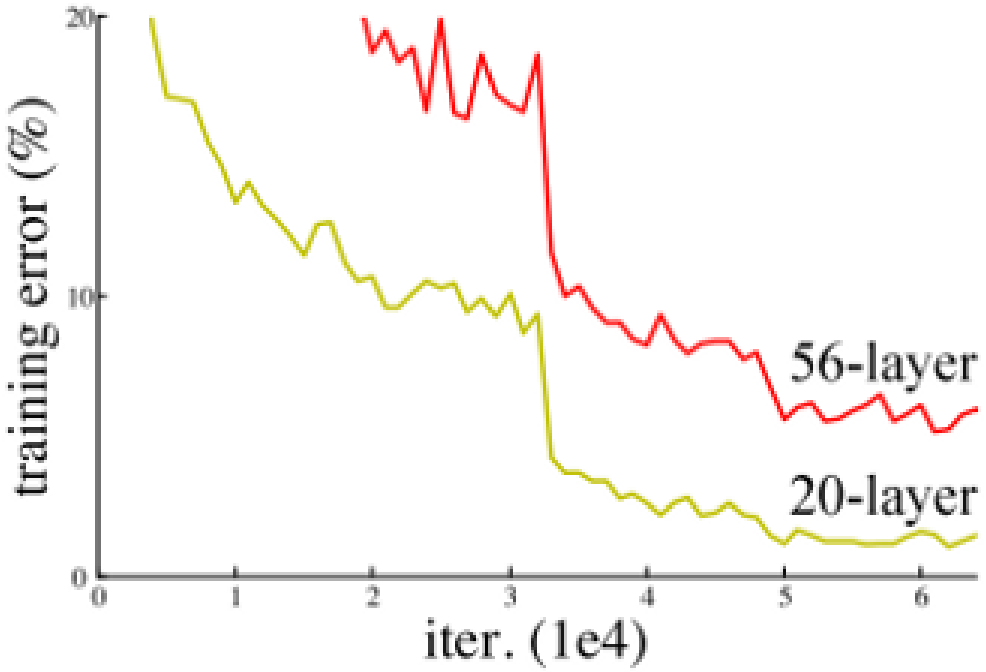
Simplify the "stem"

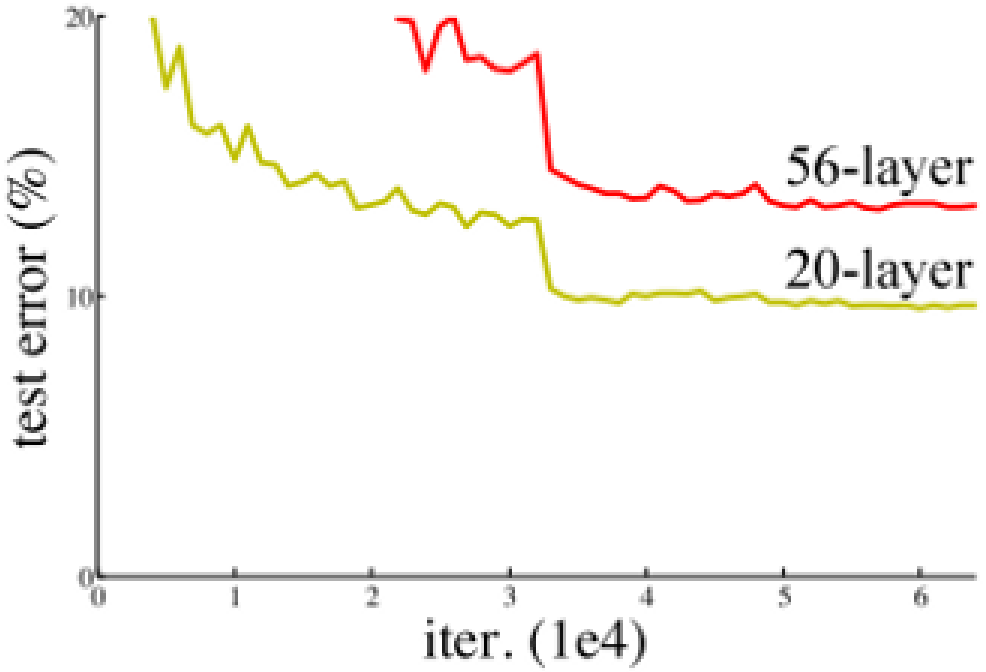




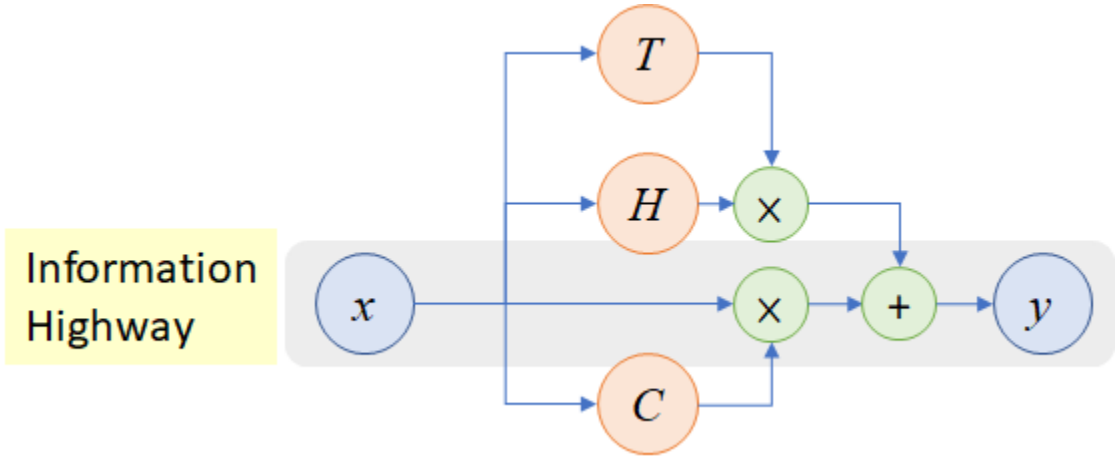

Network Depth of ImageNet Challenge Winner
160
140
120
100
80
60
40
20
0
Number of Layers
2010
2011
2012
2013
2014
2015
Year

Going deeper in CNN





Highway network



In plain network

$$\mathbf{y} = H(\mathbf{x}, \mathbf{W_H}).$$

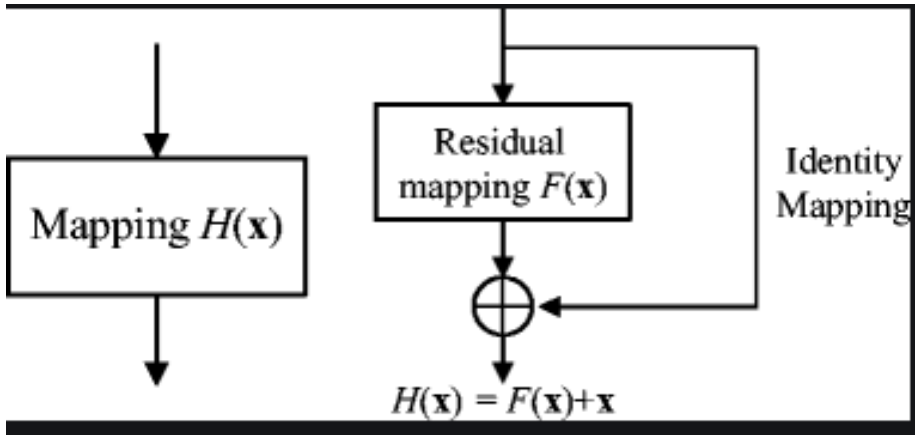
$$\mathbf{y} = H(\mathbf{x}, \mathbf{W_H}) \cdot T(\mathbf{x}, \mathbf{W_T}) + \mathbf{x} \cdot C(\mathbf{x}, \mathbf{W_C}).$$

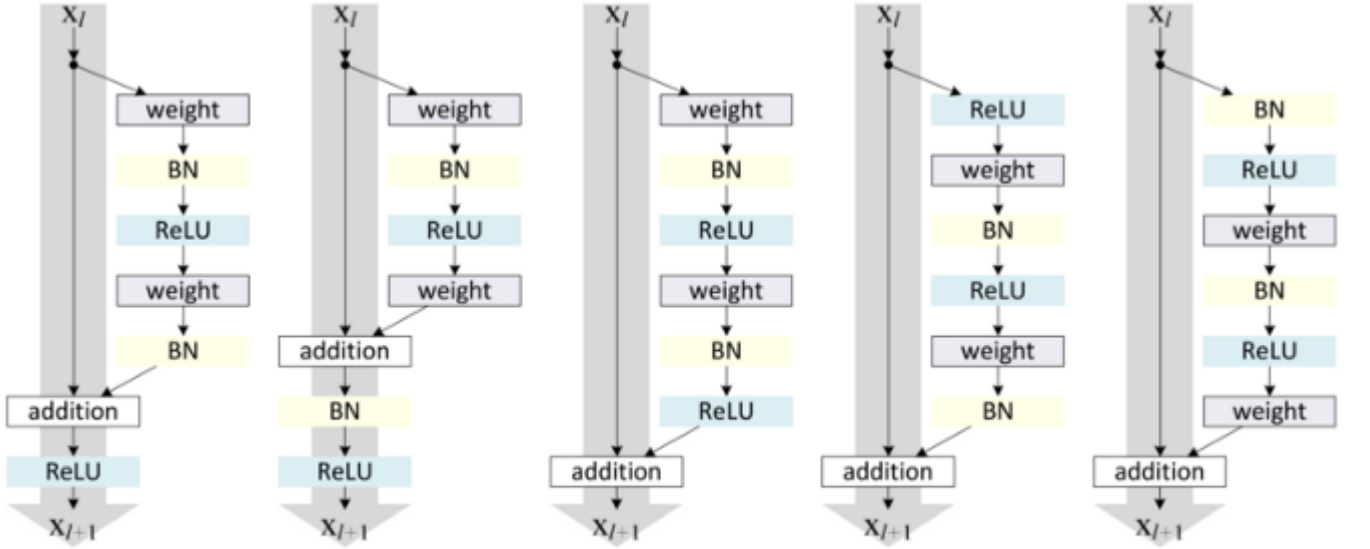
T: Transform gate $T(\mathbf{x}) = \sigma(\mathbf{W_T}^T \mathbf{x} + \mathbf{b_T})$

C: Carry gate, usually C = 1 - T

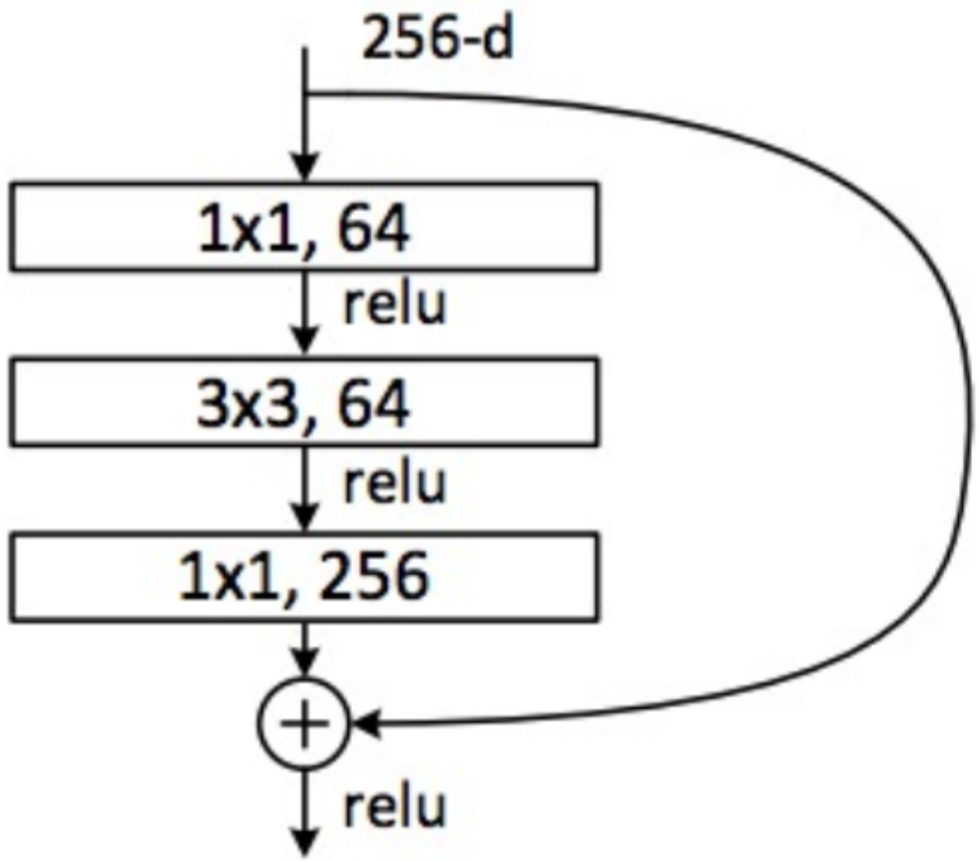
$$\mathbf{y} = \begin{cases} \mathbf{x}, & \text{if } T(\mathbf{x}, \mathbf{W_T}) = \mathbf{0}, \\ H(\mathbf{x}, \mathbf{W_H}), & \text{if } T(\mathbf{x}, \mathbf{W_T}) = \mathbf{1}. \end{cases}$$

Residual networks

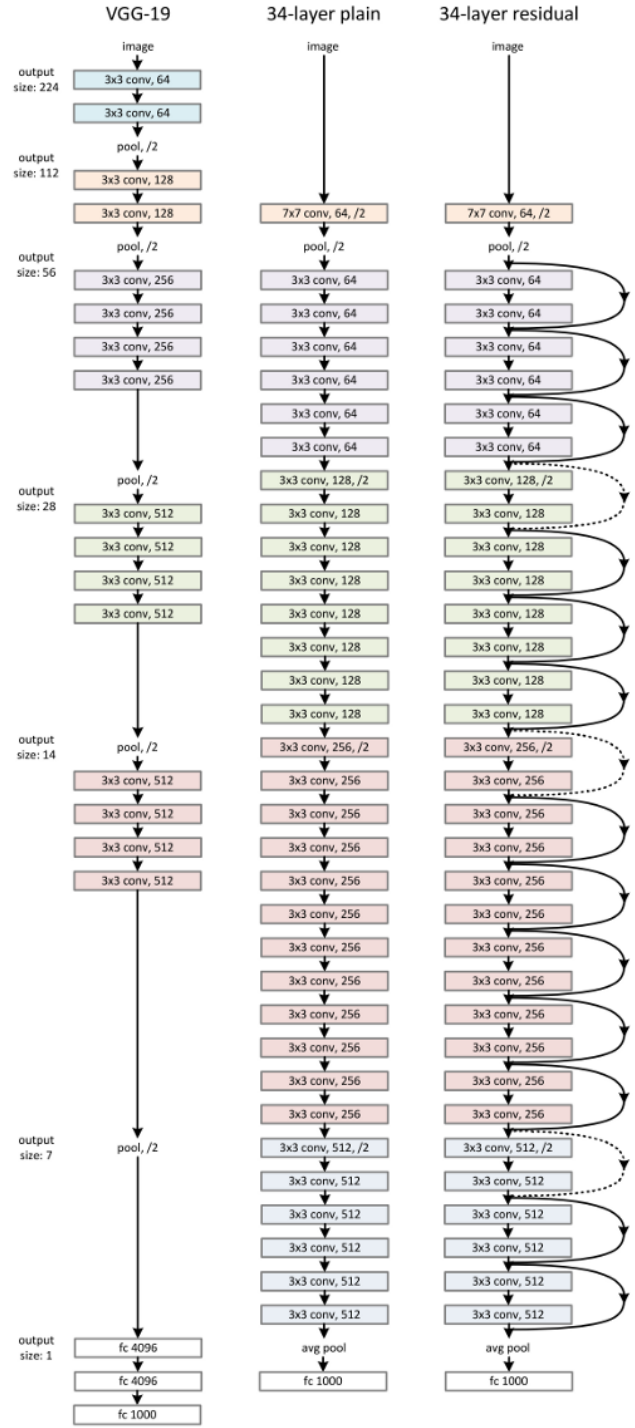




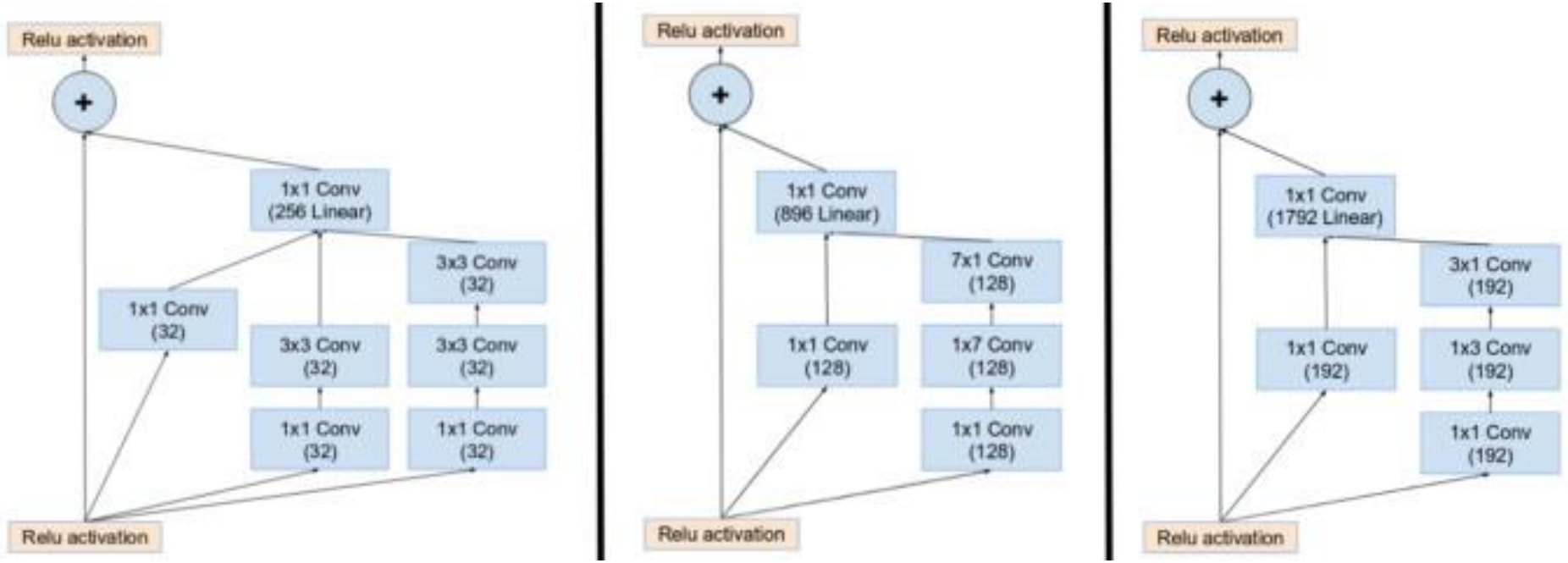
Deeper residual module (bottleneck)



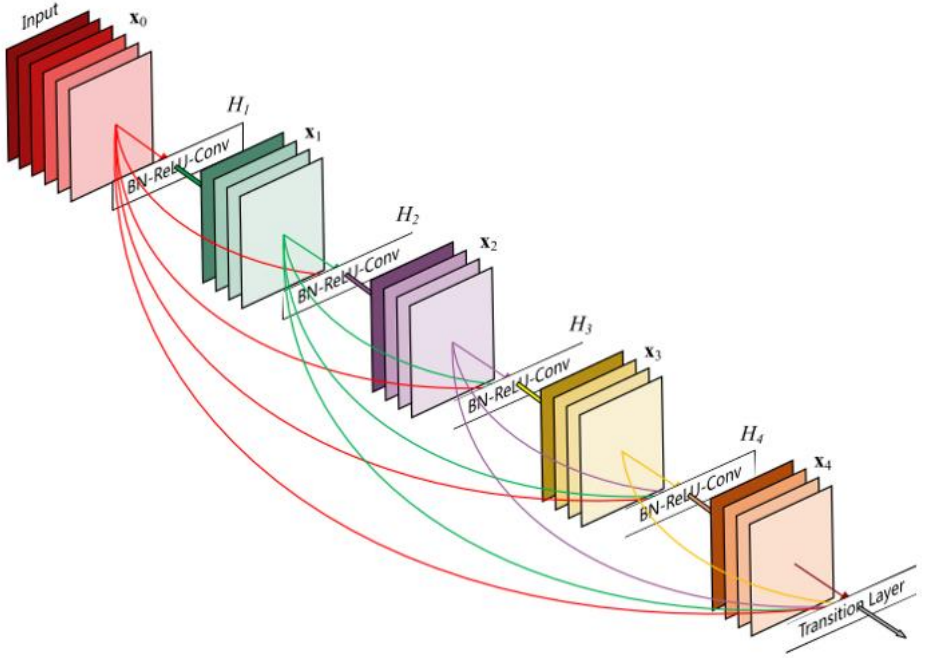
- Directly performing 3x3 convolutions with 256 feature maps at input and output:
  256 x 256 x 3 x 3 ~ 600K operations
- Using 1x1 convolutions to reduce 256 to 64 feature maps, followed by 3x3 convolutions, followed by 1x1 convolutions to expand back to 256 maps:
  256 x 64 x 1 x 1 ~ 16K
  64 x 64 x 3 x 3 ~ 36K
  64 x 256 x 1 x 1 ~ 16K
  Total: ~70K

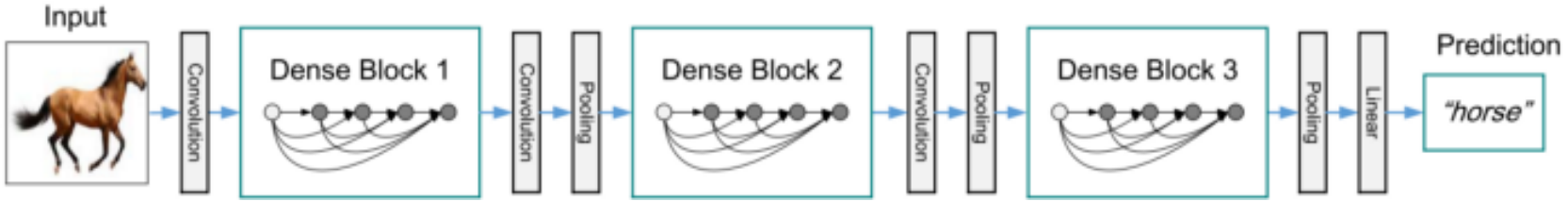

VGG-19
34-layer plain
34-layer residual
image
output size: 224
3x3 conv, 64
3x3 conv, 64
pool, /2
output size: 112
3x3 conv, 128
3x3 conv, 128
pool, /2
output size: 56
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
pool, /2
output size: 28
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
pool, /2
output size: 14
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
pool, /2
output size: 7
fc 4096
fc 4096
fc 1000
output size: 1
7x7 conv, 64, /2
3x3 conv, 128, /2
3x3 conv, 256, /2
3x3 conv, 512, /2
avg pool

Inception resnet

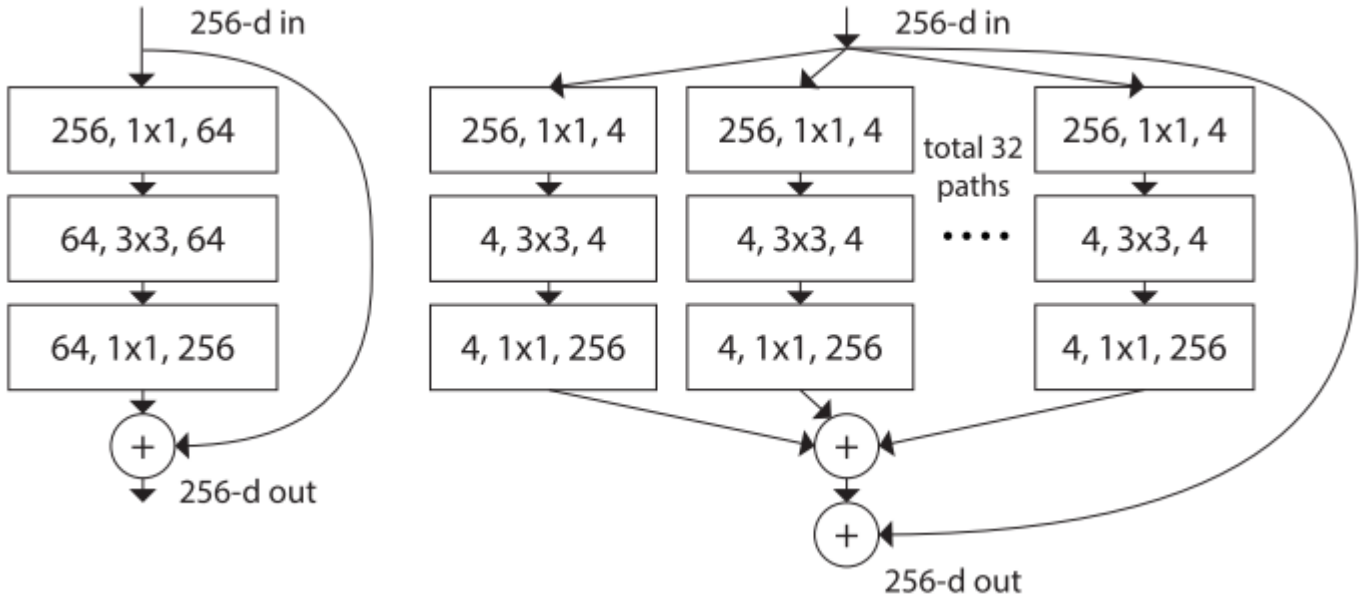


Densely connected CNN





ResNext

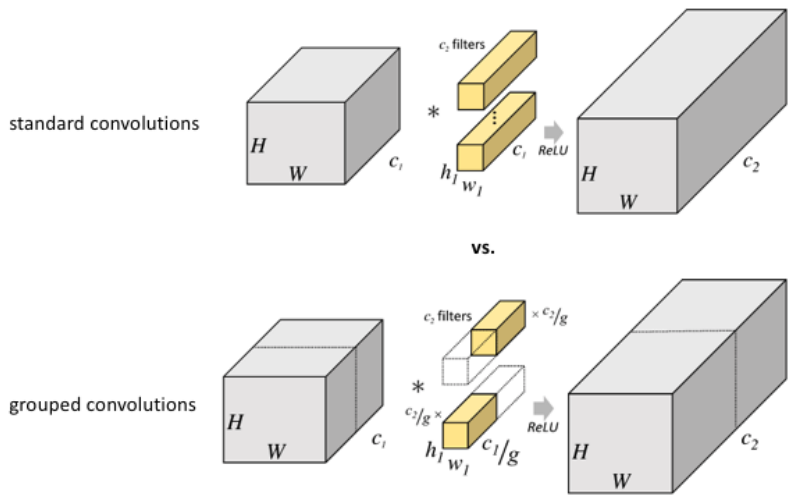


Resnext block with cardinality 32

stage	output	ResNet-50	**ResNeXt-50 (32×4d)**
conv1	112×112	7×7, 64, stride 2	7×7, 64, stride 2
conv2	56×56	3×3 max pool, stride 2 $\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	3×3 max pool, stride 2 $\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128, C{=}32 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3	28×28	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256, C{=}32 \\ 1\times1, 512 \end{bmatrix}\times4$
conv4	14×14	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512, C{=}32 \\ 1\times1, 1024 \end{bmatrix}\times6$
conv5	7×7	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 1024 \\ 3\times3, 1024, C{=}32 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	global average pool 1000-d fc, softmax	global average pool 1000-d fc, softmax
# params.		**25.5**×10^6	**25.0**×10^6
FLOPs		**4.1**×10^9	**4.2**×10^9

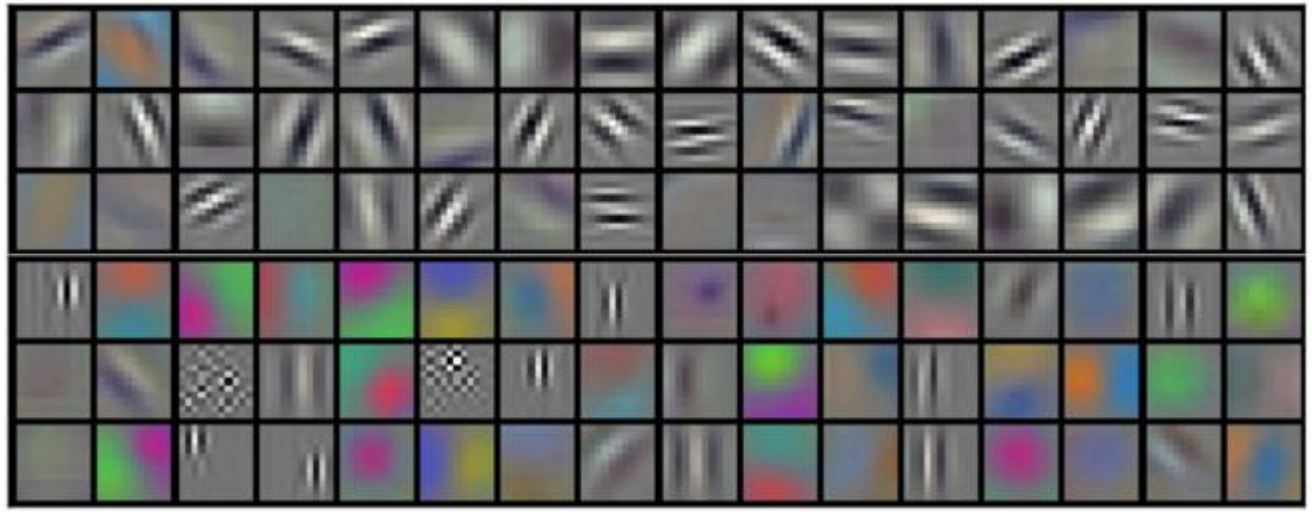
Table 1. (**Left**) ResNet-50. (**Right**) ResNeXt-50 with a 32×4d template (using the reformulation in Fig. 3(c)). Inside the brackets are the shape of a residual block, and outside the brackets is the number of stacked blocks on a stage. "C=32" suggests grouped convolutions [24] with 32 groups. *The numbers of parameters and FLOPs are similar between these two models.*

Group convolution



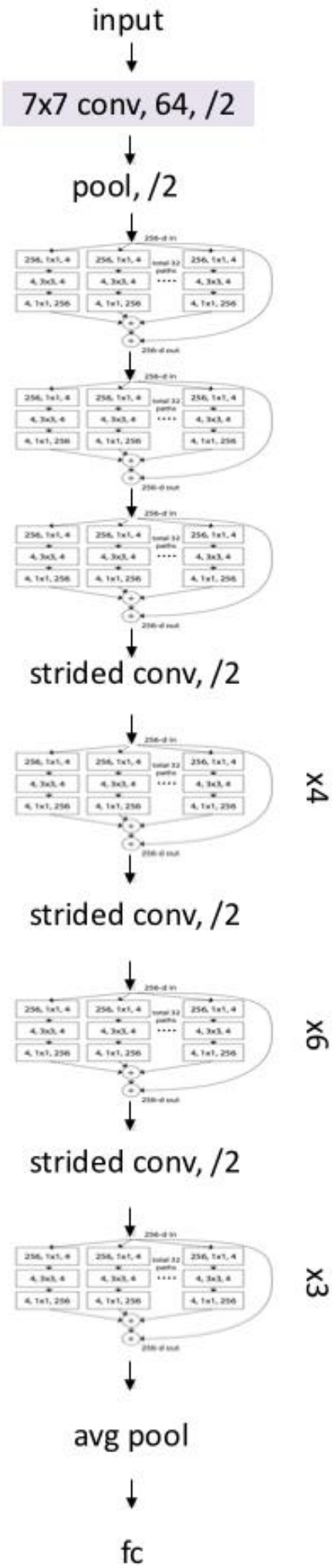
Group 1

Group 2



black and white shapes

colors


input
7x7 conv, 64, /2
pool, /2
strided conv, /2
x4
strided conv, /2
x6
strided conv, /2
x3
avg pool
fc

Stochastic depth

- Can the depth of the network change during training?

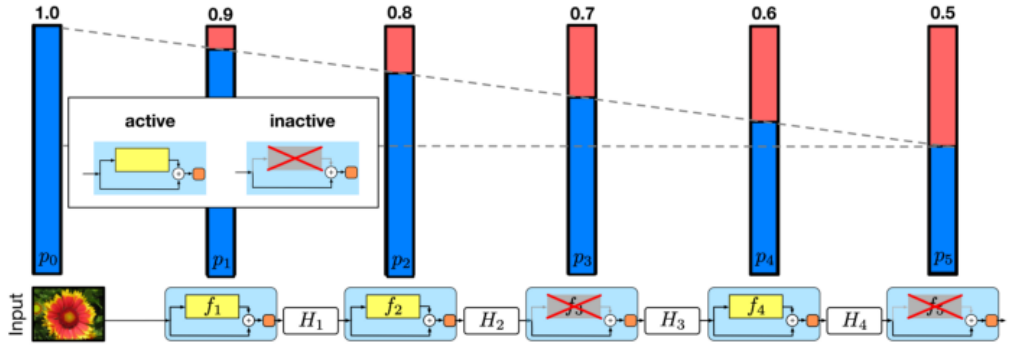
$$H_l = ReLU(b_l * f_l(H_{l-1}) + id(H_{l-1})).$$

If b=0

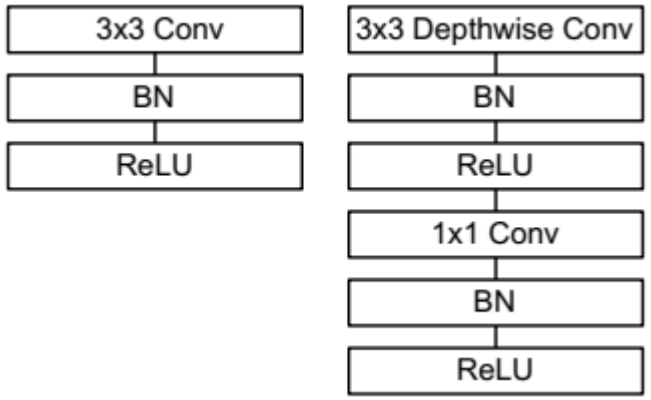
$$H_l = ReLU(id(H_{l-1})).$$

$$H_l = ReLU(p_l * f_l(H_{l-1}) + id(H_{l-1})).$$

$$p_l = 1 - \frac{l}{L}(1 - p_L).$$



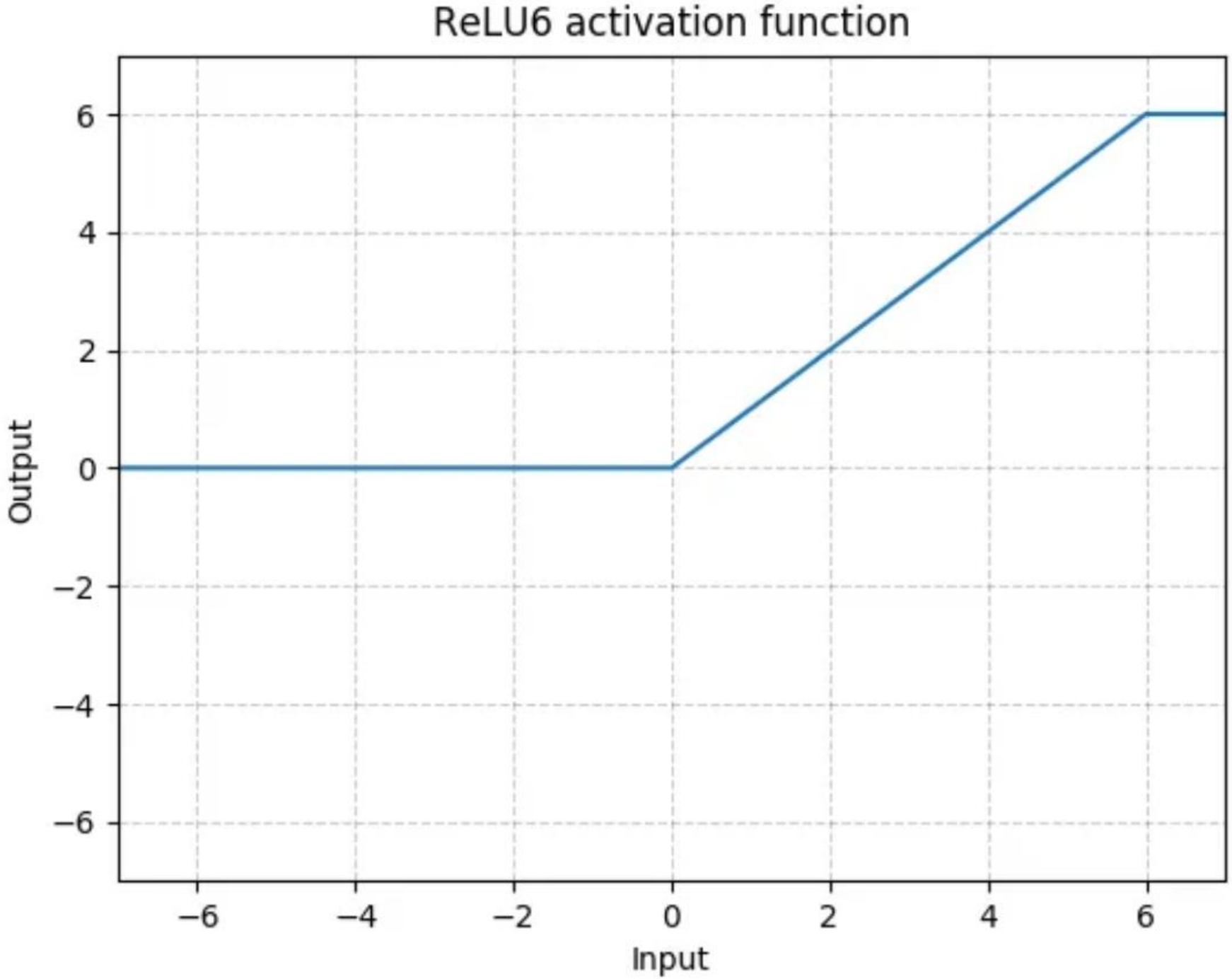
Mobilenet



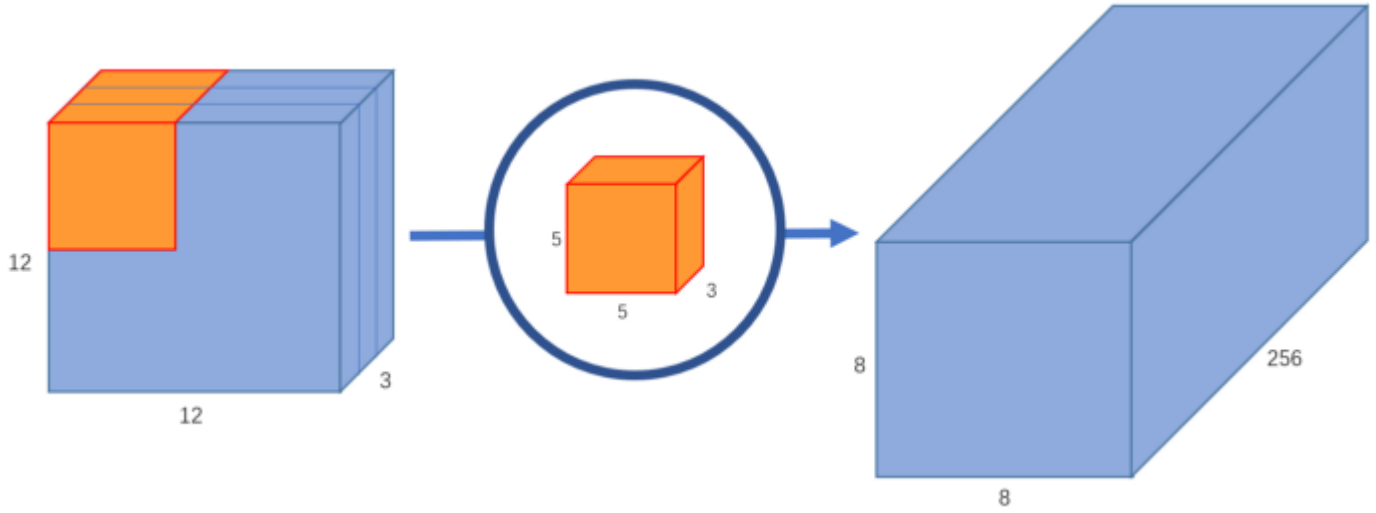
The idea of depthwise
Separable convolution

Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$


ReLU6 activation function
Output
Input
6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

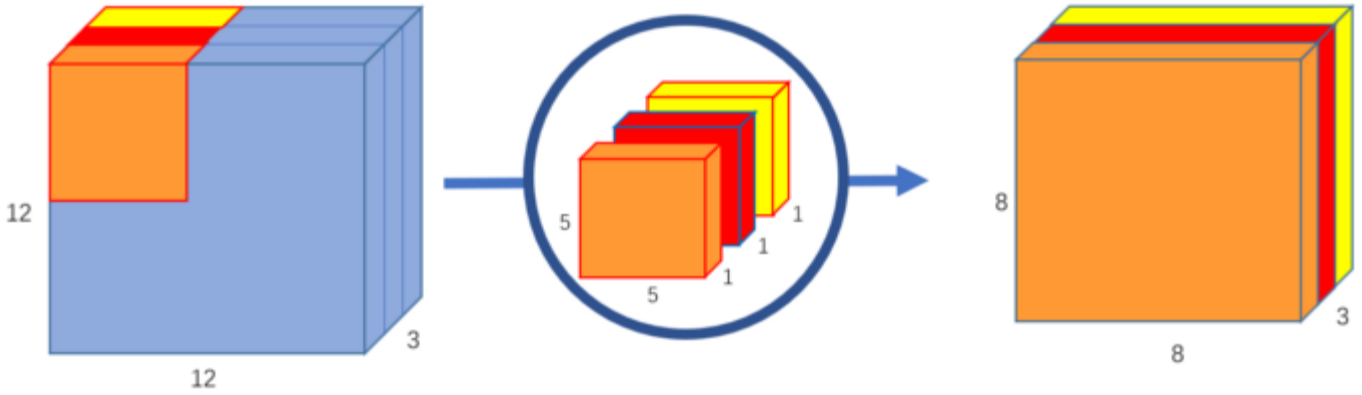
Depthwise separable convolution



Convolving by 256 5x5 kernels over the input volume

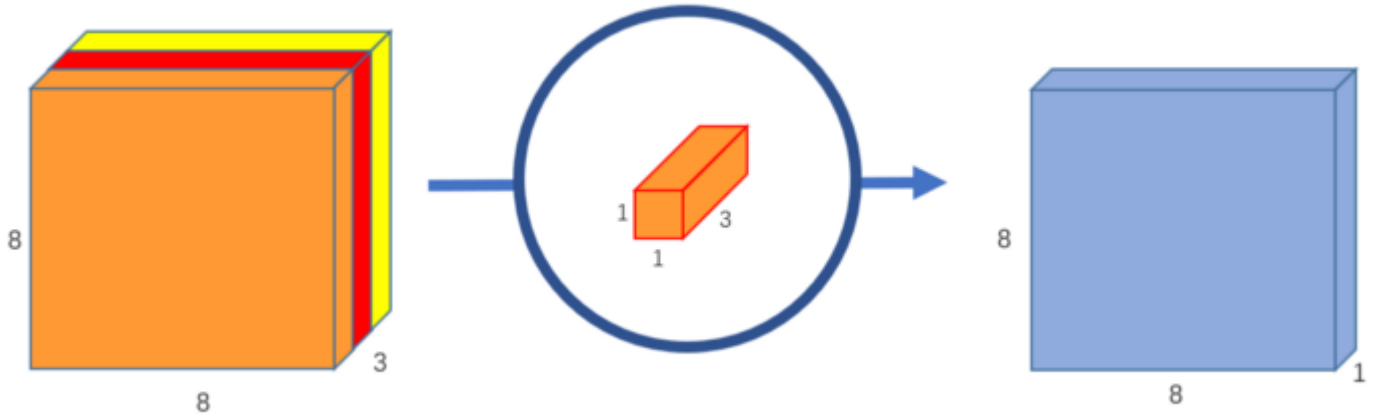
Depthwise separable convolution – step1

Along depth

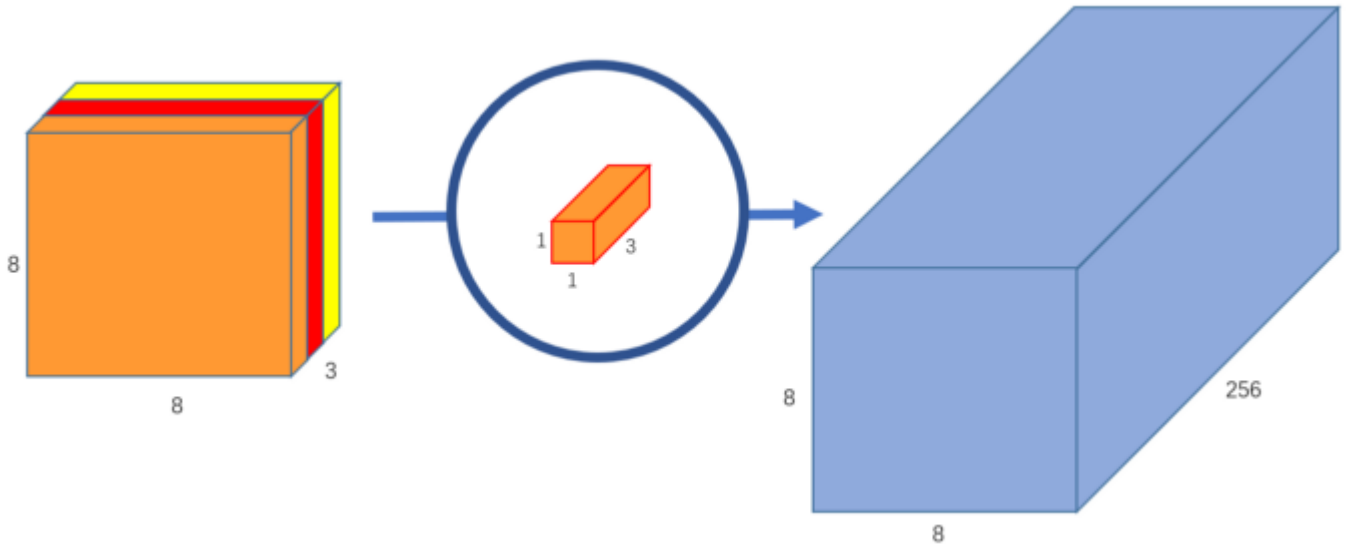


Each 5x5x1 kernel iterates 1 channel of the image (note: **1 channel**, not all channels), getting the scalar products of every 25 pixel group, giving out a 8x8x1 image. Stacking these images together creates a 8x8x3 image.

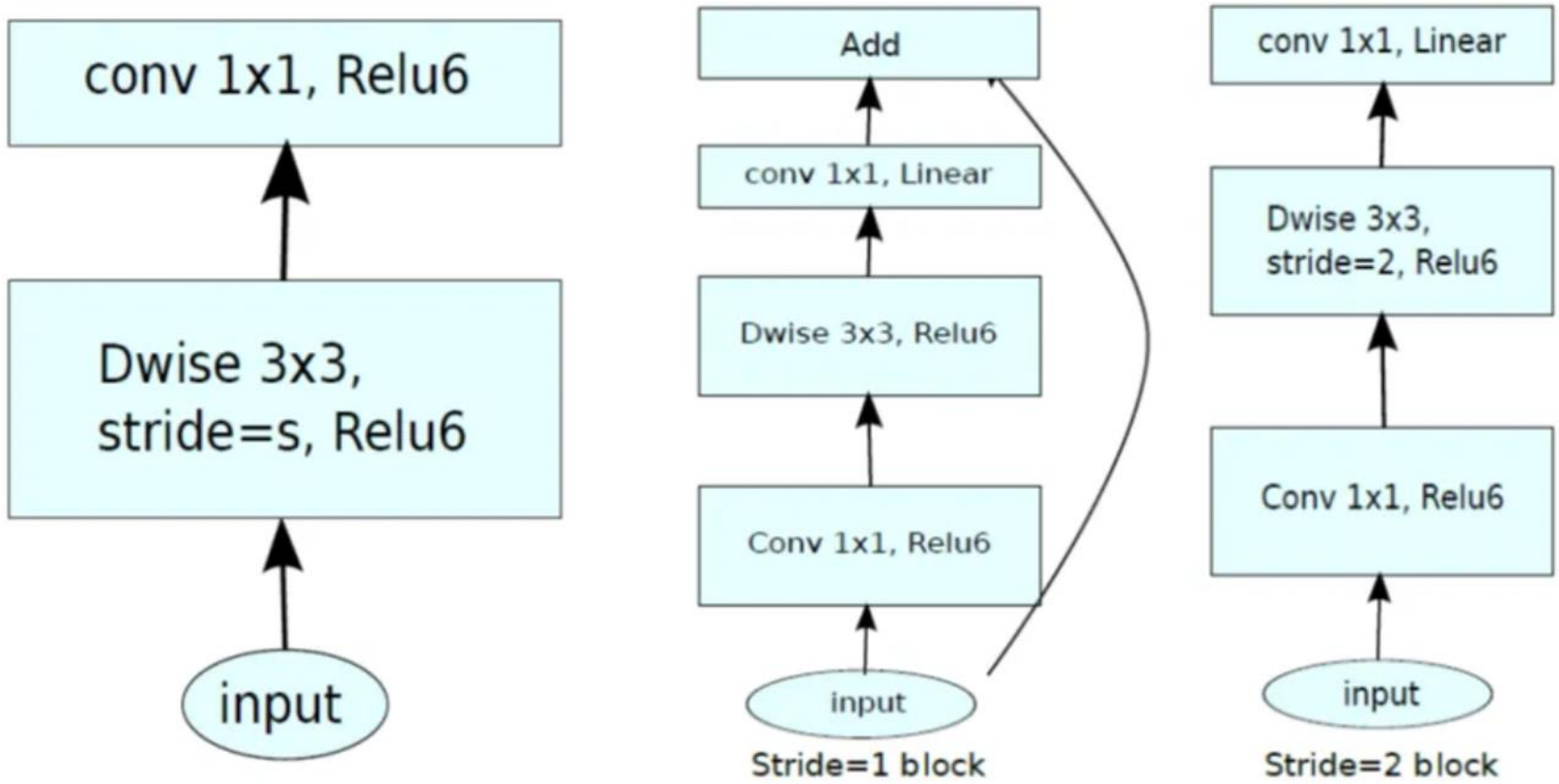
Depthwise separable convolution – step2



Pointwise



1x1 conv



MobileNetV1

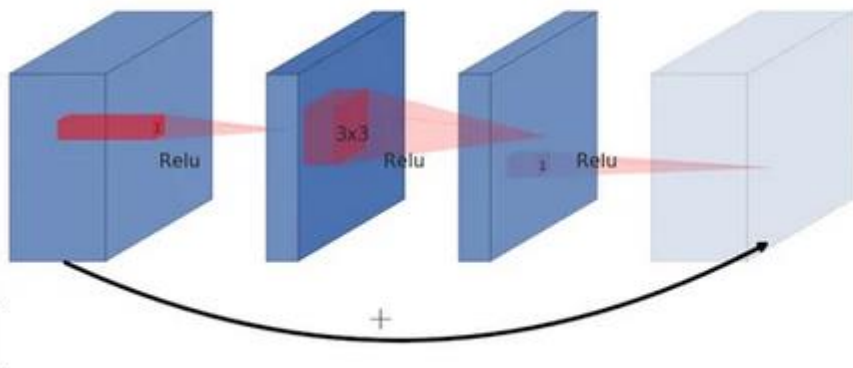
MobileNetV2

MobileNet v2

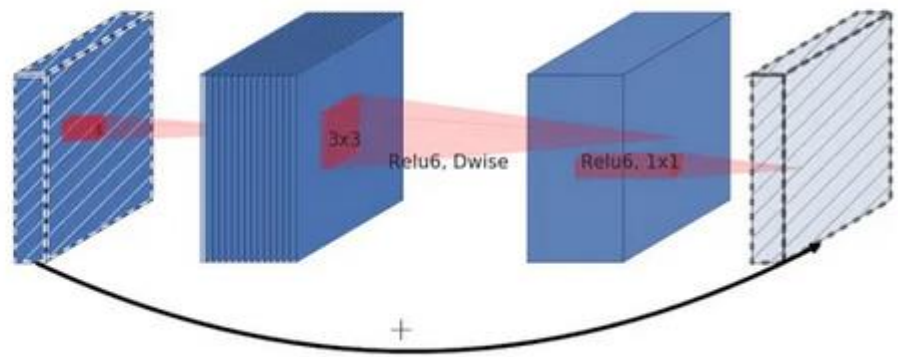
Input	Operator	Output
$h \times w \times k$	1x1 conv2d , ReLU6	$h \times w \times (tk)$
$h \times w \times tk$	3x3 dwise s=s, ReLU6	$\frac{h}{s} \times \frac{w}{s} \times (tk)$
$\frac{h}{s} \times \frac{w}{s} \times tk$	linear 1x1 conv2d	$\frac{h}{s} \times \frac{w}{s} \times k'$

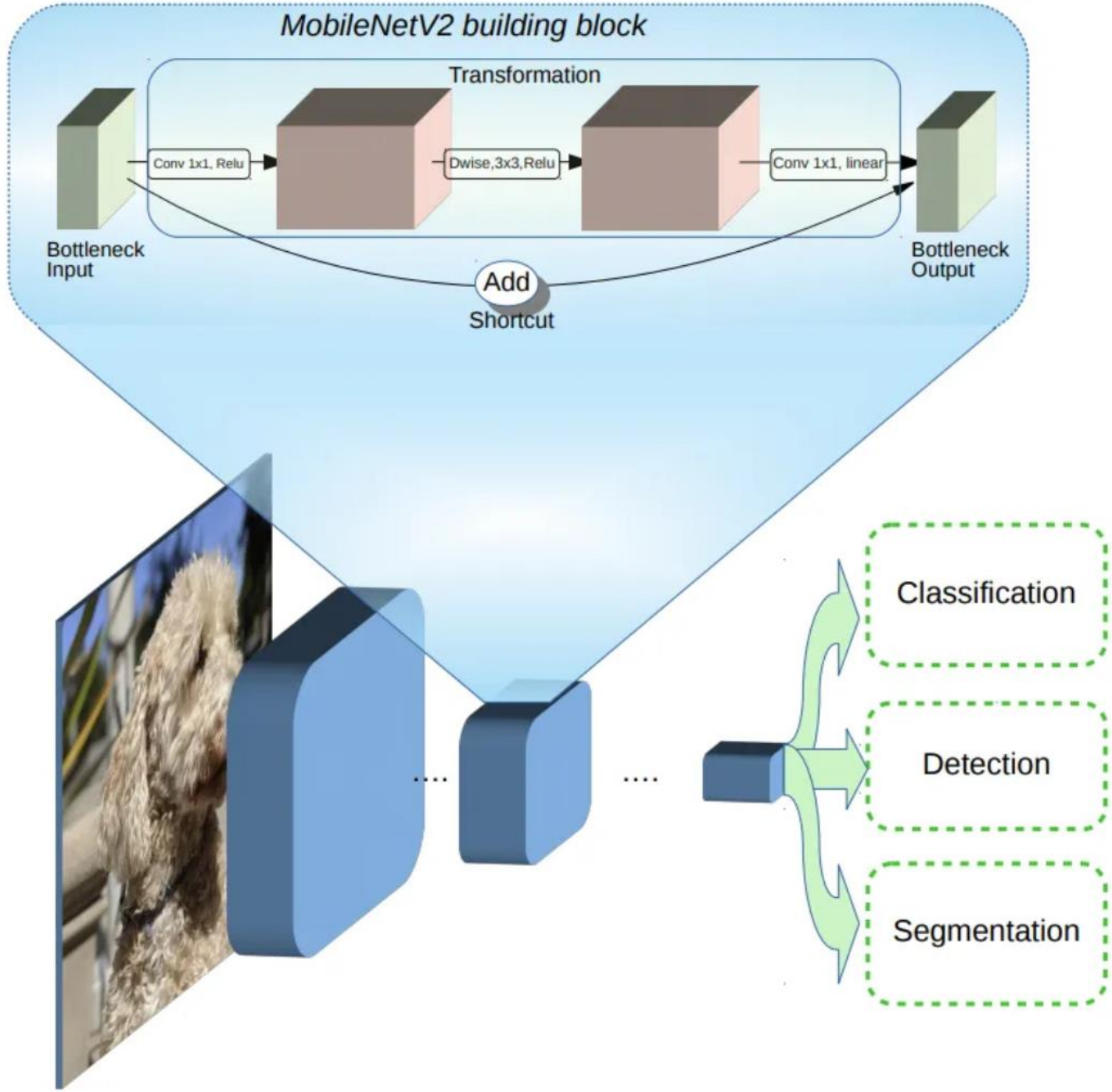
Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

(a) Residual block

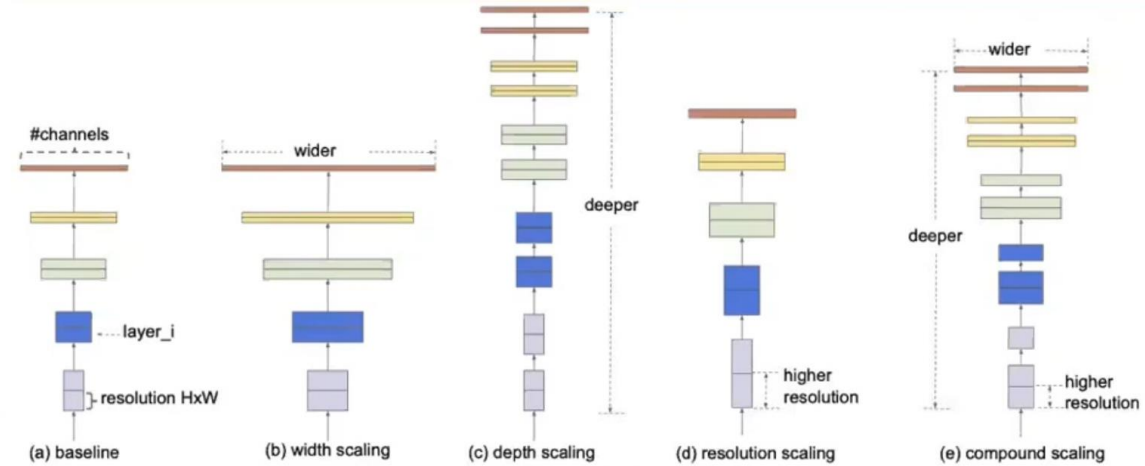


(b) Inverted residual block




MobileNetV2 building block
Transformation
Conv 1x1, Relu
Dwise,3x3,Relu
Conv 1x1, linear
Bottleneck Input
Bottleneck Output
Add
Shortcut
....
....
Classification
Detection
Segmentation

Efficientnet



$d \rightarrow$ depth scaling coefficient
$w \rightarrow$ width scaling coefficient
$r \rightarrow$ resolution scaling coefficient

$$\max_{d,w,r} \quad \text{accuracy}$$

$$\text{s.t.} \quad \text{memory} \leq \text{target\_memory}$$
$$\text{flops} \leq \text{target\_flops}$$

Compound Scaling

$\phi \rightarrow$ compound coefficient

$$d = \alpha^\phi, w = \beta^\phi, r = \gamma^\phi$$
$$\alpha\beta^2\gamma^2 \approx 2, \underbrace{\alpha \geq 1, \beta \geq 1, \gamma \geq 1}_{\text{small grid search}}$$

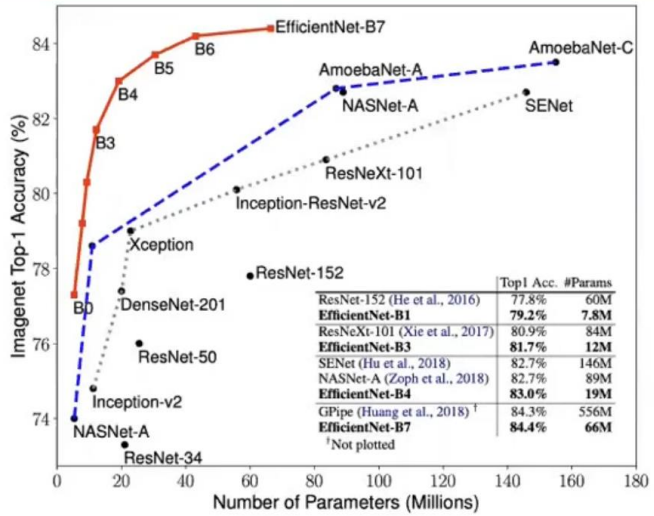
EfficientNet

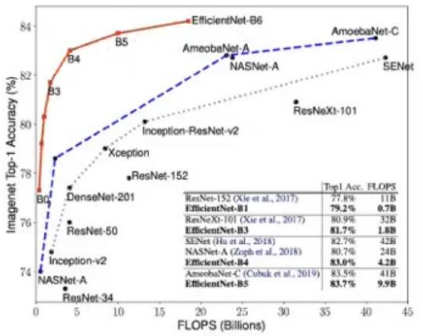
$$\max \text{accuracy} \cdot (\text{flops}/\text{target\_flops})^\omega$$

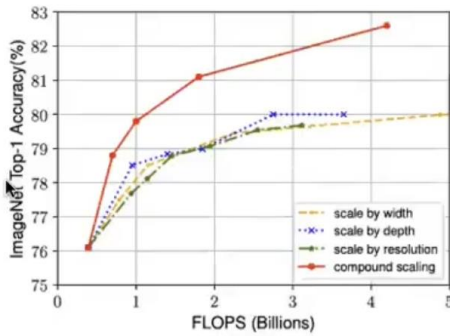
$\omega = -0.07 \rightarrow$ controls the tradeoff btw accuracy & flops

Stage i	Operator $\hat{\mathcal{F}}_i$	Resolution $\hat{H}_i \times \hat{W}_i$	#Channels $\hat{C}_i$	#Layers $\hat{L}_i$
1	Conv3x3	224×224	32	1
2	MBConv1, k3x3	112×112	16	1
3	MBConv6, k3x3	112×112	24	2
4	MBConv6, k5x5	56×56	40	2
5	MBConv6, k3x3	28×28	80	3
6	MBConv6, k5x5	14×14	112	3
7	MBConv6, k5x5	14×14	192	4
8	MBConv6, k3x3	7×7	320	1
9	Conv1x1 & Pooling & FC	7×7	1280	1

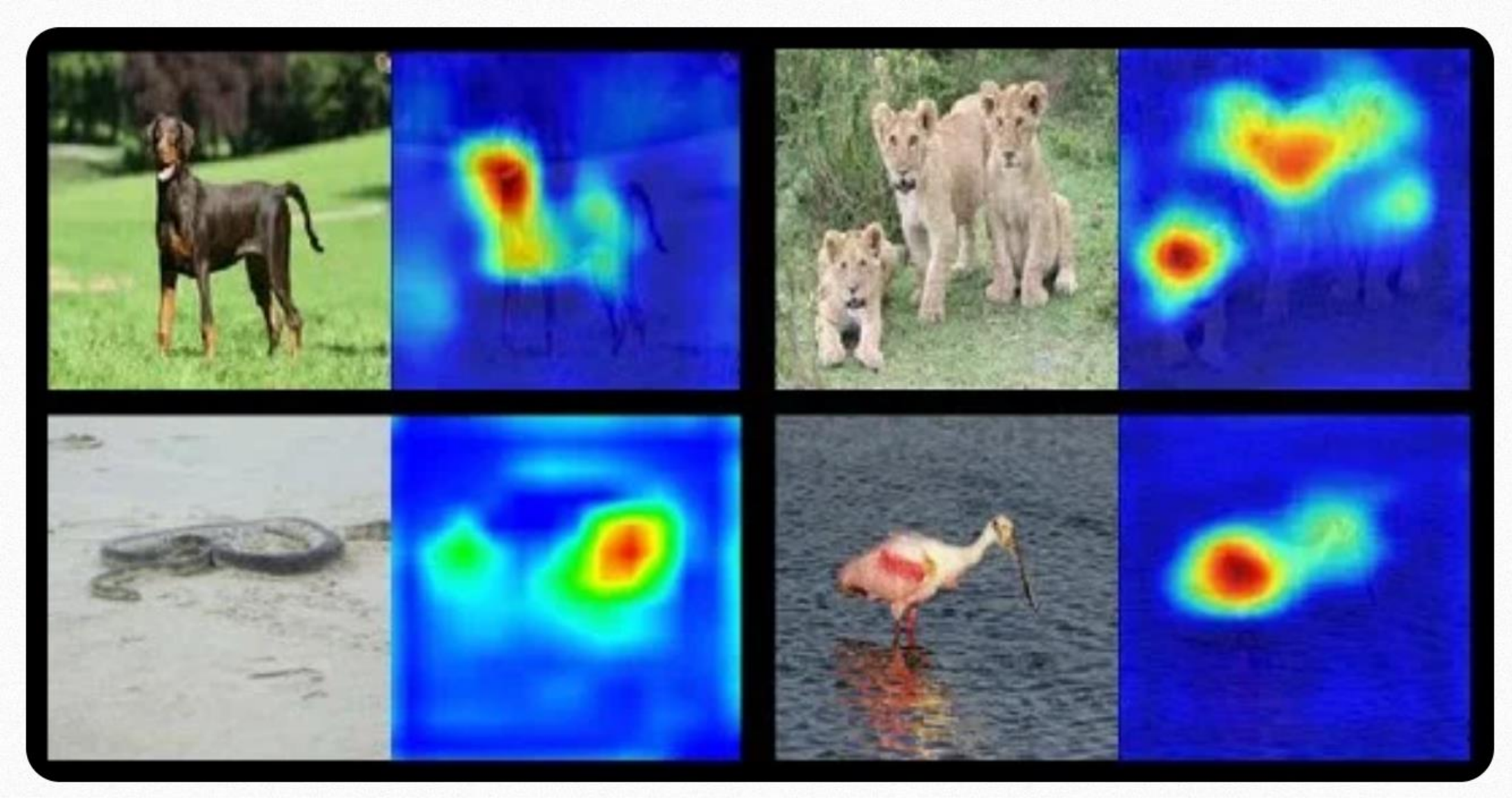
MBConv: mobile inverted bottleneck



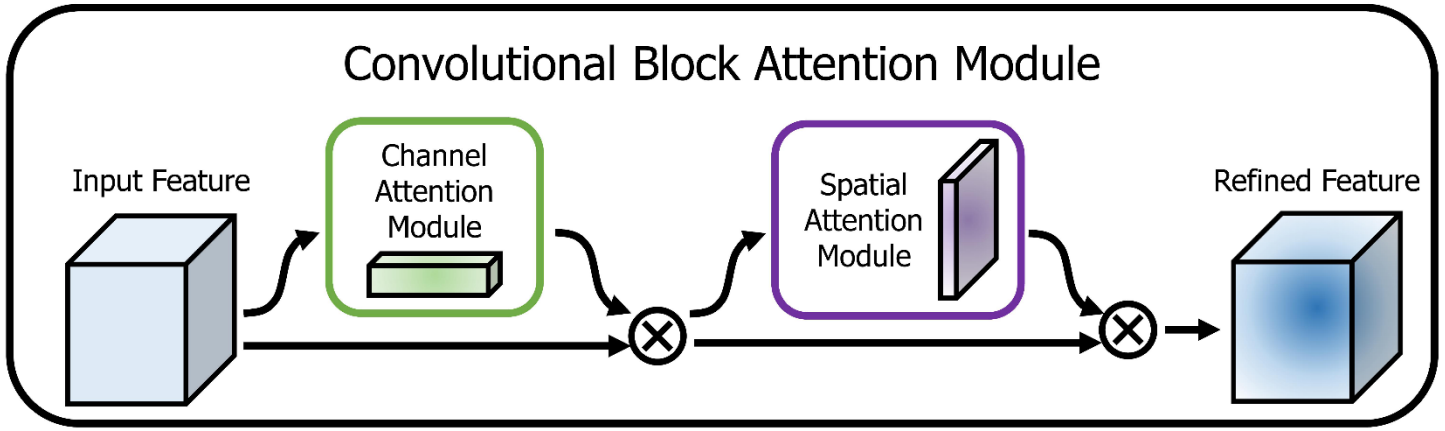


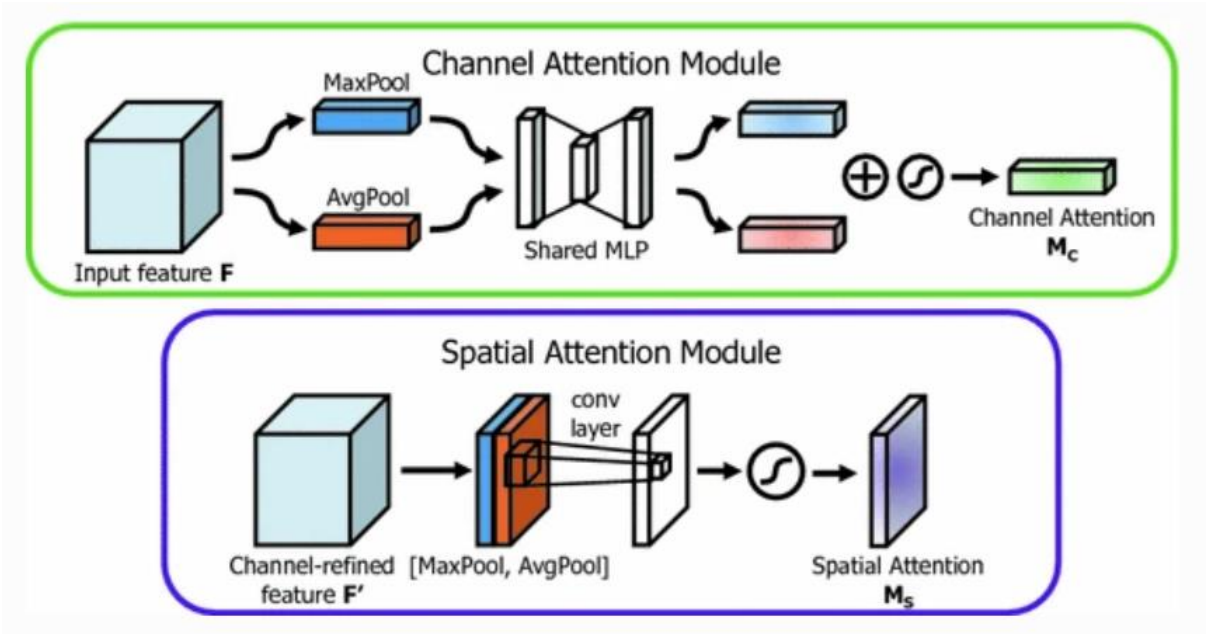


Visual attention

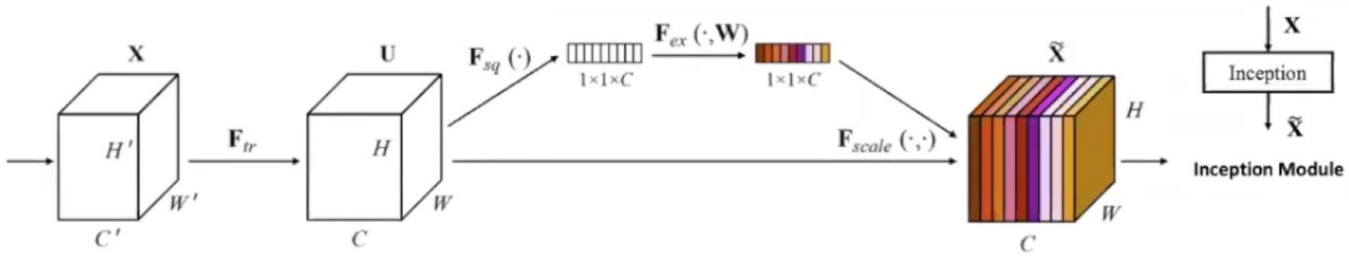


CBAM – spectral spatial attention





SE ResNet



Inception Module

$X \in \mathbb{R}^{H' \times W' \times C'}$ $U \in \mathbb{R}^{H \times W \times C}$

$F_{tr} : X \mapsto U$

$V = [v_1, \ldots, v_C] \rightarrow$ learned set of filter kernels

$v_c \rightarrow$ parameters of the c-th filter

$U = [u_1, \ldots, u_C]$

$$u_c = v_c * X = \sum_{s=1}^{C'} v_c^s * x^s$$

$v_c = [v_c^1, \ldots, v_c^{C'}]$

$X = [x^1, \ldots, x^{C'}]$

Squeeze

$z \in \mathbb{R}^C$ $$z_c = F_{sq}(u_c) = \frac{1}{HW} \sum_{i=1}^{H} \sum_{j=1}^{W} u_c(i,j)$$

Excitation

$$s = F_{ex}(z, W) = \sigma(g(z, W)) = \sigma(W_2 \delta(W_1 z))$$

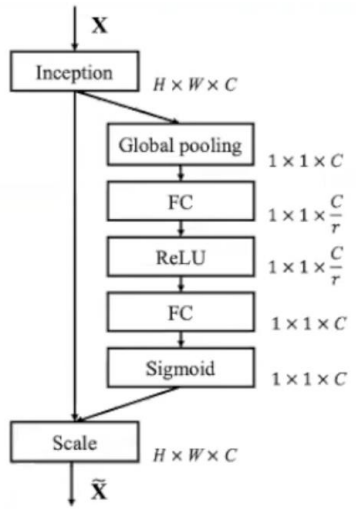
$W_1 \in \mathbb{R}^{\frac{C}{r} \times C}$ $W_2 \in \mathbb{R}^{C \times \frac{C}{r}}$

$$\tilde{x}_c = F_{scale}(u_c, s_c) = s_c u_c$$

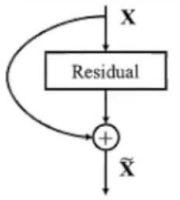
$u_c \in \mathbb{R}^{H \times W}$

$\tilde{X} = [\tilde{x}_1, \ldots, \tilde{x}_C]$

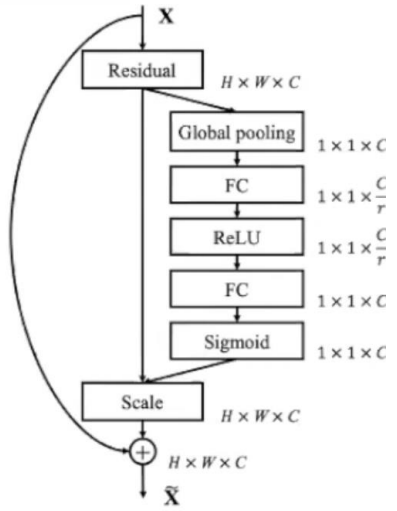
	224 × 224		320 × 320 / 299 × 299	
	top-1 err.	top-5 err.	top-1 err.	top-5 err.
ResNet-152 [10]	23.0	6.7	21.3	5.5
ResNet-200 [11]	21.7	5.8	20.1	4.8
Inception-v3 [44]	-	-	21.2	5.6
Inception-v4 [42]	-	-	20.0	5.0
Inception-ResNet-v2 [42]	-	-	19.9	4.9
ResNeXt-101 (64 × 4d) [47]	20.4	5.3	19.1	4.4
DenseNet-264 [14]	22.15	6.12	-	-
Attention-92 [46]	-	-	19.5	4.8
Very Deep PolyNet [51] †	-	-	18.71	4.25
PyramidNet-200 [8]	20.1	5.4	19.2	4.7
DPN-131 [5]	19.93	5.12	18.55	4.16
SENet-154	**18.68**	**4.47**	**17.28**	**3.79**
NASNet-A (6@4032) [55] †	-	-	17.3‡	3.8‡
SENet-154 (post-challenge)	-	-	**16.88‡**	**3.58‡**



SE-Inception Module

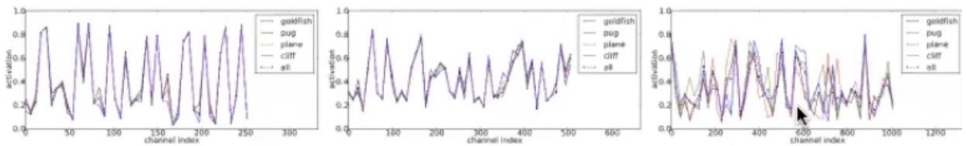


ResNet Module



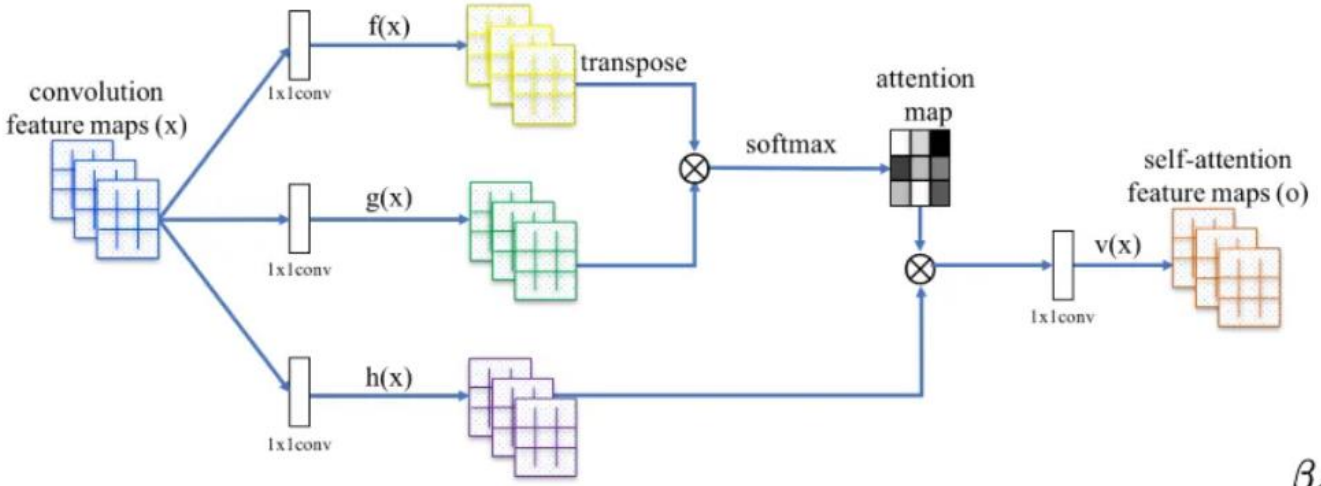
SE-ResNet Module

	original		re-implementation			SENet		
	top-1 err.	top-5 err.	top-1err.	top-5 err.	GFLOPs	top-1 err.	top-5 err.	GFLOPs
ResNet-50 [10]	24.7	7.8	24.80	7.48	3.86	$23.29_{(1.51)}$	$6.62_{(0.86)}$	3.87
ResNet-101 [10]	23.6	7.1	23.17	6.52	7.58	$22.38_{(0.79)}$	$6.07_{(0.45)}$	7.60
ResNet-152 [10]	23.0	6.7	22.42	6.34	11.30	$21.57_{(0.85)}$	$5.73_{(0.61)}$	11.32
ResNeXt-50 [47]	22.2	-	22.11	5.90	4.24	$21.10_{(1.01)}$	$5.49_{(0.41)}$	4.25
ResNeXt-101 [47]	21.2	5.6	21.18	5.57	7.99	$20.70_{(0.48)}$	$5.01_{(0.56)}$	8.00
VGG-16 [39]	-	-	27.02	8.81	15.47	$25.22_{(1.80)}$	$7.70_{(1.11)}$	15.48
BN-Inception [16]	25.2	7.82	25.38	7.89	2.03	$24.23_{(1.15)}$	$7.14_{(0.75)}$	2.04
Inception-ResNet-v2 [42]	19.9†	4.9†	20.37	5.21	11.75	$19.80_{(0.57)}$	$4.79_{(0.42)}$	11.76



(a) SE_2_3 (b) SE_3_4 (c) SE_4_6

Idea of self-attention in CNN



$$\beta_{j,i} = \frac{\exp(s_{ij})}{\sum_{i=1}^{N} \exp(s_{ij})}, \text{ where } s_{ij} = \boldsymbol{f}(\boldsymbol{x_i})^T \boldsymbol{g}(\boldsymbol{x_j}),$$

$$f(x) = W_f x, g(x) = W_g x$$

$$h(x) = W_h x$$

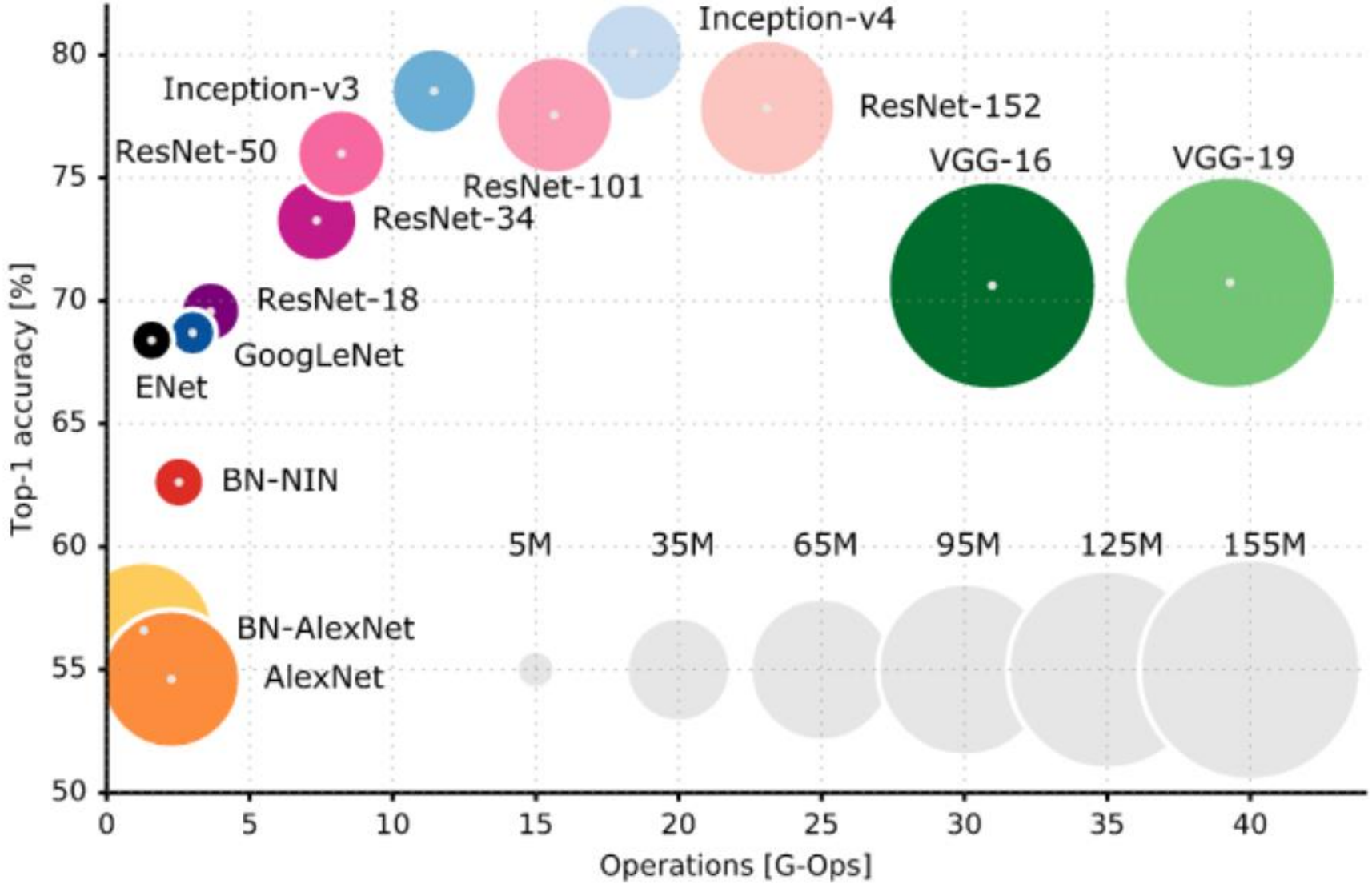
$$o_j = v\left(\sum_{i=1}^{N} \beta_{j,i} h(x_i)\right)$$

$$W_f, W_g, W_h \in \mathbb{R}^{C^* \times C}$$

$$v(x) = W_v x, W_v \in \mathbb{R}^{C \times C^*}$$

Team	Year	Place	Error (top-5)	External data
SuperVision – Toronto (AlexNet, 7 layers)	2012	-	16.4%	no
SuperVision	2012	1st	15.3%	ImageNet 22k
Clarifai – NYU (7 layers)	2013	-	11.7%	no
Clarifai	2013	1st	11.2%	ImageNet 22k
VGG – Oxford (16 layers)	2014	2nd	7.32%	no
GoogLeNet (19 layers)	2014	1st	6.67%	no
ResNet (152 layers)	2015	1st	3.57%	
Human expert*			5.1%	

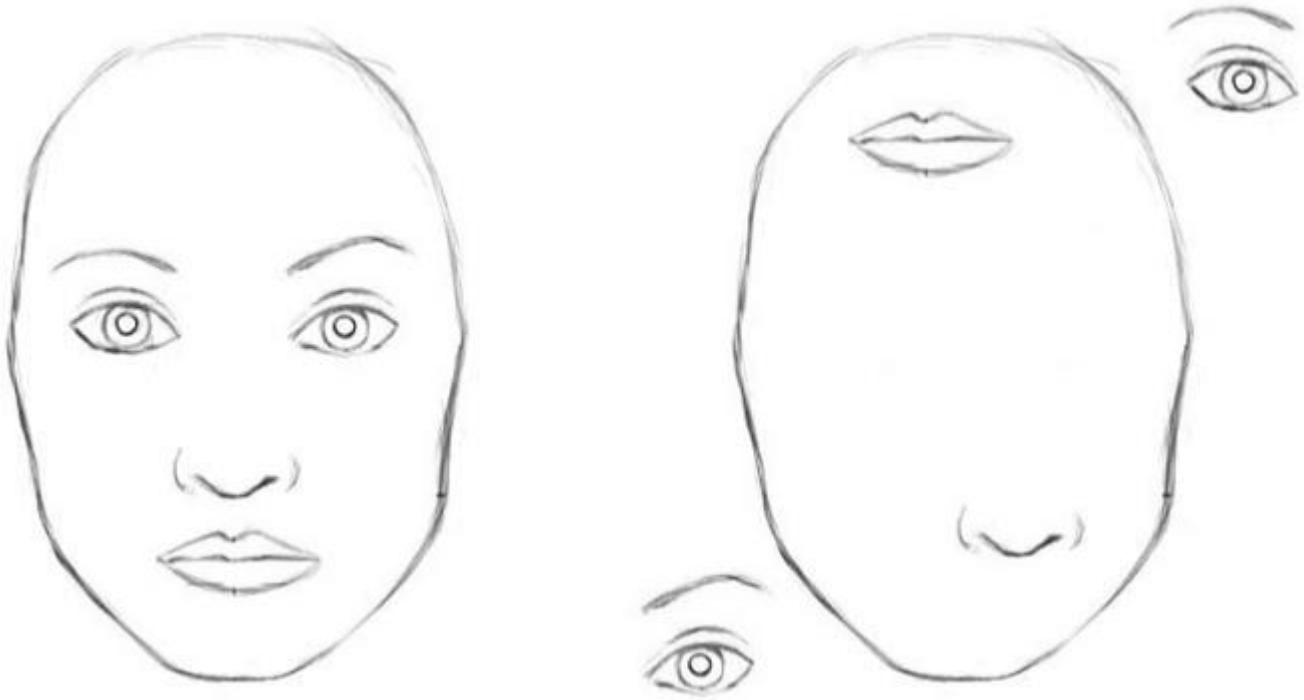
http://karpathy.github.io/2014/09/02/what-i-learned-from-competing-against-a-convnet-on-imagenet/



https://culurciello.github.io/tech/2016/06/04/nets.html

What’s wrong with deep CNN?

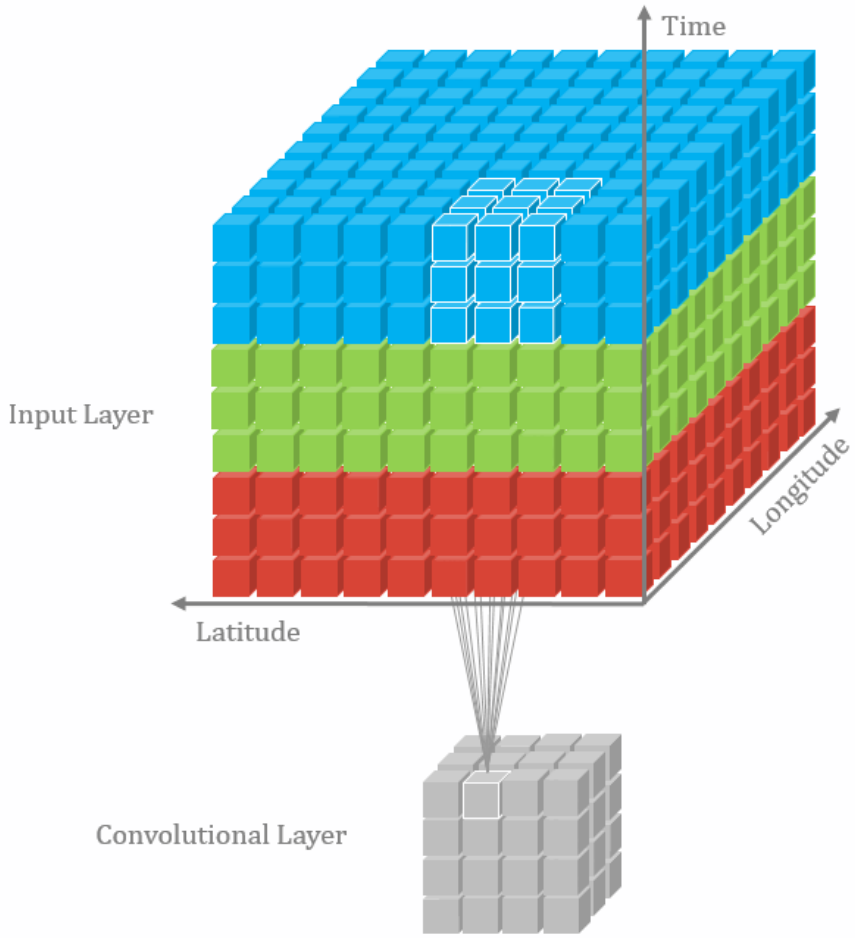
- Pooling causes much information loss



Some design tricks

- Reduce filter sizes (except possibly at the lowest layer), factorize filters aggressively
- Use 1x1 convolutions to reduce and expand the number of feature maps judiciously
- Use skip connections and/or create multiple paths through the network

3D CNN



Must suggested reading

A Survey of the Recent Architectures of Deep Convolutional Neural Networks

Asifullah Khan, Anabia Sohail, Umme Zahoora, Aqsa Saeed Qureshi

Deep Convolutional Neural Network (CNN) is a special type of Neural Networks, which has shown exemplary performance on several competitions related to Computer Vision and Image Processing. Some of the exciting application areas of CNN include Image Classification and Segmentation, Object Detection, Video Processing, Natural Language Processing, and Speech Recognition. The powerful learning ability of deep CNN is primarily due to the use of multiple feature extraction stages that can automatically learn representations from the data. The availability of a large amount of data and improvement in the hardware technology has accelerated the research in CNNs, and recently interesting deep CNN architectures have been reported. Several inspiring ideas to bring advancements in CNNs have been explored, such as the use of different activation and loss functions, parameter optimization, regularization, and architectural innovations. However, the significant improvement in the representational capacity of the deep CNN is achieved through architectural innovations. Notably, the ideas of exploiting spatial and channel information, depth and width of architecture, and multi-path information processing have gained substantial attention. Similarly, the idea of using a block of layers as a structural unit is also gaining popularity. This survey thus focuses on the intrinsic taxonomy present in the recently reported deep CNN architectures and, consequently, classifies the recent innovations in CNN architectures into seven different categories. These seven categories are based on spatial exploitation, depth, multi-path, width, feature-map exploitation, channel boosting, and attention. Additionally, the elementary understanding of CNN components, current challenges, and applications of CNN are also provided.

Checking codebase

- https://github.com/fchollet/deep-learning-models

- Written in Keras

Suggested readings

- https://culurciello.github.io/tech/2016/06/04/nets.html
- Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, Gradient-based learning applied to document recognition, Proc. IEEE 86(11): 2278–2324, 1998.
- A. Krizhevsky, I. Sutskever, and G. Hinton, ImageNet Classification with Deep Convolutional Neural Networks, NIPS 2012
- M. Zeiler and R. Fergus, Visualizing and Understanding Convolutional Networks, ECCV 2014
- K. Simonyan and A. Zisserman, Very Deep Convolutional Networks for Large-Scale Image Recognition, ICLR 2015
- M. Lin, Q. Chen, and S. Yan, Network in network, ICLR 2014
- C. Szegedy et al., Going deeper with convolutions, CVPR 2015
- C. Szegedy et al., Rethinking the inception architecture for computer vision, CVPR 2016
- K. He, X. Zhang, S. Ren, and J. Sun, Deep Residual Learning for Image Recognition, CVPR 2016