An Introduction to Optimization and Regularization Methods in Deep Learning

Yuan YAO

HKUST

Acknowledgement

- Feifei Li, Stanford cs231n
- Ruder, Sebastian (2016). An overview of gradient descent optimization algorithms. arXiv:1609.04747.
  - http://ruder.io/deep-learning-optimization-2017/

Image Classification

Example Dataset: **CIFAR10**

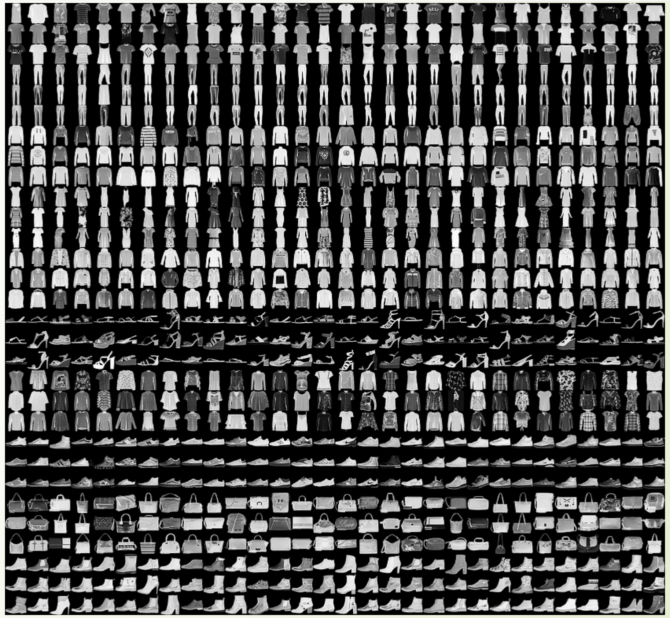
10 classes
50,000 training images
10,000 testing images

Alex Krizhevsky, "Learning Multiple Layers of Features from Tiny Images", Technical Report, 2009.

Example Dataset: **Fashion MNIST**

28x28 grayscale images
60,000 training and 10,000 test examples
10 classes

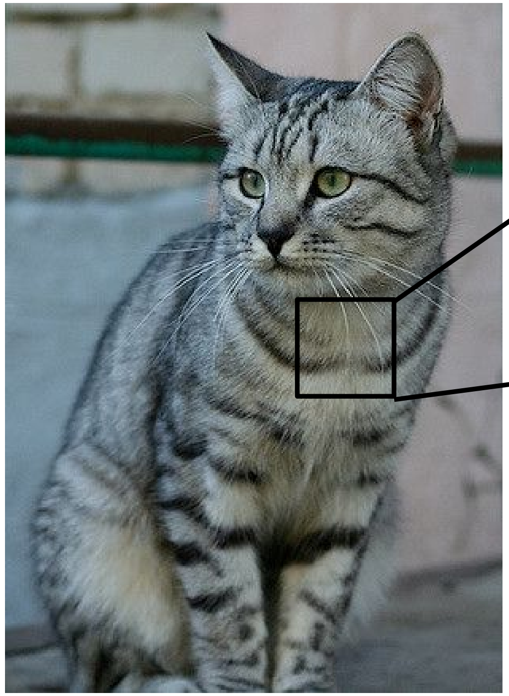
index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Type	T-shirt/top	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle boot



Jason WU, Peng XU, and Nayeon LEE

The Challenge of Human-Instructing-Computers

The Problem: Semantic Gap



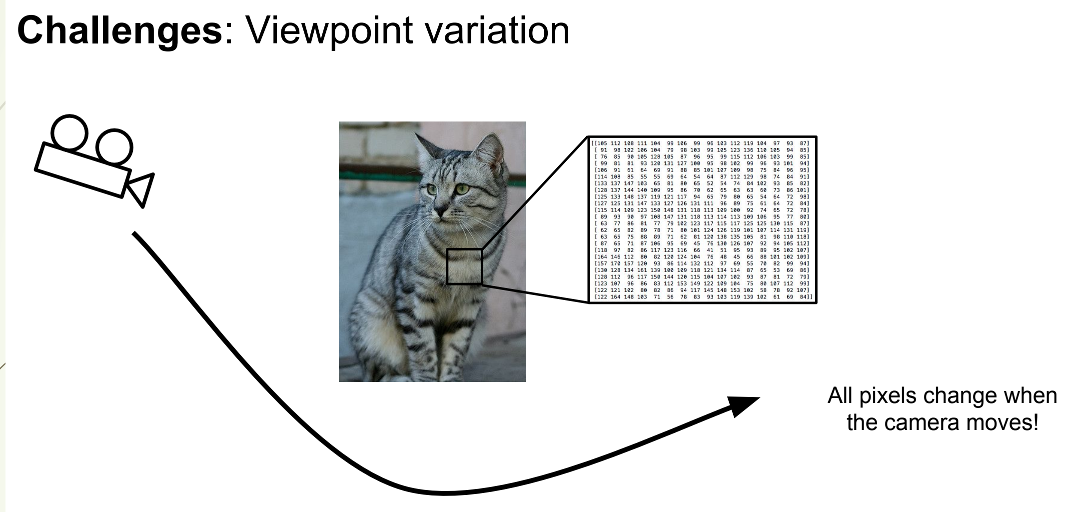

```
[[105 112 108 111 104  99 106  99  96 103 112 119 104  97  93  87]
 [ 91  98 102 106 104  79  98 103  99 105 123 136 110 105  94  85]
 [ 76  85  90 105 128 105  87  96  95  99 115 112 106 103  99  85]
 [ 99  81  81  93 120 131 127 100  95  98 102  99  96  93 101  94]
 [106  91  61  64  69  91  88  85 101 107 109  98  75  84  96  95]
 [114 108  85  55  55  69  64  54  64  87 112 129  98  74  84  91]
 [133 137 147 103  65  81  80  65  52  54  74  84 102  93  85  82]
 [128 137 144 140 109  95  86  70  62  65  63  63  60  73  86 101]
 [125 133 148 137 119 121 117  94  65  79  80  65  54  64  72  98]
 [127 125 131 147 133 127 126 131 111  96  89  75  61  64  72  84]
 [115 114 109 123 150 148 131 118 113 109 100  92  74  65  72  78]
 [ 89  93  90  97 108 147 131 118 113 114 113 109 106  95  77  80]
 [ 63  77  86  81  77  79 102 123 117 115 117 125 125 130 115  87]
 [ 62  65  82  89  78  71  80 101 124 126 119 101 107 114 131 119]
 [ 63  65  75  88  89  71  62  81 120 138 135 105  81  98 110 118]
 [ 87  65  71  87 106  95  69  45  76 130 126 107  92  94 105 112]
 [118  97  82  86 117 123 116  66  41  51  95  93  89  95 102 107]
 [164 146 112  80  82 120 124 104  76  48  45  66  88 101 102 109]
 [157 170 157 120  93  86 114 132 112  97  69  55  70  82  99  94]
 [130 128 134 161 139 100 109 118 121 134 114  87  65  53  69  86]
 [128 112  96 117 150 144 120 115 104 107 102  93  87  81  72  79]
 [123 107  96  86  83 112 153 149 122 109 104  75  80 107 112  99]
 [122 121 102  80  82  86  94 117 145 148 153 102  58  78  92 107]
 [122 164 148 103  71  56  78  83  93 103 119 139 102  61  69  84]]
```

What the computer sees

An image is just a big grid of numbers between [0, 255]:

e.g. 800 x 600 x 3
(3 channels RGB)

Complex Invariance



Euclidean transform

Large scale deformation

Complex Invariance

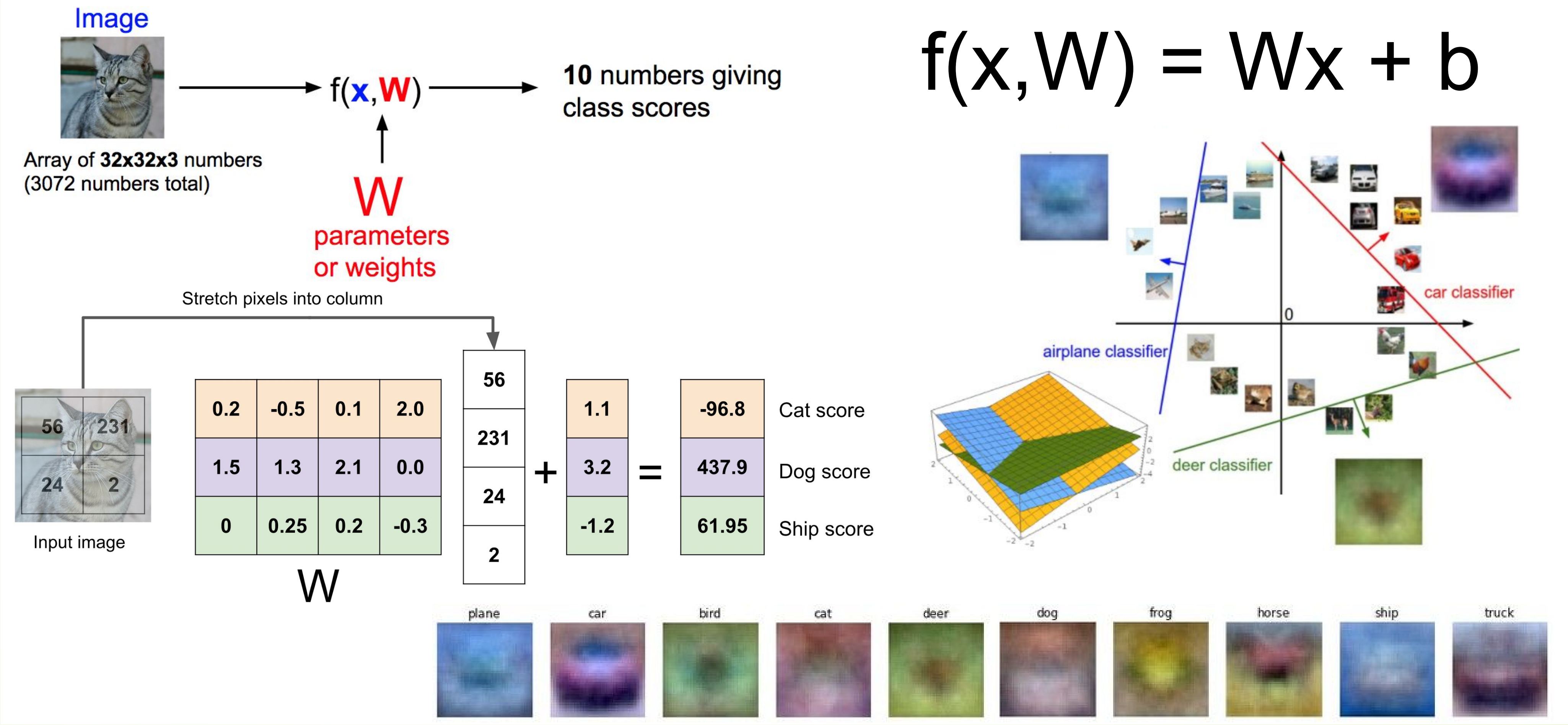
Challenges: Illumination

Challenges: Background Clutter

Challenges: Occlusion

Challenges: Intraclass variation

Data Driven Learning of the invariants: linear discriminant/classification



(Empirical) Loss or Risk Function

Suppose: 3 training examples, 3 classes.
With some W the scores $f(x,W) = Wx$ are:

cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**

A **loss function** tells how good our current classifier is

Given a dataset of examples

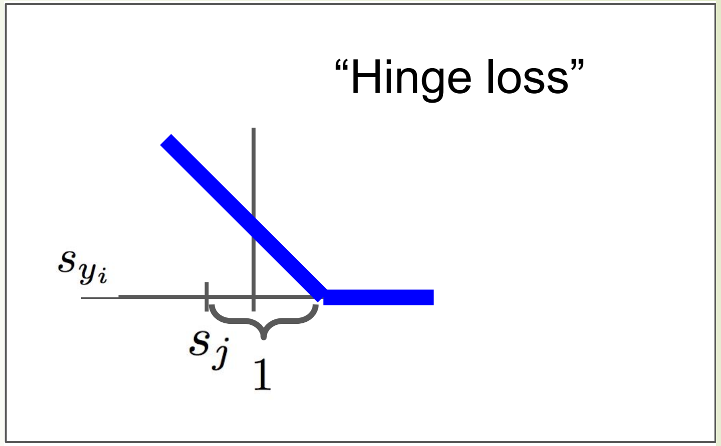
$$\{(x_i, y_i)\}_{i=1}^{N}$$

Where x_i is image and y_i is (integer) label

Loss over the dataset is a sum of loss over examples:

$$L = \frac{1}{N} \sum_i L_i(f(x_i, W), y_i)$$

Hing Loss



Suppose: 3 training examples, 3 classes.
With some W the scores $f(x, W) = Wx$ are:

cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
Losses:	2.9	0	12.9

Multiclass SVM loss:

Given an example (x_i, y_i)
where x_i is the image and
where y_i is the (integer) label,

and using the shorthand for the scores vector: $s = f(x_i, W)$

the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \begin{cases} 0 & \text{if } s_{y_i} \geq s_j + 1 \\ s_j - s_{y_i} + 1 & \text{otherwise} \end{cases}$$
$$= \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Cross Entropy (Negative Log-likelihood) Loss

Softmax Classifier (Multinomial Logistic Regression)

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

unnormalized probabilities

	unnormalized log probabilities		unnormalized probabilities		probabilities	
cat	**3.2**	exp →	**24.5**	normalize →	**0.13**	→ L_i = -log(0.13) = **0.89**
car	5.1		164.0		0.87	
frog	-1.7		0.18		0.00	

unnormalized log probabilities

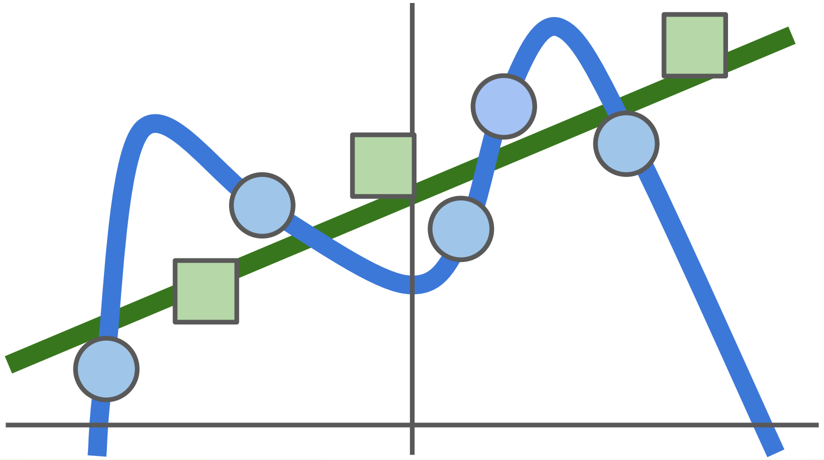
probabilities

Loss + Regularization

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)} + \underbrace{\lambda R(W)}$$

Data loss: Model predictions should match training data

Regularization: Model should be "simple", so it works on test data



Occam's Razor:
"Among competing hypotheses, the simplest is the best"
William of Ockham, 1285 - 1347

Regularizations

- Explicit regularization
  - L2-regularization
  - L1-regularization (Lasso)
  - Elastic-net (L1+L2)
  - Max-norm regularization
- Implicit regularization
  - Dropout
  - Batch-normalization
  - Earlystopping

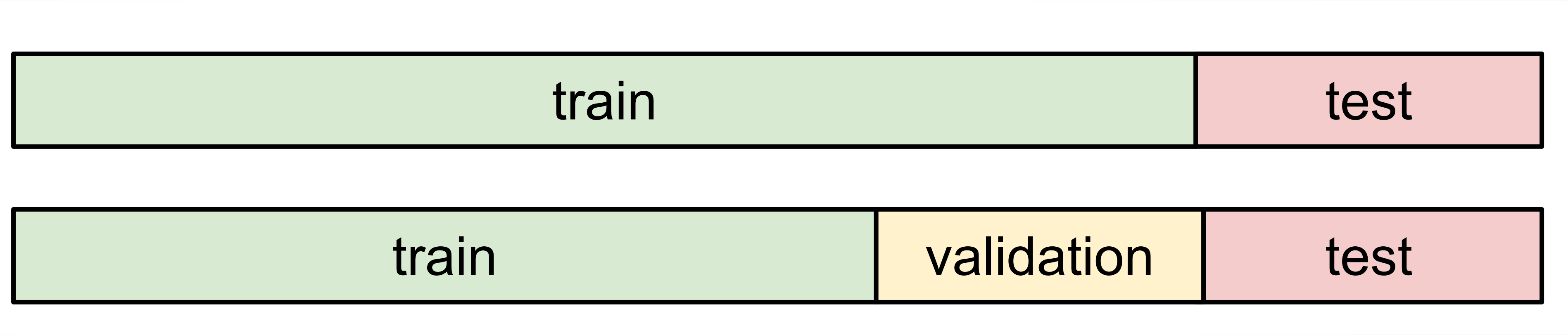
$$R(W) = \sum_k \sum_l W_{k,l}^2$$

$$R(W) = \sum_k \sum_l |W_{k,l}|$$

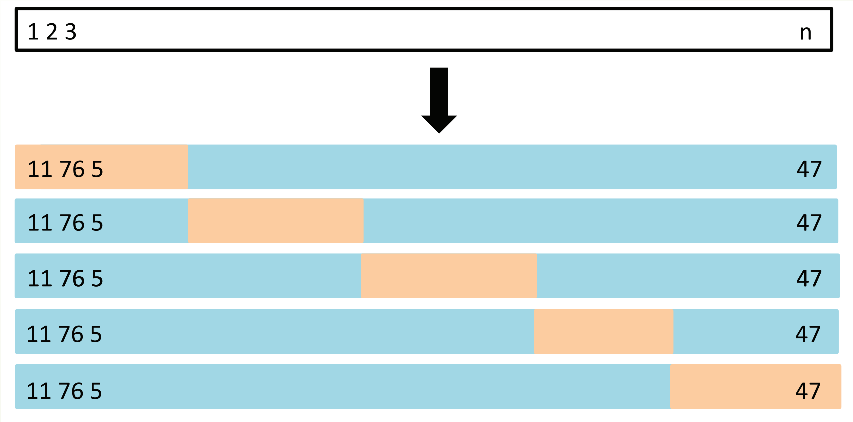
$$R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$$

Hyperparameter (Regularization) Tuning

Data rich:



Data poverty: cross-validation



Recap

How do we find the best W?

- We have some dataset of (x,y)
- We have a **score function:** $s = f(x; W) \overset{\text{e.g.}}{=} Wx$
- We have a **loss function**:

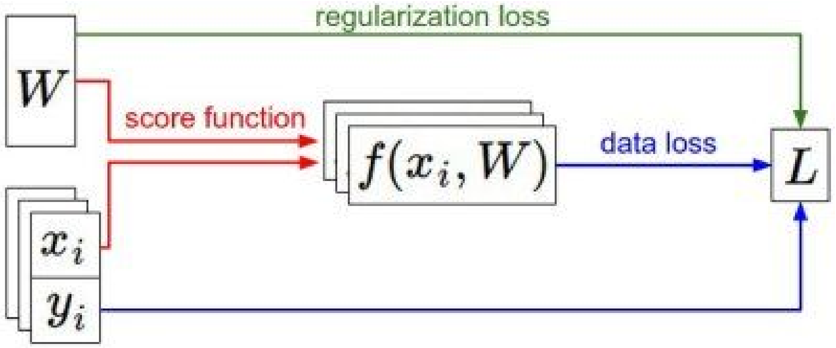
Softmax

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

SVM

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i + R(W)$$ Full loss



In regression, square loss is often used instead.

Optimization Methods to find minima of the Loss Landscape?

Gradient Descent Method

- Gradient descent is a way to minimize an objective function $J(\theta)$
  - $\theta \in \mathbb{R}^d$: model parameters
  - η: learning rate
  - $\nabla_\theta J(\theta)$: gradient of the objective function with regard to the parameters
- Updates parameters **in opposite direction** of gradient.
- Update equation: $\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$

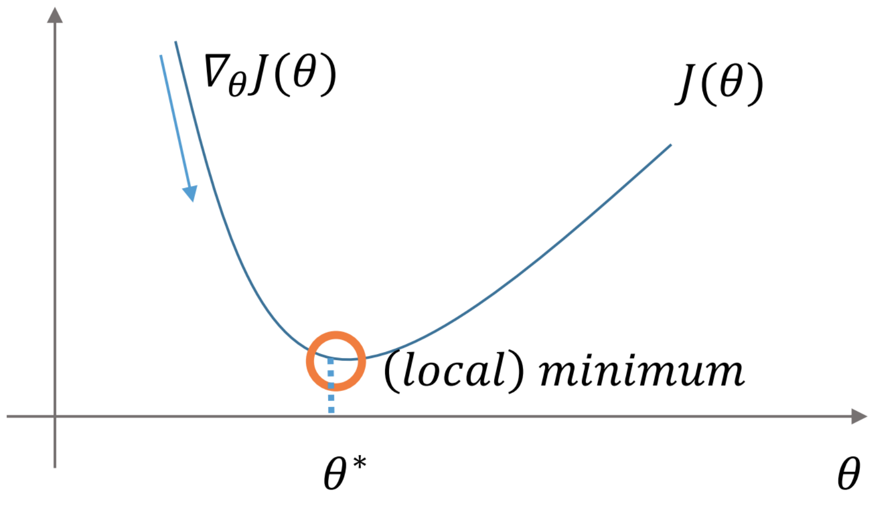


Figure: Optimization with gradient descent

Gradient Descent Variants

- Batch Gradient Descent
- Stochastic Gradient Descent
- Mini-batch Gradient Descent

- Difference: how much data we use in computing the *gradients*

Batch Gradient Descent

- Computes gradient with the **entire** dataset
- Update rule: $$\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$$

```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(
    loss_function, data, params)
  params = params - learning_rate * params_grad
```

Listing 1: Code for batch gradient descent update

- Pros:
  - Guaranteed to converge to **global** minimum for **convex** objective function and to a **stationary/critical** point for **non-convex** ones.
  - Exponentially fast (linear) convergence rates in **strongly convex** landscape
  - Sublinear convergence rates in **convex** landscape
- Cons:
  - Slow in big data.
  - Intractable for big datasets that do **not fit in memory**.
  - No **online** learning.

Stochastic Gradient Descent

- Computes update for each example $(x^{(i)}, y^{(i)})$, usually uniformly sampled from the training dataset
- Update equation:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

- The expectation of stochastic gradient is the batch gradient

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(
      loss_function, example, params)
    params = params - learning_rate * params_grad
```

Listing 2: Code for stochastic gradient descent update

- Pros:
  - Guaranteed to converge to **global** minimum for **convex** losses and to a local optima for **non-convex** ones, may **escape saddle** points polynomially fast
  - O(1/k) convergence rates in convex losses, possibly dimension-free
  - Much faster than batch in big data
  - Online learning algorithms
- Cons:
  - High variance in gradients and outcomes

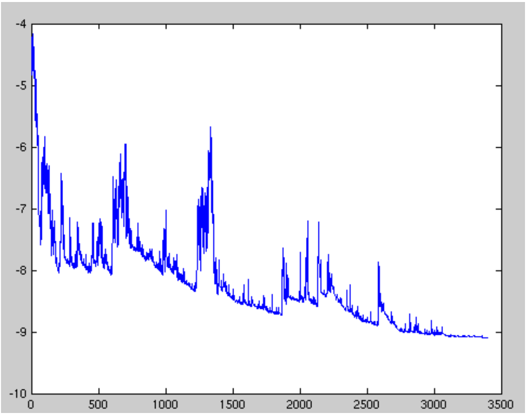


Figure: SGD fluctuation (Source: Wikipedia)

Batch GD vs. Stochastic GD

- SGD shows same convergence behaviour as batch gradient descent if learning rate is slowly decreased (annealed) over time.

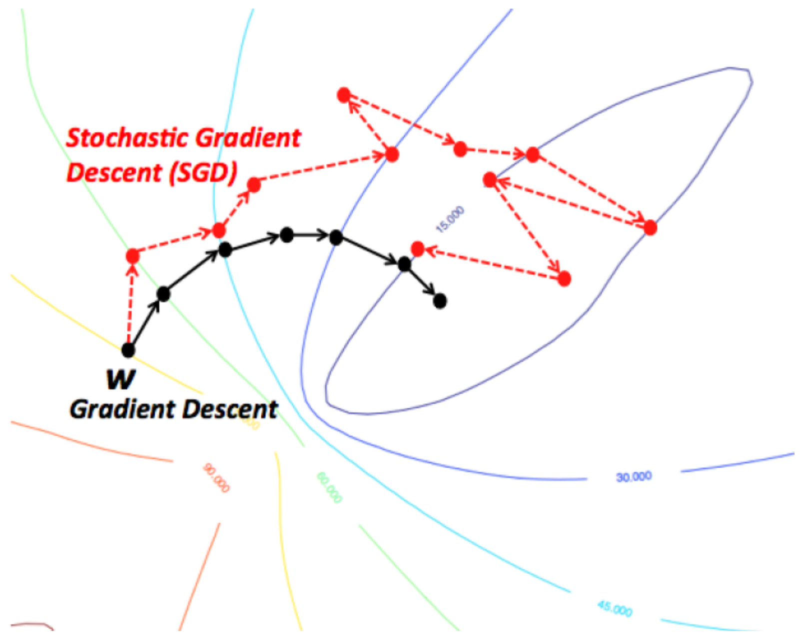


Figure: Batch gradient descent vs. SGD fluctuation (Source: wikidocs.net)

Mini-batch Gradient Descent

- Performs update for every **mini-batch** of random n examples.
- Update equation:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

- The expectation of gradient is the same as the batch gradient

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(
      loss_function, batch, params)
    params = params - learning_rate * params_grad
```

Listing 3: Code for mini-batch gradient descent update

- Pros
  - Reduces variance of updates.
  - Can exploit matrix multiplication primitives.
- Cons
  - Mini-batch size is a hyperparameter. Common sizes are 50-256.
- Typically the algorithm of choice.
- Usually referred to as **SGD** in deep learning even when **mini-batches** are used.

Method	**Accuracy**	**Update Speed**	**Memory Usage**	**Online Learning**
Batch gradient descent	Good	Slow	High	No
Stochastic gradient descent	Good (with annealing)	High	Low	Yes
Mini-batch gradient descent	Good	Medium	Medium	Yes

Table: Comparison of trade-offs of gradient descent variants

Challenges

- Choosing a learning rate.
- Defining an annealing (learning rate decay) schedule.
- Escaping saddles and suboptimal minima.

Variants of Gradient Descent Algorithms

- Momentum
- Nesterov accelerated gradient
- Adagrad
- Adadelta
- RMSprop
- Adam
- Adam extensions

Momentum by Polyak 1964, heavy ball

As has been known at least since the advent of conjugate gradient algorithms, improvements to gradient descent can be obtained within a first-order framework by using the history of past gradients. Modern research on such extended first-order methods arguably dates to Polyak [Pol64, Pol87], whose *heavy-ball method* incorporates a momentum term into the gradient step. This approach allows past gradients to influence the current step, while avoiding the complexities of conjugate gradients and permitting a stronger theoretical analysis. Explicitly, starting from an initial point $x_0, x_1 \in \mathbb{R}^n$, the heavy-ball method updates the iterates according to

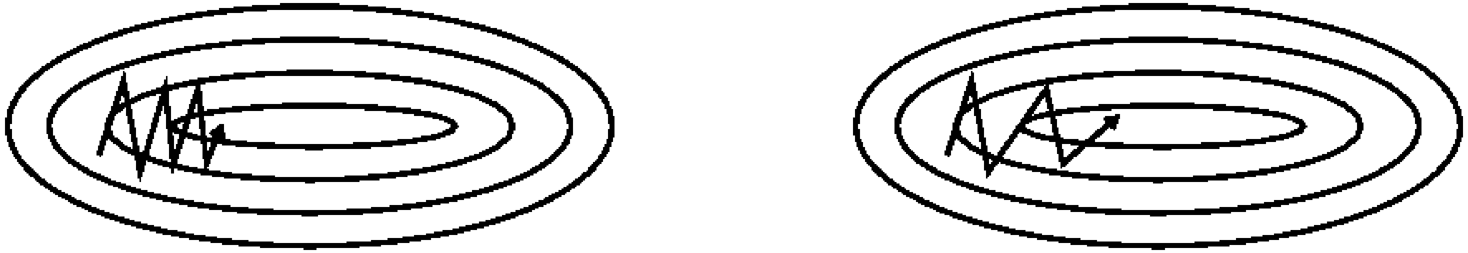
$$x_{k+1} = x_k + \alpha\,(x_k - x_{k-1}) - s\nabla f(x_k), \tag{1.2}$$

where $\alpha > 0$ is the momentum coefficient. While the heavy-ball method provably attains a faster rate of *local* convergence than gradient descent near a minimum of f, it does not come with *global* guarantees. Indeed, [LRP16] demonstrate that even for strongly convex functions the method can fail to converge for some choices of the step size.[1]

Momentum in Deep Learning

- SGD has trouble navigating **ravines**.
- Momentum [Qian, 1999] helps SGD **accelerate**.
- Adds a fraction γ of the update vector of the past step v_{t-1} to current update vector v_t. Momentum term γ is usually set to 0.9.

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \\ \theta &= \theta - v_t \end{aligned} \tag{1}$$



(a) SGD without momentum

(b) SGD with momentum

Figure: Source: Genevieve B. Orr

- **Reduces updates** for dimensions whose gradients **change directions**.
- **Increases updates** for dimensions whose gradients **point in the same directions**.

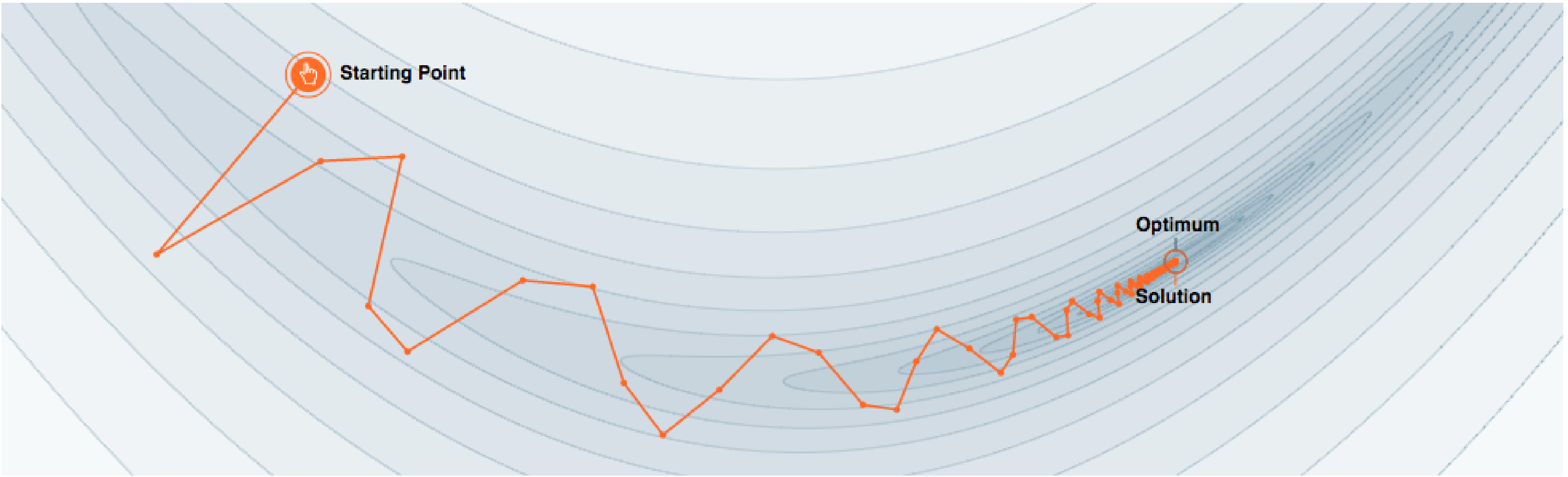


Figure: Optimization with momentum (Source: distill.pub)

Nesterov Accelerated Gradient

- Momentum **blindly accelerates** down slopes: First computes gradient, then makes a big jump.
- Nesterov accelerated gradient (NAG) [Nesterov, 1983] first makes a big jump in the direction of the previous accumulated gradient $\theta - \gamma v_{t-1}$. Then measures where it ends up and makes a correction, resulting in the complete update vector.

$$
\begin{aligned}
v_t &= \gamma\, v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1}) \\
\theta &= \theta - v_t
\end{aligned}
\tag{2}
$$

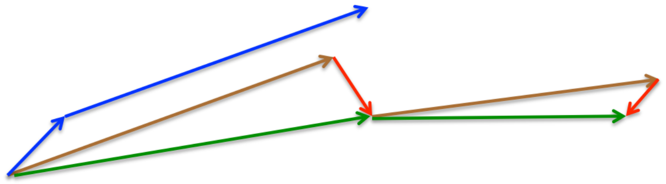


Figure: Nesterov update (Source: G. Hinton's lecture 6c)

Nesterov ODE: convex

- f is convex and has L-Lipschitz gradient, Nesterov Acceleration (NAG-C):

$$y_{k+1} = x_k - s\nabla f(x_k)$$
$$x_{k+1} = y_{k+1} + \frac{k}{k+3}(y_{k+1} - y_k),$$

- [Weijie Su, Stephen Boyd, Emmanuel Candes'2016] Nesterov ODE:

$$\ddot{X}(t) + \frac{3}{t}\dot{X}(t) + \nabla f(X(t)) = 0,$$

Nesterov ODE: strongly convex

(NAG-SC)

descent [Nes83, Nes13]. For a μ-strongly convex objective f with L-Lipschitz gradients, Nesterov's accelerated gradient method (NAG-$\mathtt{SC}$) involves the following pair of update equations:

$$\begin{aligned} y_{k+1} &= x_k - s\nabla f(x_k) \\ x_{k+1} &= y_{k+1} + \frac{1-\sqrt{\mu s}}{1+\sqrt{\mu s}}\left(y_{k+1} - y_k\right), \end{aligned} \tag{1.3}$$

between the heavy-ball method and NAG-$\mathtt{SC}$. In particular, these two methods have the *same* limiting ODE (see, for example, [WRJ16]):

$$\ddot{X}(t) + 2\sqrt{\mu}\dot{X}(t) + \nabla f(X(t)) = 0, \tag{1.9}$$

High Resolution Nesterov ODE

- [Bin Shi, Simon S. Du, Michael I. Jordan, Weijie J. Su 2018]

(a) The high-resolution ODE for the heavy-ball method (1.2):

$$\ddot{X}(t) + 2\sqrt{\mu}\dot{X}(t) + (1 + \sqrt{\mu s})\nabla f(X(t)) = 0, \tag{1.10}$$

with $X(0) = x_0$ and $\dot{X}(0) = -\frac{2\sqrt{s}\nabla f(x_0)}{1+\sqrt{\mu s}}$.

(b) The high-resolution ODE for NAG-SC (1.3):

$$\ddot{X}(t) + 2\sqrt{\mu}\dot{X}(t) + \sqrt{s}\nabla^2 f(X(t))\dot{X}(t) + (1 + \sqrt{\mu s})\nabla f(X(t)) = 0, \tag{1.11}$$

with $X(0) = x_0$ and $\dot{X}(0) = -\frac{2\sqrt{s}\nabla f(x_0)}{1+\sqrt{\mu s}}$.

(c) The high-resolution ODE for NAG-C (1.5):

$$\ddot{X}(t) + \frac{3}{t}\dot{X}(t) + \sqrt{s}\nabla^2 f(X(t))\dot{X}(t) + \left(1 + \frac{3\sqrt{s}}{2t}\right)\nabla f(X(t)) = 0 \tag{1.12}$$

for $t \geq 3\sqrt{s}/2$, with $X(3\sqrt{s}/2) = x_0$ and $\dot{X}(3\sqrt{s}/2) = -\sqrt{s}\nabla f(x_0)$.

Adagrad

- Previous methods: **Same learning rate** η for all parameters θ.
- Adagrad [Duchi et al., 2011] **adapts** the learning rate to the parameters (**large** updates for **infrequent** parameters, **small** updates for **frequent** parameters).
- SGD update: $\theta_{t+1} = \theta_t - \eta \cdot g_t$
  - $g_t = \nabla_{\theta_t} J(\theta_t)$
- Adagrad divides the learning rate by the **square root of the sum of squares of historic gradients**.
- Adagrad update:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t \tag{3}$$

  - $G_t \in \mathbb{R}^{d \times d}$: diagonal matrix where each diagonal element i, i is the sum of the squares of the gradients w.r.t. θ_i up to time step t
  - ϵ: smoothing term to avoid division by zero
  - $\odot$: element-wise multiplication

- Pros
  - Well-suited for dealing with sparse data.
  - Significantly improves robustness of SGD.
  - Lesser need to manually tune learning rate.
- Cons
  - Accumulates squared gradients in denominator.
  - Causes the learning rate to shrink and become infinitesimally small.

Adadelta

- Adadelta [Zeiler, 2012] restricts the window of accumulated past gradients to a **fixed size**. SGD update:

$$\begin{aligned} \Delta\theta_t &= -\eta \cdot g_t \\ \theta_{t+1} &= \theta_t + \Delta\theta_t \end{aligned} \tag{4}$$

- Defines **running average** of squared gradients $E[g^2]_t$ at time t:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma)g_t^2 \tag{5}$$

  - γ: fraction similarly to momentum term, around 0.9
- Adagrad update:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t \tag{6}$$

- Preliminary Adadelta update:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \tag{7}$$

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \tag{8}$$

- Denominator is just root mean squared (RMS) error of gradient:

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t \tag{9}$$

- Note: **Hypothetical units do not match**.
- Define **running average of squared parameter updates** and RMS:

$$\begin{aligned} E[\Delta\theta^2]_t &= \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2 \\ RMS[\Delta\theta]_t &= \sqrt{E[\Delta\theta^2]_t + \epsilon} \end{aligned} \tag{10}$$

- Approximate with $RMS[\Delta\theta]_{t-1}$, replace η for **final Adadelta update**:

$$\begin{aligned} \Delta\theta_t &= -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t \\ \theta_{t+1} &= \theta_t + \Delta\theta_t \end{aligned} \tag{11}$$

RMSprop

- Developed independently from Adadelta around the same time by Geoff Hinton.
- Also divides learning rate by a **running average of squared gradients**.
- RMSprop update:

$$
\begin{aligned}
E[g^2]_t &= \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2 \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t
\end{aligned} \tag{12}
$$

  - γ: decay parameter; typically set to 0.9
  - η: learning rate; a good default value is 0.001

Adam

- Adaptive Moment Estimation (Adam) [Kingma and Ba, 2015] also stores **running average of past squared gradients** v_t like Adadelta and RMSprop.
- Like Momentum, stores **running average of past gradients** m_t.

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2
\end{aligned}
\tag{13}
$$

  - m_t: first moment (mean) of gradients
  - v_t: second moment (uncentered variance) of gradients
  - β_1, β_2: decay rates

- m_t and v_t are initialized as 0-vectors. For this reason, they are biased towards 0.
- Compute bias-corrected first and second moment estimates:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \end{aligned} \tag{14}$$

- Adam update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t \tag{15}$$

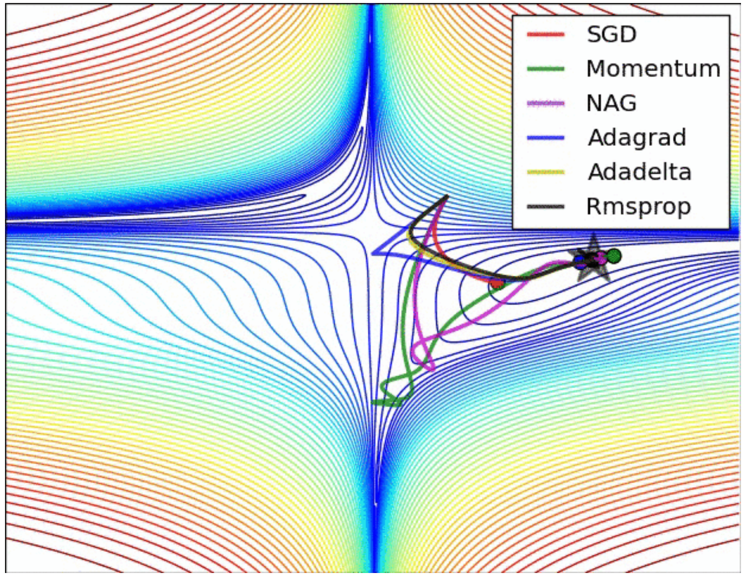
Adam Extensions

1. AdaMax [Kingma and Ba, 2015]
   - Adam with ℓ_∞ norm
2. Nadam [Dozat, 2016]
   - Adam with Nesterov accelerated gradient

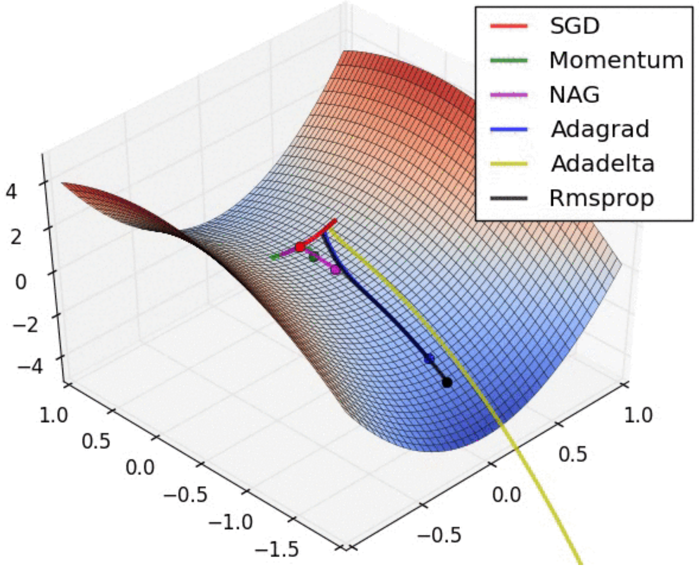
Update Equations

Method	Update equation
SGD	$g_t = \nabla_{\theta_t} J(\theta_t)$ $\Delta\theta_t = -\eta \cdot g_t$ $\theta_t = \theta_t + \Delta\theta_t$
Momentum	$\Delta\theta_t = -\gamma\ v_{t-1} - \eta g_t$
NAG	$\Delta\theta_t = -\gamma\ v_{t-1} - \eta \nabla_\theta J(\theta - \gamma v_{t-1})$
Adagrad	$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$
Adadelta	$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$
RMSprop	$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$
Adam	$\Delta\theta_t = -\frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$

Visualization of algorithms



(a) SGD optimization on loss surface contours



(b) SGD optimization on saddle point

Figure: Source and full animations: Alec Radford

Comparisons

- Adaptive learning rate methods (**Adagrad**, **Adadelta**, **RMSprop**, **Adam**) are particularly useful for sparse features.
- Adagrad, Adadelta, RMSprop, and Adam work well in similar circumstances.
- [**Kingma and Ba, 2015**] show that bias-correction helps **Adam** slightly outperform RMSprop.

On Convergence Analysis

- [Xiangyi Chen, Sijia Liu, Ruoyu Sun, Mingyi Hong 2018] On the Convergence of A Class of Adam-type Algorithms for Non-Convex Optimization, arXiv: 1808.02941:
  - Under mild conditions, this class of methods, which we refer to as the "Adam-type", includes the popular algorithms such as the Adam, AMSGrad and AdaGrad, can achieve convergence rate of order O(log T/\sqrt{T}) for nonconvex stochastic optimization.

Parallel and Distributed SGD

- **Hogwild!** [**Niu et al., 2011**]
  - Parallel SGD updates on CPU
  - Shared memory access without parameter lock Only works for sparse input data
- Downpour SGD [**Dean et al., 2012**]
  - Multiple replicas of model on subsets of training data run in parallel
  - Updates sent to parameter server;
  - updates fraction of model parameters
- Delay-tolerant Algorithms for SGD [**Mcmahan and Streeter, 2014**]
  - Methods also adapt to update delays
- TensorFlow [**Abadi et al., 2015**]
  - Computation graph is split into a subgraph for every device
  - Communication takes place using Send/Receive node pairs
- Elastic Averaging SGD [**Zhang et al., 2015**]
  - Links parameters elastically to a center variable stored by parameter server

Additional Strategies for SGD

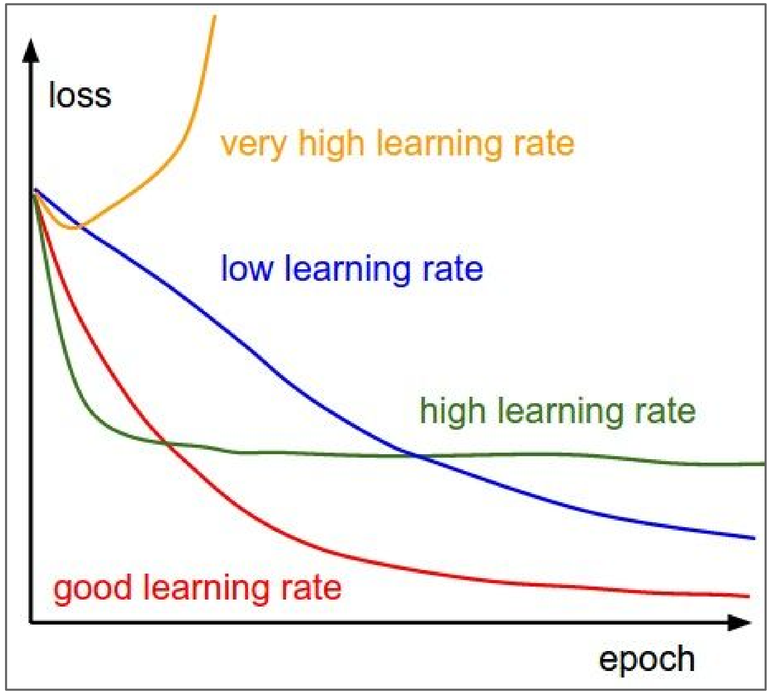
- Shuffling and Curriculum Learning [**Bengio et al., 2009**]
  - Shuffle training data after every epoch to break biases
  - Order training examples to solve progressively harder problems; infrequently used in practice
- Batch normalization [**Ioffe and Szegedy, 2015**]
  - Re-normalizes every mini-batch to zero mean, unit variance
  - Must-use for computer vision
- Early stopping
  - "*Early stopping (is) beautiful free lunch*" (**Geoff Hinton**)
- Gradient noise [**Neelakantan et al., 2015**]
  - Add Gaussian noise to gradient
  - Makes model more robust to poor initializations
  - Escape saddles or local optima

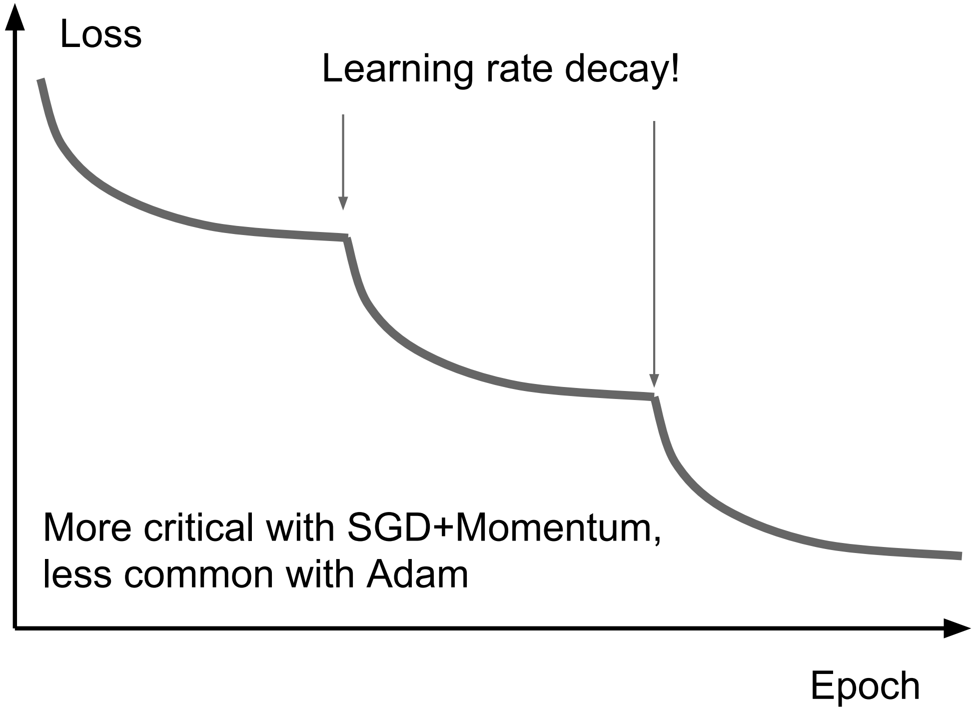
Adam vs. Tuned SGD

- Many recent papers use SGD with learning rate annealing.
- SGD with tuned learning rate and momentum is competitive with Adam [Zhang et al., 2017b].
- Adam converges faster, but oscillates and may underperform SGD on some tasks, e.g. Machine Translation [Wu et al., 2016].
- Adam with restarts and SGD-style annealing converges faster and outperforms SGD [Denkowski and Neubig, 2017].
- Increasing the batch size may have the same effect as decaying the learning rate [Smith et al., 2017].

Second Order Methods

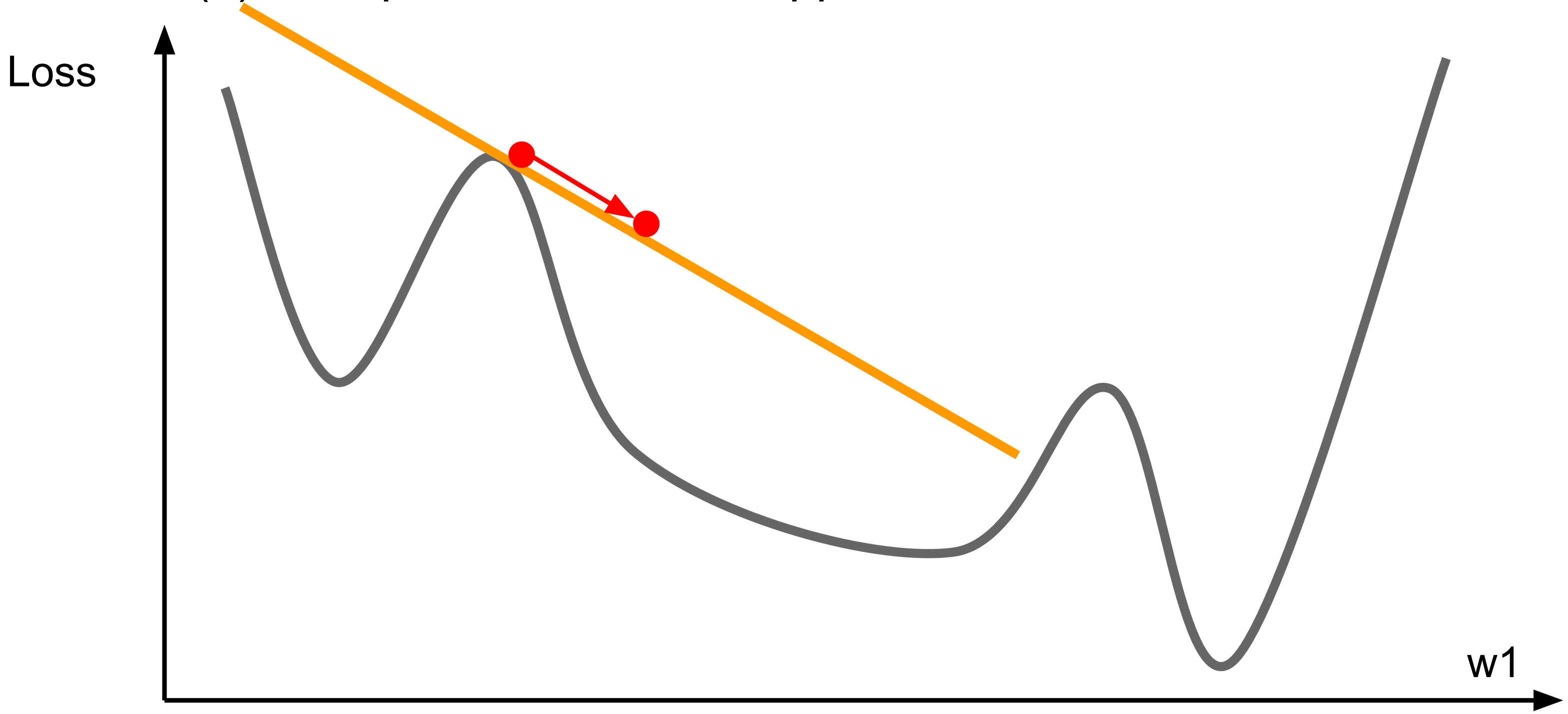
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.





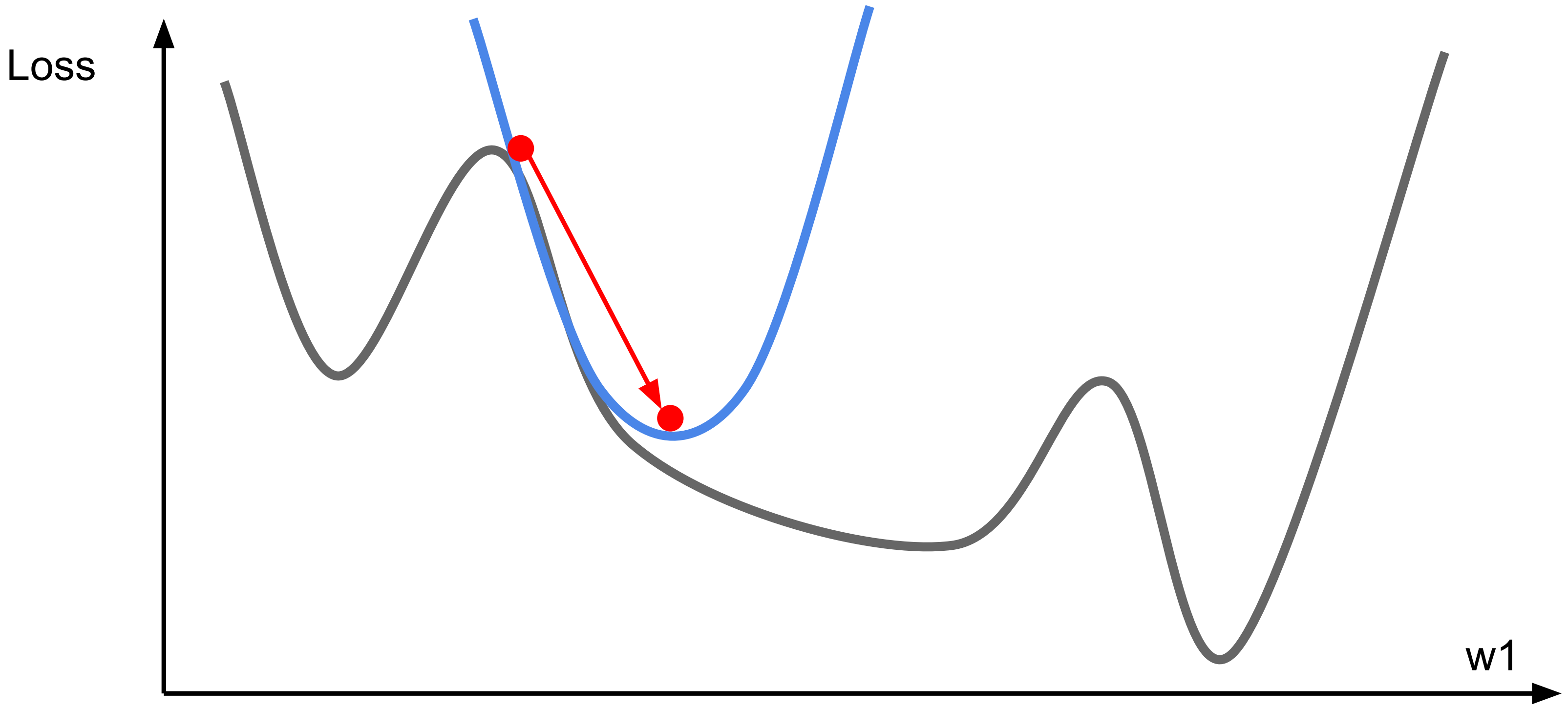
First-Order Optimization

(1) Use gradient form linear approximation
(2) Step to minimize the approximation



Second-Order Optimization

(1) Use gradient **and Hessian** to form **quadratic** approximation
(2) Step to the **minima** of the approximation



Newton Method

Second-Order Optimization

second-order Taylor expansion:

$$\boxed{J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)}$$

Solving for the critical point we obtain the Newton parameter update:

$$\boxed{\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)}$$

Q: What is nice about this update?

Second-Order Optimization

second-order Taylor expansion:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

No hyperparameters!
No learning rate!

Q: What is nice about this update?

But, ...

Second-Order Optimization

second-order Taylor expansion:

$$\boxed{J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)}$$

Solving for the critical point we obtain the Newton parameter update:

$$\boxed{\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)}$$

Hessian has O(N^2) elements
Inverting takes O(N^3)
N = (Tens or Hundreds of) Millions

Q2: Why is this bad for deep learning?

Second-Order Optimization

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

- Quasi-Newton methods (**BGFS** most popular):
  instead of inverting the Hessian (O(n^3)), approximate inverse Hessian with rank 1 updates over time (O(n^2) each).

- **L-BFGS** (Limited memory BFGS):
  Does not form/store the full inverse Hessian.

L-BFGS

- **Usually works very well in full batch, deterministic mode** i.e. if you have a single, deterministic f(x) then L-BFGS will probably work very nicely

- **Does not transfer very well to mini-batch setting**. Gives bad results. Adapting L-BFGS to large-scale, stochastic setting is an active area of research.

Le et al, “On optimization methods for deep learning, ICML 2011”

In practice

- **Adam** is a good default choice in most cases
  - **Adam+SGD** may achieve fast speed and better accuracy
- If you can afford to do full batch updates then try out **L-BFGS** (and don’t forget to disable all sources of noise)

Regularizations

Regularization: Add term to loss

$$L = \frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) + \boxed{\lambda R(W)}$$

In common use:

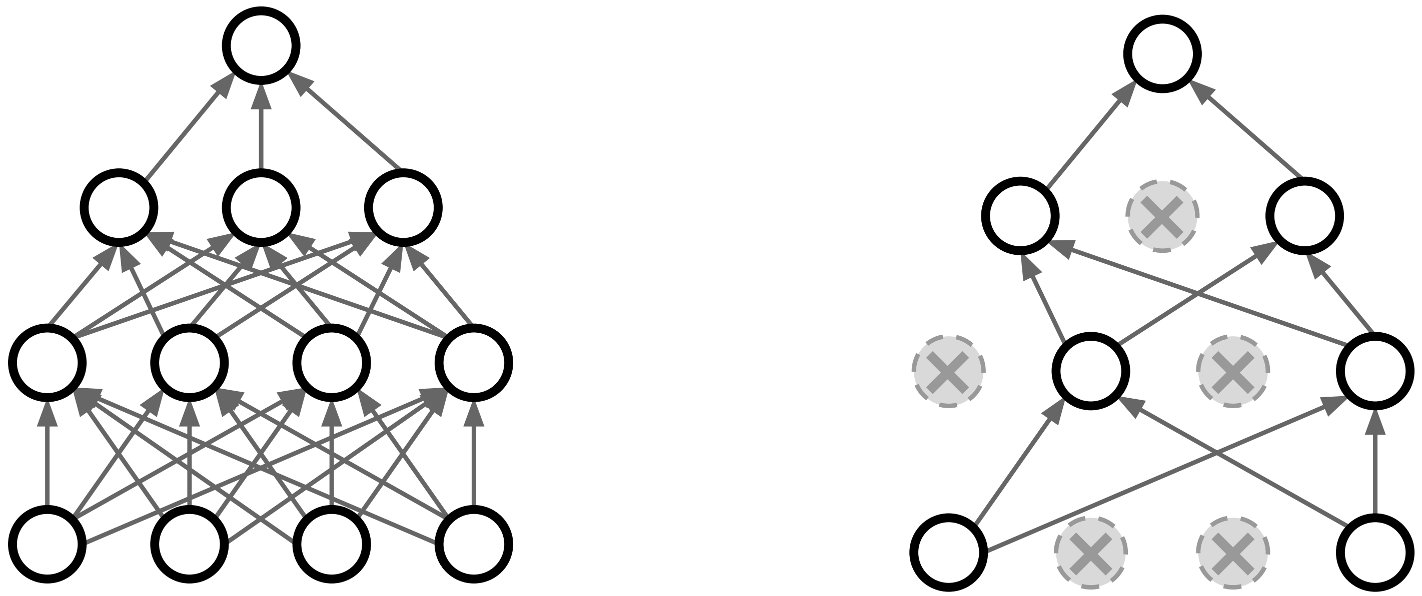
L2 regularization $R(W) = \sum_k \sum_l W_{k,l}^2$ (Weight decay)

L1 regularization $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2) $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

Regularization: Dropout

In each forward pass, randomly set some neurons to zero
Probability of dropping is a hyperparameter; 0.5 is common

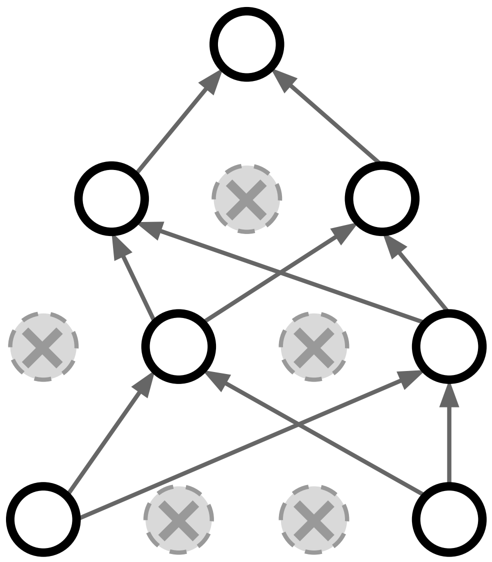


Srivastava et al, “Dropout: A simple way to prevent neural networks from overfitting”, JMLR 2014

Regularization: Dropout

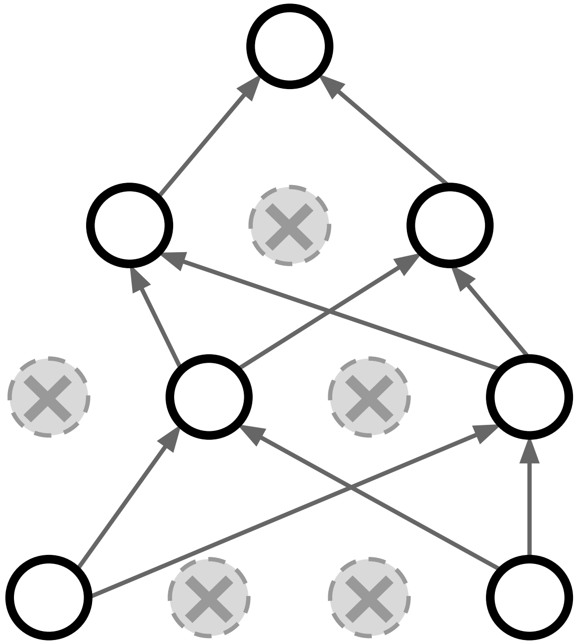
```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)
```

Example forward pass with a 3-layer network using dropout



Regularization: Dropout

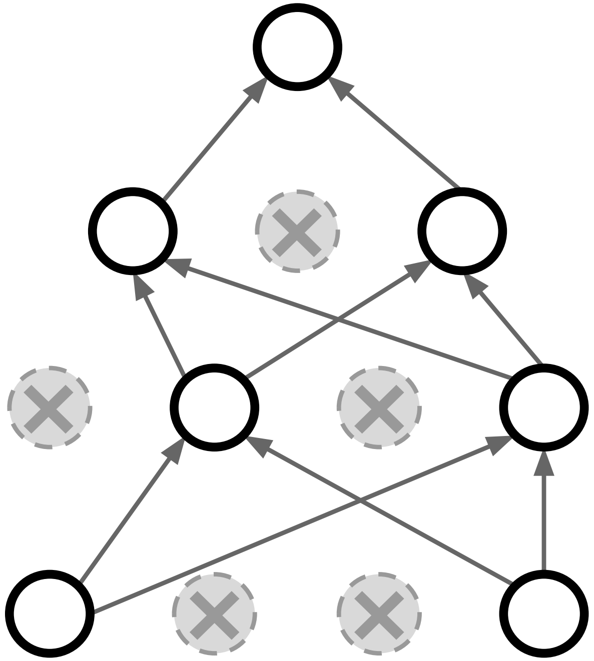
How can this possibly be a good idea?



Forces the network to have a redundant representation;
Prevents co-adaptation of features

Regularization: Dropout

How can this possibly be a good idea?



Another interpretation:

Dropout is training a large **ensemble** of models (that share parameters).

Each binary mask is one model

An FC layer with 4096 units has $2^{4096} \sim 10^{1233}$ possible masks!
Only $\sim 10^{82}$ atoms in the universe...

Dropout: Test time

Dropout makes our output random!

$$\underbrace{\boxed{y}}_{\text{Output (label)}} = f_W(\underbrace{\boxed{x}}_{\text{Input (image)}}, \underbrace{\boxed{z}}_{\text{Random mask}})$$

Want to "average out" the randomness at test-time

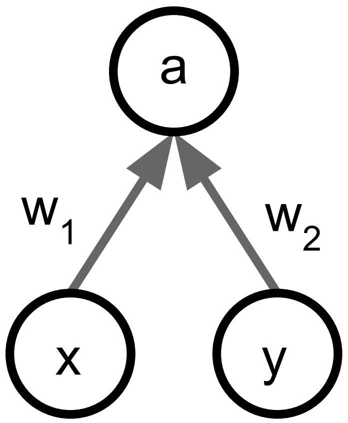
$$y = f(x) = E_z[f(x, z)] = \int p(z) f(x, z) dz$$

But this integral seems hard …

Dropout: Test time

Want to approximate the integral

$$y = f(x) = E_z\big[f(x,z)\big] = \int p(z) f(x,z) dz$$



Consider a single neuron.

At test time we have: $E\big[a\big] = w_1 x + w_2 y$

During training we have:
$$\begin{aligned} E\big[a\big] &= \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) \\ &+ \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2y) \\ &= \frac{1}{2}(w_1x + w_2y) \end{aligned}$$

At test time, **multiply** by dropout probability

Dropout: Test time

```
def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

At test time all neurons are active always
=> We must scale the activations so that for each neuron:
<u>output at test time</u> = <u>expected output at training time</u>

Dropout Summary

```python
""" Vanilla Dropout: Not recommended implementation (see notes below) """

p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

drop in forward pass

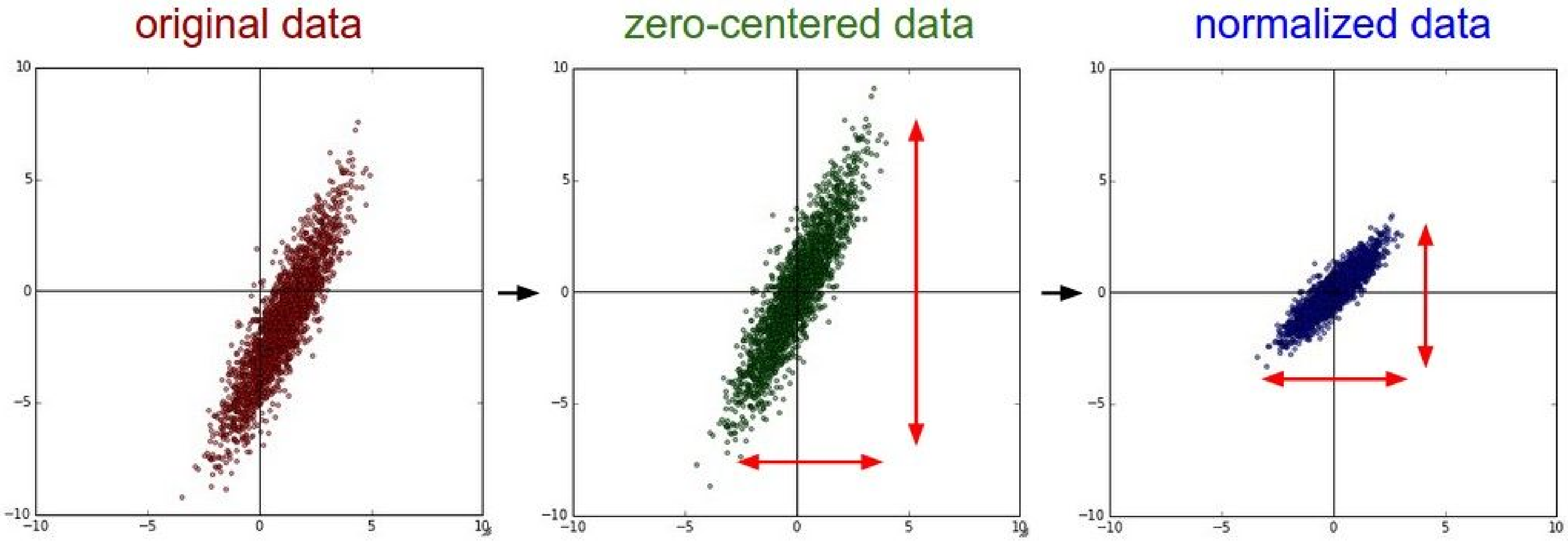
scale at test time

More common: “Inverted dropout”

```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

test time is unchanged!

Regularization: Batch normalization




```
X -= np.mean(X, axis = 0)
```

```
X /= np.std(X, axis = 0)
```

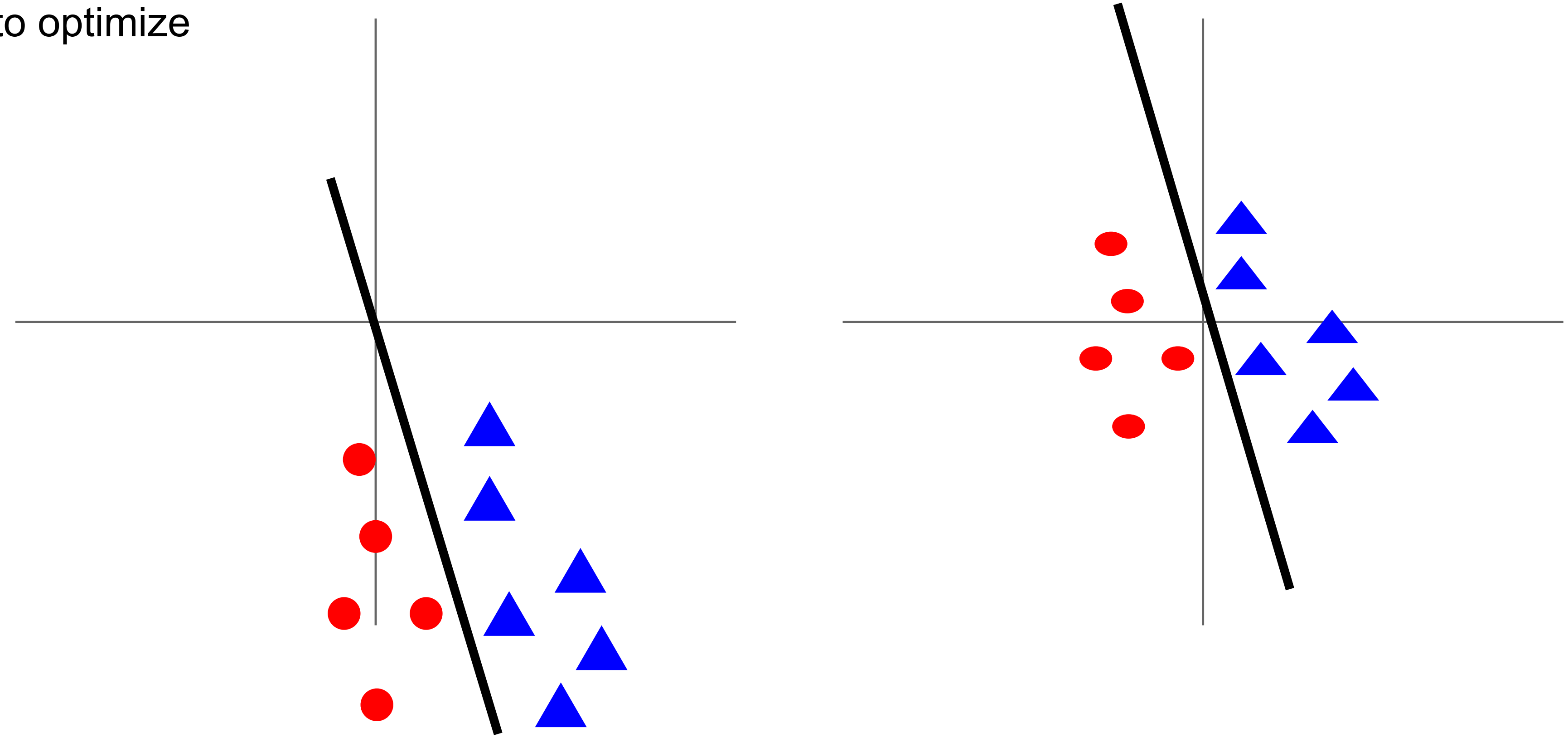
(Assume X [NxD] is data matrix, each example in a row)

$$f\left(\sum_i w_i x_i + b\right)$$

Data normalization

Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize

After normalization: less sensitive to small changes in weights; easier to optimize



e.g. consider CIFAR-10 example with [32,32,3] images

- Subtract the mean image (e.g. AlexNet)
  (mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet)
  (mean along each channel = 3 numbers)

Not common to normalize variance, to do PCA or whitening

Regularization: Batch Normalization

Batch Normalization

[Ioffe and Szegedy, 2015]

"you want unit gaussian activations? just make them so."

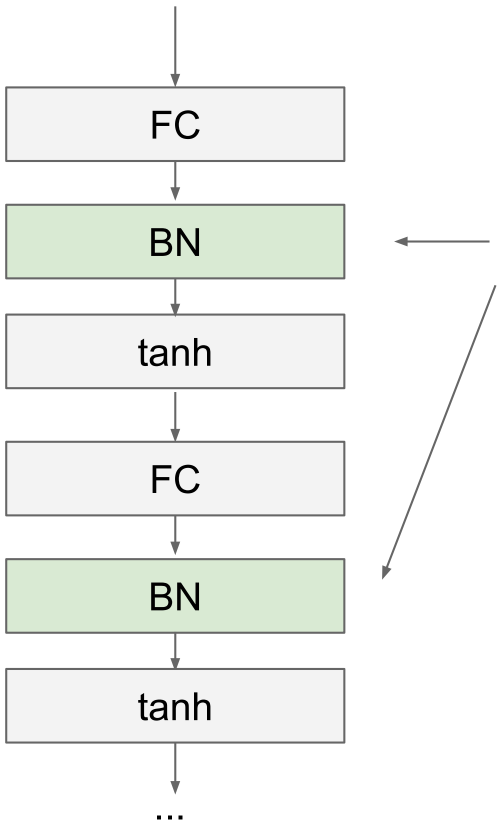
consider a batch of activations at some layer.
To make each dimension unit gaussian, apply:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

this is a vanilla
differentiable function...

Batch Normalization

[Ioffe and Szegedy, 2015]

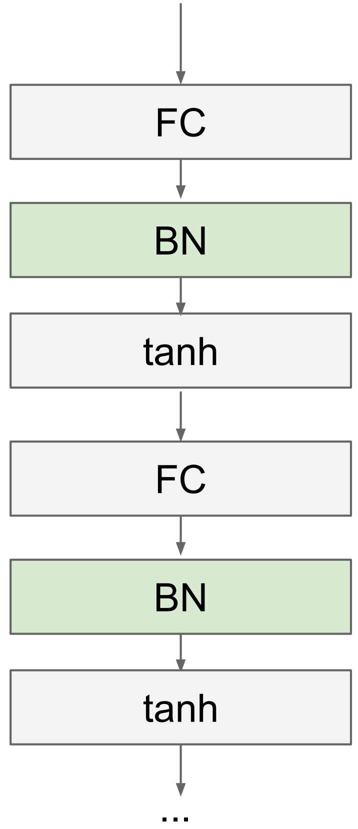


Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Batch Normalization

[Ioffe and Szegedy, 2015]



Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

Problem: do we necessarily want a unit gaussian input to a tanh layer?

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Batch Normalization

[Ioffe and Szegedy, 2015]

Normalize:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

And then allow the network to squash the range if it wants to:

$$y^{(k)} = \gamma^{(k)} \widehat{x}^{(k)} + \beta^{(k)}$$

Note, the network can learn:

$$\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$$

$$\beta^{(k)} = \mathrm{E}[x^{(k)}]$$

to recover the identity mapping.

Batch Normalization

[Ioffe and Szegedy, 2015]

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

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

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

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

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

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

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

Batch Normalization

[Ioffe and Szegedy, 2015]

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

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

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

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

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

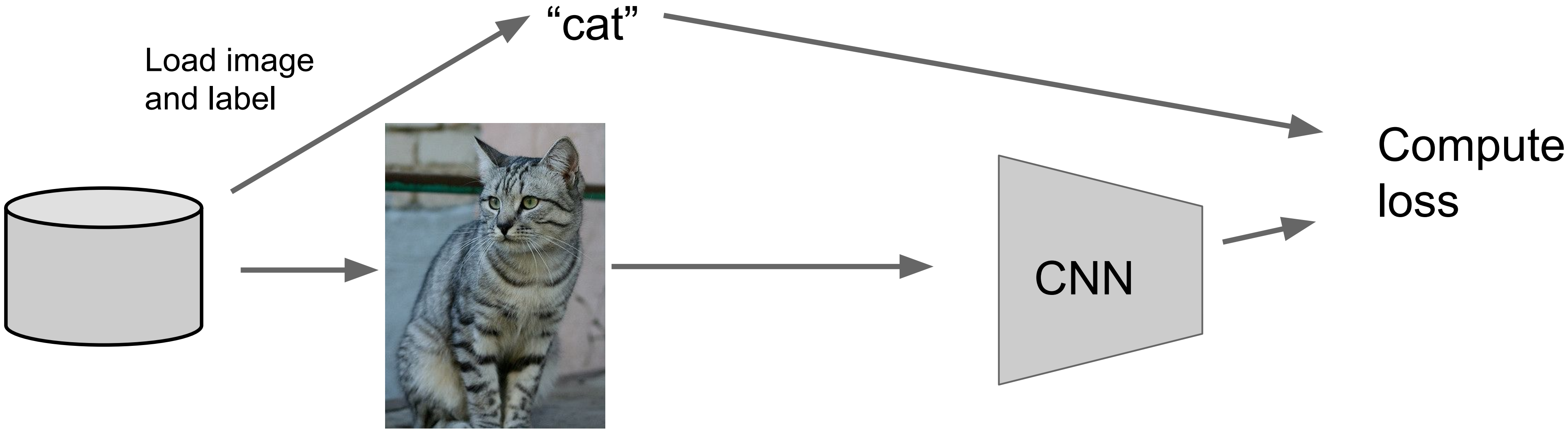
$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Note: at test time BatchNorm layer functions differently:

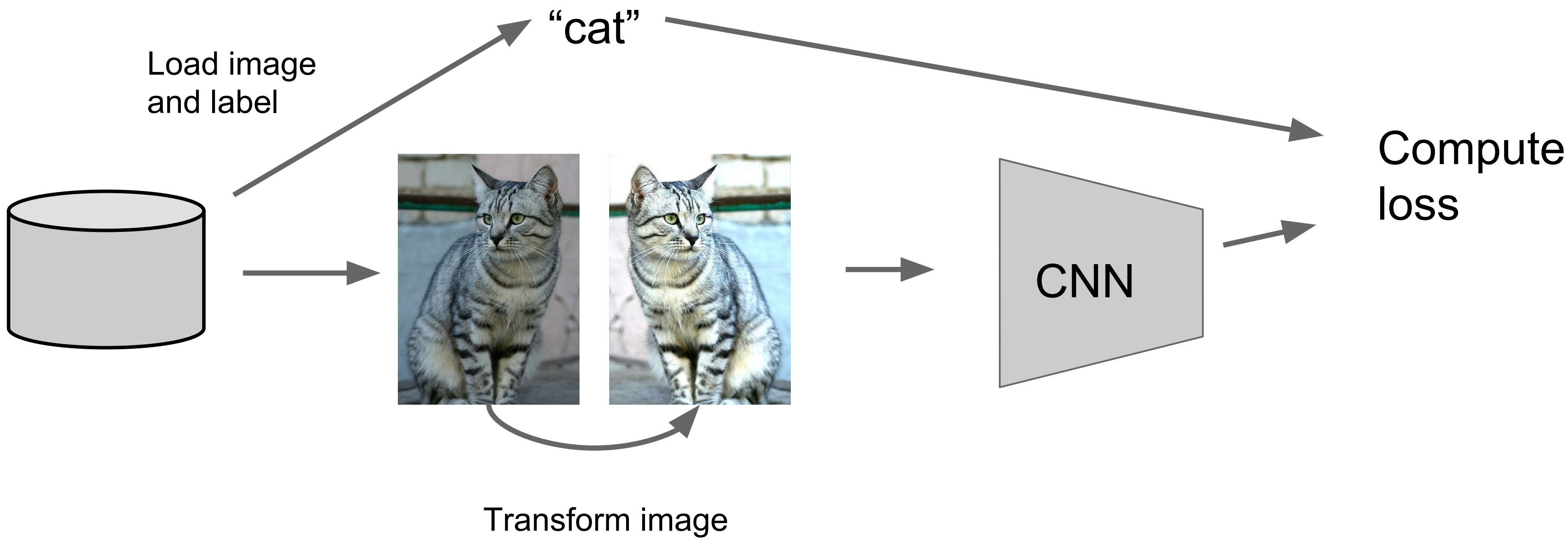
The mean/std are not computed based on the batch. Instead, a single fixed empirical mean of activations during training is used.

(e.g. can be estimated during training with running averages)

Regularization: Data Augmentation

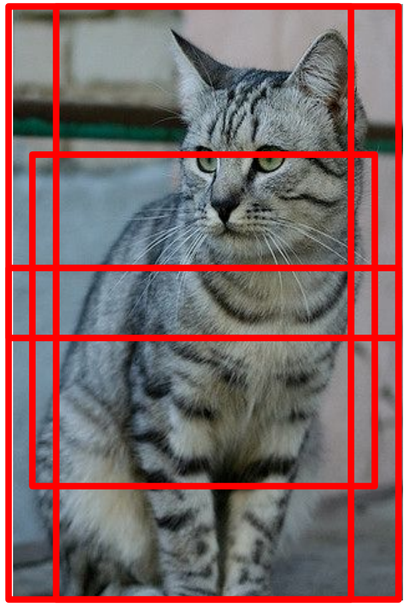


Regularization: Data Augmentation



Data Augmentation

Random crops and scales



Training: sample random crops / scales

ResNet:

1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch

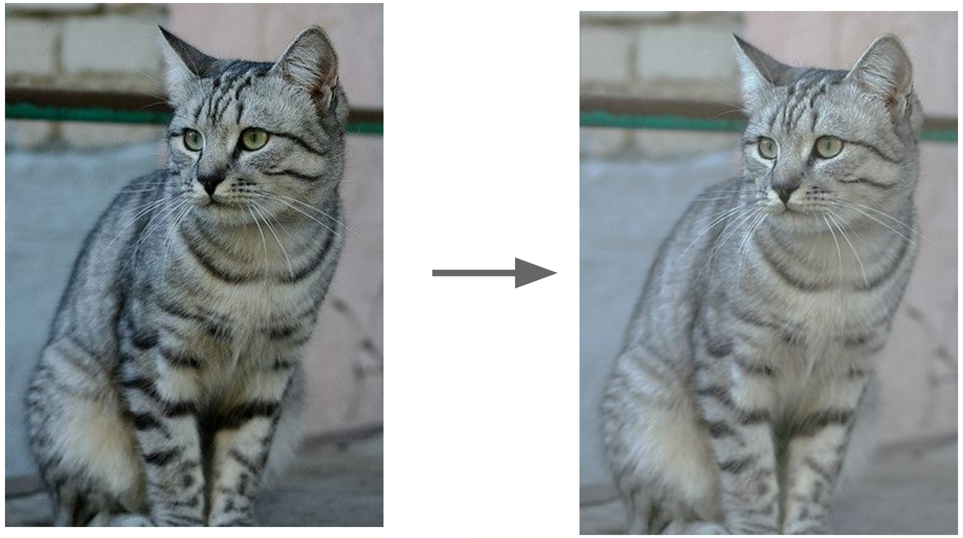
Testing: average a fixed set of crops

ResNet:

1. Resize image at 5 scales: {224, 256, 384, 480, 640}
2. For each size, use 10 224 x 224 crops: 4 corners + center, + flips

Data Augmentation
Color Jitter

Simple: Randomize contrast and brightness



More Complex:

1. Apply PCA to all [R, G, B] pixels in training set
2. Sample a “color offset” along principal component directions
3. Add offset to all pixels of a training image

(As seen in *[Krizhevsky et al. 2012]*, ResNet, etc)

Data Augmentation
Get creative for your problem!

- Random mix/combinations of
  - Translation
  - Rotation
  - Stretching
  - Shearing
  - Lens distortions
  - Style transform
  - Adversarials … (go crazy)

Randomized Algorithms

Regularization: A common pattern

Training: Add some kind of randomness

$$y = f_W(x, z)$$

Testing: Average out randomness (sometimes approximate)

$$y = f(x) = E_z\big[f(x, z)\big] = \int p(z) f(x, z) dz$$

Regularization: A common pattern

Training: Add random noise

Testing: Marginalize over the noise

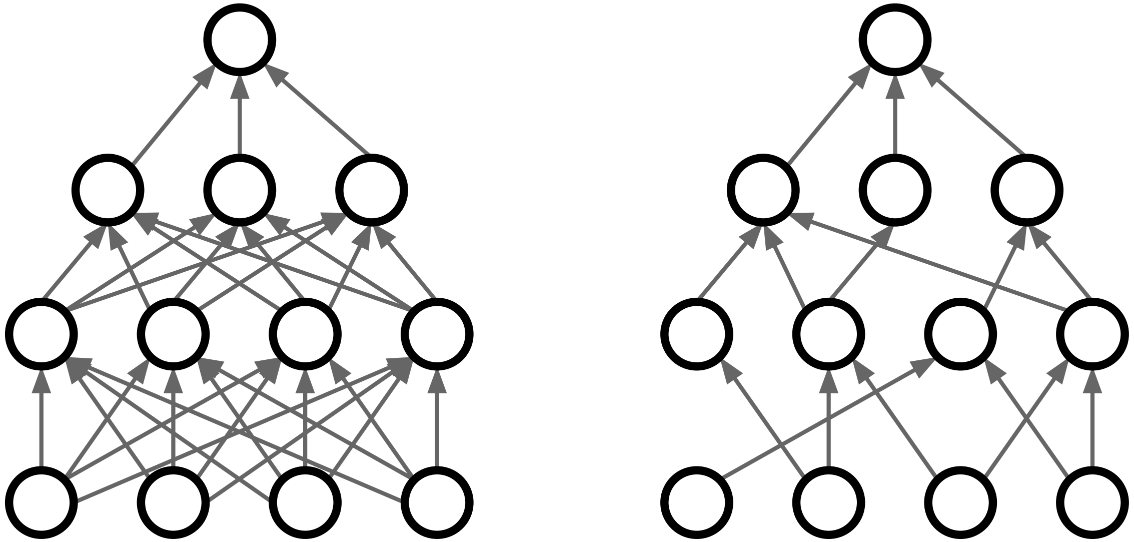
Examples:

Dropout

Batch Normalization

Data Augmentation

DropConnect

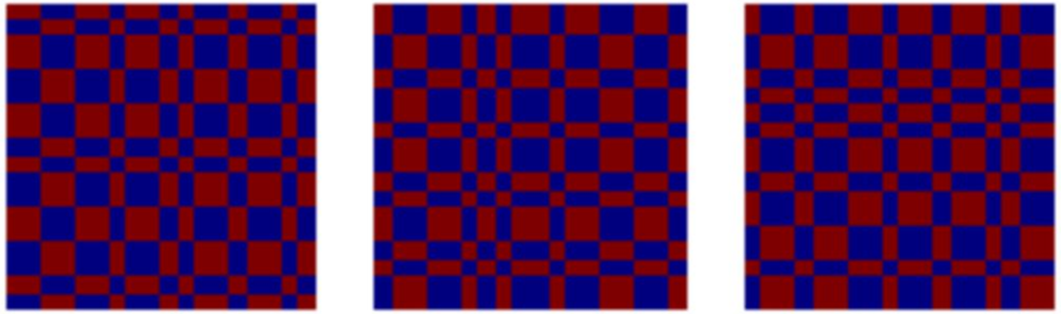


Wan et al, “Regularization of Neural Networks using DropConnect”, ICML 2013

Regularization: A common pattern

Training: Add random noise
Testing: Marginalize over the noise

Examples:
Dropout
Batch Normalization
Data Augmentation
DropConnect
Fractional Max Pooling

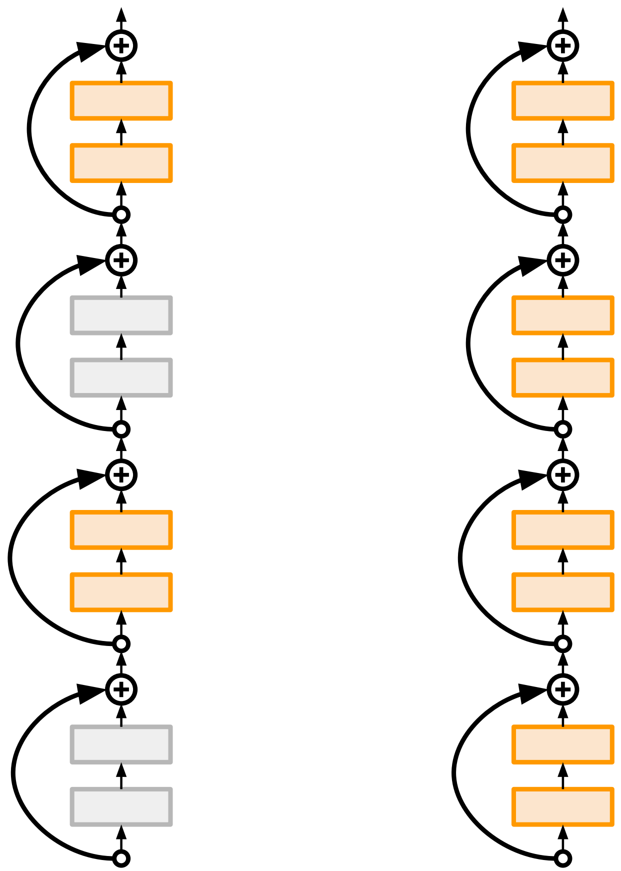


Graham, “Fractional Max Pooling”, arXiv 2014

Regularization: A common pattern

Training: Add random noise
Testing: Marginalize over the noise

Examples:
Dropout
Batch Normalization
Data Augmentation
DropConnect
Fractional Max Pooling
Stochastic Depth



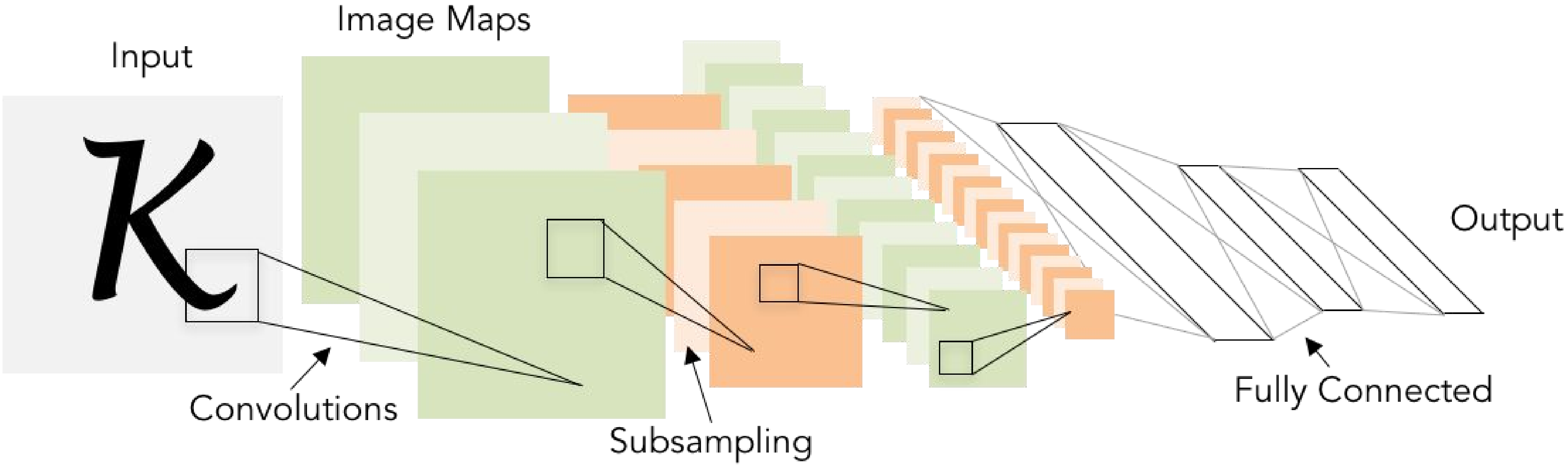
Huang et al, “Deep Networks with Stochastic Depth”, ECCV 2016

Randomization can be more:

- Regularization, that we have seen
- *Privacy (Differential Privacy): Dwork et al.*
- *Robustness: Osher et al., Daniel Hsu et al.*

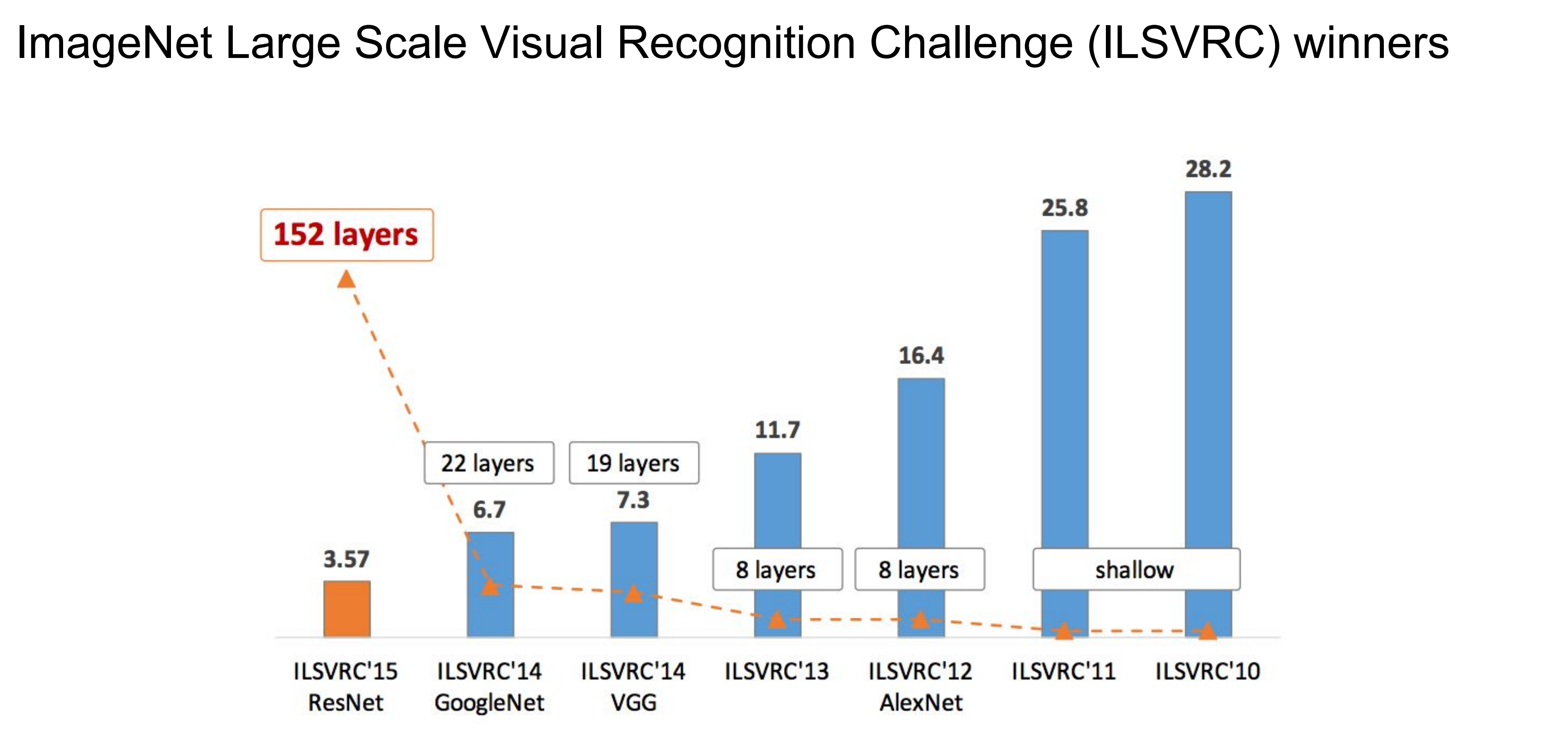
Review: LeNet-5

[LeCun et al., 1998]

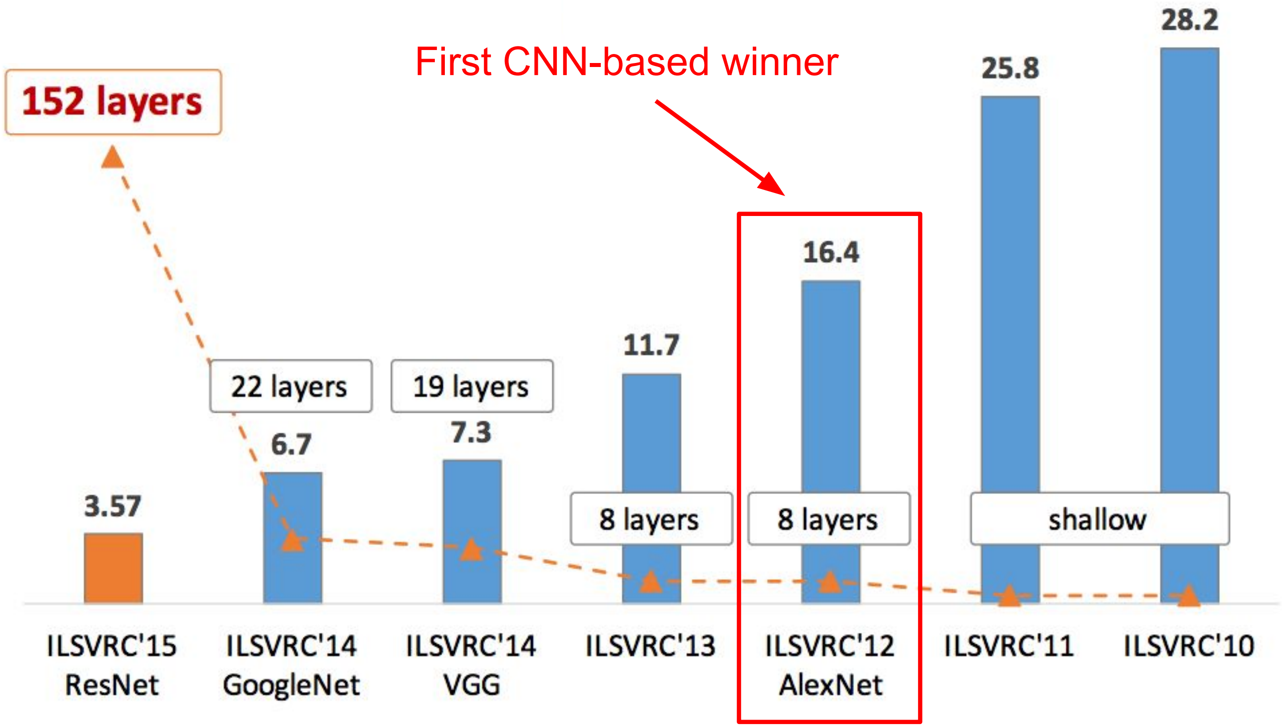


Conv filters were 5x5, applied at stride 1
Subsampling (Pooling) layers were 2x2 applied at stride 2
i.e. architecture is [CONV-POOL-CONV-POOL-FC-FC]

Popular Architectures

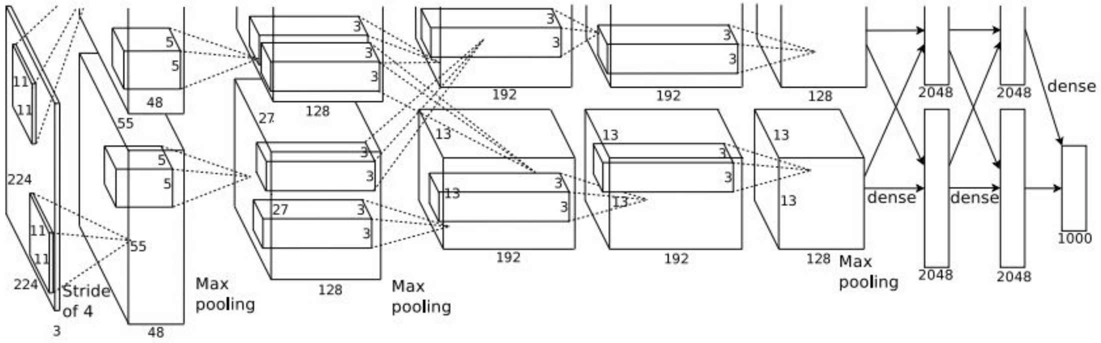


ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



Case Study: AlexNet

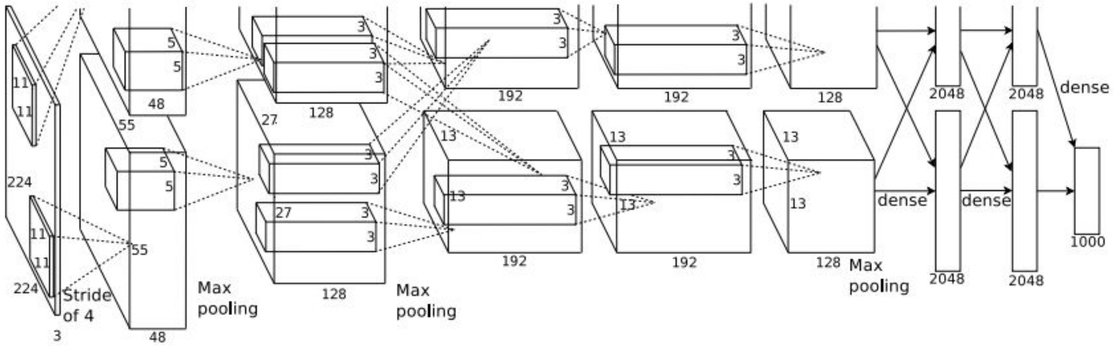
[Krizhevsky et al. 2012]



Architecture:
CONV1
MAX POOL1
NORM1
CONV2
MAX POOL2
NORM2
CONV3
CONV4
CONV5
Max POOL3
FC6
FC7
FC8

Case Study: AlexNet

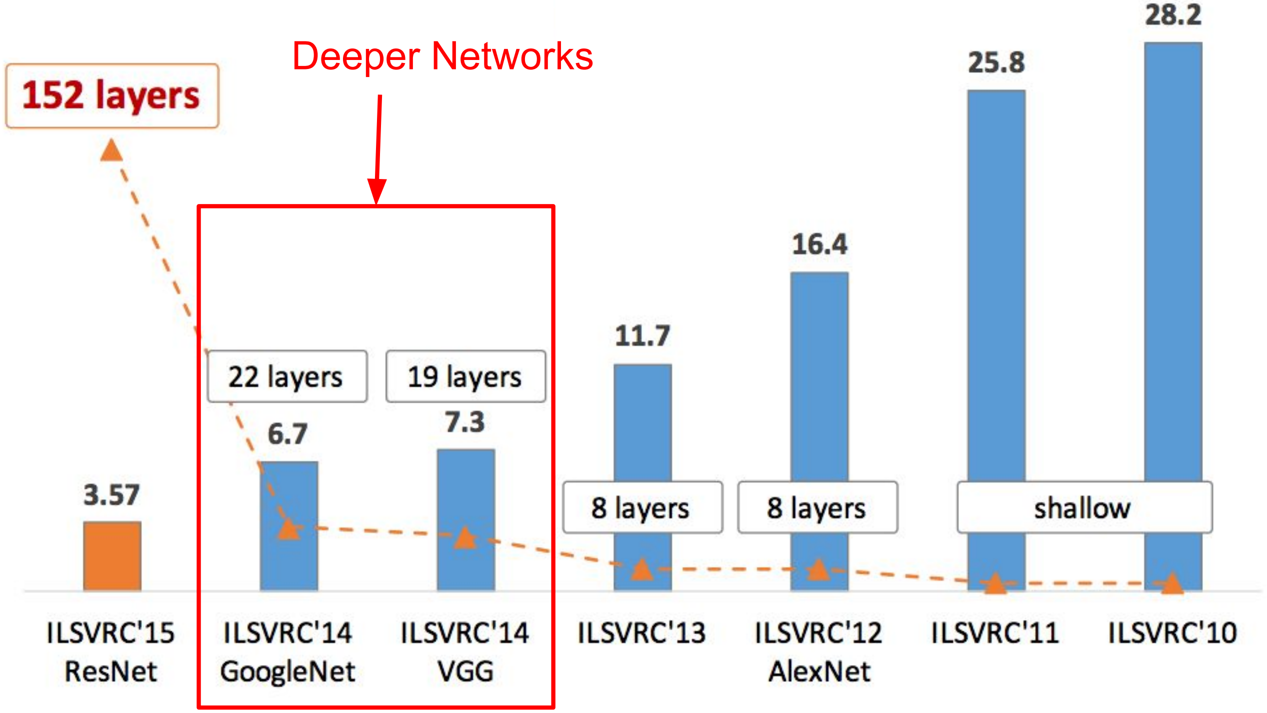
[Krizhevsky et al. 2012]



Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)

Details/Retrospectives:
- first use of ReLU
- used Norm layers (not common anymore)
- heavy data augmentation
- dropout 0.5
- batch size 128
- SGD Momentum 0.9
- Learning rate 1e-2, reduced by 10 manually when val accuracy plateaus
- L2 weight decay 5e-4
- 7 CNN ensemble: 18.2% -> 15.4%

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

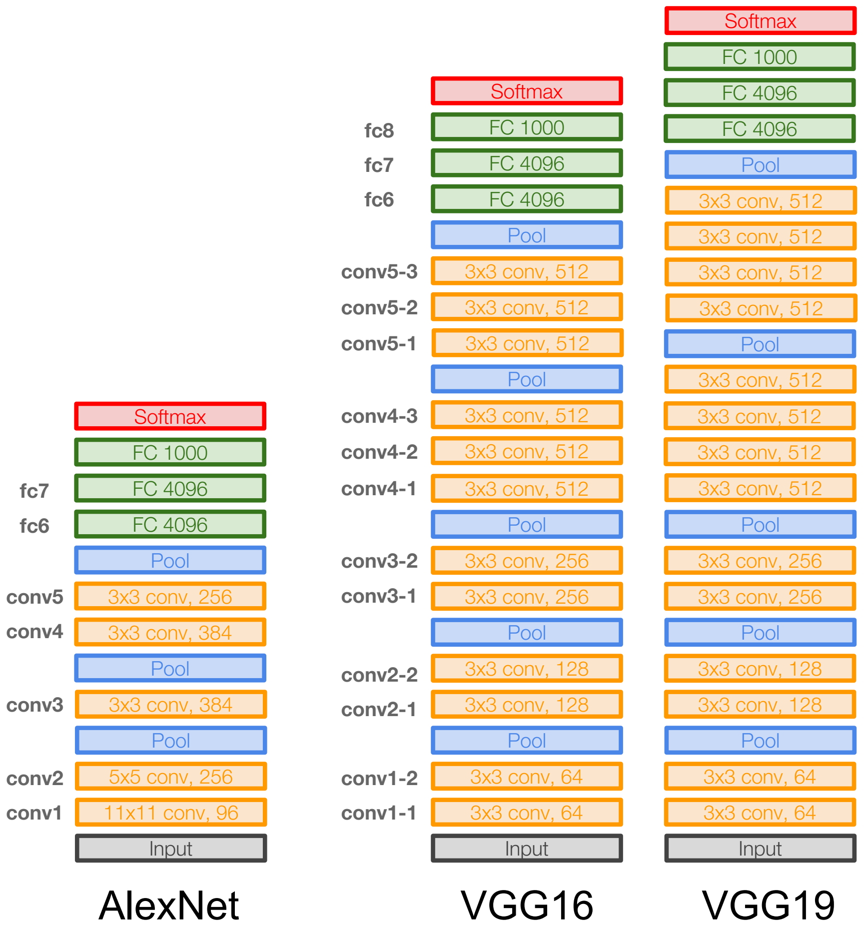


Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Details:

- ILSVRC'14 2nd in classification, 1st in localization
- Similar training procedure as Krizhevsky 2012
- No Local Response Normalisation (LRN)
- Use VGG16 or VGG19 (VGG19 only slightly better, more memory)
- Use ensembles for best results
- FC7 features generalize well to other tasks



Case Study: VGGNet

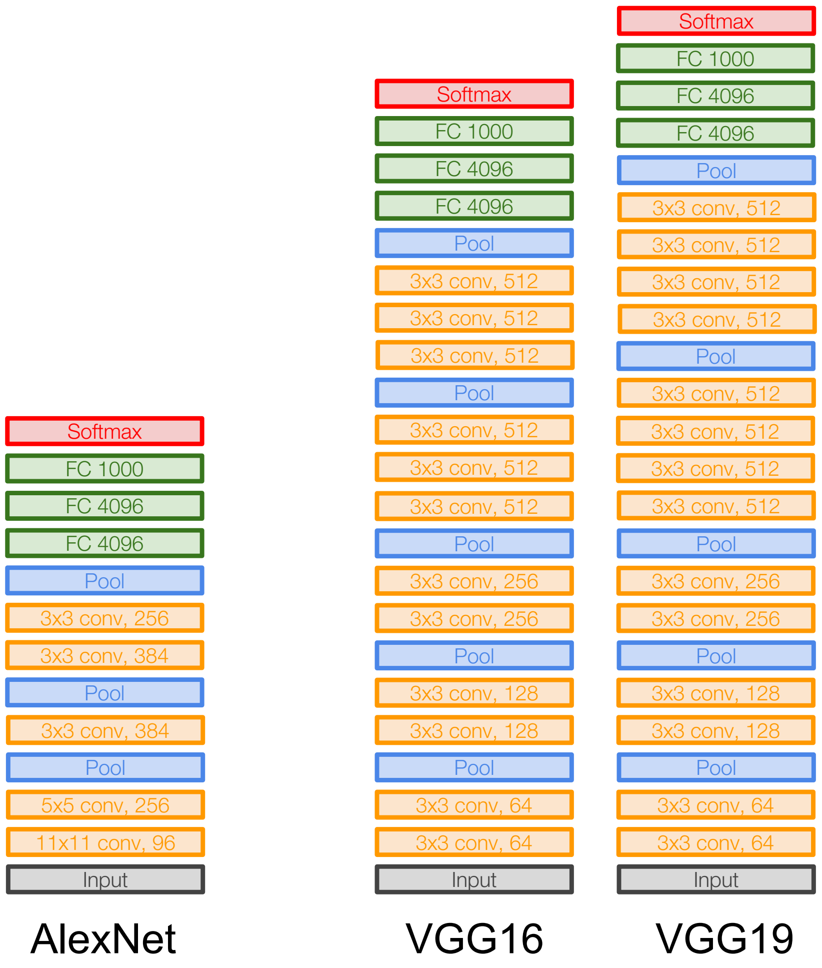
[Simonyan and Zisserman, 2014]

Small filters, Deeper networks

8 layers (AlexNet)
-> 16 - 19 layers (VGG16Net)

Only 3x3 CONV stride 1, pad 1
and 2x2 MAX POOL stride 2

11.7% top 5 error in ILSVRC'13 (ZFNet)
-> 7.3% top 5 error in ILSVRC'14



Case Study: VGGNet

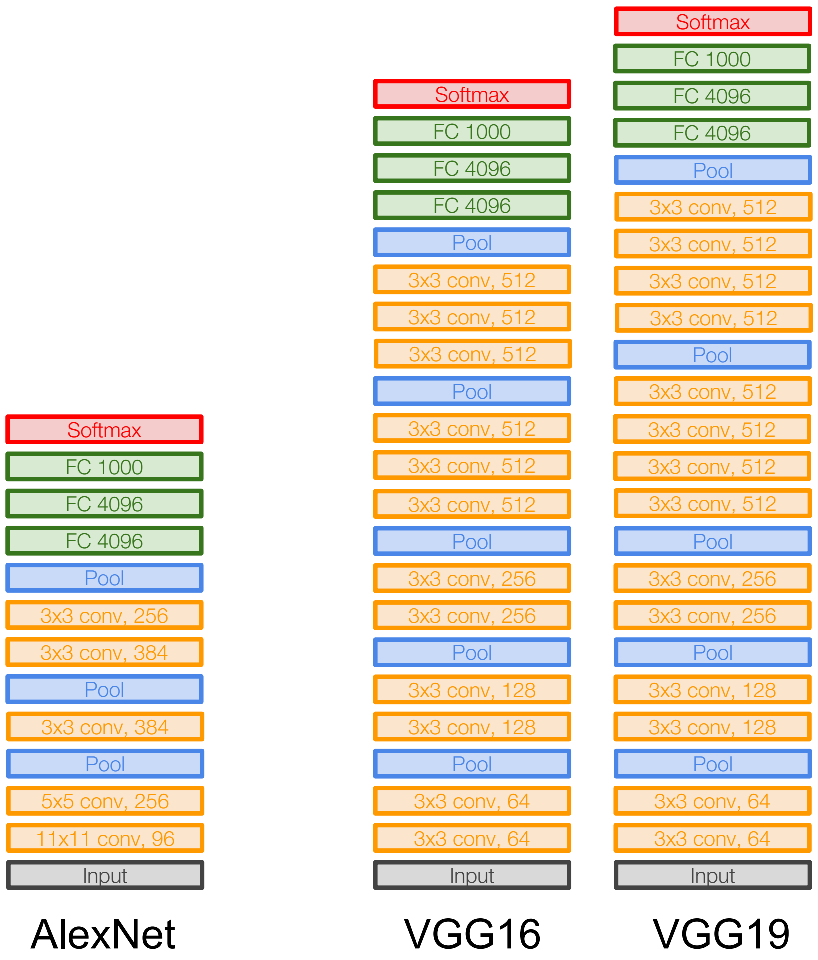
[Simonyan and Zisserman, 2014]

Q: Why use smaller filters? (3x3 conv)

Stack of three 3x3 conv (stride 1) layers has same **effective receptive field** as one 7x7 conv layer

But deeper, more non-linearities

And fewer parameters: $3 * (3^2C^2)$ vs. 7^2C^2 for C channels per layer



(not counting biases)

INPUT: [224x224x3] memory: 224*224*3=150K params: 0

CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*3)*64 = 1,728

CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*64)*64 = 36,864

POOL2: [112x112x64] memory: 112*112*64=800K params: 0

CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*64)*128 = 73,728

CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*128)*128 = 147,456

POOL2: [56x56x128] memory: 56*56*128=400K params: 0

CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*128)*256 = 294,912

CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824

CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824

POOL2: [28x28x256] memory: 28*28*256=200K params: 0

CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*256)*512 = 1,179,648

CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296

CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296

POOL2: [14x14x512] memory: 14*14*512=100K params: 0

CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296

CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296

CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296

POOL2: [7x7x512] memory: 7*7*512=25K params: 0

FC: [1x1x4096] memory: 4096 params: 7*7*512*4096 = 102,760,448

FC: [1x1x4096] memory: 4096 params: 4096*4096 = 16,777,216

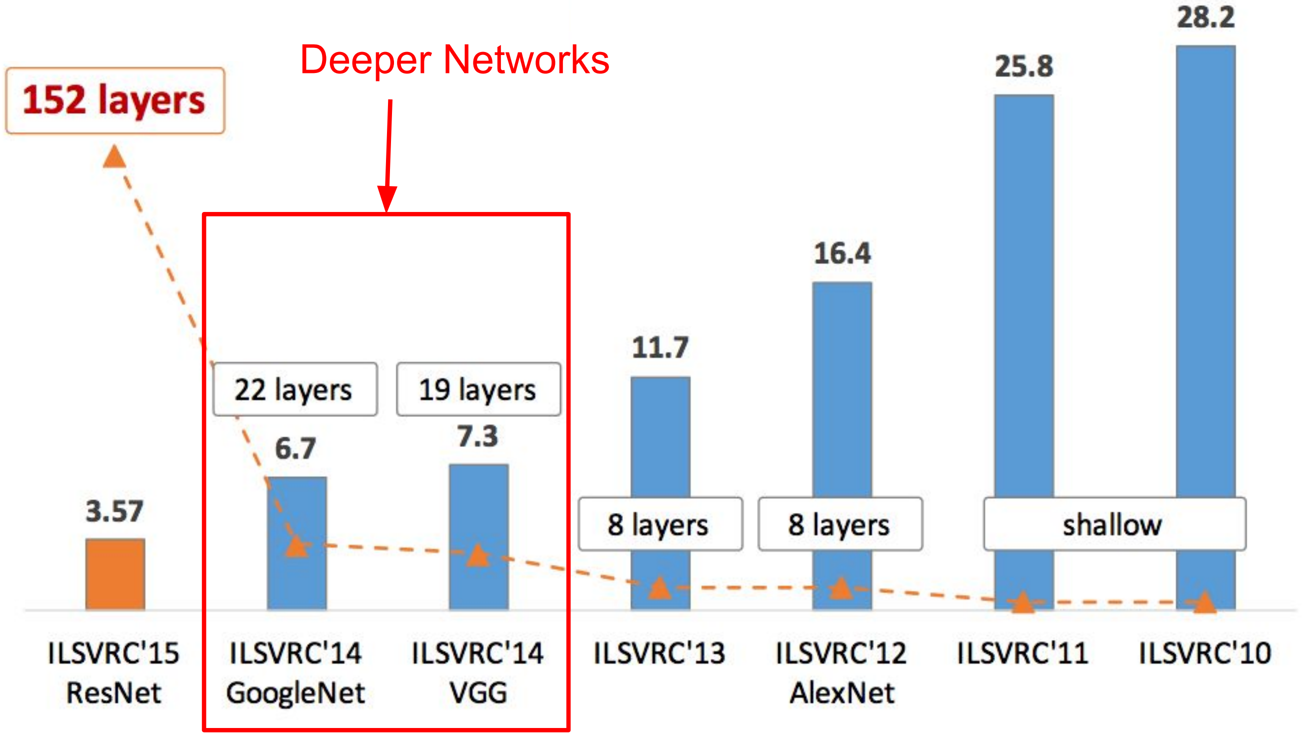
FC: [1x1x1000] memory: 1000 params: 4096*1000 = 4,096,000

TOTAL memory: 24M * 4 bytes ~= 96MB / image (only forward! ~*2 for bwd)

TOTAL params: 138M parameters

VGG16

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

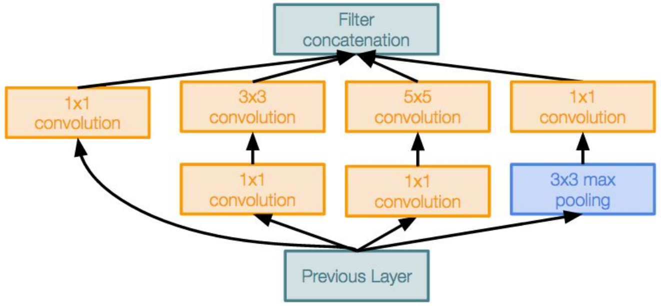


Case Study: GoogLeNet

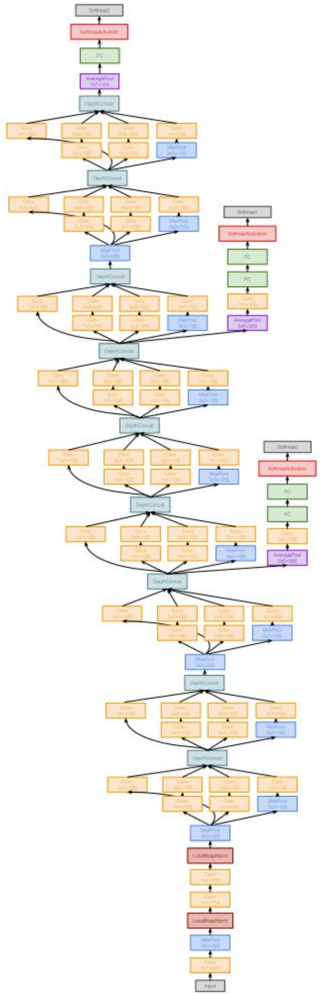
[Szegedy et al., 2014]

Deeper networks, with computational efficiency

- 22 layers
- Efficient "Inception" module
- No FC layers
- Only 5 million parameters!
  12x less than AlexNet
- ILSVRC'14 classification winner
  (6.7% top 5 error)



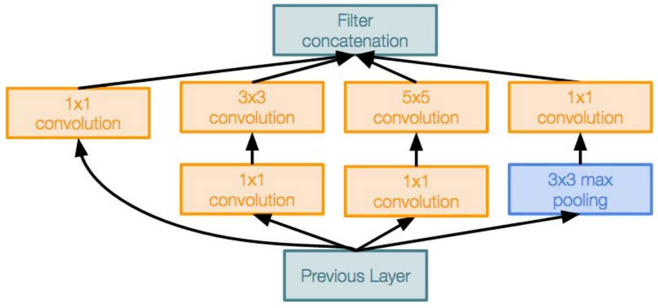
Inception module



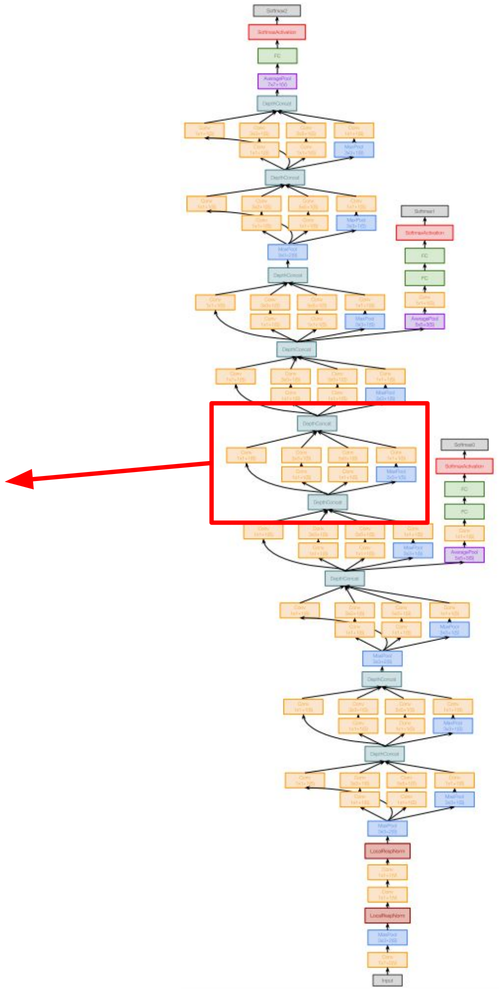
Case Study: GoogLeNet

[Szegedy et al., 2014]

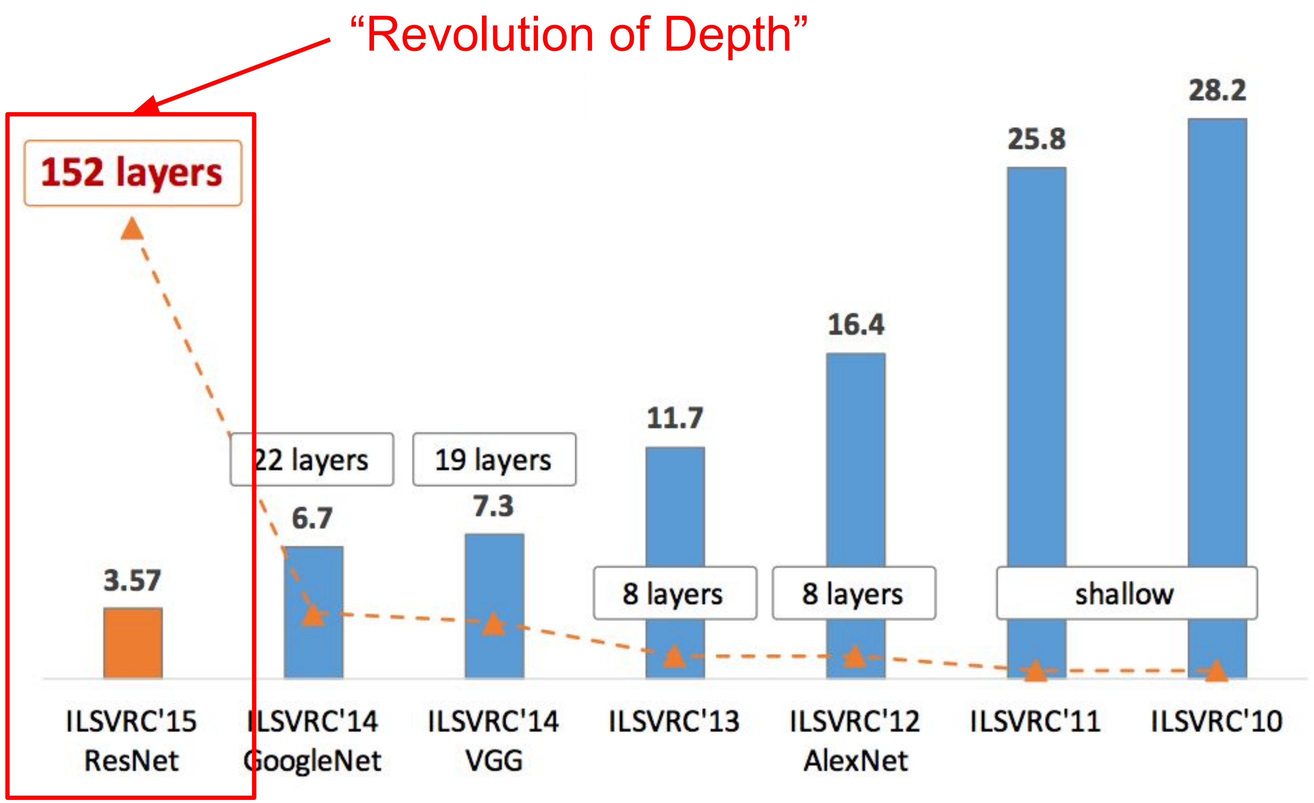
"Inception module": design a good local network topology (network within a network) and then stack these modules on top of each other



Inception module



ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

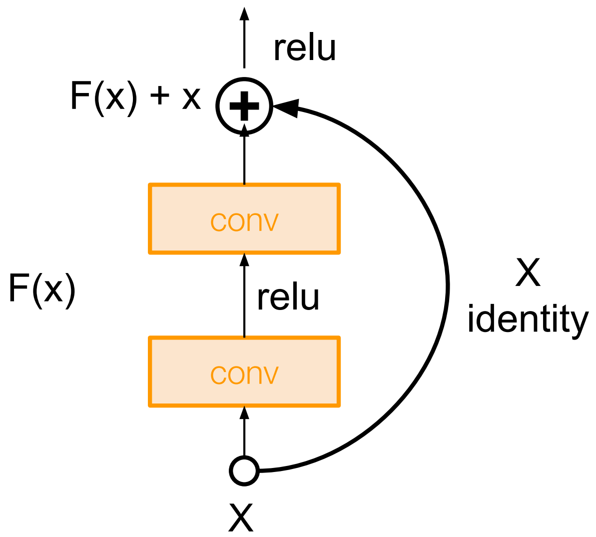


Case Study: ResNet

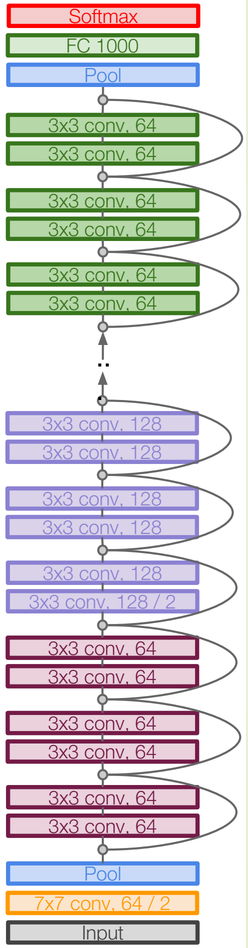
[He et al., 2015]

Very deep networks using residual connections

- 152-layer model for ImageNet
- ILSVRC’15 classification winner (3.57% top 5 error)
- Swept all classification and detection competitions in ILSVRC’15 and COCO’15!



Residual block



Case Study: ResNet

[He et al., 2015]

What happens when we continue stacking deeper layers on a “plain” convolutional neural network?

56-layer model performs worse on both training and test error
-> The deeper model performs worse, but it’s not caused by overfitting!

Case Study: ResNet

[He et al., 2015]

Hypothesis: the problem is an *optimization* problem, deeper models are harder to optimize

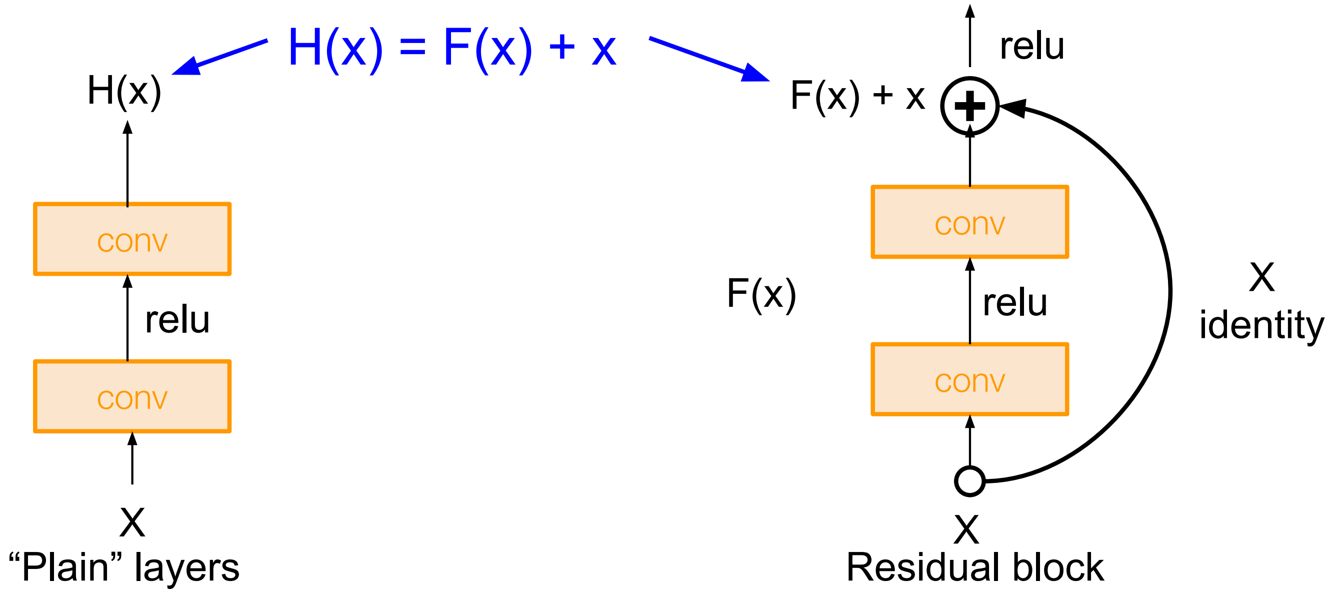
The deeper model should be able to perform at least as well as the shallower model.

A solution by construction is copying the learned layers from the shallower model and setting additional layers to identity mapping.

Case Study: ResNet

[He et al., 2015]

Solution: Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping



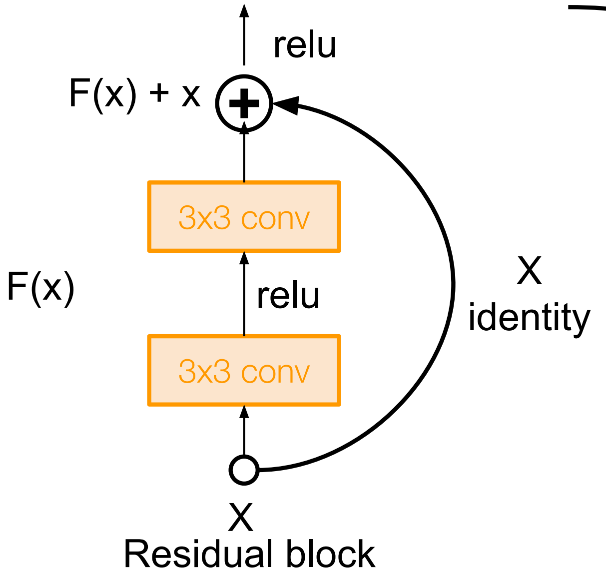
Use layers to fit residual F(x) = H(x) - x instead of H(x) directly

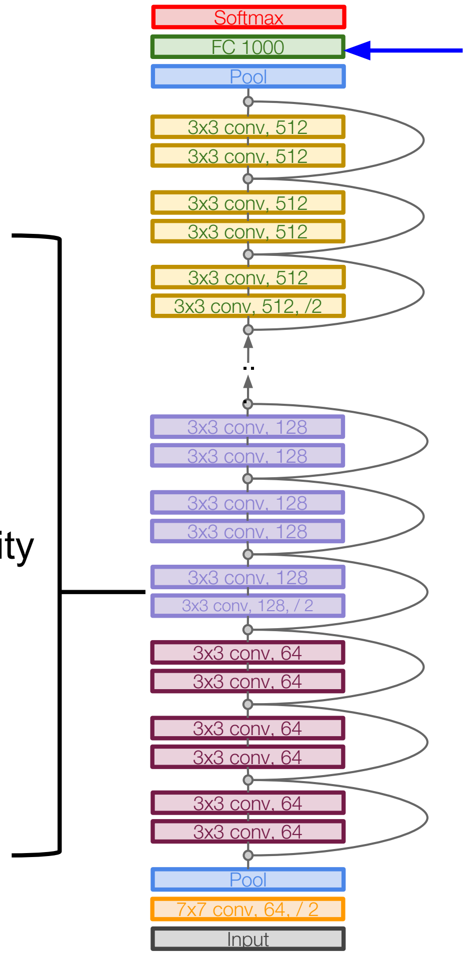
Case Study: ResNet

[He et al., 2015]

Full ResNet architecture:

- Stack residual blocks
- Every residual block has two 3x3 conv layers
- Periodically, double # of filters and downsample spatially using stride 2 (/2 in each dimension)
- Additional conv layer at the beginning
- No FC layers at the end (only FC 1000 to output classes)





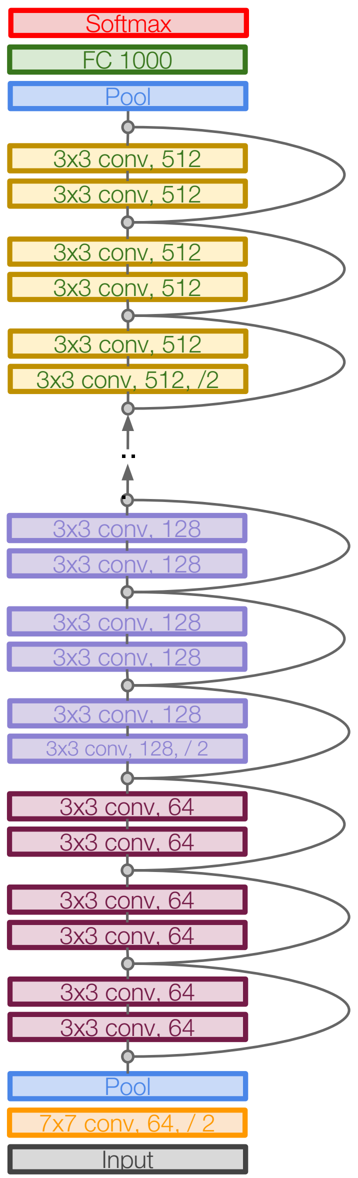
No FC layers besides FC 1000 to output classes

Global average pooling layer after last conv layer

Case Study: ResNet

[He et al., 2015]

Total depths of 34, 50, 101, or 152 layers for ImageNet



Case Study: ResNet

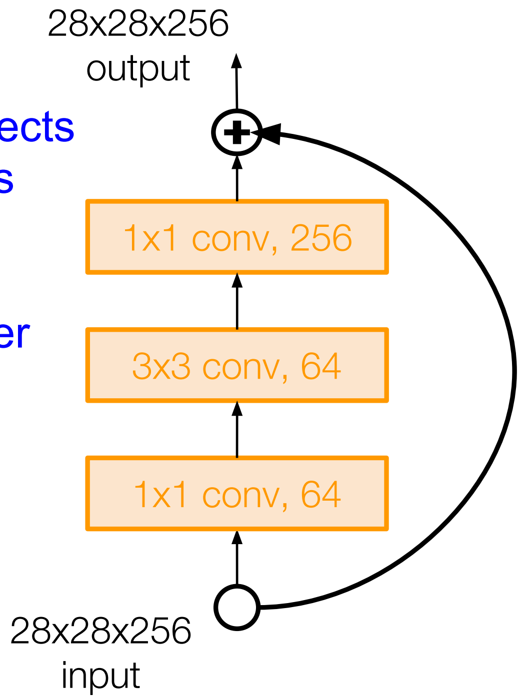
[He et al., 2015]

For deeper networks (ResNet-50+), use “bottleneck” layer to improve efficiency (similar to GoogLeNet)

1x1 conv, 256 filters projects back to 256 feature maps (28x28x256)

3x3 conv operates over only 64 feature maps

1x1 conv, 64 filters to project to 28x28x64



Case Study: ResNet

[He et al., 2015]

Training ResNet in practice:

- Batch Normalization after every CONV layer
- Xavier/2 initialization from He et al.
- SGD + Momentum (0.9)
- Learning rate: 0.1, divided by 10 when validation error plateaus
- Mini-batch size 256
- Weight decay of 1e-5
- No dropout used

Case Study: ResNet

[He et al., 2015]

Experimental Results

- Able to train very deep networks without degrading (152 layers on ImageNet, 1202 on Cifar)
- Deeper networks now achieve lowing training error as expected
- Swept 1st place in all ILSVRC and COCO 2015 competitions

MSRA @ ILSVRC & COCO 2015 Competitions

- **1st places in all five main tracks**
  - ImageNet Classification: *"Ultra-deep"* (quote Yann) 152-layer nets
  - ImageNet Detection: 16% better than 2nd
  - ImageNet Localization: 27% better than 2nd
  - COCO Detection: 11% better than 2nd
  - COCO Segmentation: 12% better than 2nd

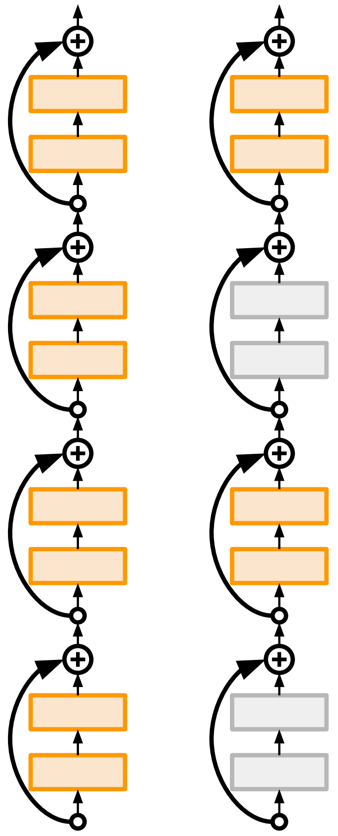
ILSVRC 2015 classification winner (3.6% top 5 error) -- better than "human performance"! (Russakovsky 2014)

Improving ResNets...

Deep Networks with Stochastic Depth

[Huang et al. 2016]

- Motivation: reduce vanishing gradients and training time through short networks during training
- Randomly drop a subset of layers during each training pass
- Bypass with identity function
- Use full deep network at test time

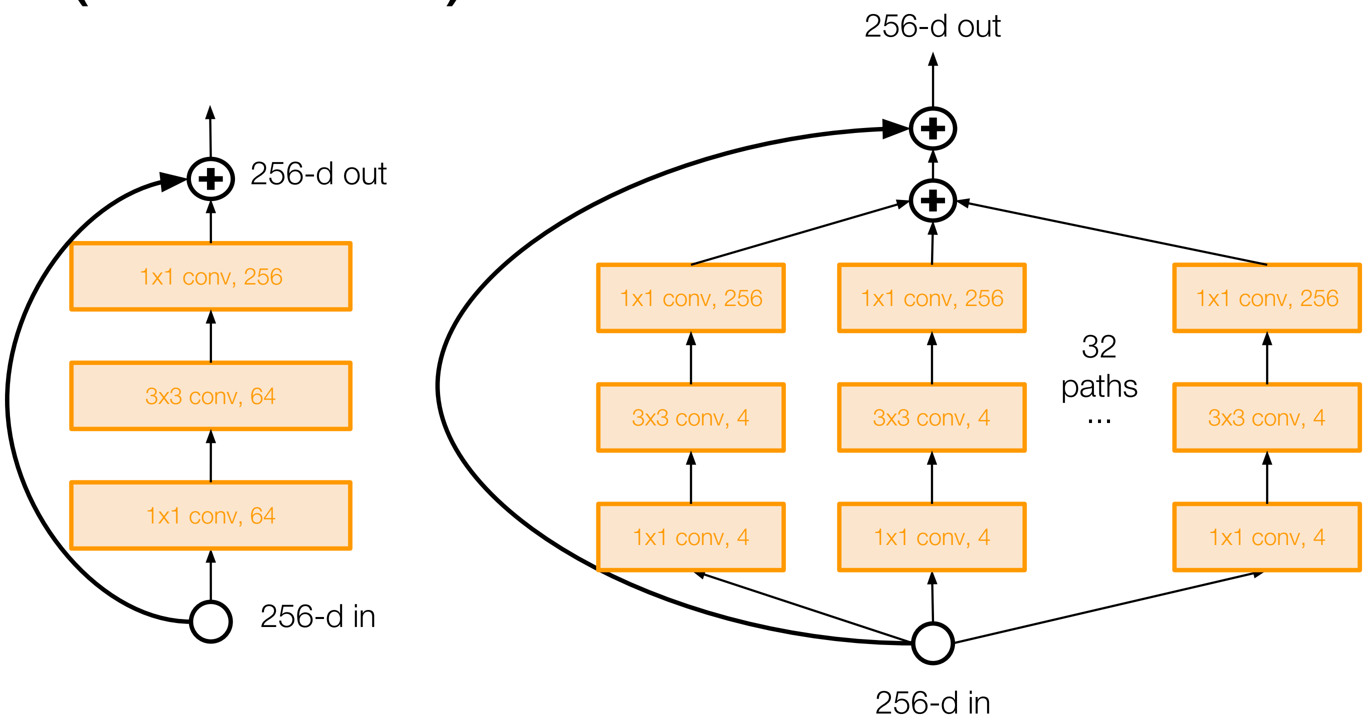


Improving ResNets...

Aggregated Residual Transformations for Deep Neural Networks (ResNeXt)

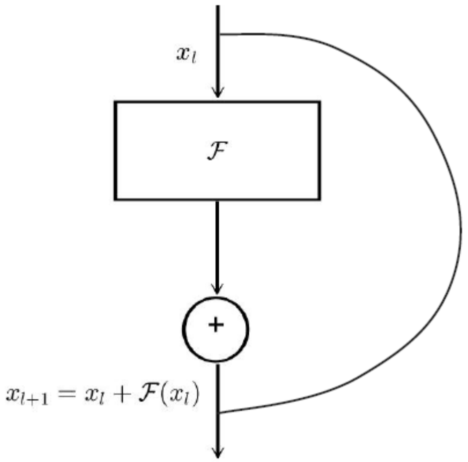
[Xie et al. 2016]

- Also from creators of ResNet
- Increases width of residual block through multiple parallel pathways ("cardinality")
- Parallel pathways similar in spirit to Inception module



ResNet in Noisy Ensembles: Feynman-Kac Equations

- ResNet as a discretization of transport PDE



Plain Net: $x_{l+1} = \mathcal{G}(x_l)$
ResNet: $x_{l+1} = x_l + \mathcal{F}(x_l)$

$$\begin{cases} x(0) = \hat{x}, \\ x(t_{k+1}) = x(t_k) + \Delta t \cdot \overline{F}(x(t_k), W(t_k)), \;\; k = 0, 1, \ldots, L-1, \\ \hat{y} \doteq f(x(1)), \end{cases}$$

where $\overline{F} \doteq \frac{1}{\Delta t}\mathcal{F}$, and $f(x) = \mathrm{softmax}(W_{\mathrm{FC}} \cdot x)$.

Continuous limit

$$\begin{cases} \frac{dx(t)}{dt} = \overline{F}(x(t), W(t)), \\ x(0) = \hat{x}, \\ \hat{y} = f(x(1)), \end{cases}$$

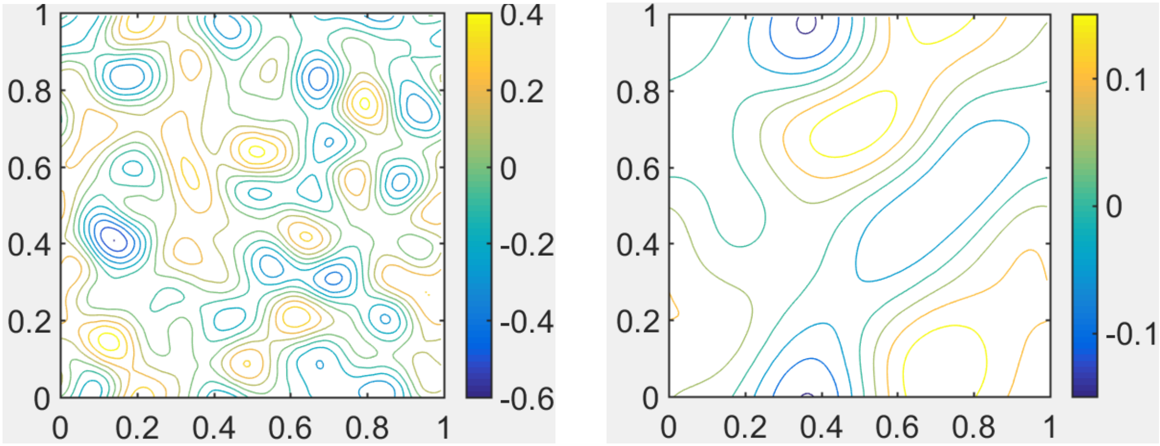
characteristic curves of the following transport equation (TE)

$$\frac{\partial u}{\partial t}(x,t) + \overline{F}(x, W(t)) \cdot \nabla u(x,t) = 0, \quad x \in \mathbb{R}^d.$$

[Bao Wang, B. Yuan, Zuoqiang Shi, Stan Osher, arXiv:1811.10745]

- Feynman-Kac Equation by injective Noise:

$$\begin{cases} \frac{\partial u}{\partial t} + F(x, W(t)) \cdot \nabla u + \frac{1}{2}\sigma^2 \Delta u = 0, \quad x \in \mathbb{R}^d, \ t \in [0,1), \\ u(x,1) = f(x). \end{cases}$$



(a) $\sigma = 0.01$ (b) $\sigma = 0.1$

Figure: (a) and (b) are solutions of the convection-diffusion equation, Eq. (1), at $t = 0$ with different diffusion coefficients σ.

Provable Robustness

[O. Ladyzhenskaja et al. Linear and Quasilinear Equations of Parabolic Type]

Theorem (Stability) Let $\overline{F}(x,t)$ be Lipschitz in both x and t, and $f(x)$ is bounded. For the following terminal value problem of convection-diffusion equation ($\sigma \neq 0$)

$$\begin{cases} \frac{\partial u}{\partial t}(x,t) + \overline{F}(x, W(t)) \cdot \nabla u(x,t) + \frac{1}{2}\sigma^2 \Delta u(x,t) = 0, \quad x \in \mathbb{R}^d, \quad t \in [0,1), \\ u(x,1) = f(x). \end{cases}$$

we have

$$|u(x+\delta,0) - u(x,0)| \leq C \left(\frac{\|\delta\|_2}{\sigma} \right)^{\alpha}$$

for some constant $\alpha > 0$ if $\sigma \leq 1$. C is a constant that depends on d, $\|f\|_\infty$, and $\|\overline{F}\|_{L^\infty_{x,t}}$.

Reference

- [Abadi et al., 2015] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Kaiser, L., Kudlur, M., Levenberg, J., Man, D., Monga, R., Moore, S., Murray, D., Shlens, J., Steiner, B., Sutskever, I., Tucker, P., Vanhoucke, V., Vasudevan, V., Vinyals, O., Warden, P., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems.
- [Bello et al., 2017] Bello, I., Zoph, B., Vasudevan, V., and Le, Q. V. (2017). Neural Optimizer Search with Reinforcement Learning. In Proceedings of the 34th International Conference on Machine Learning.
- [Bengio et al., 2009] Bengio, Y., Louradour, J., Collobert, R., and Weston, J. (2009). Curriculum learning. Proceedings of the 26th annual international conference on machine learning, pages 41–48.
- [Dean et al., 2012] Dean, J., Corrado, G. S., Monga, R., Chen, K., Devin, M., Le, Q. V., Mao, M. Z., Ranzato, M. A., Senior, A., Tucker, P., Yang, K., and Ng, A. Y. (2012). Large Scale Distributed Deep Networks. NIPS 2012: Neural Information Processing Systems, pages 1–11.
- [Denkowski and Neubig, 2017] Denkowski, M. and Neubig, G. (2017). Stronger Baselines for Trustable Results in Neural Machine Translation. In Workshop on Neural Machine Translation (WNMT).

Reference

- [Dinh et al., 2017] Dinh, L., Pascanu, R., Bengio, S., and Bengio, Y. (2017). Sharp Minima Can Generalize For Deep Nets. In Proceedings of the 34 th International Conference on Machine Learning.
- [Dozat, 2016] Dozat, T. (2016). Incorporating Nesterov Momentum into Adam. ICLR Workshop, (1):2013–2016.
- [Dozat and Manning, 2017] Dozat, T. and Manning, C. D. (2017). Deep Biaffine Attention for Neural Dependency Parsing. In ICLR 2017.
- [Duchi et al., 2011] Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. Journal of Machine Learning Research, 12:2121–2159.
- [Huang et al., 2017] Huang, G., Li, Y., Pleiss, G., Liu, Z., Hopcroft, J. E., and Weinberger, K. Q. (2017). Snapshot Ensembles: Train 1, get M for free. In Proceedings of ICLR 2017.
- [Ioffe and Szegedy, 2015] Ioffe, S. and Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. arXiv preprint arXiv:1502.03167v3.
- [Ruder, 2016] Ruder, S. (2016). An overview of gradient descent optimization algorithms. arXiv preprint arXiv:1609.04747.

Reference

- [Nesterov, 1983] Nesterov, Y. (1983). A method for unconstrained convex minimization problem with the rate of convergence o(1/k2). Doklady ANSSSR (translated as Soviet.Math.Docl.), 269:543–547.
- [Niu et al., 2011] Niu, F., Recht, B., Christopher, R., and Wright, S. J. (2011). Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. pages 1–22.
- [Qian, 1999] Qian, N. (1999). On the momentum term in gradient descent learning algorithms. Neural networks : the official journal of the International Neural Network Society, 12(1):145–151.
- [Wang, Yuan, Shi, Osher, 2018] Bao Wang, B. Yuan, Z. Shi, S. Osher (2019). ResNets Ensemble via the Feynman-Kac Formalism to Improve Natural and Robust Accuracies, arXiv:1811.10745, NeurIPS 2019.
- [Zeiler, 2012] Zeiler, M. D. (2012). ADADELTA: An Adaptive Learning Rate Method. arXiv preprint arXiv:1212.5701.
- [Zhang et al., 2015] Zhang, S., Choromanska, A., and LeCun, Y. (2015). Deep learning with Elastic Averaging SGD. Neural Information Processing Systems Conference (NIPS 2015), pages 1–24.

Thank you!