Deep Learning in Computer Vision

June 7th, 2019

Janus Nørtoft Jensen, jnje@dtu.dk

Topics of this lecture

- Better optimization
-

Optimization

Learning rate annealing

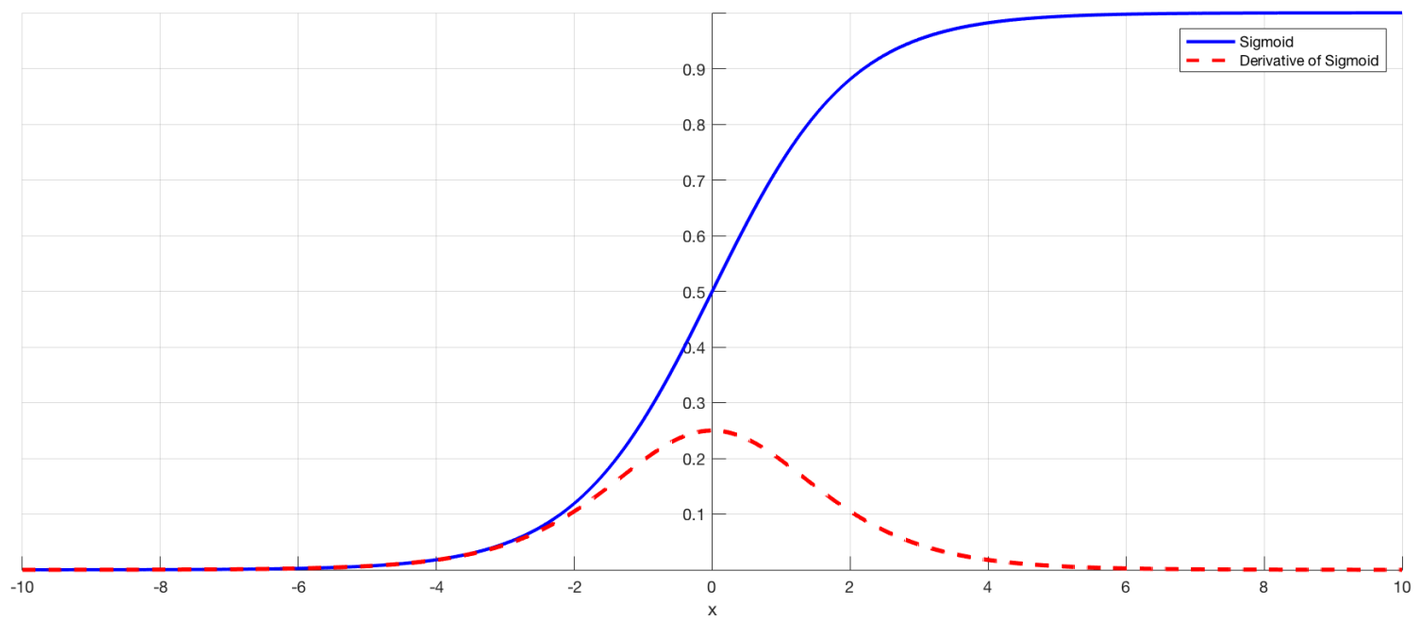
- Often it is helpful to change to learning rate over time when training deep neural networks.
  - If we are far from a minimum we maybe want a high learning rate
  - If we are close to a minimum we want a smaller learning rate.
- If the learning rate is to large the the updated parameters will just bounce around and not reach a minimum
- If it is to small it takes ‘forever’ to reach a minimum

Better optimisation

- A lot have been done to do faster training of deep neural networks
- The gradients in a deep neural networks are unstable
  - Many gradients are multiplied together
    - Vanishing gradient / exploding gradient
- Better optimisation methods

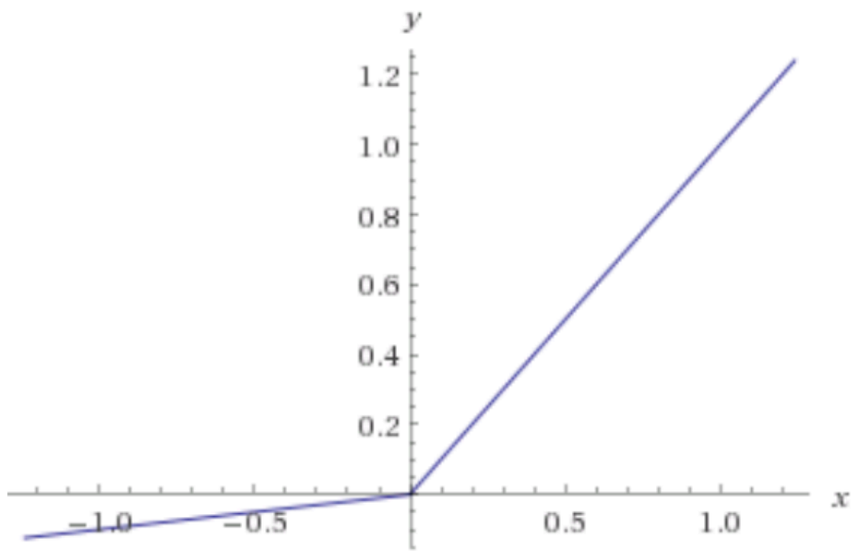
Vanishing gradients

- The gradients in the early layers of a network is the product of the gradients from later layers
  - If we use a sigmoid activation function we may see the ‘vanishing gradient problem’



Vanishing gradient

- Solution?
  - Use ReLU activation function
- Drawbacks
  - No gradient below 0
  - Not differentiable in 0
    - Usually not a problem
- Sometimes people use a ‘Leaky ReLU’ to overcome this



Unstable gradients

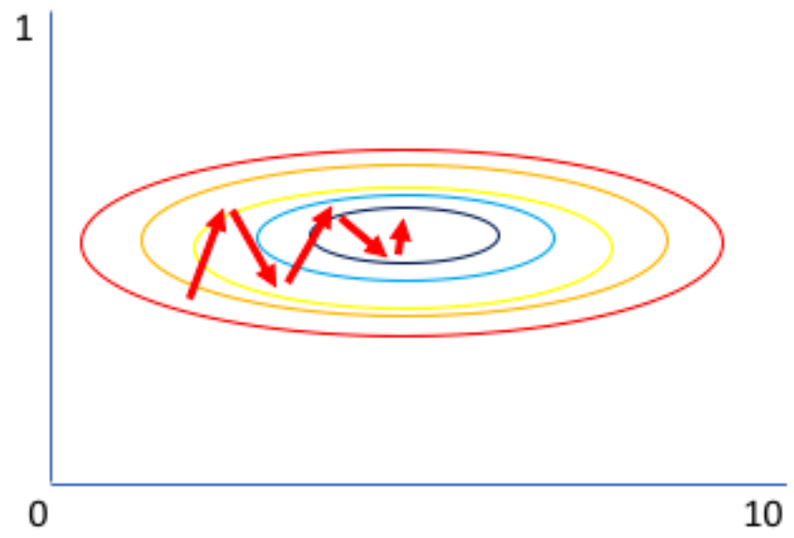
- In general, deep neural networks have unstable gradients and layers may not learn at the same rate.

Input normalization

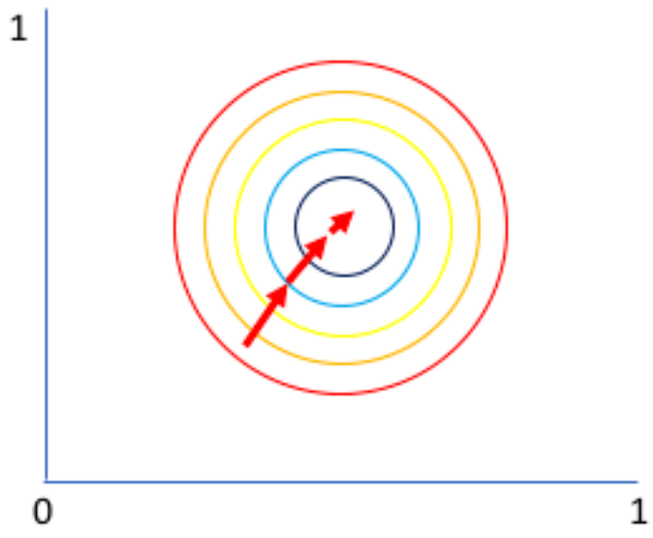
- Consider two images:
  - One of a person in very dark room
  - One of a person in bright sunlight
- We want to train a network on both types of images
- The difference between them is mainly their means (and std. deviations)

Input normalization

- We (almost) always normalize our input to some specified range ([0,1] or [-1,1]) or to zero mean and std. dev. 1
- Intuitively this makes it easier for the network to learn since the output will also be in the same(ish) range

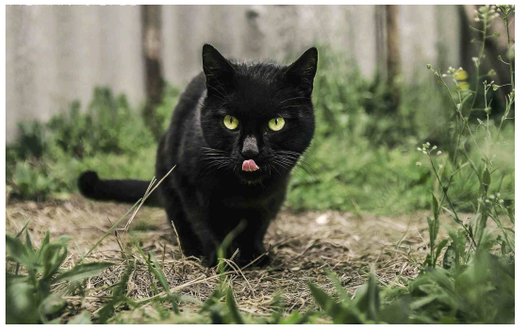


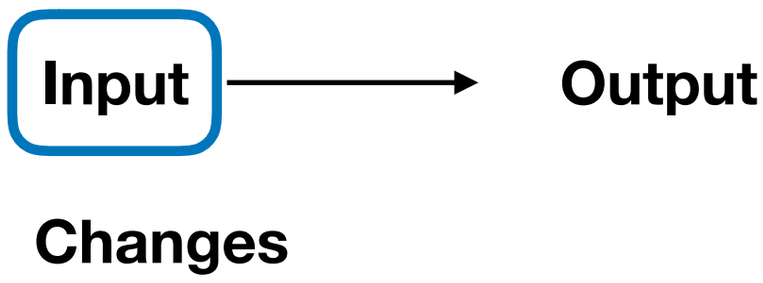
Gradient of larger parameter dominates the update



Both parameters can be updated in equal proportions

Input normalization





Learning rate annealing

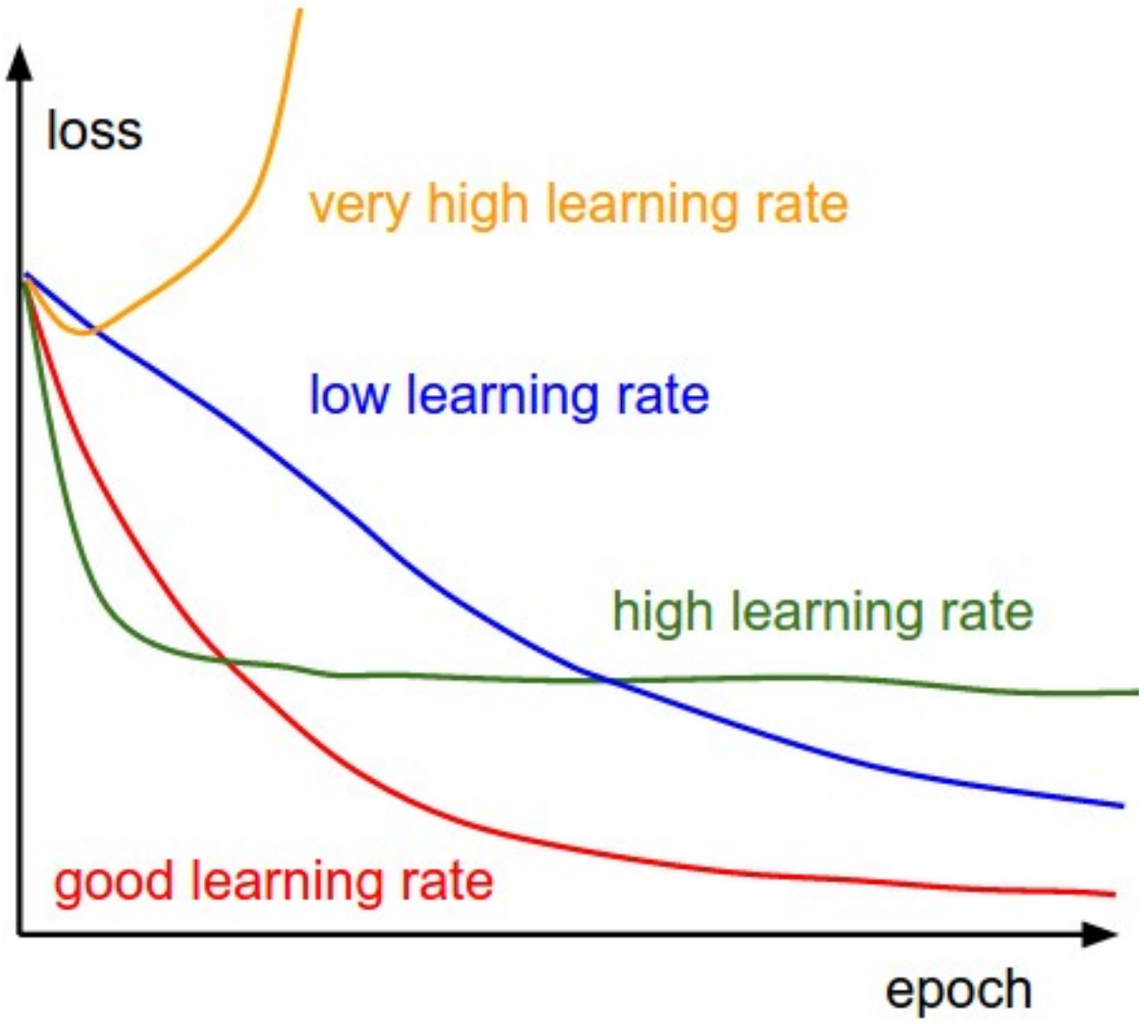
- Two common types of learning rate decay are
- Step decay
  - Reduce the learning rate by some factor $k < 1$ every few epochs. The factor and how many epochs is very dependent on the problem

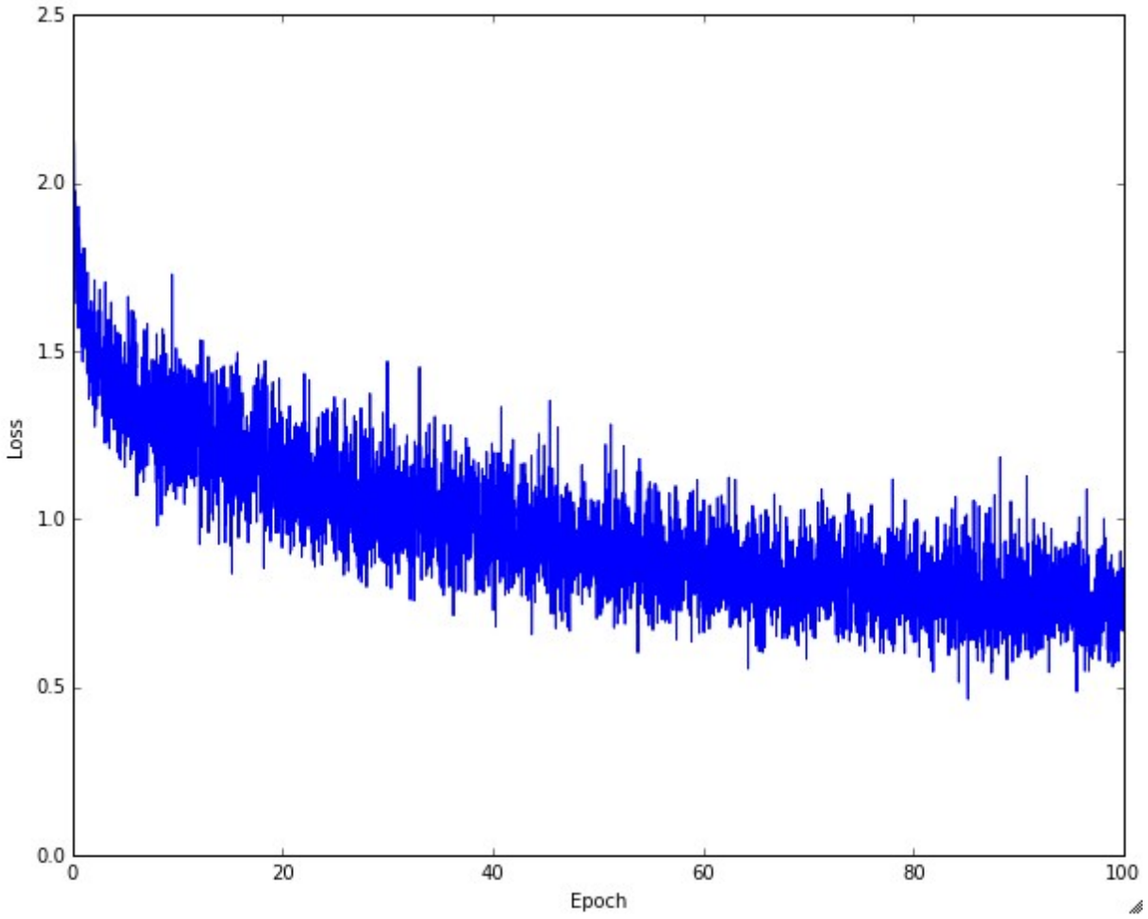
- Exponential decay

$$\alpha = \alpha_0 e^{-kt}$$

  - Where a_0, k are hyperparameters and t is the time step (epoch number)

Learning rate annealing





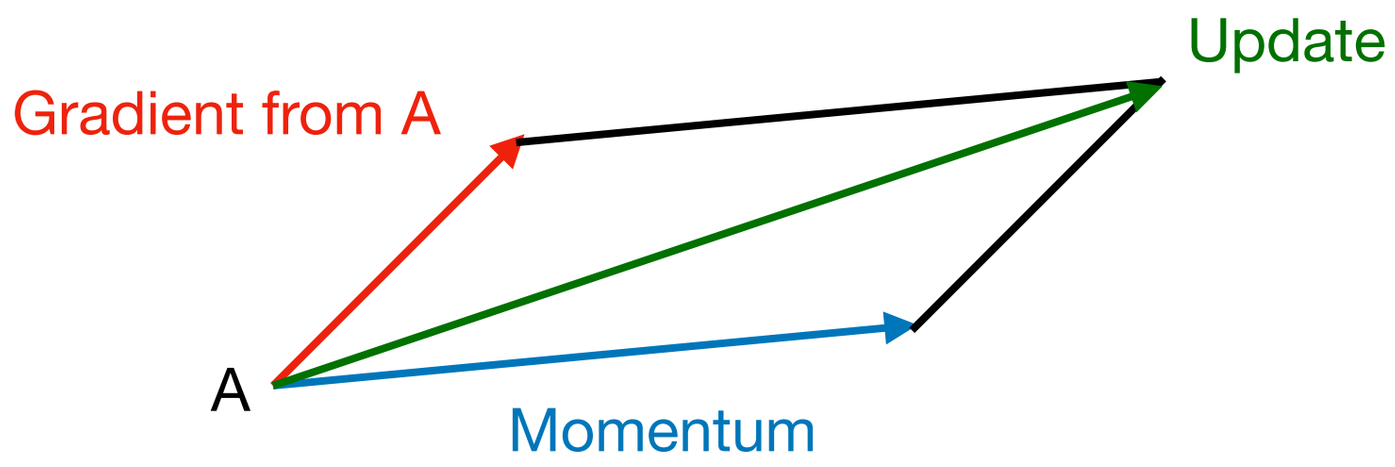
Li & Karparthy, CS231n Convoluation Neural Networks for Visual Recognition, Stanford University

Using other optimisers

- Simply using SGD is sufficient for many tasks
- Sometimes, however, training takes a long time which may be reduced if we can take better steps adapting to the loss surface
- Many SGD alterations have been invented to do just this

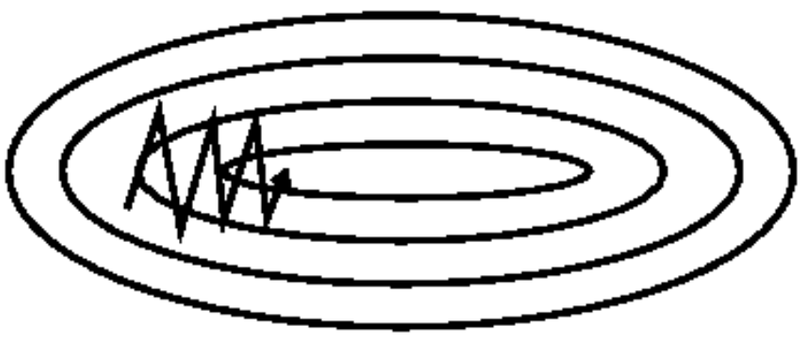
SGD with momentum

- Consider a ball rolling down a hillside
  - Just because the ball meets a small obstacle does not mean that it completely changes direction
  - Instead it keeps some of the velocity in it's current direction and add some velocity in another direction
  - We call this momentum
  - We can use this idea to take more ideal steps than SGD

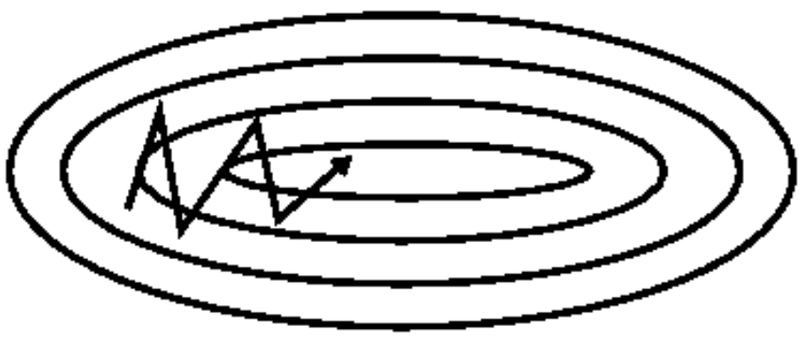


SGD with momentum

- If a surface curves more steeply in one dimension than in another it takes many steps for SGD to move to the minimum
- SGD would oscillate across the slopes with high curvature with only minor progress towards the minima



SGD



SGD with momentum

SGD with momentum

- Standard SGD

$$w_{j,k}^{\ell} \rightarrow w_{j,k}^{\bar{\ell}} = w_{j,k}^{\ell} - \alpha \frac{\partial L}{\partial w_{j,k}^{\ell}} \qquad \mathbf{W} \rightarrow \bar{W} = \mathbf{W} - \alpha \nabla_W \mathcal{L}$$

- SGD + momentum

$$\mathbf{W} \rightarrow \bar{W} = \mathbf{W} - \mathbf{v}_t$$

$$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} + \alpha \nabla_W \mathcal{L}$$

- Typically $\gamma = 0.9$

RMSProp

- Root Mean Square Propagation (RMSProp)
- The idea is to adapt the learning rate α for each individual parameter such that we that we take large steps in "good" dimensions and smaller steps in "bad" dimensions

$$v_t = \gamma v_{t-1} + (1-\gamma)\left(\frac{\partial \mathcal{L}}{\partial w_{j,k}^{\ell}}\right)^2$$

$$w_{j,k}^{\ell} \rightarrow w_{j,k}^{\bar{\ell}} = w_{j,k}^{\ell} - \frac{\alpha}{\sqrt{v_t}} \frac{\partial \mathcal{L}}{\partial w_{j,k}^{\ell}}$$

RMSProp

$$w_{j,k}^{\ell} \rightarrow w_{j,k}^{\overline{\ell}} = w_{j,k}^{\ell} - \frac{\alpha}{\sqrt{v_t}} \frac{\partial \mathcal{L}}{\partial w_{j,k}^{\ell}}$$

Adam

- Adaptive Moment Estimation (Adam) is a combination of momentum and RMSProp

$$m_w^t = \gamma_1 m_w^{t-1} + (1-\gamma_1)\frac{\partial \mathcal{L}}{\partial w_{j,k}^\ell}$$

$$v_w^t = \gamma_2 v_w^{t-1} + (1-\gamma_2)\left(\frac{\partial \mathcal{L}}{\partial w_{j,k}^\ell}\right)^2$$

- Bias corrections

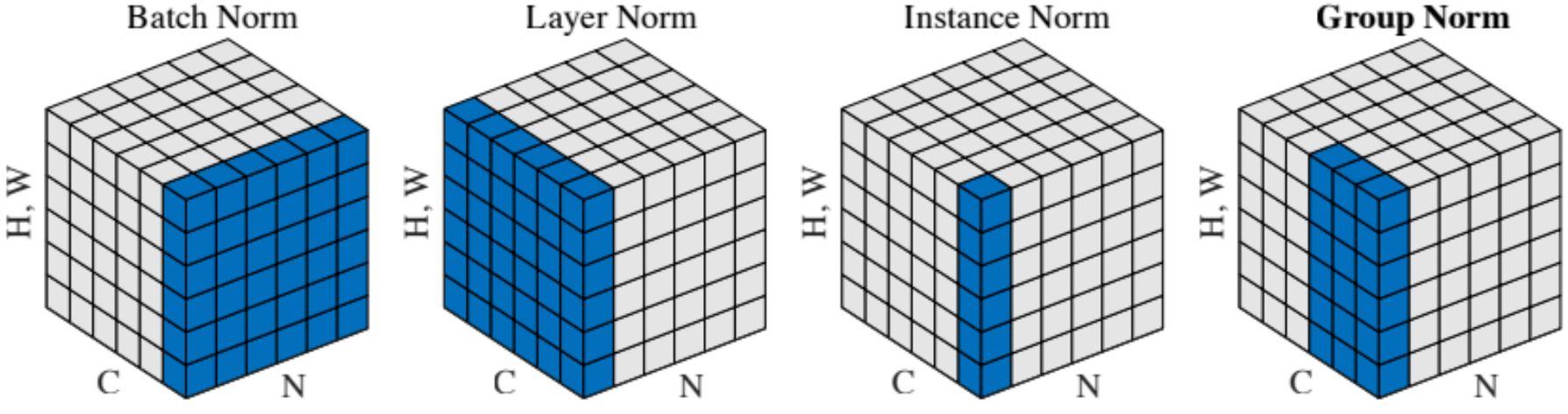
$$\hat{m_w^t} = \frac{m_w^t}{1-\gamma_1^{t+1}}, \quad \hat{v_w^t} = \frac{v_w^t}{1-\gamma_2^{t+1}}$$

- Update

$$w_{j,k}^\ell \rightarrow w_{j,k}^{\overline{\ell}} = w_{j,k}^\ell - \alpha \frac{m_w^t}{\sqrt{v_w^t}}$$

Batch normalisation

- We normalize the input to the network for faster convergence and more stability
  - Why not do the same for the outputs from the hidden layers?
- This is called batch normalisation
- Batch normalisation normalises the output of each layer by subtracting the mean and dividing by the standard deviation of the of mini-batch outputs



Batch normalisation

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

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

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

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

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

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

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

Ioffe & Szegedy, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Batch normalisation

- Batch normalisation approximately makes that the output distribution from each layer is always the same
- Batch normalisation adds some noise to the output of each neuron (similar to dropout)
- This functions as a regulariser

Batch normalisation at test time

- At test time we might only process images one at a time
  - We cannot do mini-batch statistics

- We keep a running average of the minibatch statistics across the minibatches to use at test time

$$\mu_t = \gamma \mu_{t-1} + (1 - \gamma)\mu_B$$

Common architectures

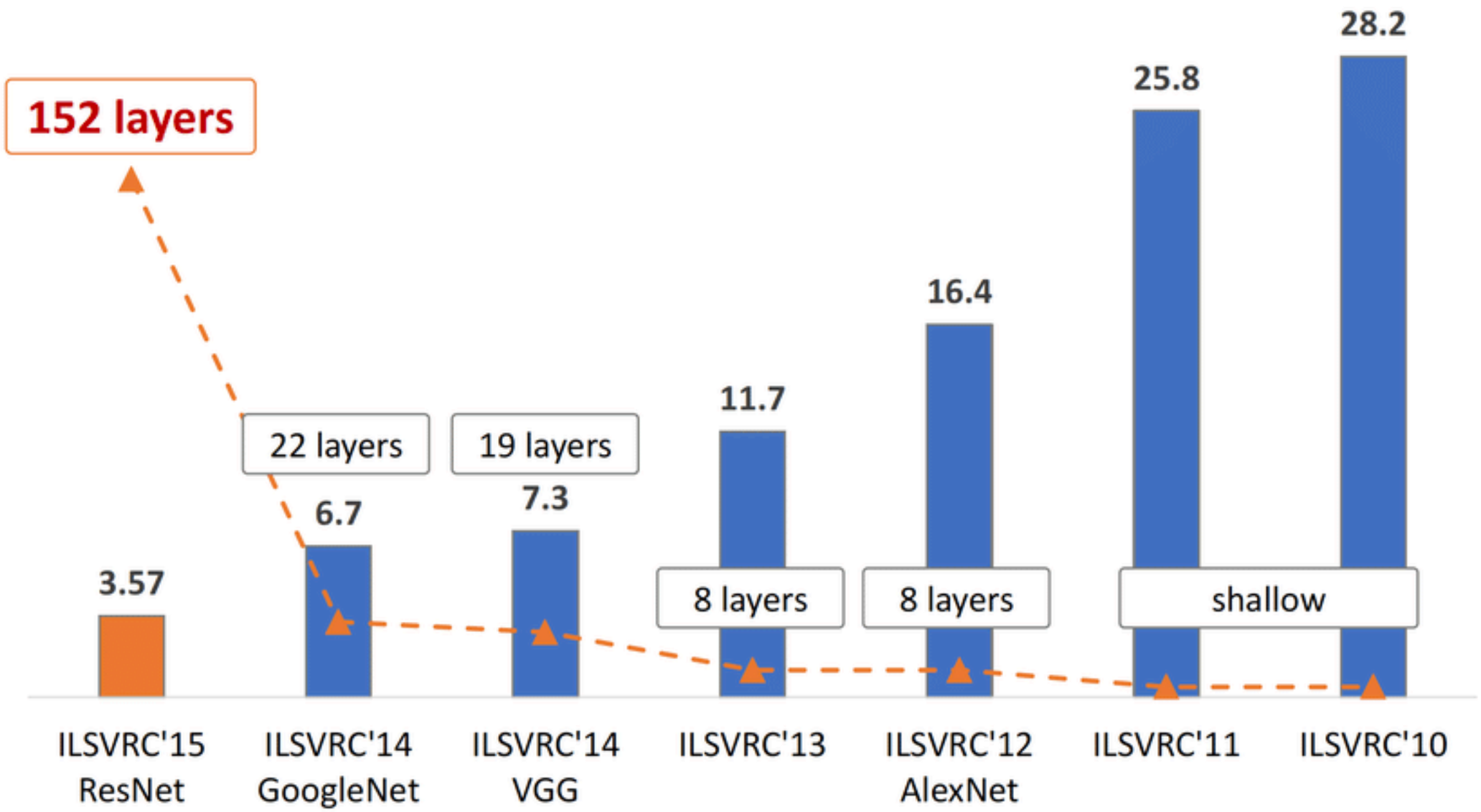
ImageNet Large Scale Visual Recognition Challenge

- The ILSVRC is a large scale challenge for object detection and image classification

- ImageNet has more than 14 million images in more than 20.000 categories

- The yearly challenge has spurred some of the most widely used architectures for visual recognition today

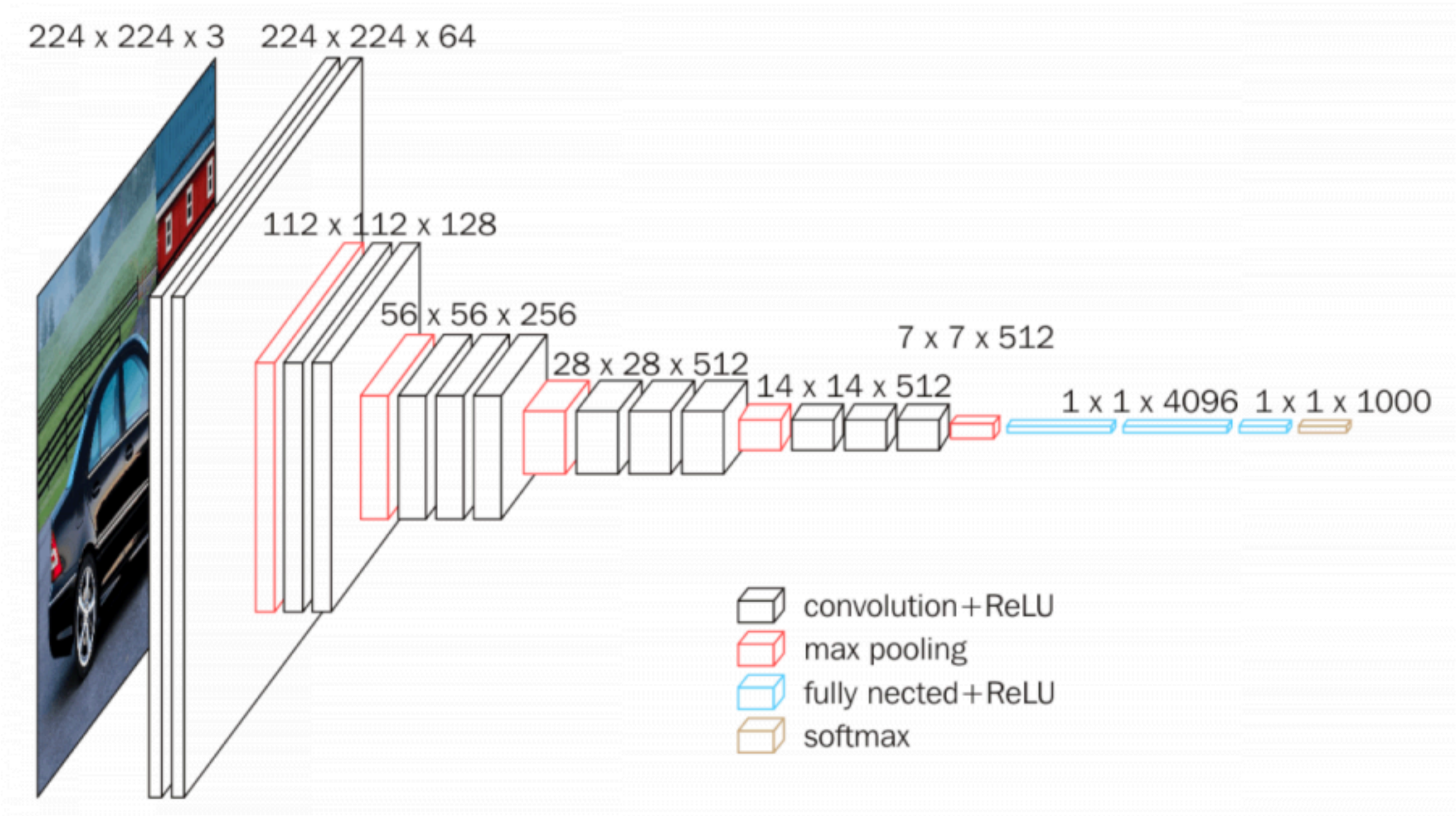
ILSVRC


152 layers
22 layers
19 layers
8 layers
8 layers
shallow
3.57
6.7
7.3
11.7
16.4
25.8
28.2
ILSVRC'15 ResNet
ILSVRC'14 GoogleNet
ILSVRC'14 VGG
ILSVRC'13
ILSVRC'12 AlexNet
ILSVRC'11
ILSVRC'10

VGGNet

- Simonyan & Zisserman pushed the standard convolution neural network with 3x3 convolutions to 19 layers in 2014
- Runner-up of ILSVRC 2014 (Their best model had 16 layers)
- VGG16 has around 138 million parameters
  - Using 4 GPUs it took 2-3 weeks to train

VGGnet

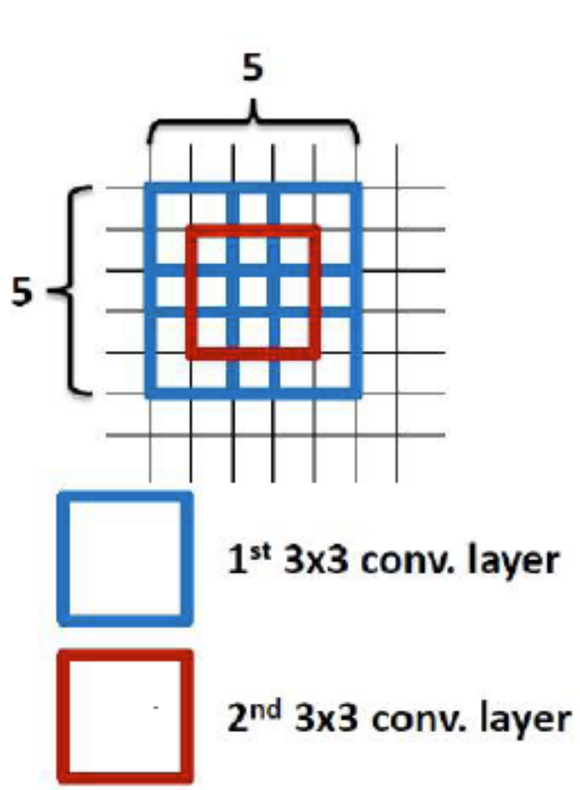

224 x 224 x 3
224 x 224 x 64
112 x 112 x 128
56 x 56 x 256
28 x 28 x 512
14 x 14 x 512
7 x 7 x 512
1 x 1 x 4096
1 x 1 x 1000
convolution+ReLU
max pooling
fully nected+ReLU
softmax

VGGNet

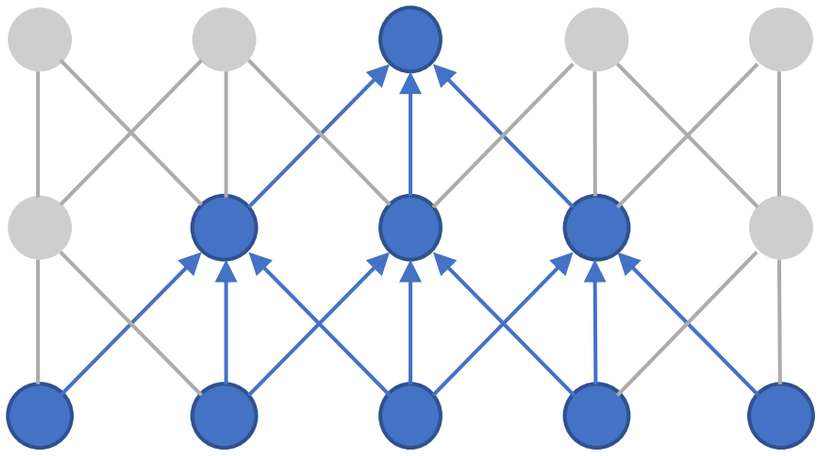
- Why only use 3x3 layers?

VGGNet

- Why only use 3x3 layers?
  - Stacking layers have a large receptive field
    - Two 3x3 layers - 5x5 receptive field
    - Three 3x3 layers 7x7 receptive field
    - …
- Less parameters to learn



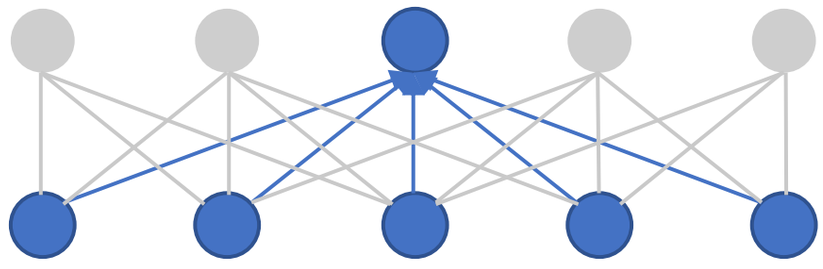
VGGNet



Two 3x3 convolutions

Parameters: $2 \cdot (3^2 + 1) = 20$

Operations: $2 \cdot (2 \cdot 3^2) = 36$



One 5x5 convolution

Parameters: $5^2 + 1 = 26$

Operations: $2 \cdot 5^2 = 50$

Three 3x3 convolutions

Parameters: $3 \cdot (3^2 + 1) = 30$

Operations: $3 \cdot (2 \cdot 3^2) = 54$

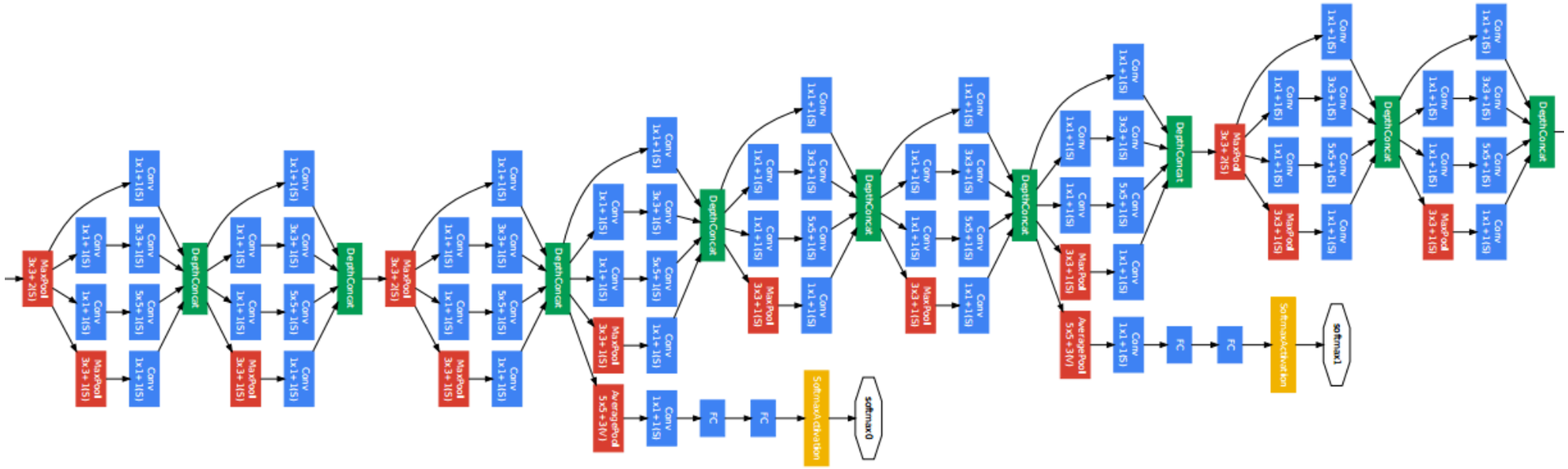
One 7x7 convolution

Parameters: $7^2 + 1 = 50$

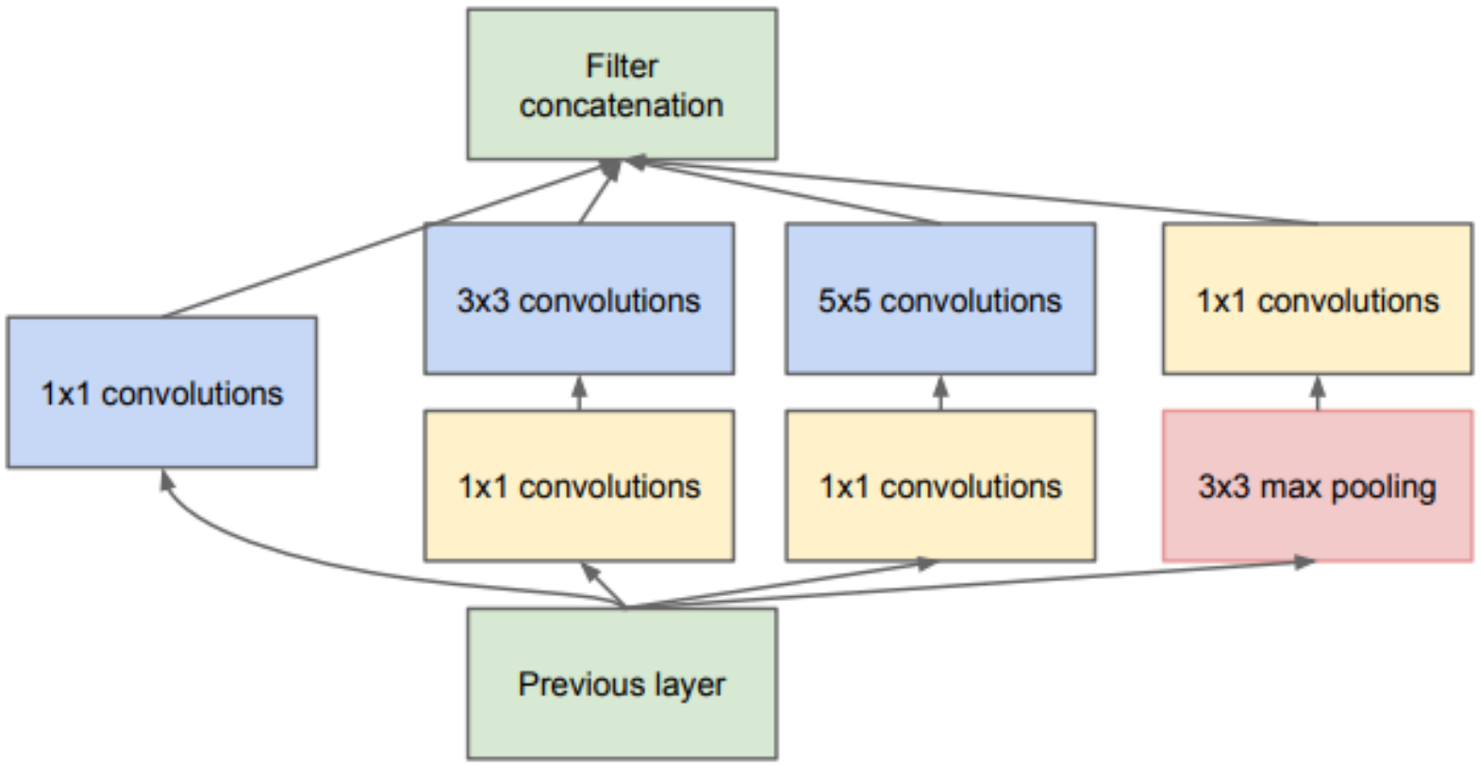
Operations: $2 \cdot 5^2 = 98$

GoogLeNet/Inception

- Winner of the ILSVRC 2014 Classification challenge
  - Able to train network with 23 layers

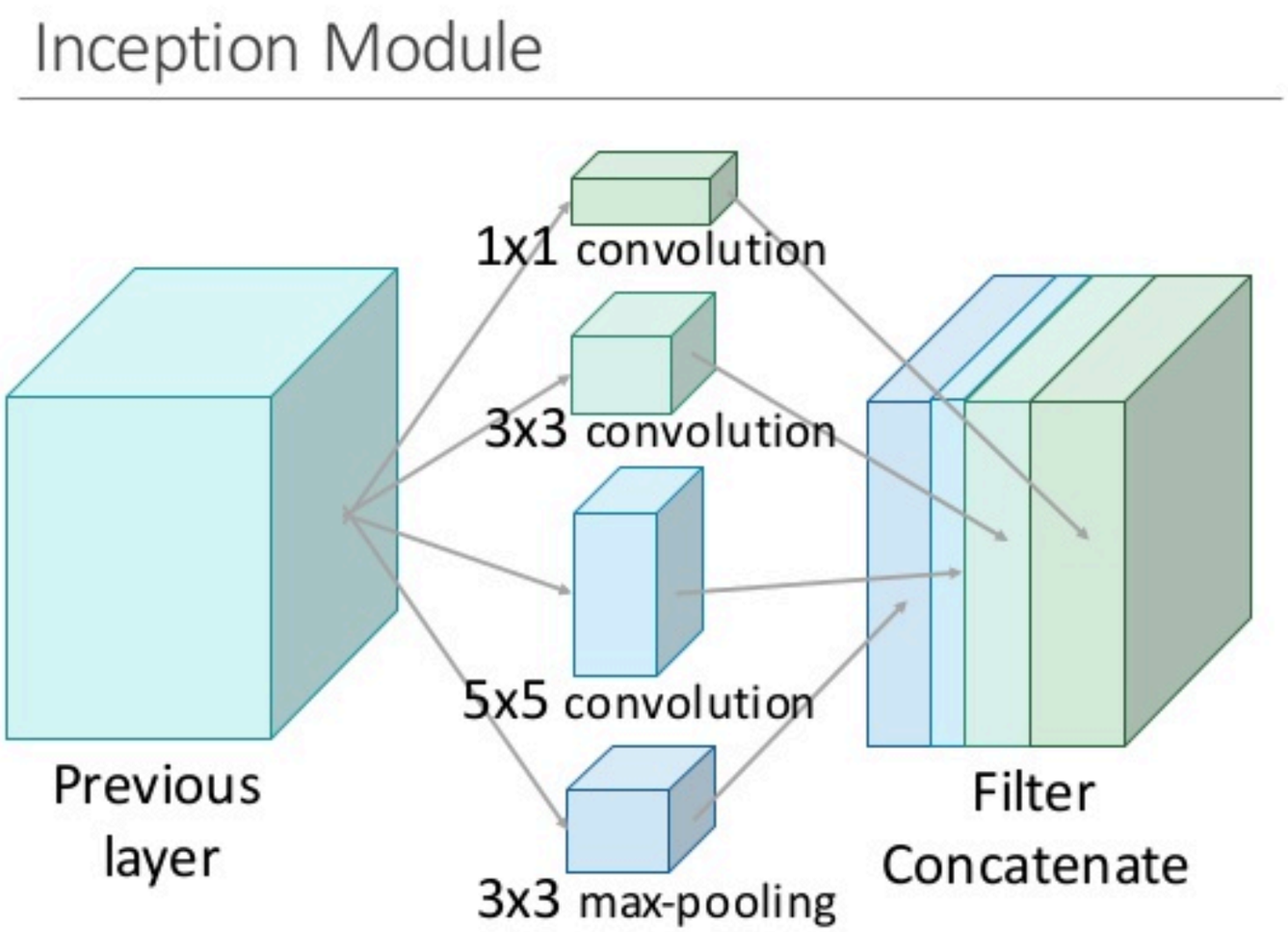


InceptionNet

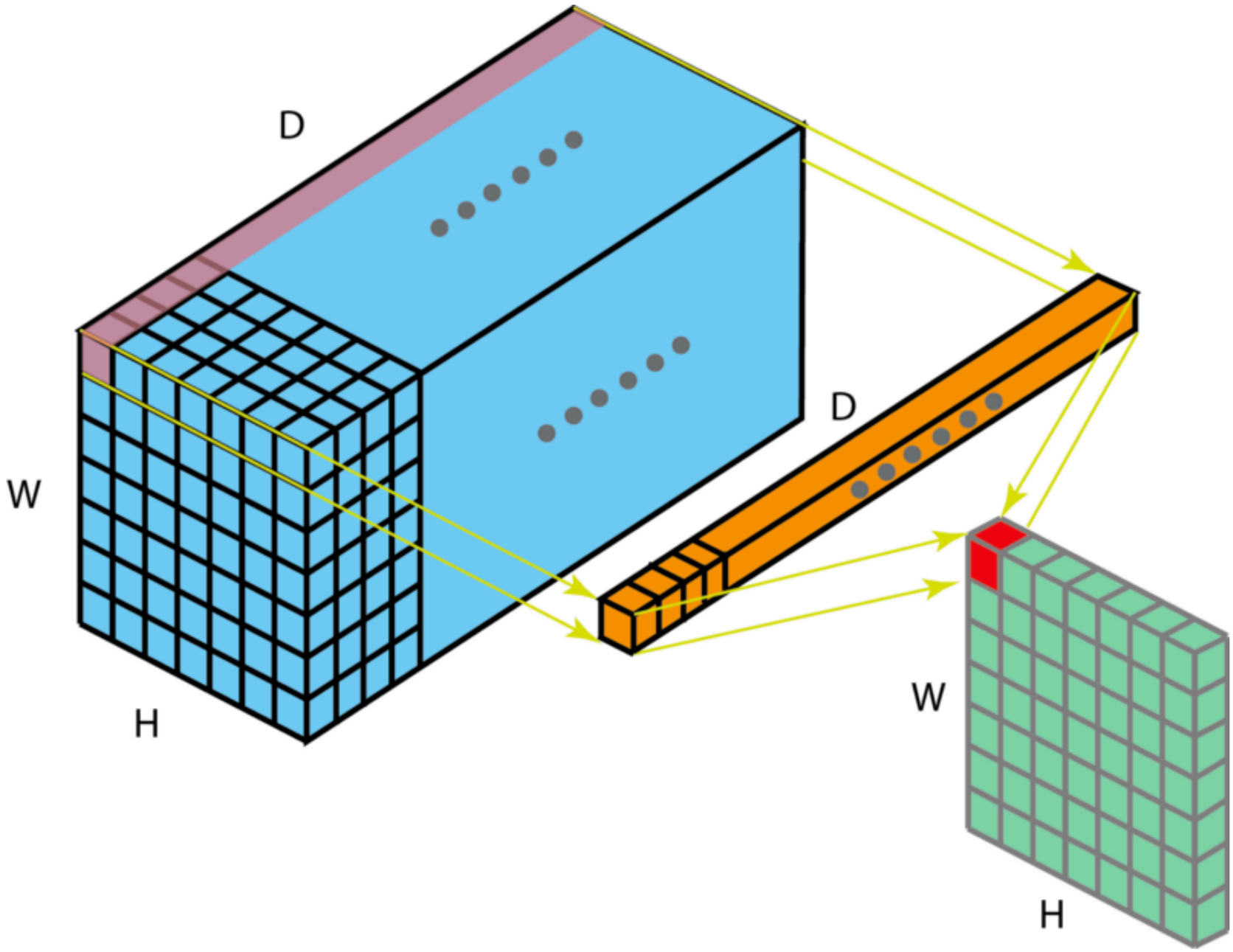


(b) Inception module with dimension reductions

InceptionNet

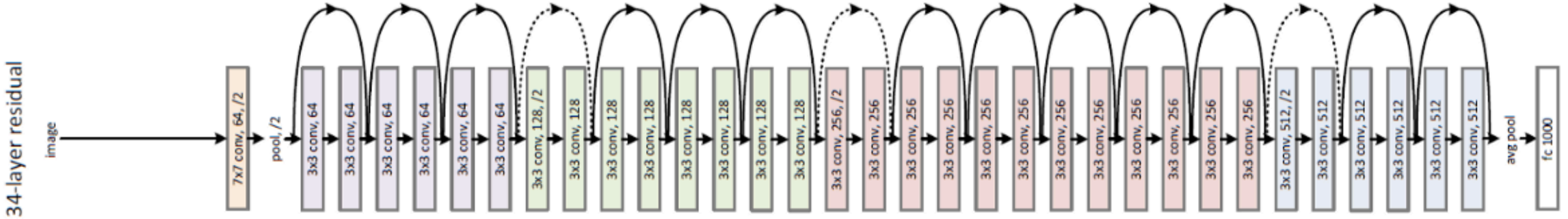


1x1 convolution

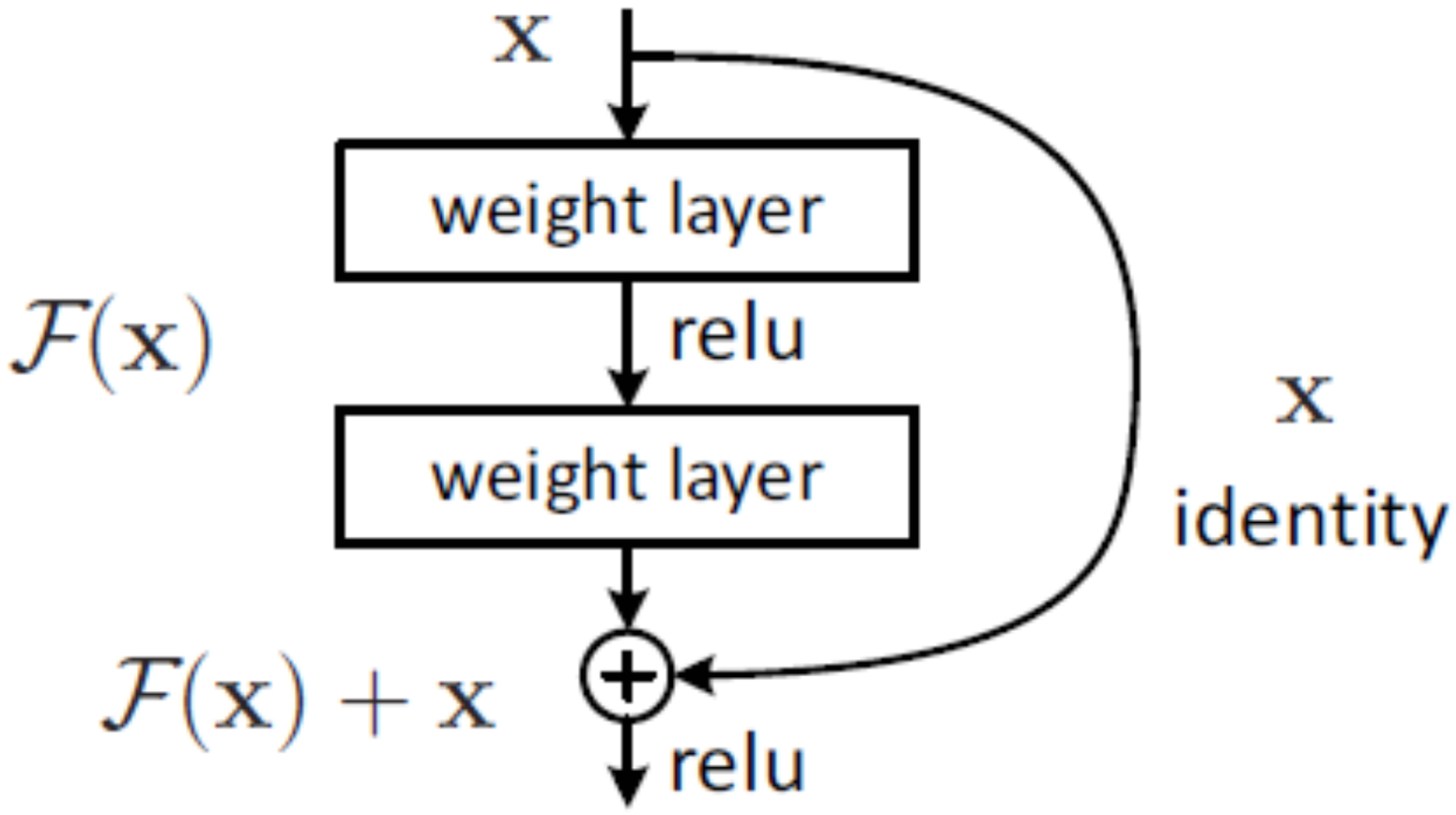


ResNet

- Won the ILSVRC 2015 Challenge
  - Could train 152 layers on ImageNet

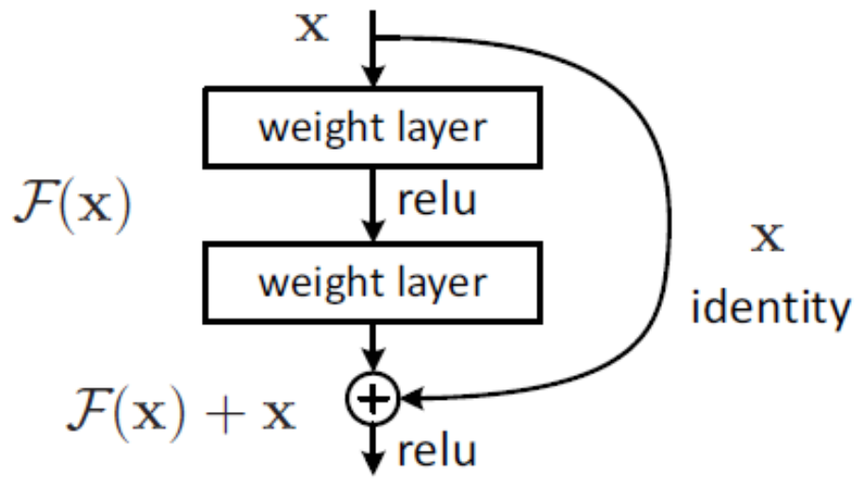


The building block of ResNets



ResNets

- Why can we train very large networks using residual blocks?
- If our weights are zero in a residual block we just feed the values through - so it can function as a more shallow network
- We can now train ResNets with more than 1000 layers
- In back propagation the gradients can use the identity mappings to move faster back to the early layers



```
conda install matplotlib
```

Remember to use gpustat