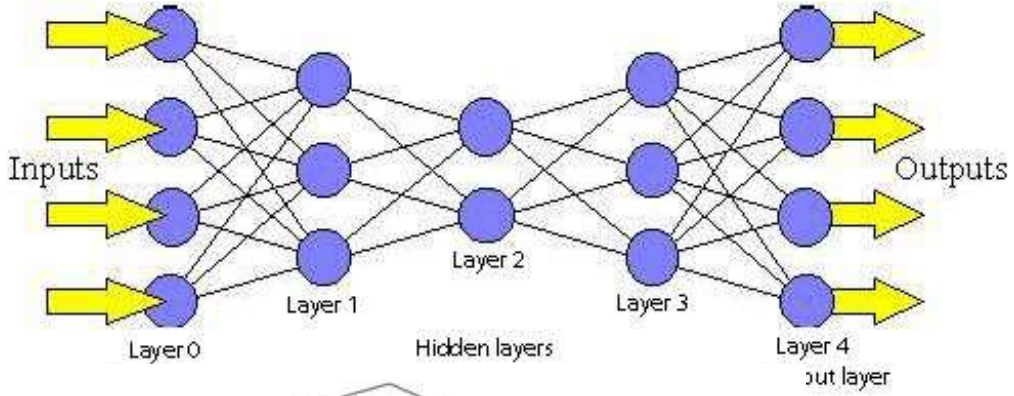
Optimization

7

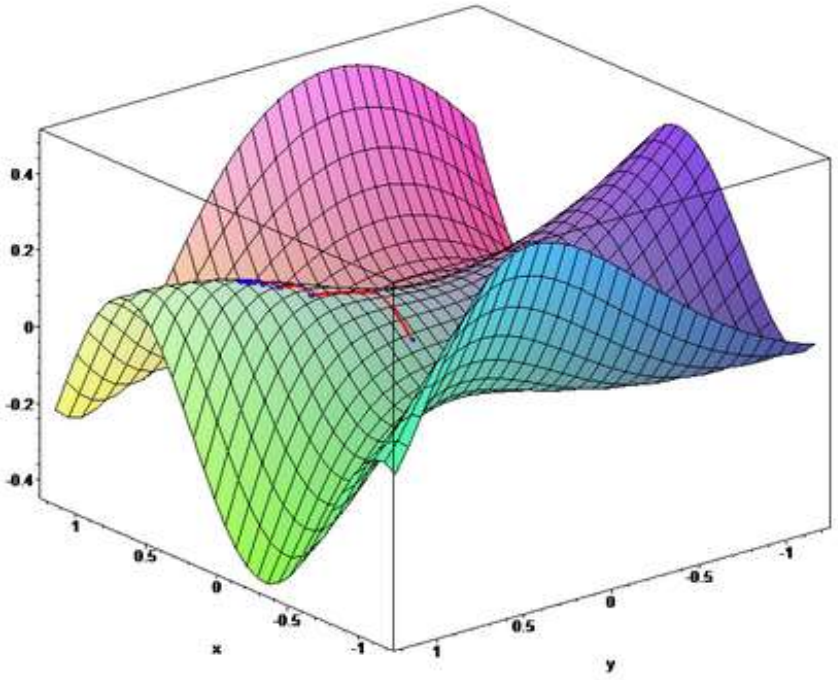
Feb 2020

Feed forward/Backpropagation Neural Network



Feed forward algorithm:

Activate the neurons from the bottom to the top.



Gradient Descent Method is a first-order optimization algorithm. To find a local minimum of a function, one takes a step proportional to the negative of the gradient of the function at the current point.

Gradient Descent

- Neural network: $\mathrm{f}(x;\theta)$
  - x: input (vector)
  - θ: model parameters (weights, biases)
- Cost Function: $C(\theta)$
  - How good your model parameters θ is
  - $C(\theta) = \sum_{(x,\hat{y})\in\mathbb{T}} C_x(\theta)$
  - $C_x(\theta) = l(\mathrm{f}(x;\theta), \hat{y})$
    - $\hat{y}$: reference output when x is input
    - $\mathrm{l}(\mathrm{f}(x;\theta), \hat{y})$: loss function between $\hat{y}$ and the output of neural network with model parameters θ
- Target: find θ that minimizes $C(\theta)$

Gradient Descent

Taylor series by definition

$$C(\theta_{new}) = C(\theta_{old}) + \nabla C \cdot [\theta_{new} - \theta_{old}] + ..$$

Here $\Delta\theta = [\theta_{new} - \theta_{old}]$, $\nabla C = \frac{\partial C}{\partial \theta}$

$$C(\theta_{new}) \approx C(\theta_{old}) + \frac{\partial C}{\partial \theta} \cdot \Delta\theta \qquad (1)$$

If we set $\Delta\theta = -\eta \frac{\partial C}{\partial \theta}$ with η a small positive learning rate

equation (1) becomes:

$$C(\theta_{new}) \approx C(\theta_{old}) + \frac{\partial C}{\partial \theta} \cdot \left(-\eta \frac{\partial C}{\partial \theta} \right) = C(\theta_{old}) - \eta \left(\frac{\partial C}{\partial \theta} \right) \left(\frac{\partial C}{\partial \theta} \right)$$

$C(\theta_{new}) \leq C(\theta_{old})$, since $\eta \left(\frac{\partial C}{\partial \theta} \right) \left(\frac{\partial C}{\partial \theta} \right)$ is always positive

Conclusion: if we set $\Delta\theta = -\eta \frac{\partial C}{\partial \theta}$ it will decrease C

- Target: find θ that minimizes $C(\theta)$
- Need to Compute $\frac{\partial C}{\partial \theta}$

How to back propagate?

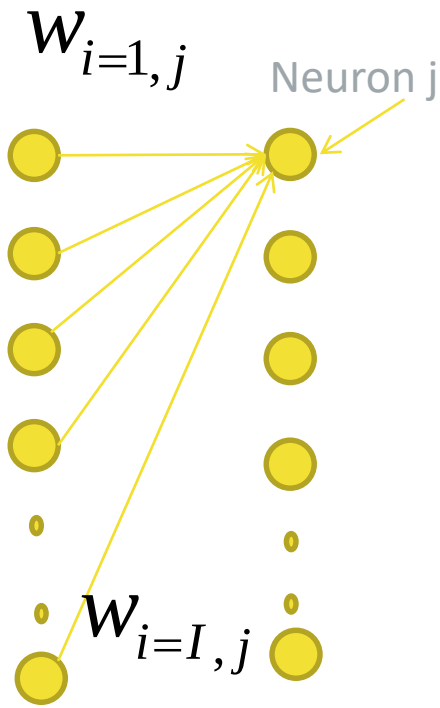
For a neuron j, Output is y_j

By definition , $u_j = \sum_{i=1}^{i=I} x_i w_{ij} + b_j$

$y_j = f(u_j) = f\left(\sum_{i=1}^{i=I} x_i w_{ij} + b_j\right)$

We want to find $\frac{\partial C}{\partial w_{ij}}$, *so*

$\frac{\partial C}{\partial w_{ij}} = \frac{\partial C}{\partial y_j}\frac{\partial y_j}{\partial u_j}\frac{\partial u_j}{\partial w_{ij}}$, by chain rule we will compute all derivative



i=1,2,..,I
I inputs to neuron j
Output of neuron j is y_j

Gradient Descent variants

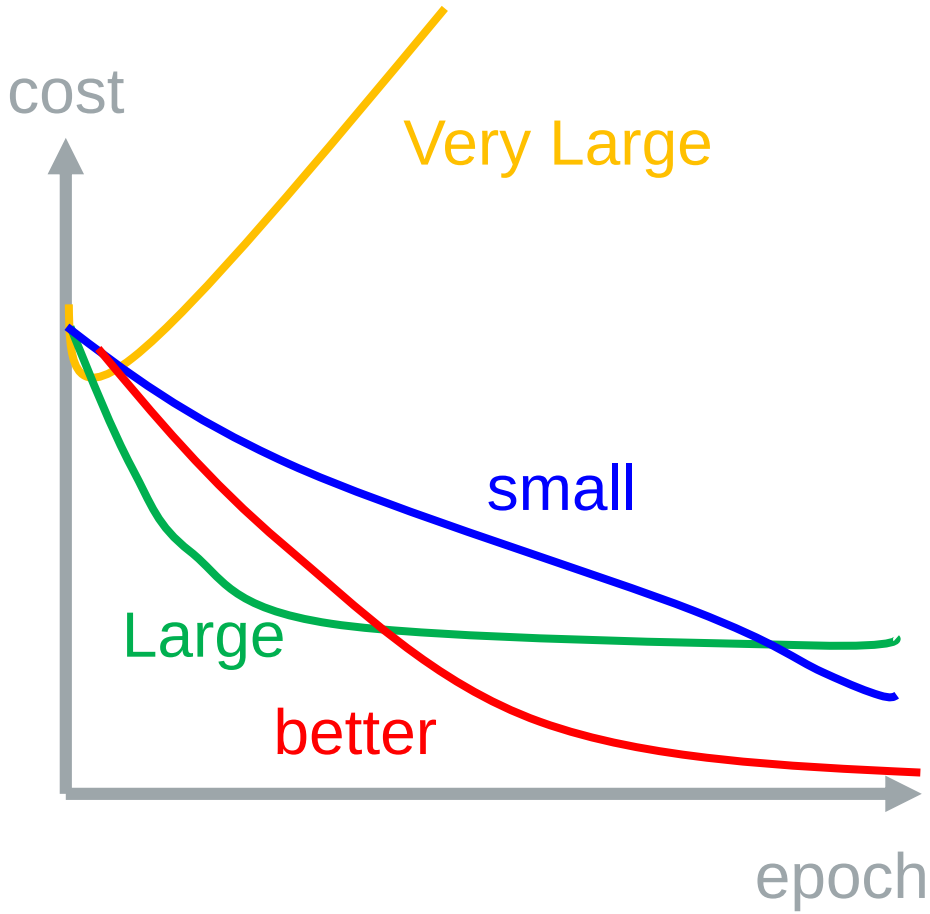
- Target: find θ that minimizing $C(\theta)$
  - $C(\theta) = \sum_{(x,\hat{y})\in\mathbb{T}} C_x(\theta)$
- Gradient descent
  - Initialization: θ^0
  - $\theta^t \leftarrow \theta^{t-1} - \eta\nabla C(\theta^{t-1})$
- SGD = Stochastic gradient descent
  - Pick $(x, \hat{y})$ from training data set $\mathbb{T}$
  - $\theta^t \leftarrow \theta^{t-1} - \eta\nabla C_x(\theta^{t-1})$

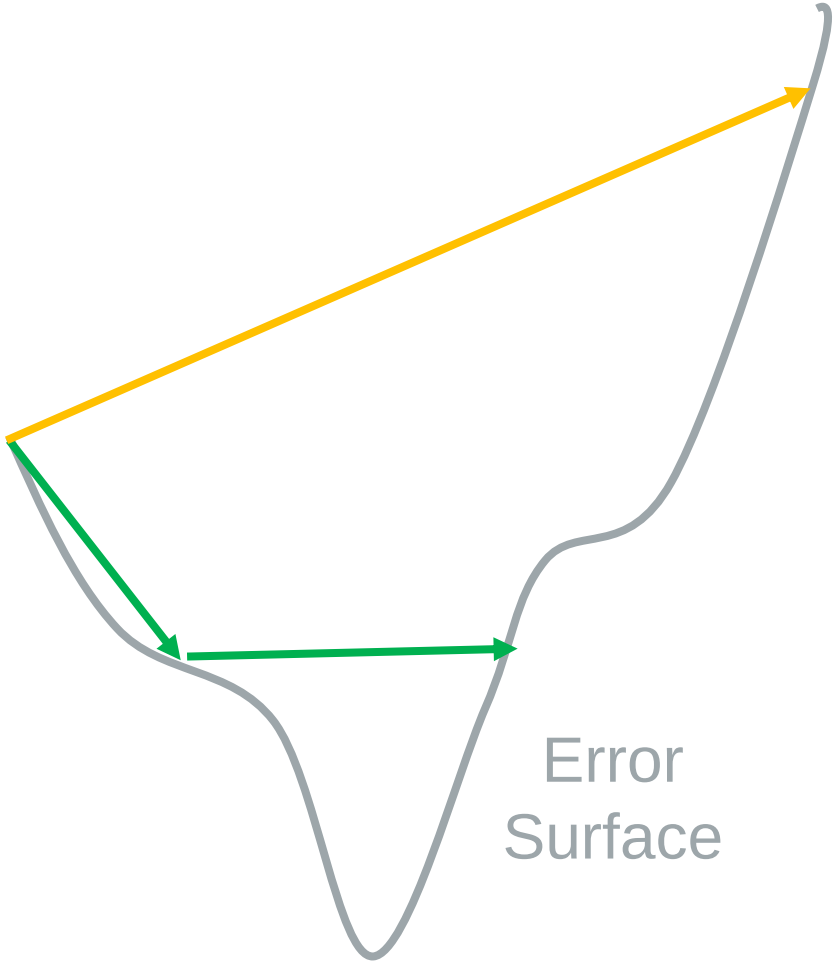
Tensorflow works with batches.

Gradients in a batch are computes in parallel on GPU.

Learning Rates

- $\theta^t \leftarrow \theta^{t-1} - \eta \nabla C(\theta^{t-1})$
- What is a good learning rate η ?





An iteration = back propagation of a batch batches.
An epoch = back propagation of the database

Learning Rates

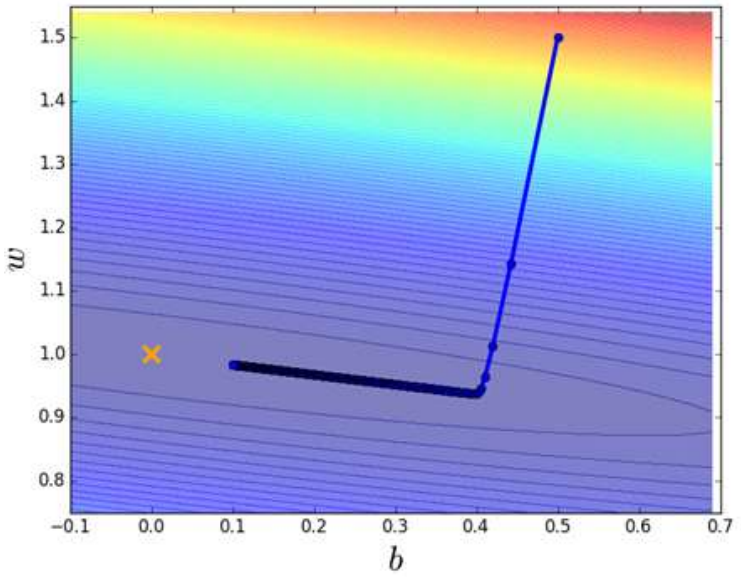
Popular & Simple Idea: Reduce the learning rate by some factor every few epochs.

At the beginning, we are far from a minimum, so we use larger learning rate

After several epochs, we are close to a minimum, so we reduce the learning rate

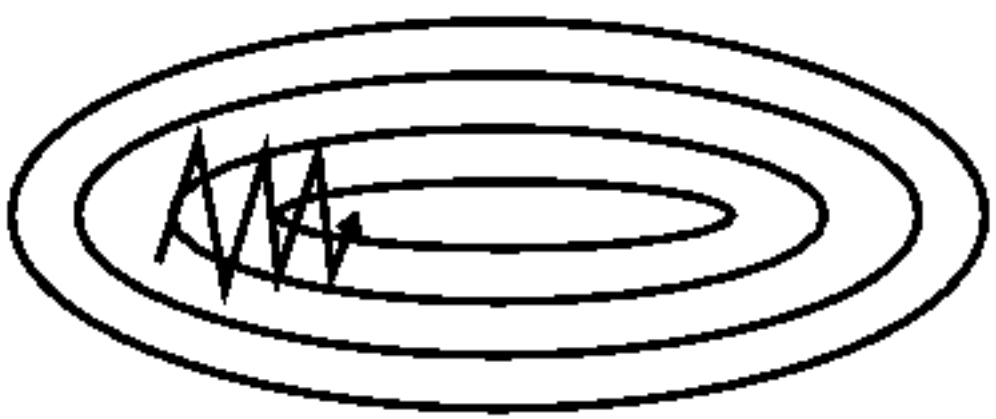
E.g. 1/t decay: $\eta^t = \eta/(t+1)$

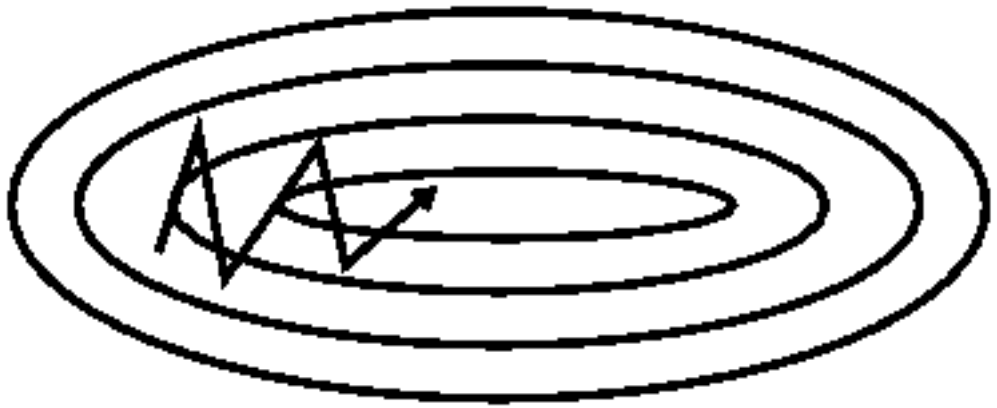
Not always true



Momentum

Momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations. It does this by adding a fraction of the update vector of the past time step to the current update vector:



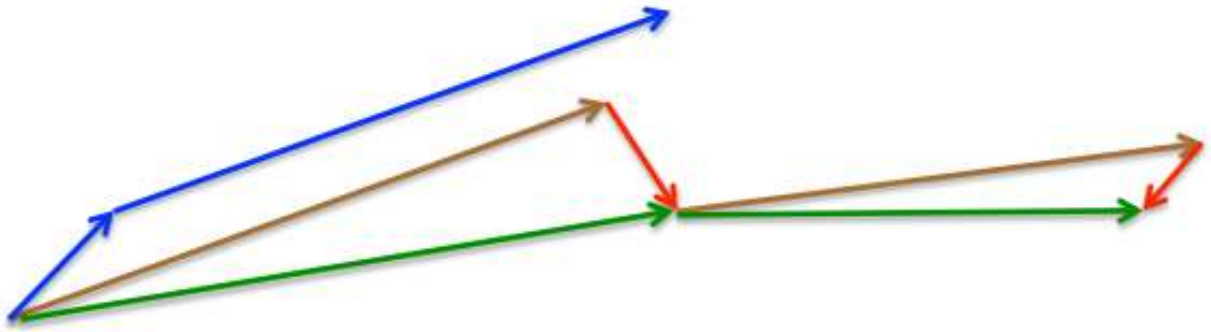


http://doi.org/10.1016/S0893-6080(98)00116-6

A better type of momentum (Nesterov 1983)

The standard momentum method first computes the gradient at the current location and then takes a big jump in the direction of the updated accumulated gradient.

- First make a big jump in the direction of the previous accumulated gradient.
- Then measure the gradient where you end up and make a correction.



brown vector = jump, red vector = correction, green vector = accumulated gradient

blue vectors = standard momentum

Adagrad

In a multilayer net, the magnitudes of the gradients are often very different for different layers, especially if the initial weights are small. The appropriate learning rates can vary widely between weights.

$$w^{t+1} \leftarrow w^{t} - \frac{\eta}{\sigma} g^{t}$$

σ: Average gradient of parameter w

If w has small average gradient σ → Larger learning rate

If w has large average gradient σ → Smaller learning rate

Adagrad

Divide the learning rate by "*average" gradient*

The "average" gradient is obtained while updating the parameters

$$\sigma^t = \sqrt{\frac{1}{t+1}\sum_{i=0}^{t}(g^i)^2}$$

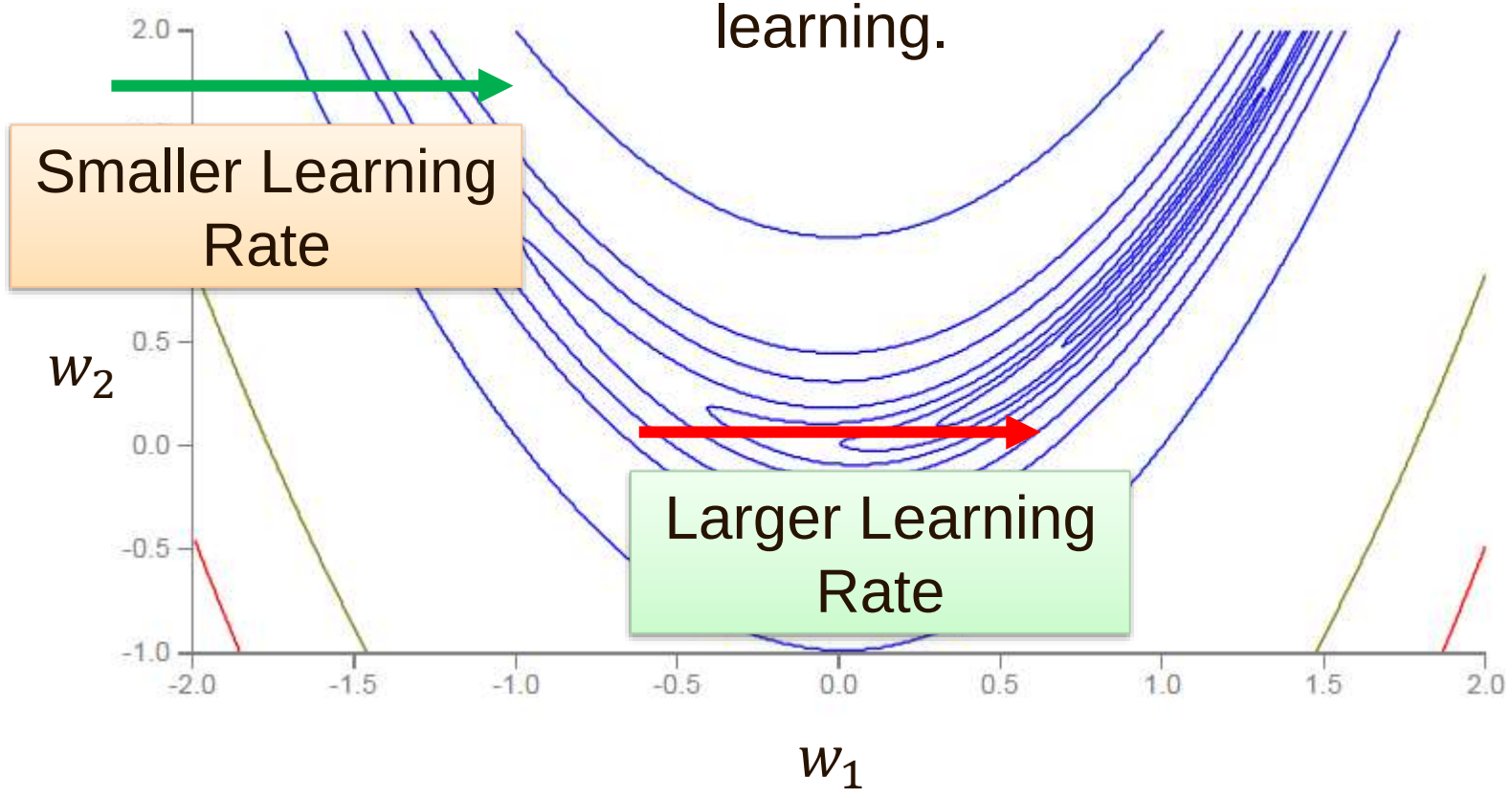
$$\eta^t = \frac{\eta}{\sqrt{t+1}}$$

$$w^{t+1} \leftarrow w^t - \frac{\eta^t}{\sigma^t} g^t$$

$$w^{t+1} \leftarrow w^t - \frac{\eta}{\sqrt{\sum_{i=0}^{t}(g^i)^2}} g^t$$

RMSProp

Error Surface can be complex in deep learning.



RMSProp

Same as adagrad , but with an exponentially decaying average

$$w^1 \leftarrow w^0 - \frac{\eta}{\sigma^0} g^0 \qquad \sigma^0 = g^0$$

$$w^2 \leftarrow w^1 - \frac{\eta}{\sigma^1} g^1 \qquad \sigma^1 = \sqrt{\alpha(\sigma^0)^2 + (1-\alpha)(g^1)^2}$$

$$w^3 \leftarrow w^2 - \frac{\eta}{\sigma^2} g^2 \qquad \sigma^2 = \sqrt{\alpha(\sigma^1)^2 + (1-\alpha)(g^2)^2}$$

$$\vdots$$

$$w^{t+1} \leftarrow w^t - \frac{\eta}{\sigma^t} g^t \qquad \sigma^t = \sqrt{\alpha(\sigma^{t-1})^2 + (1-\alpha)(g^t)^2}$$

Adam – ADAptive Moment estimation

Like RMSprop, but in addition to storing an exponentially decaying average of past squared gradients g_t, Adam also keeps an exponentially decaying average of past gradients m_t, similar to momentum:

$$m_{t+1} = \gamma_1 m_t + (1 - \gamma_1) \nabla \mathcal{L}(\theta_t)$$

$$g_{t+1} = \gamma_2 g_t + (1 - \gamma_2) \nabla \mathcal{L}(\theta_t)^2$$

$$\hat{m}_{t+1} = \frac{m_{t+1}}{1 - \gamma_1^{t+1}}$$

$$\hat{g}_{t+1} = \frac{g_{t+1}}{1 - \gamma_2^{t+1}}$$

$$\theta_{t+1} = \theta_t - \frac{\alpha \hat{m}_{t+1}}{\sqrt{\hat{g}_{t+1}} + \epsilon}$$

Weight Initialization

- Can be deceptively important!
- Bias terms are often initialized to 0 with no issues
- Weight matrices more problematic, ex.
  - If initialized to all 0s, can be shown that the tanh activation function will yield zero gradients
  - If the weights are all the same, hidden units will produce same gradients and behave the same as each other (wasting params)
- One solution: initialize all elements of weight matrix from uniform distribution over interval [–b, b]

Weight Initialization

Random from a distribution with zero mean and a specific variance

- **Standard initialization**

$$var(W) = \frac{1}{N^{in}}$$

- **“Xavier” initialization. picks std according to blob size.**

See “Understanding the difficulty of training deep feedforward neural networks”. Glorot and Bengio 2010.

$$var(W) = \frac{2}{N^{in} + N^{out}}$$

Weight Initialization

- **Whitening**
  A whitening transformation is a linear transformation that transforms a vector of random variables with a known covariance matrix into a set of new variables whose covariance is the identity matrix meaning that they are uncorrelated and all have variance 1.

  idea : avoid initialization with close filters

- **Mishkin & Matas. All you need is a good init. https://arxiv.org/pdf/1511.06422**
  init W with gaussian distribution
  SVD on W
  minibatch, rescale to have variance 1, iterates on layers

Unsupervised pre-training

Idea: model the distribution of unlabeled data using a method that allows the parameters of the learned model to inform or be somehow transferred to the network

- **Can be an effective way to both initialize and regularize a feedforward network**
- **Particularly useful when the volume of labeled data is small relative to the model's capacity.**
- **The use of activation functions such as rectified linear units (which improve gradient flow in deep networks), along with good parameter initialization techniques, can mitigate the need for sophisticated pre-training methods**

OVERFITTING

8

Overfitting

We want to builds models that performs well on unknown data, by only learning on known data !

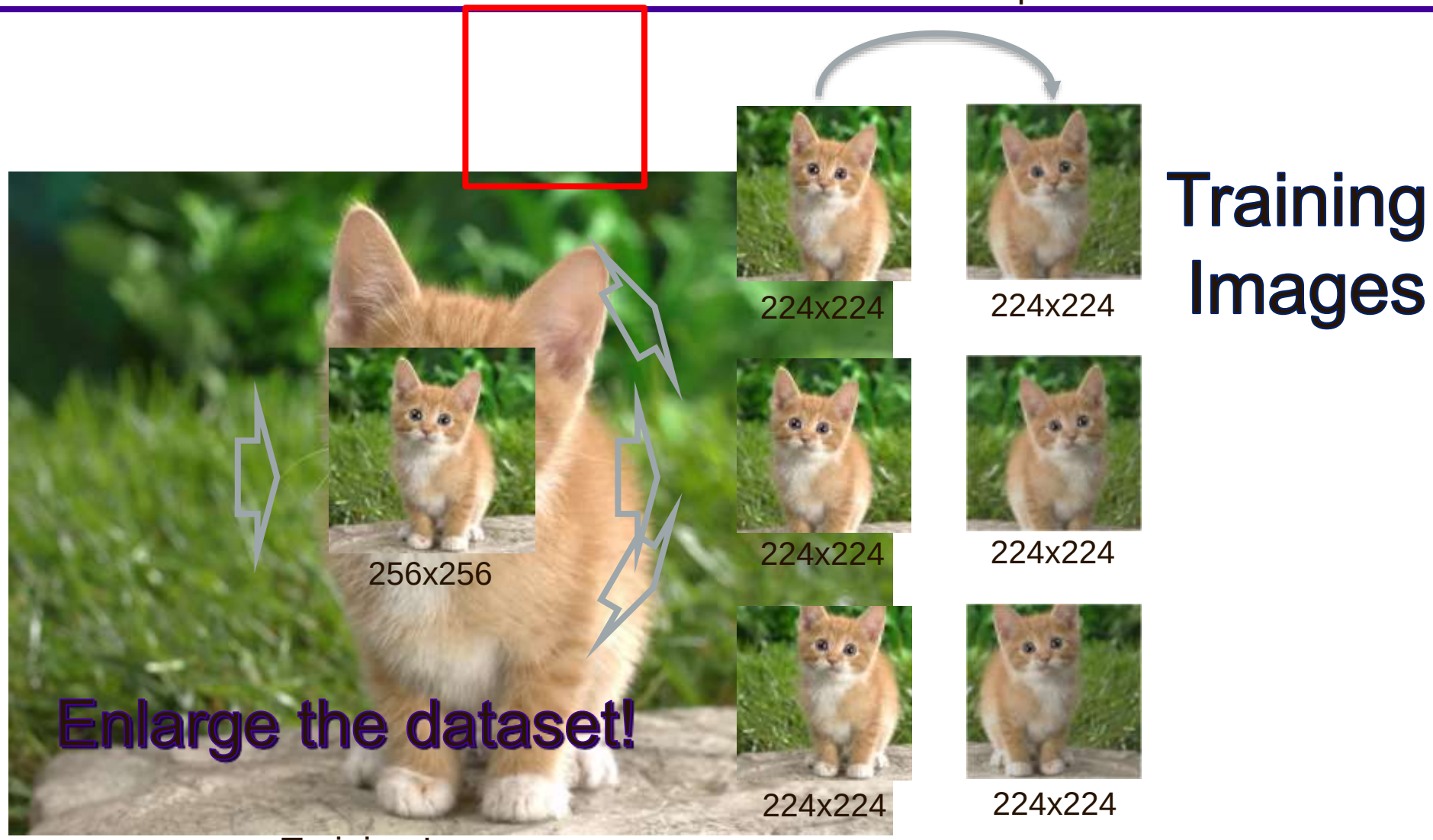
Different levels on overfitting exists :

- **Overfitting on the learning data.**
  Technical solutions exist to prevent this. (we will see this !)

- **Overfitting by experience.**
  Working too long on the same test data, may lead to tune meta-parameters and algorithmic choices.

- **Overfitting on a scenario.**
  Test data are drawn from the same distribution as the training data.
  Test performances are ok, but not on other use-cases

Data Augmentation



Dropout

Independently set each hidden unit activity to zero with α probability

- **Intention: reducing hidden unit co-adaptation & combat over-fitting**
- **Has been argued it corresponds to sampling from an exponential number of networks with shared parameters & missing connections**
- **One averages over models at test time by using original network without dropped-out connections, but with scaled-down weights**
- **If a unit is retained with probability *p* during training, its outgoing weights are rescaled or multiplied by a factor of *p* at test time**
- **By performing dropout a neural network with *n* units can be made to behave like an ensemble of 2^n smaller networks**
- **Often only used in the fully-connected layers at the net's output**

Regularization

- To avoid over-fitting, it is possible to regularize the cost function.
- Here we use L2 regularization, by changing the cost function to:

$$\widetilde{L}(\mathbf{w}) = L(\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^2$$

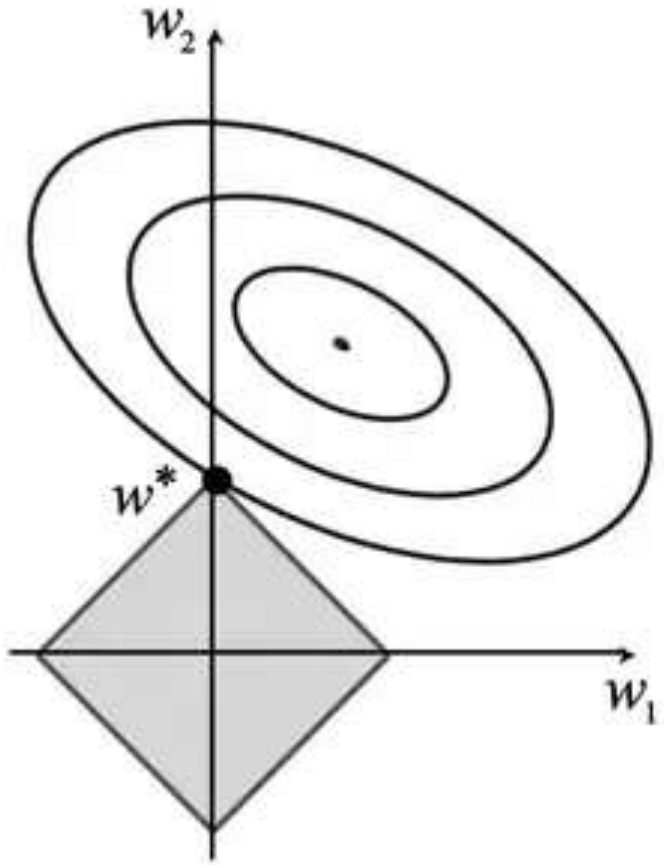
- In practice this penalizes large weights and effectively limits the freedom in the model.
- The regularization parameter λ determines how you trade off the original loss L with the large weights penalization.

- Applying gradient descent to this new cost function we obtain:

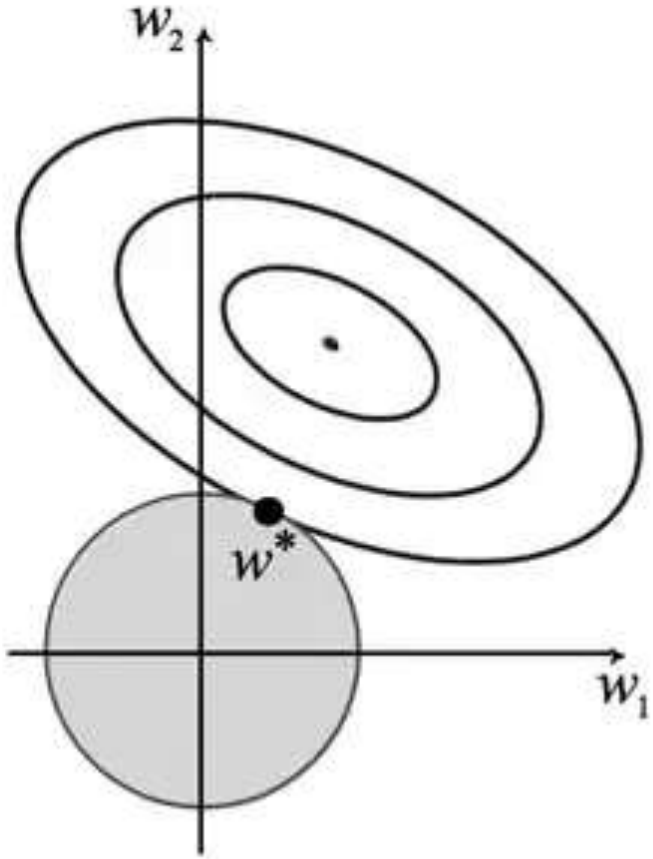
$$w_i \leftarrow w_i - \eta \frac{\partial L}{\partial w_i} - \eta \lambda w_i$$

- The new term $-\eta \lambda w_i$ coming from the regularization causes the weight to decay in proportion to its size.

Regularization



L1

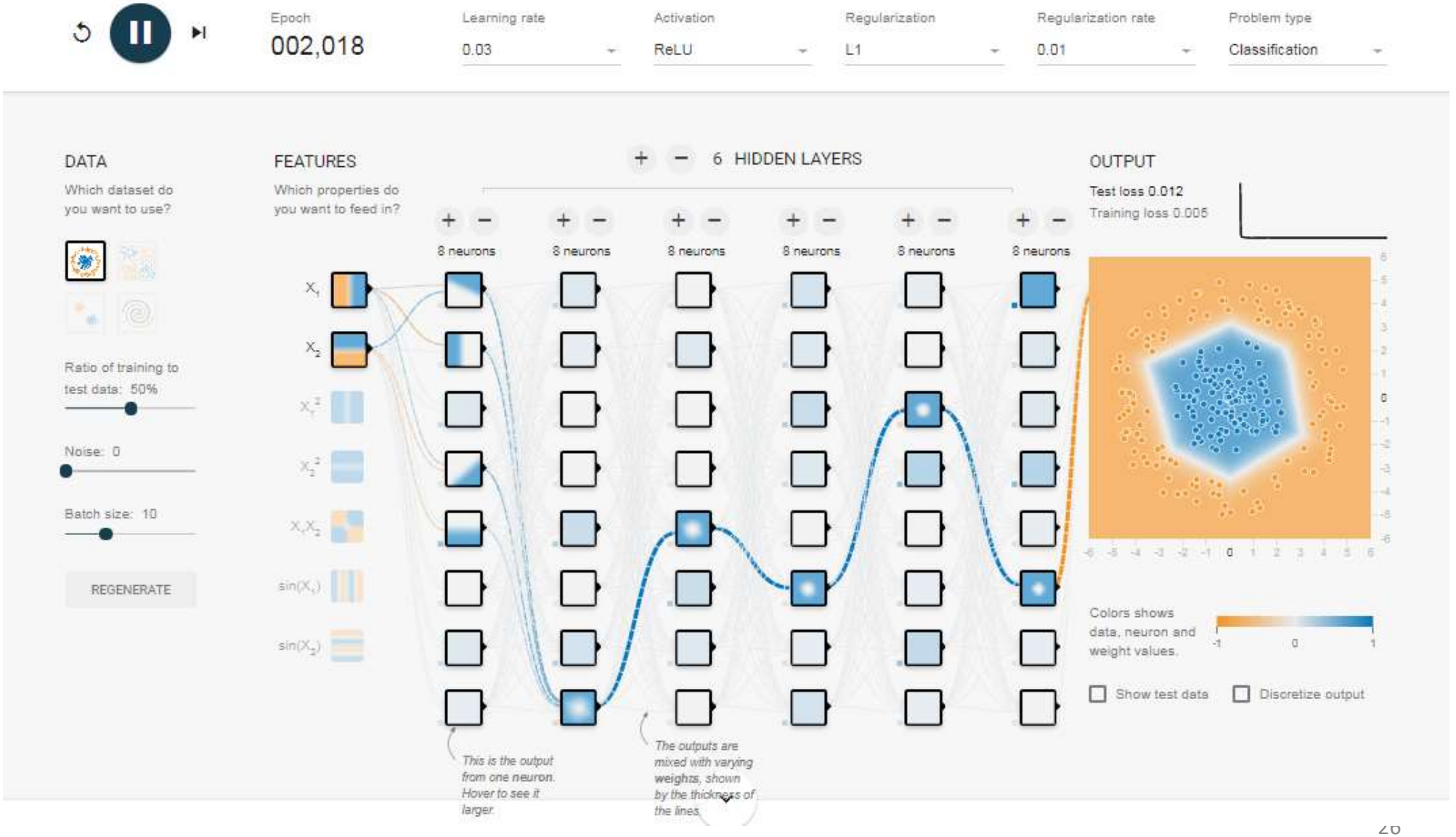


L2

Source-https://www.quora.com/What-are-the-main-regularization-methods-used-in-machine-learning

TensorFlow Playground

http://playground.tensorflow.org/



Early stopping

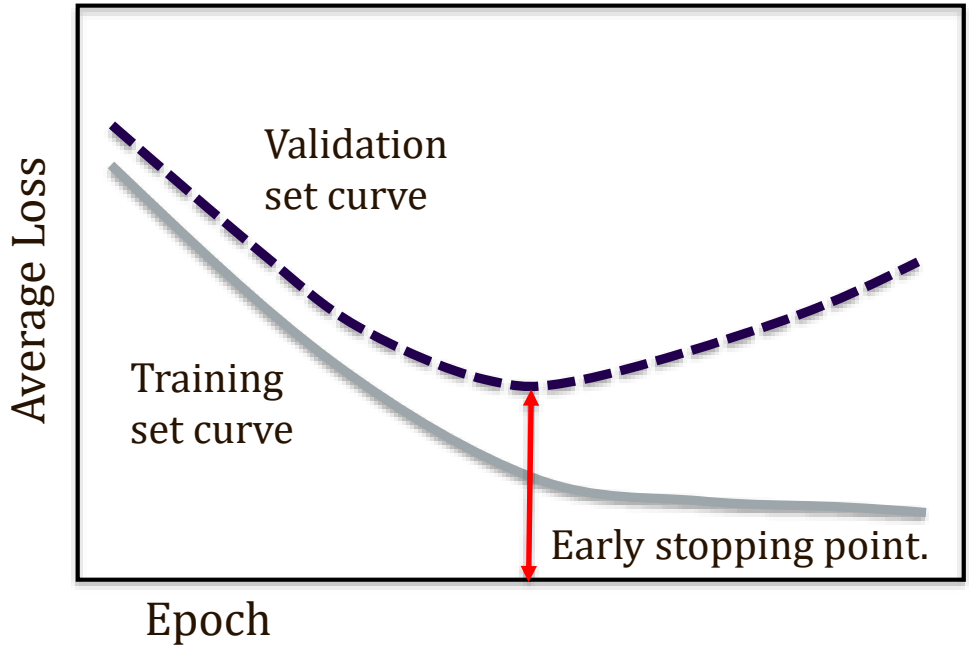
Deep learning involves high capacity architectures, which are susceptible to overfitting even when data is plentiful.

Early stopping is standard practice even when other methods to reduce overfitting are employed, ex. regularization and dropout.

The idea is to monitor learning curves that plot the average loss for the training and validation sets as a function of epoch.

The key is to find the point at which the validation set average loss begins to deteriorate

Early stopping



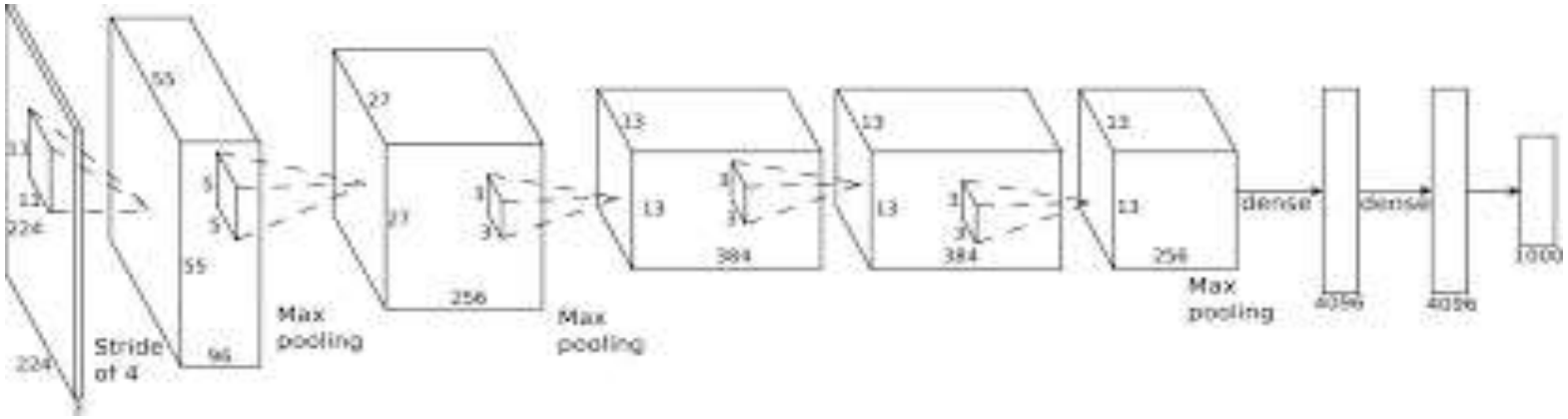
- In practice the curves above can be more noisy due to the use of stochastic gradient descent
- As such, it is common to keep the history of the validation set curve when looking for the minimum – even if it goes back up it might come back down

MORE DNN ARCHITECTURES

9

Batch Normalization

- Batch normalization comes from Sergey Ioffe and Christian Szegedy [ICML2015]



- We apply whitening to the input, and choose our initial weights in such a way that the activations in between are close to whitened

  It would be nice to also ensure whitened activations during training for all layers.

- Training of networks is complicated because the distribution of layer inputs changes during training *(internal covariate shift)*

  Making normalization at all layers part of the training prevents the internal covariate shift.

Batch Normalization

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

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

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

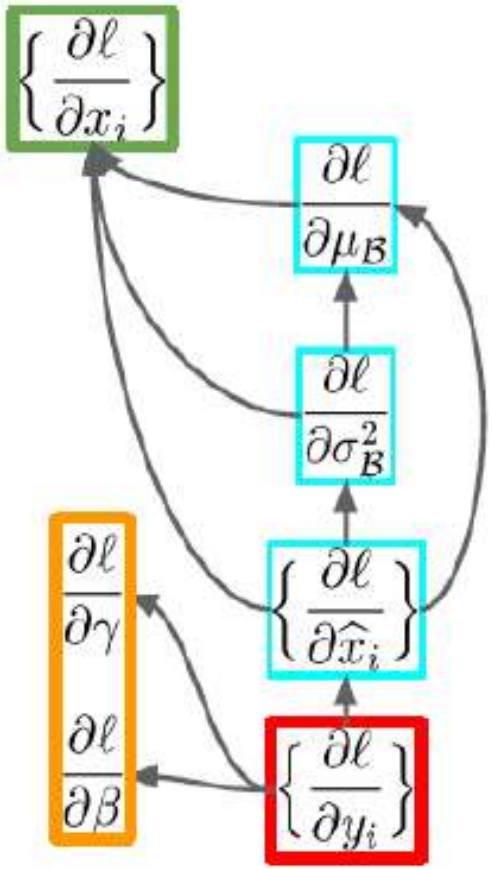
$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

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

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

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

Batch Normalization



(a)

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

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

$$\frac{\partial \ell}{\partial \mu_{\mathcal{B}}} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial \hat{x}_i} \cdot \frac{-1}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}}$$

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

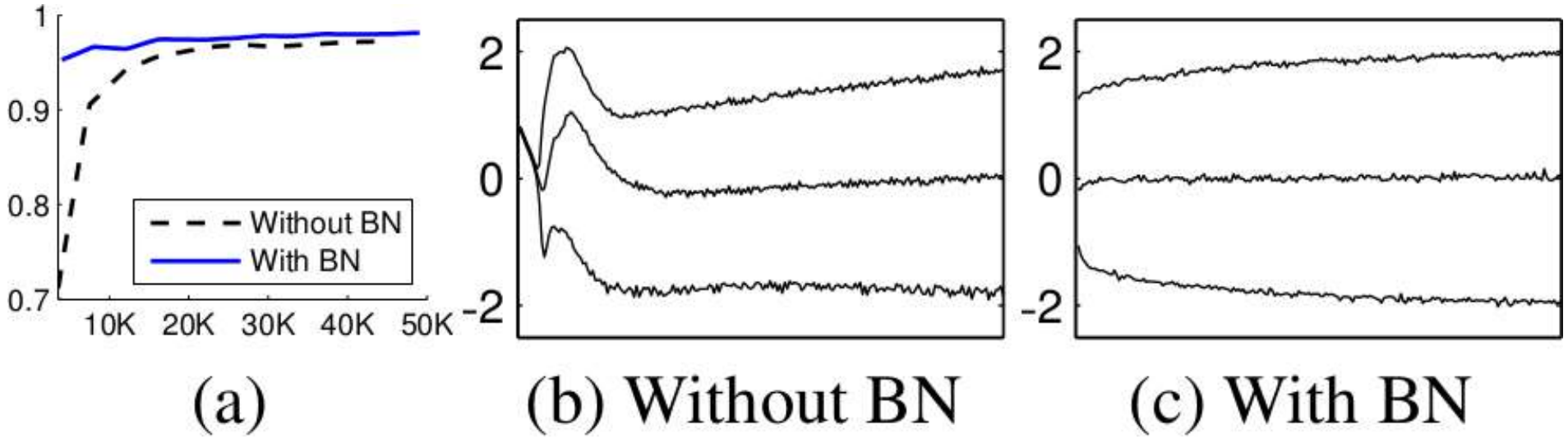
$$\frac{\partial \ell}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial y_i} \cdot \hat{x}_i$$

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

(b)

Backpropagation with Batch Normalization

Batch normalization: Better accuracy , faster.



BN applied to MNIST (a), and activations of a randomly selected neuron over time (b, c), where the middle line is the median activation, the top line is the 15th percentile and the bottom line is the 85th percentile.

Results on MNIST [Ioffe & Szegedy2015]

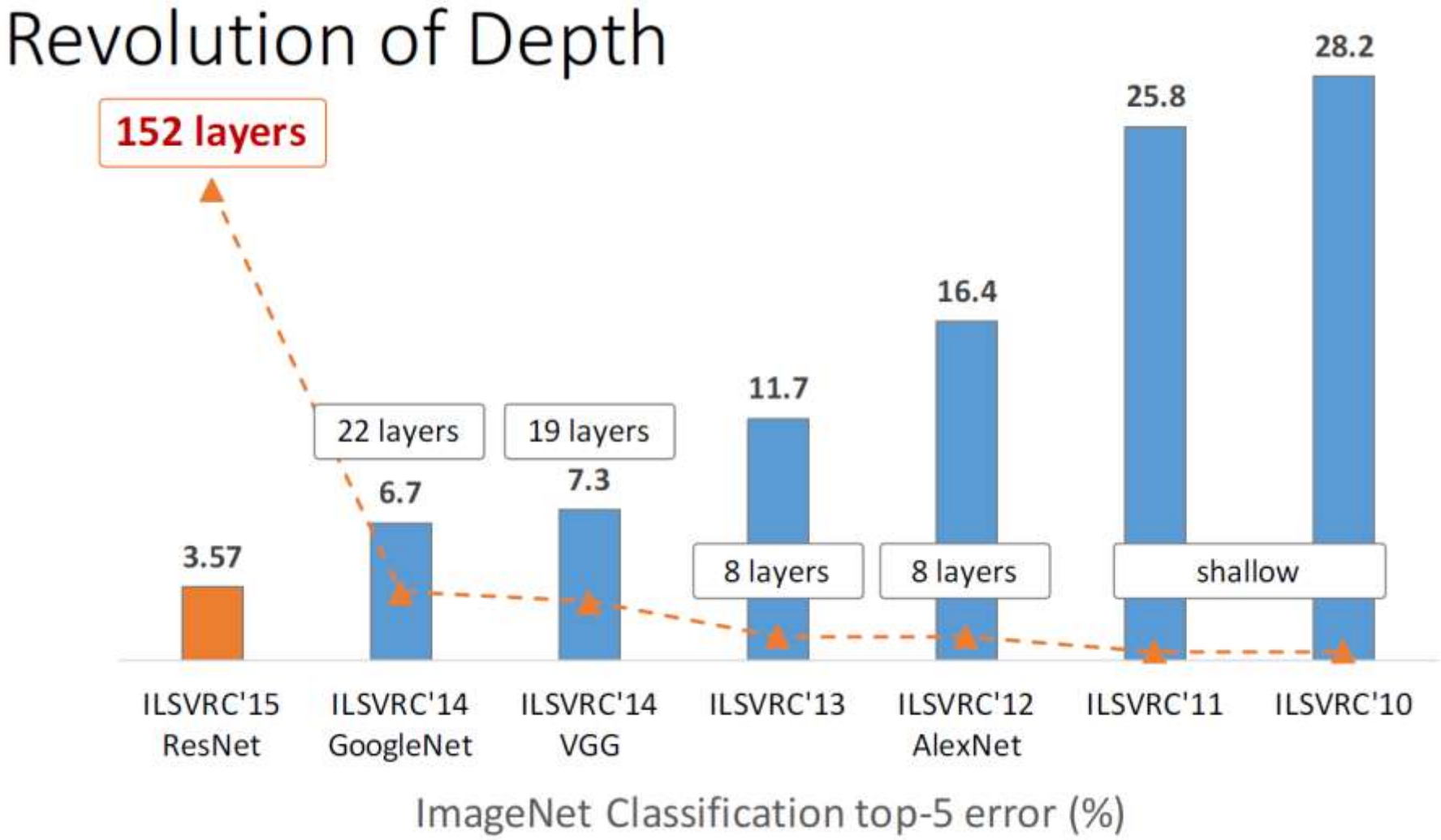
The deeper, the better

The deeper network can cover more complex problems

- Receptive field size ↑
- Non-linearity ↑

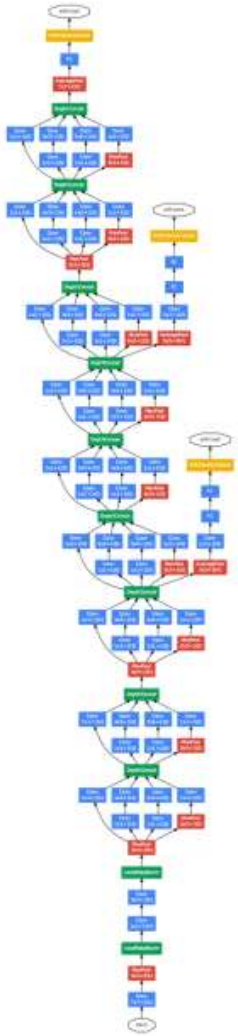
However, training the deeper network is more difficult because of vanishing/exploding gradients problem

Evolution of deep networks



Inception Network

[Szegedy et al. 2014]



"*Going deeper* with convolutions"

Inception network

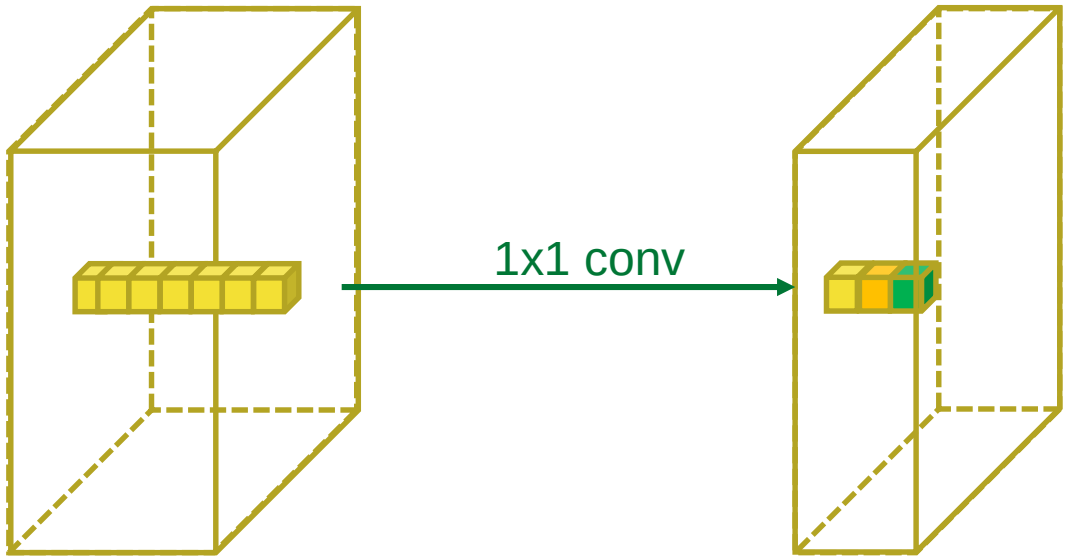
22 layers network
6.7% Top-5 error rate!
Winners of Imagenet 2014

Materials partly taken from
Stanford's *Convolutional Neural Networks for Visual Recognition course*

Inception V1 : GoogLeNet

[Szegedy et al. 2014]

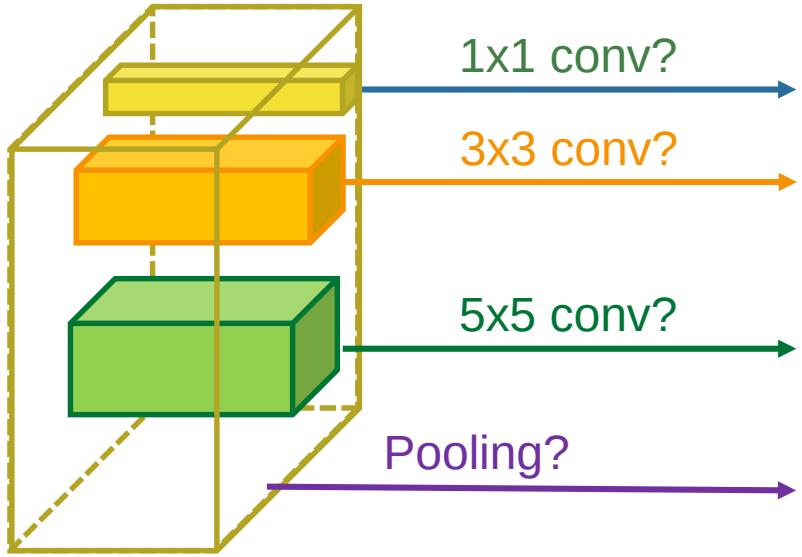
- Does it make any sense to do 1x1 convolutions?
- Can we do dimensionality reduction on the depth?



Inception V1 : GoogLeNet

[Szegedy et al. 2014]

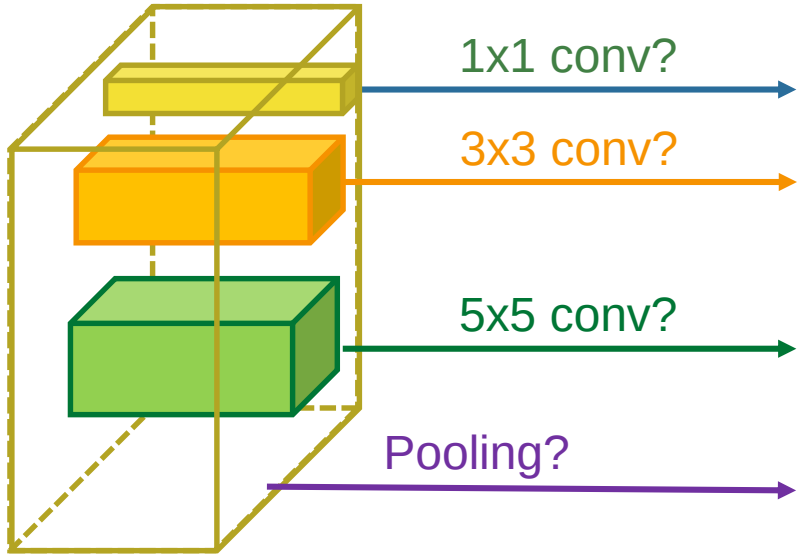
- Which one?



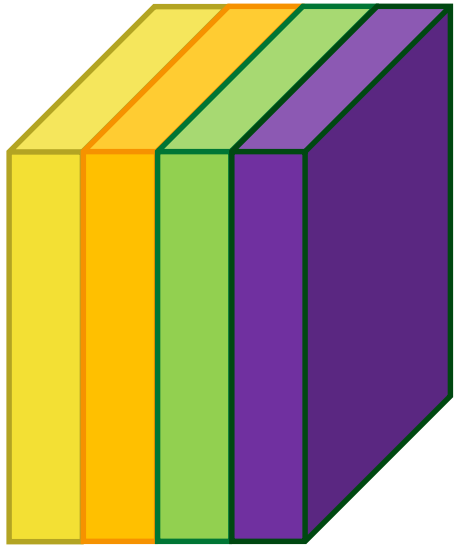
Inception V1 : GoogLeNet

[Szegedy et al. 2014]

- Which one?



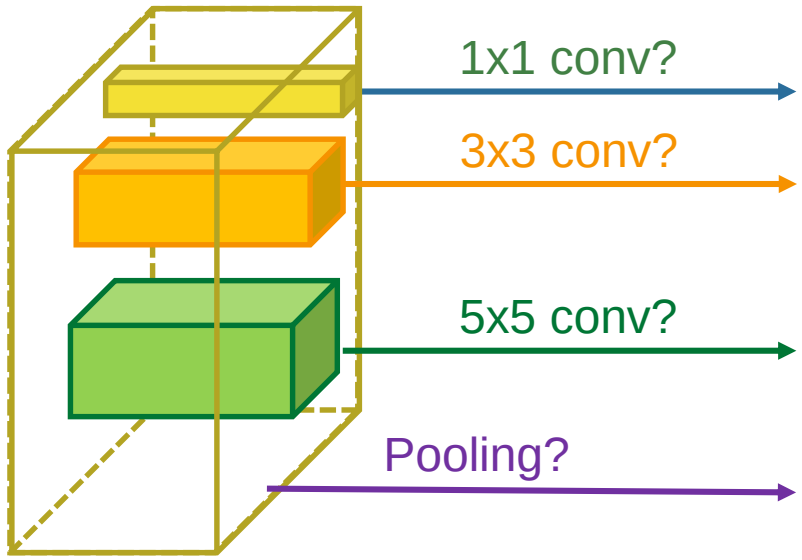
Pick them all!!



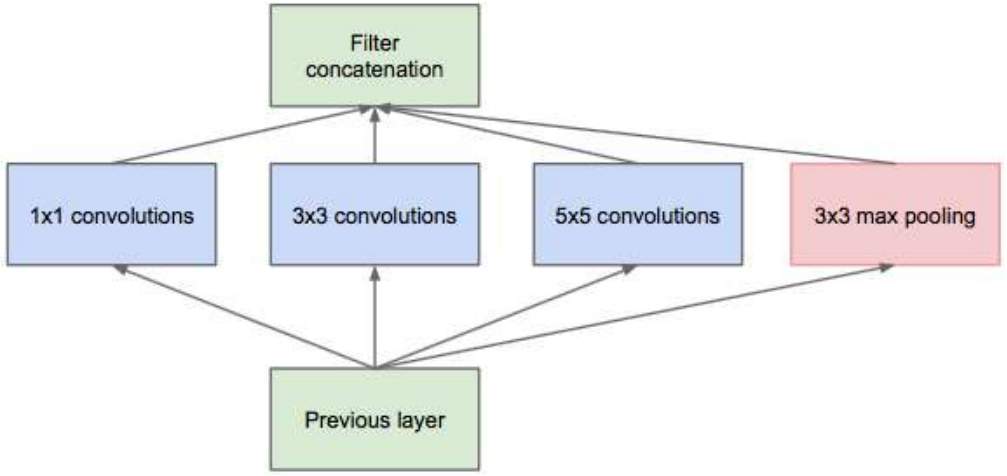
Inception V1 : GoogLeNet

[Szegedy et al. 2014]

- What to do?



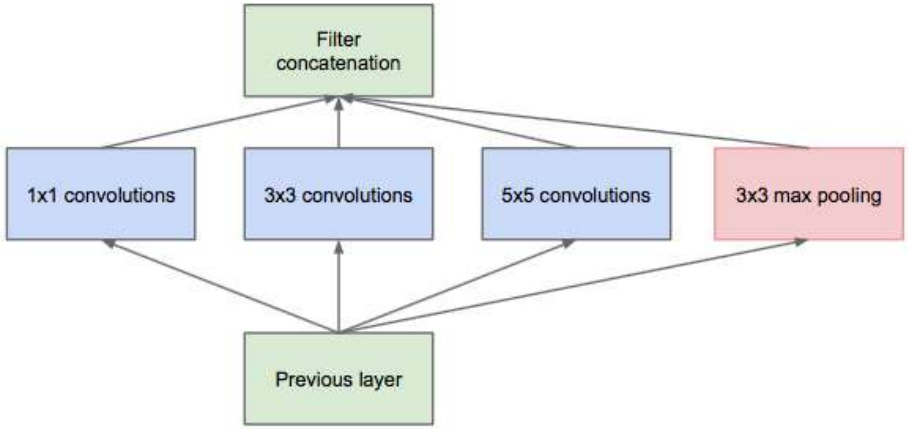
Pick them all!!



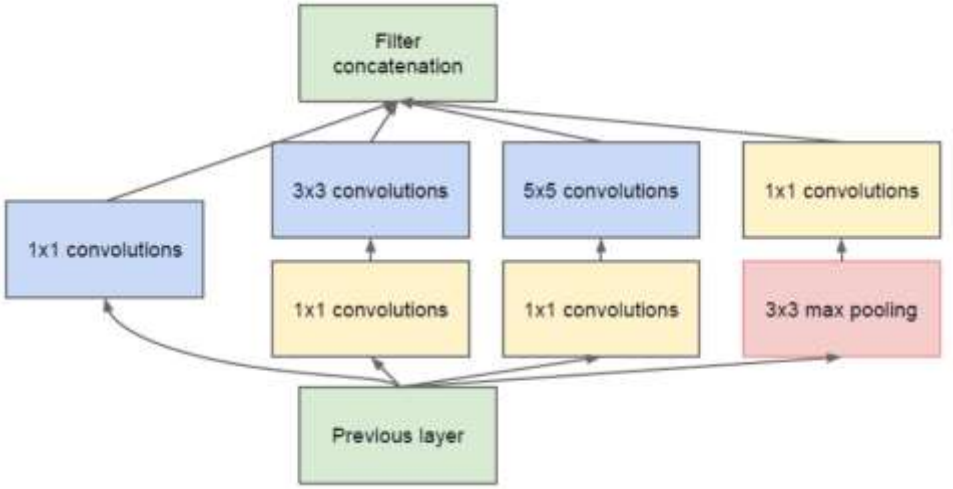
Inception module (naïve version)

Inception V1 : GoogLeNet

[Szegedy et al. 2014]



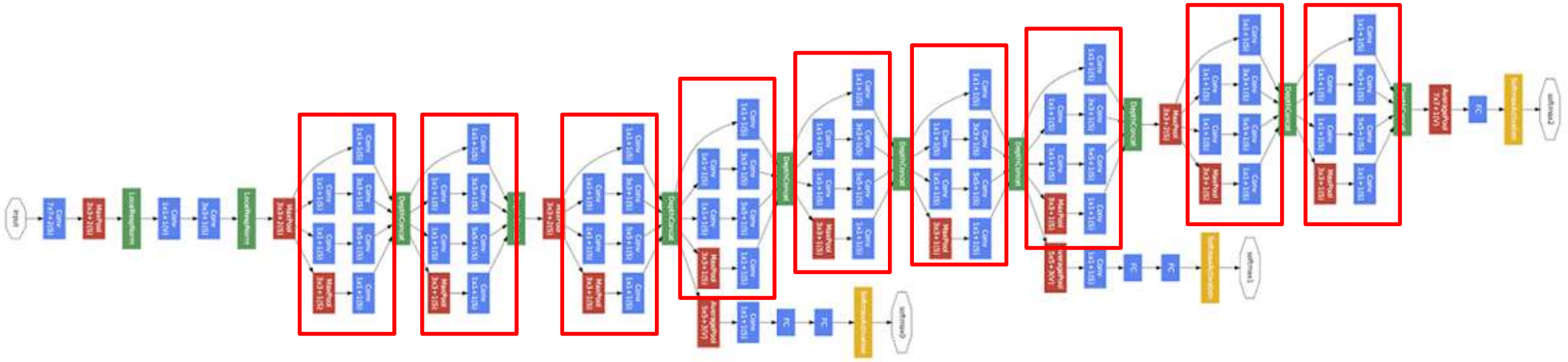
Inception module (naïve version)



Inception module with dimensionality reduction

Inception V1 : GoogLeNet

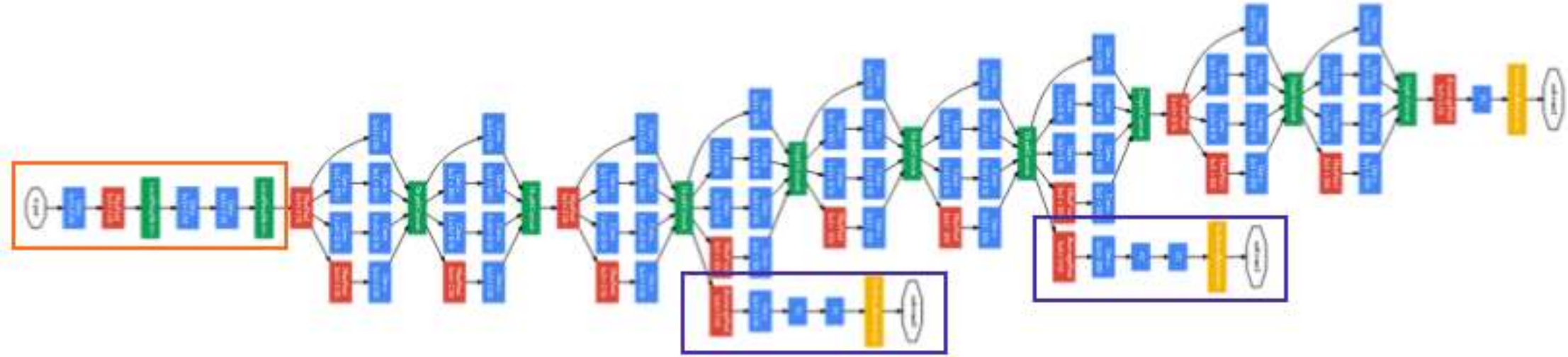
[Szegedy et al. 2014]



9 inception layers

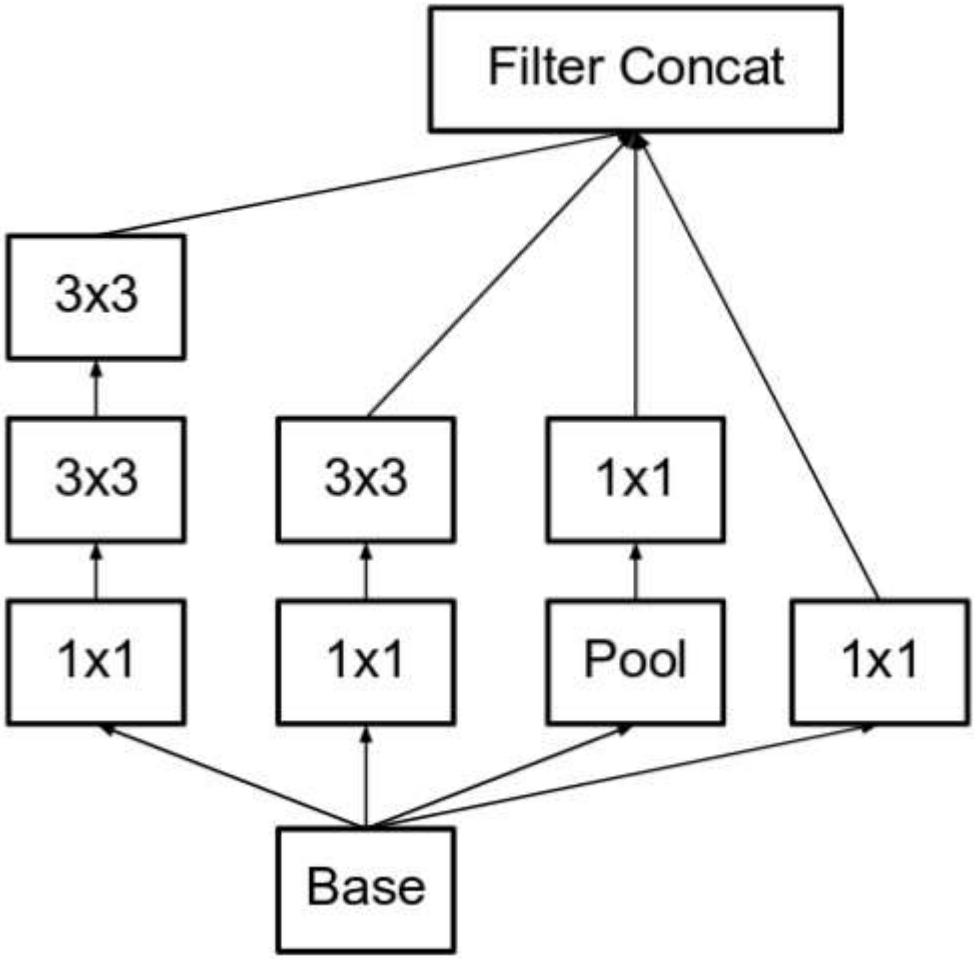
Inception V1 : GoogLeNet

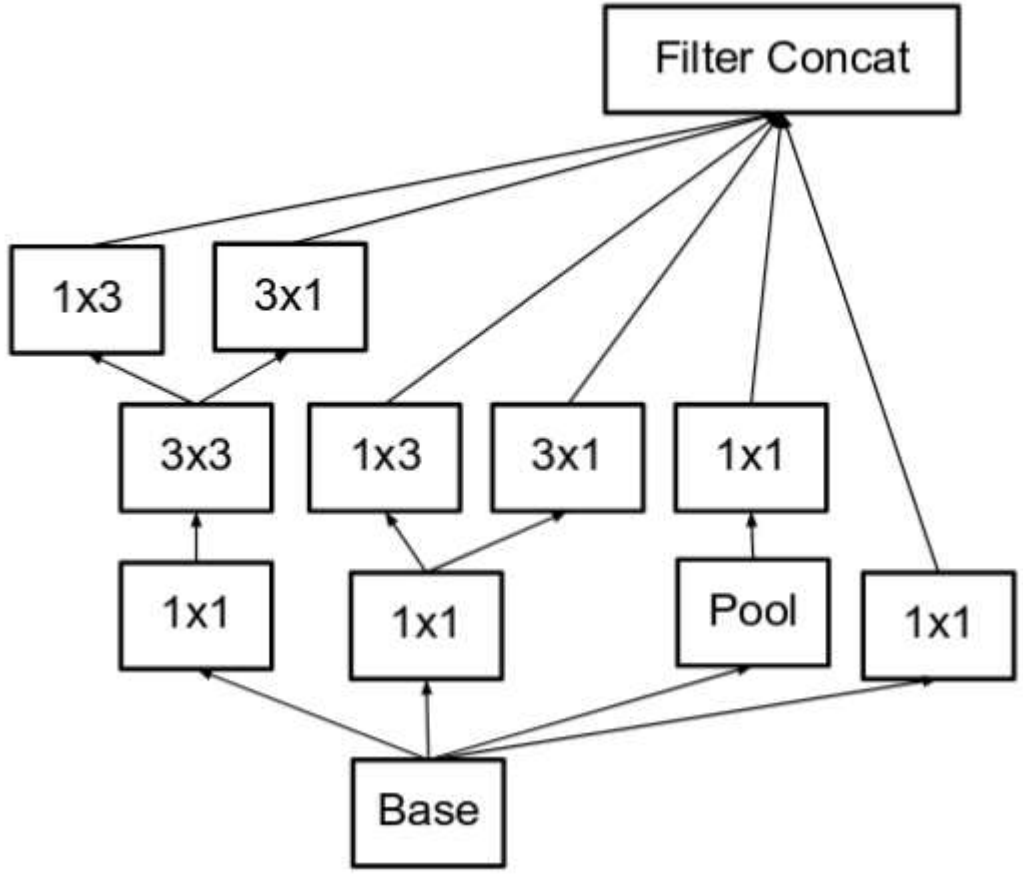
[Szegedy et al. 2014]



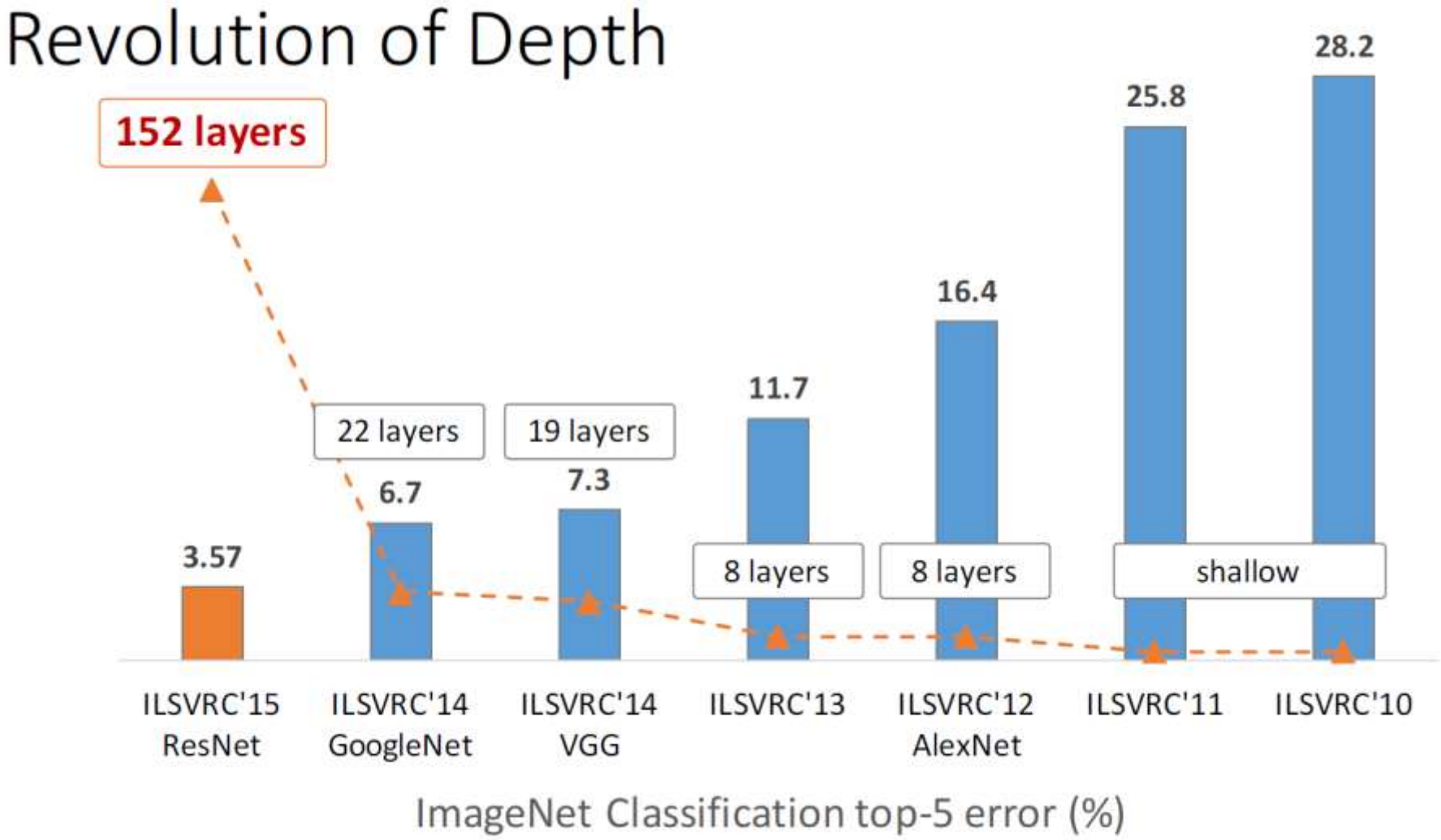
The **total loss function** is a **weighted sum** of the **auxiliary loss** and the **real loss**

Inception V2 & V3



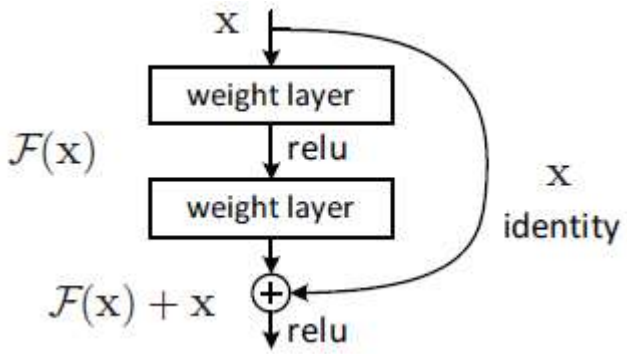


Evolution of deep networks



Residual Learning Block

Define H(x) = F(x) + x, the stacked weight layers try to approximate F(x) instead of H(x).



If the optimal function is close to identity mapping, the nonlinear stacked weight layers can capture the small perturbations easier.

① No extra parameter and computation complexity introduced.

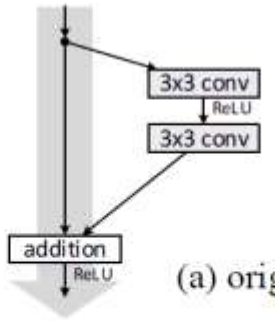
② Element-wise addition is performed on all feature maps.

Residual Learning Block

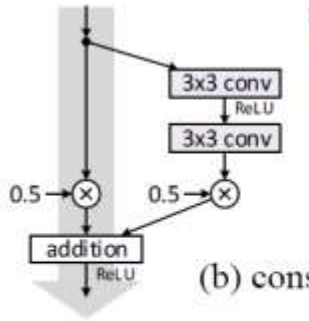
What if shortcut mapping h(x) ≠ identity?

* ResNet-110 on CIFAR-10

$h(x) = x$
error: 6.6%



(a) original

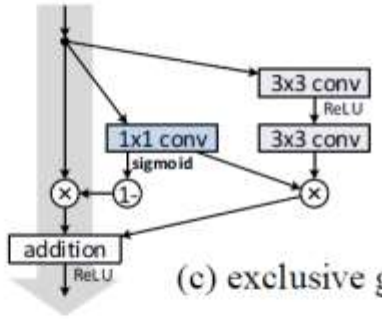


(b) constant scaling

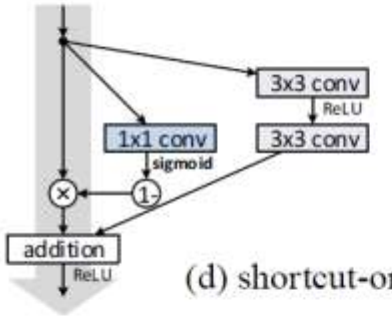
$h(x) = 0.5x$
error: 12.4%

$h(x) = \text{gate} \cdot x$
error: 8.7%

*similar to "Highway Network"



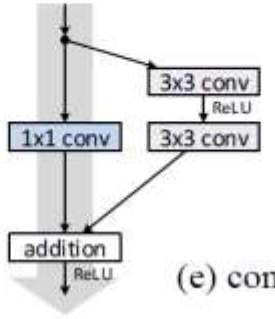
(c) exclusive gating



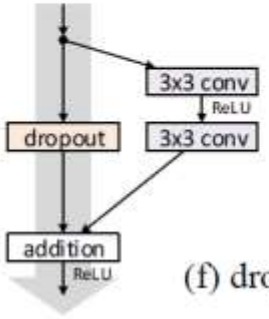
(d) shortcut-only gating

$h(x) = \text{gate} \cdot x$
error: 12.9%

$h(x) = \text{conv}(x)$
error: 12.2%



(e) conv shortcut

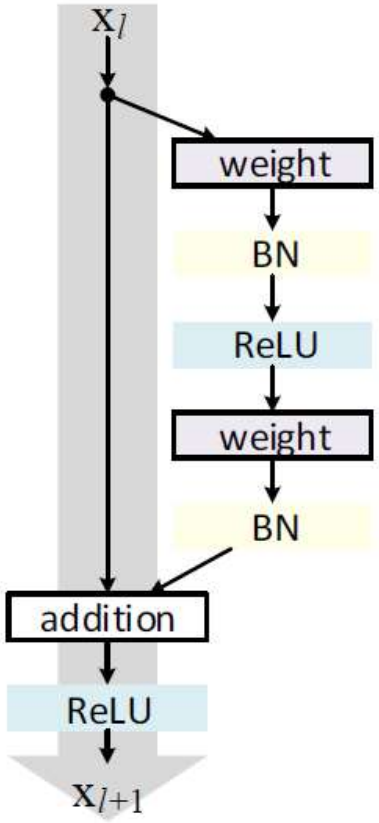


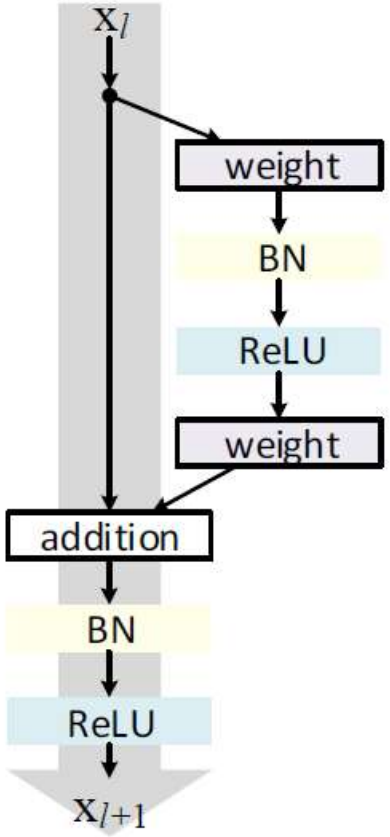
(f) dropout shortcut

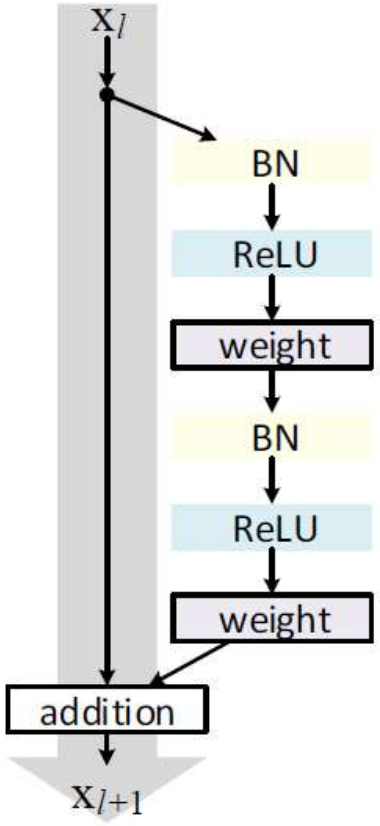
$h(x) = \text{dropout}(x)$
error: > 20%

If after-adding f(x) is identity mapping ?

Keep the shortest path as smooth(clean) as possible!







Deeper Neural Network

Escape from few layers

ReLU for solving gradient vanishing problem

Dropout …

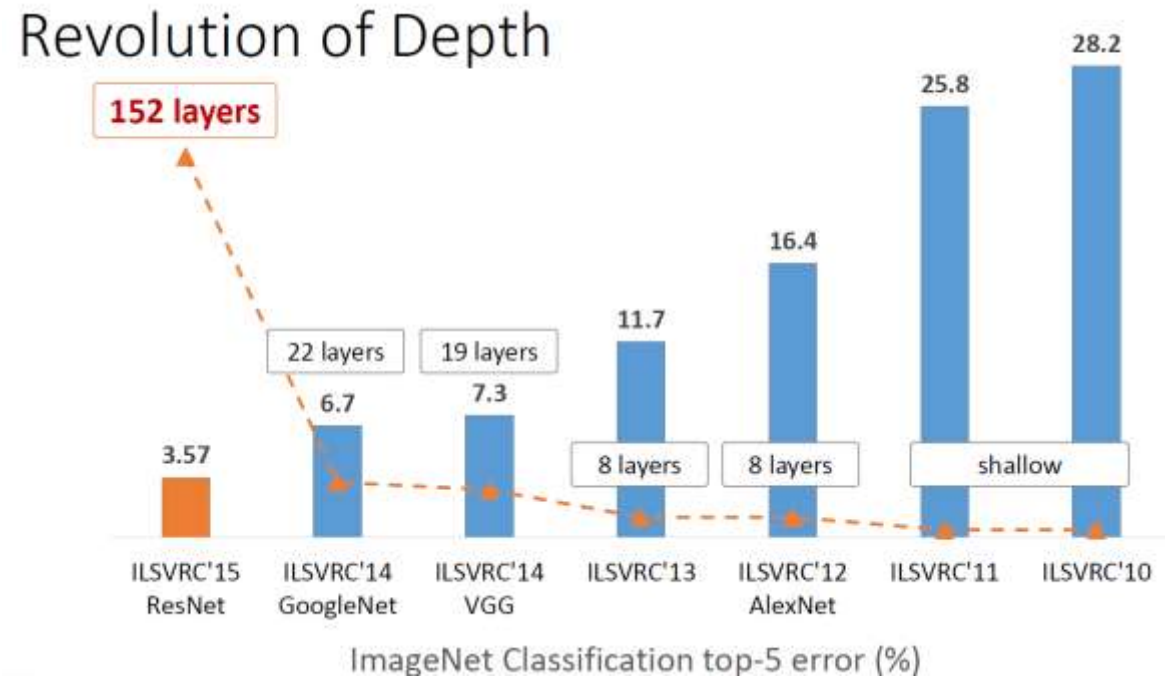
Escape from 10 layers

Normalized initialization

Intermediate normalization

Escape from 100 layers

Residual network layers



Residual networks with tensorflow

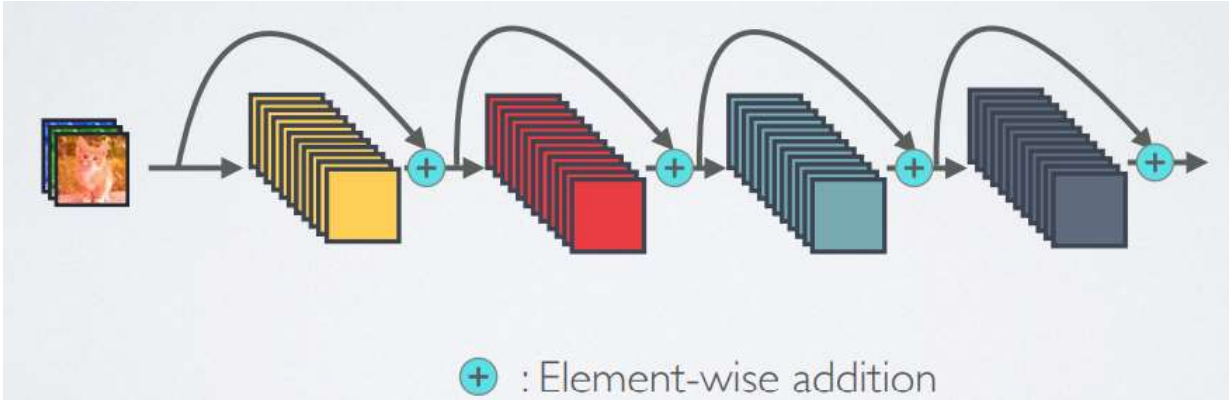
```
class Residual(tf.Module):
    def __init__(self, name, output_dim, filterSize, stride):
        self.lin = conv('lin', output_dim, 1, stride)
        self.cv1 = conv('conv_1', output_dim, filterSize, stride)
        self.cv2 = conv('conv_2', output_dim, filterSize, stride)

    def __call__(self, x, log_summary):
        y = self.lin(x, log_summary)
        x = self.cv1(x, log_summary)
        x = self.cv2(x, log_summary)
        return x+y
```

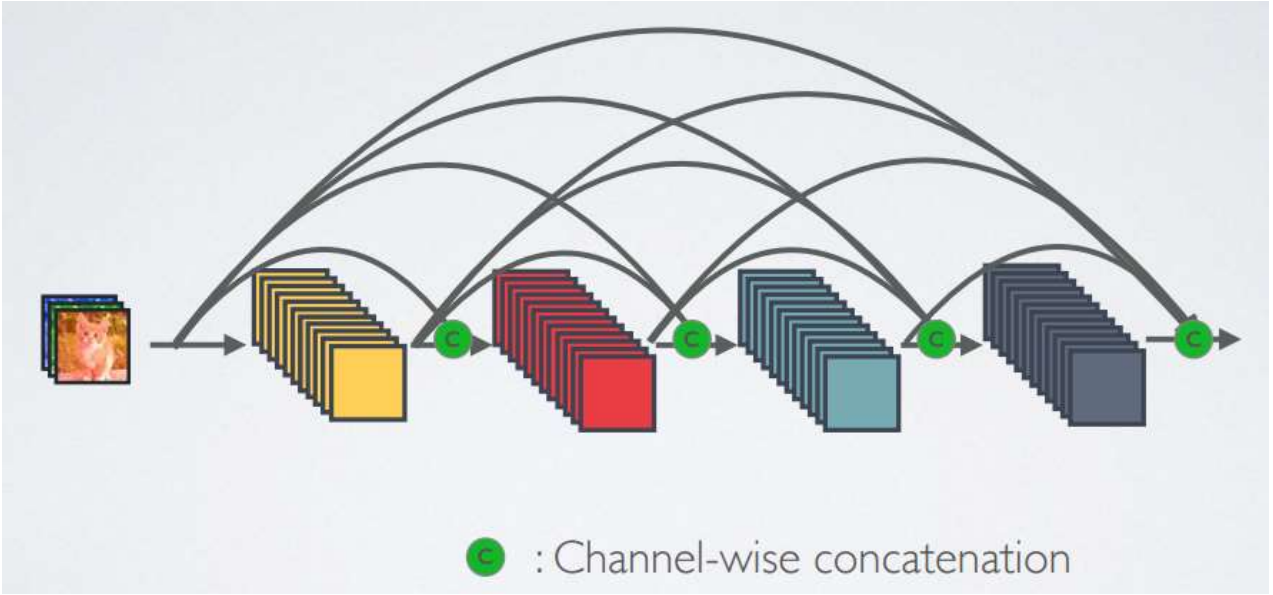
Solution for input_dim ≠ output_dim
Convolution with a 1x1 filter

DenseNet

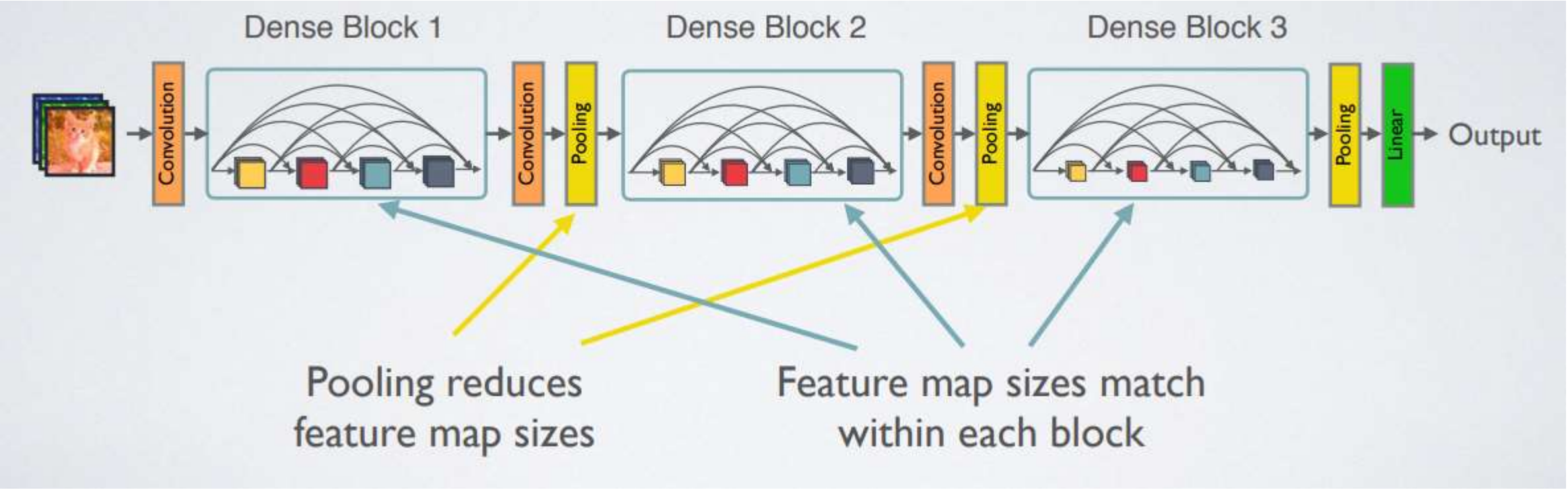
ResNet connectivity



DenseNet connectivity

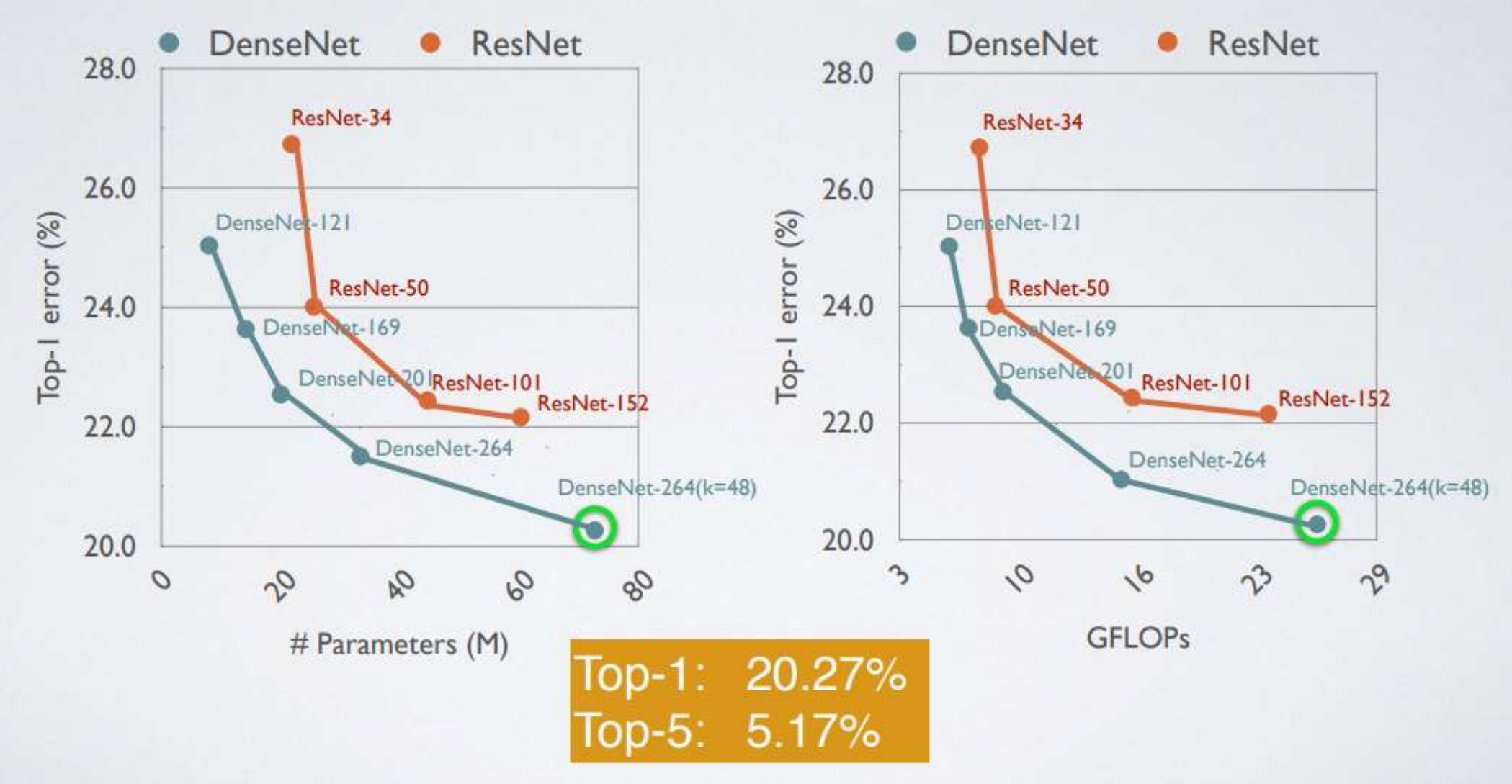


DenseNet



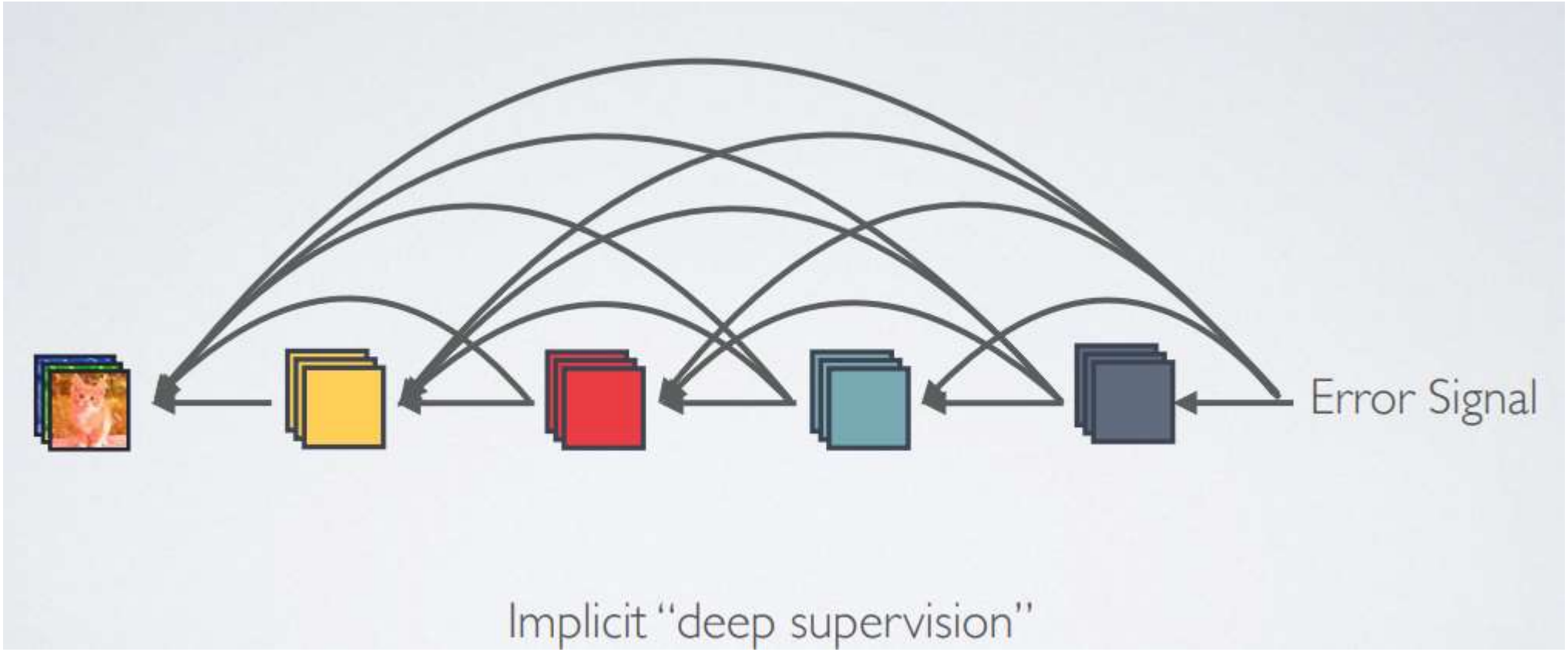
Deep residual learning for image recognition: [He, Zhang, Ren, Sun] (CVPR 2015)

DenseNet : Results on Imagenet



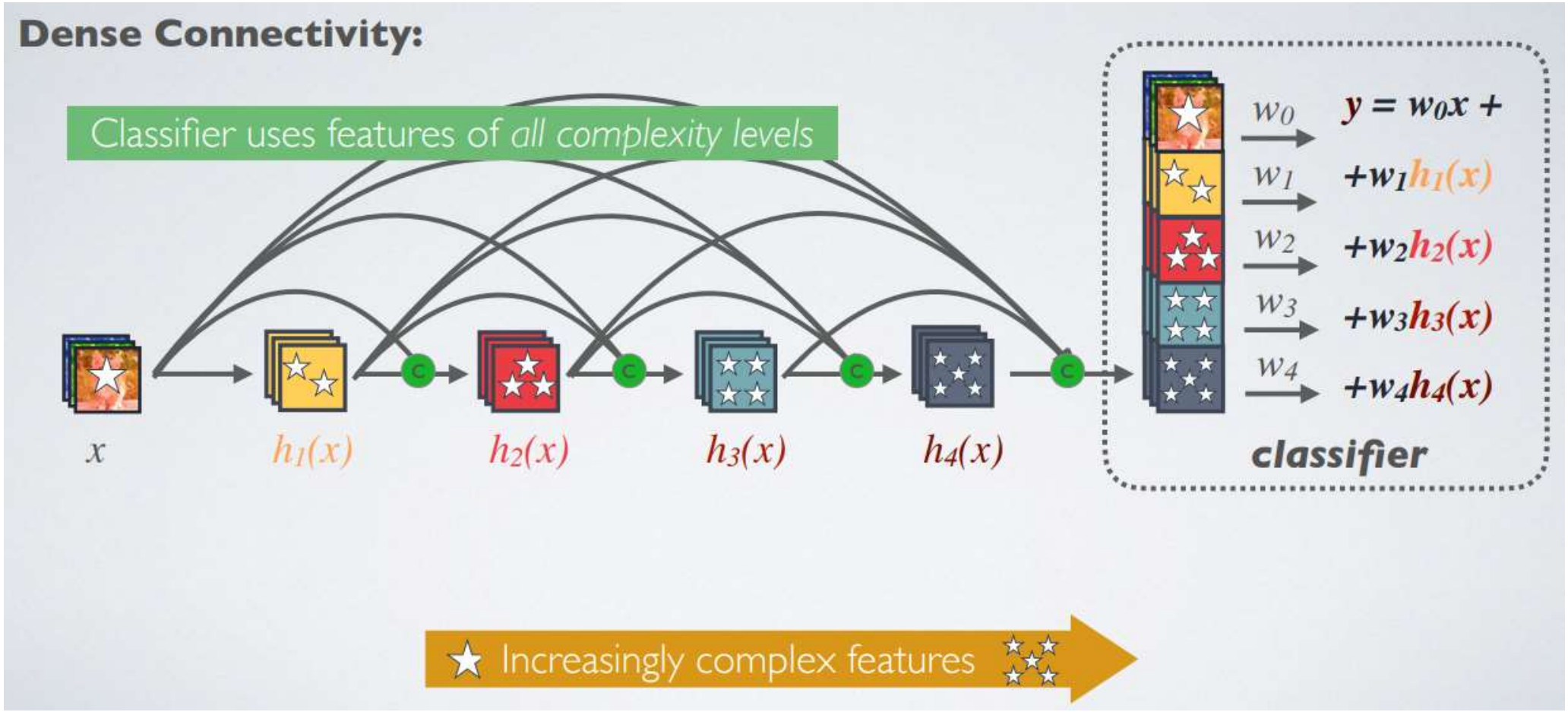
DenseNet

Avantage 1 : Strong gradient flow



DenseNet

Avantage : Maintains Low Complexity features



Highway Networks

Deeper and Deeper

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

H : a standard layer. Learned how to modify x, with weight W_H

- **The transform gate operation (T). T ϵ [0,1]**
- **The carry gate operation (C or just 1 - T).**

What happens is that when the transform gate is 1, we pass through our activation (H) and suppress the carry gate (since it will be 0). When the carry gate is 1, we pass through the unmodified input (x), while the activation is suppressed.

Deep Learning

Mini Projet

IDEMIA

Mini Project

1/ learn your best gender CNN detector.

Accuracy on test db > 85%

ckpt.save("./my_model")

2/ find on the web, the face image of a person

- Not androgynous
- Build a 48*48 face image

IrfanView

3/ write a small stand alone script.

- Load your model
- Load a face image
- Estimate gender

ckpt.restore("./ my_model-1")

Mini Project

4/ Modify your image in order to mislead your CNN detector.

- Load your model
- Load your image and a wrong label
- Add a Variable DX to the input in your graph
  $X_{mod} = X + DX$
- Train only the DX Variable
- Show Modified Image and DX Image
- Save the Modified Image. X_{mod}
- As DX is very small, build an amplified image, A=abs(DX)*50
- save the A Image.

5/ Test the Modified Image with your small script.

6/ Add a L2 constraint to DX to make it small

```
y = model(X+DX,False)

grads = tape.gradient(loss, [DX])

Import Pillow
Image.show()
```

Provide 3 images (raw or png)
adv_nom1_nom2_ori.raw
adv_nom1_nom2_mod.raw
adv_nom1_nom2_amp.raw
Send them to
stephane.gentric@telecom-paris.fr